Formal aspects of component software

M. Sun, B. Schatz

SEN-E0902

Copyright © 2009, Centrum Wiskunde & Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Science Park 123, 1098 XG Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Sun Meng & Bernhard Schätz (Eds.)

Formal Aspects of Component Software

6th International Workshop, FACS 2009

Eindhoven, the Netherlands

2-3 November, 2009

Pre-Proceedings

Formal aspects of component software

ABSTRACT
This is the pre-proceedings of 6th International Workshop on Formal Aspects of Component Software (FACS'09).

2000 Mathematics Subject Classification: 00
1998 ACM Computing Classification System: A.0
Keywords and Phrases: Proceedings

Preface

This volume contains the proceedings of the 6th International Workshop on Formal Aspects of Component Software (FACS09). FACS'09 was held in Eindhoven, the Netherlands on November 2-3, 2009 as a satellite event of the Formal Methods Week.

The objective of FACS'09 is to bring together researchers in the areas of component software and formal methods to promote a deep understanding of this paradigm and its applications. Component-based software has emerged as a promising paradigm to deal with the ever increasing need for mastering systems' complexity, for enabling evolution and reuse, and for bringing sound production and engineering standards into software engineering. However, many issues in component-based software development remain open and pose challenging research questions. Formal methods consist of mathematically-based techniques for the specification, development and verification of software and hardware systems. They have shown their great utility for setting up the formal foundations of component software and working out challenging issues such as mathematical models for components, their composition and adaptation, or rigorous approaches to verification, deployment, testing and certification.

FACS'09 is the 6th event in a series of workshops, founded by the International Institute for Software Technology of the United Nations University (UNU-IIST). The first FACS workshop was co-located with FM'03 (Pisa, Italy, September 2003). The following FACS workshops were organised as standalone events, respectively at UNU-IIST in Macau (October 2005), at Charles University in Prague (September 2006), at INRIA in Sophia-Antipolis (September 2007), and at University of Málaga in Spain (September 2008).

FACS'09 was scheduled as part of the first Formal Methods Week (FMweek), to be hosted by Technische Universiteit Eindhoven in Eindhoven, the Netherlands, in November 2009. The scientific program includes 13 regular papers and 2 invited talks, delivered by Gert Döhmen from AIRBUS Deutschland GmbH, Hamburg, Germany and by Jan Rutten from CWI, Amsterdam, the Netherlands. Each paper was reviewed by at least three Program Committee members. The entire reviewing process was supported by the EasyChair Conference System.

We would like to express our gratitude to all the researchers who submitted their work to the workshop and to all colleagues who served on the Program Committee and helped us in preparing a high-quality workshop program. We are also most grateful to the invited speakers for the willingness to present their research and to share their own perspectives on formal methods for component software at the workshop.

Without the support of TU Eindhoven, CWI and UNU-IIST this workshop could not have happened. We are most grateful to all organizing institutions and staff for supporting FACS and providing an organizational framework for the workshop. In particular, we are deeply indebted to Erik de Vink and Tijn Borghuis at TU Eindhoven, Jos van der Werf and Jan Schipper at CWI, and Kitty Chan Iok Sam at UNU-IIST Macau, for their help in managing the practical aspects of this event.

Sun Meng and Bernhard Schätz
FACS'09 PC Chairs

Organization

Steering Committee

Zhiming Liu (IIST UNU, Macau, China, coordinator)
Farhad Arbab (CWI, The Netherlands)
Luis Barbosa (Universidade do Minho, Portugal)
Carlos Canal (University of Málaga, Spain)
Markus Lumpe (Swinburne University of Technology, Australia)
Eric Madelaine (INRIA, Sophia-Antipolis, France)
Vladimir Mencl (Charles University, Prague, Czech Republic, and University of Canterbury, New Zealand)
Corina Pasareanu (NASA Ames Research Center, USA)
Sun Meng (CWI, the Netherlands)
Bernhard Schätz (Technical University of Munich, Germany)

Programme Chairs

Sun Meng (CWI, the Netherlands)
Bernhard Schätz (Technical University of Munich, Germany)

Programme Committee

Farhad Arbab (CWI, The Netherlands)
Luis Barbosa (Universidade do Minho, Portugal)
Frank S. de Boer (CWI, The Netherlands)
Christiano Braga (Universidad Complutense de Madrid, Spain)
Carlos Canal (Universidad de Málaga, Spain)
Paolo Ciancarini (Universita di Bologna, Italy)
Rolf Hennicker (Ludwig-Maximilians-Universitaet Muenchen, Germany)
Atsushi Igarashi (Kyoto University, Japan)
Einar Broch Johnsen (Universitetet i Oslo, Norway)
Ying Liu (IBM China Research, China)
Markus Lumpe (Swinburne University of Technology, Australia)
Eric Madelaine (INRIA, Centre Sophia Antipolis, France)
Corina Pasareanu (NASA Ames, USA)
Frantisek Plasil (Charles University, Czech Republic)
Anders Ravn (Aalborg University, Denmark)
Ralf Reussner (Universitaet Karlsruhe, Germany)
Heinrich Schmidt (RMIT University, Australia)
Marjan Sirjani (University of Tehran, Iran, and Reykjavik University, Iceland)
Volker Stolz (UNU-IIST, MACAU)
Carolyn Talcott (SRI International, USA)
Dang Van Hung (Vietnam National University, Vietnam)
Naijun Zhan (IOS, China)

External Reviewers

Table of Contents

Component-based design for avionics systems

Gert Döhmen

AIRBUS Operations GmbH
HAMBURG, Germany
Email: gert.doehmen@airbus.com

Abstract

Current trend in aeronautics goes from system-specific avionics to a network of standardized processing- and I/O-modules, so-called Integrated Modular Avionics (IMA) platform, which can host different systems. It necessitates advanced processes and methods to provide a cost-efficient development of software-intensive systems. The talk will discuss current and future IMA and how a component-based approach can pave the way from functional and extra-functional requirements of a system to its implementation to be hosted by IMA.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Coalgebraic Methods for Component Connectors

Jan Rutten

CWI
Amsterdam, the Netherlands
Email: Jan.Rutten@cwi.nl

Abstract

We present a small toolset of coalgebraic structures and methods (mainly streams, Mealy automata, and coinduction) and illustrate their use in the modelling of component connectors.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Action Prefixes: Reified Synchronization Paths in Minimal Component Interaction Automata

Markus Lumpe[1]

Faculty of Information & Communication Technologies, Swinburne University of Technology
P.O. Box 218, Hawthorn, VIC 3122, AUSTRALIA

Abstract

Component Interaction Automata provide a fitting model to capture and analyze the temporal facets of hierarchical-structured component-oriented software systems. However, the rules governing composition, as is typical for all automata-based approaches, suffer from *combinatorial state explosion*, an effect that can have significant ramifications on the successful application of the *Component Interaction Automata* formalism to real-world scenarios. We must, therefore, find some appropriate means to counteract state explosion – one of which is *partition refinement* through weak bisimulation. But, while this technique can yield the desired state space reduction, it does not consider *synchronization cliques*, *i.e.*, groups of states that are interconnected solely by internal synchronization transitions. Synchronization cliques give rise to *action prefixes* that capture pre-conditions for a component's ability to interact with the environment. Current practice does not pay attention to these cliques, but ignoring them can result in a loss of valuable information. For this reason we show, in this paper, how synchronization cliques emerge and how we can capture their behavior in order to make state space reduction aware of the presence of synchronization cliques.

Keywords: Component Interaction Automata, Partition Refinement, Emerging Properties

1 Introduction

Component-based software engineering has become the prevalent trend in present-day software and system engineering [6]. In this approach the focus is on *well-defined interfaces* [3,7,11,26] that provide appropriate means for decomposing an engineered system into logical and interacting entities, the *components*, and constructing their respective aggregations, the *composites*, to yield the desired system functionality at matching levels of abstraction and granularity. Moreover, according to this technique, new components are created by combining pre-existing ones with new software, the *glue* [24], using only the information published in the interface specifications of the components being composed.

Component interfaces can convey a variety of information [1] that collectively form a *contractual specification*. Ideally, all assumptions about a component's environment should be stated explicitly and formally as part of the interface specification [25]. However, even though interfaces must be organized in such a way that contractual specifications guarantee safe deployment in new contexts, the information contained in these interfaces must not

[1] Email: mlumpe@swin.edu.au

provide any instruments to circumvent component encapsulation. On the other hand, the purpose of contractual specifications is to prevent errors, at both design time and run-time. Therefore, component contracts should impose a well-balanced set of constraints to enforce contractual obligations, but must be defined in a manner so that the reasons why a particular contract verification has failed are self-evident [4].

In this paper, we are concerned with the specification of *behavioral* and *synchronization* contracts [1] between interacting components. In particular, we study the effectiveness of *Component Interaction Automata* [2,5], an automata-based modeling language for the specification of hierarchical-structured component-based systems. Components synchronize through answering mutual service requests. However, some service requests should only occur in certain situations [23] depending on the component's readiness to satisfy a given request (*pre-condition*) and its cumulative interaction profile (*post-condition*). The description of these temporal aspects corresponds best to *finite automata* in which *acceptable service requests* are modeled as state transitions between activating sets (*i.e.*, states of the modeling automaton) [23].

Unfortunately, automata-based formalisms suffer from *combinatorial state explosion*, a major obstacle to the successful application of these approaches for the specification of the interactive behavior in component-based systems. More precisely, to capture the complete behavior of an automata-based system, we need to construct the *product automaton* of the system's individual components [10]. This operation exhibits exponential space and time complexity and the resource consumption quickly reaches a level at which an effective specification of a composite system is not feasible anymore [13]. We need, therefore, workable abstraction methods that allow for a reduction of the composite state space at acceptable costs.

For this reason, we have developed a bisimulation-based partition refinement algorithm for *Component Interaction Automata* [13]. Partition refinement [9,20] is a state space reduction technique that, driven by a corresponding equivalence relation, merges separate equivalent states into one unifying representative. On termination, partition refinement yields a new automaton that reproduces the behavior of the original one up to the defined equivalence, but is *minimal* (*i.e.*, a fixed-point) with respect to the number of required states.

Partition refinement can effectively reduce the size and complexity of composite component interaction automata [13]. There are, however, instances in which partition refinement produces unexpected outcomes. In particular, we observe a frequent appearance of new, *non-deterministic* transitions in minimal composite component interaction automata, even when there were none before. These non-determistic transitions can cause harm, as their elimination may require exponentially more states [10], which is clearly not a desirable scenario.

Upon closer inspection we find that these new non-deterministic transitions are directly linked to states with internal component synchronizations that become unified as result of partition refinement. Following network theory [17], these states form *synchronization cliques*, groups of states, that embed in their structure *regular sublanguages* over an alphabet of internal component synchronizations. The sentences of these regular sublanguages serve as *prefixes* (or pre-conditions) in the interface of a given composite component interaction automaton. Before refinement, these prefixes are woven into the fabric of the composite automaton. Partition refinement, however, is blind for this additional information, as, independent of its presence, observable equivalence is always preserved between the original and the reduced automaton.

Synchronization cliques are *intrinsic* to automata-based approaches that enumerate internal synchronization actions [6,2,5,14,27] rather than modeling them by τ – a *perfect action* [16]. As a consequence, an external observer can monitor not only the occurrences of internal synchronizations (through the passing of time), but also the order in which actions actually trigger the internal synchronizations. We can capture the alternating sequences of states and internal synchronization actions in synchronization paths [5,13]. However, weak bisimulation is an equivalence relation that abstracts from internal actions, resulting in a loss of information, including the ability to monitor the sequence of internal synchronizations. We show, in this paper, that we can recover this information by representing the existing synchronization paths in a system as *action prefixes* in the corresponding reduced component interaction automaton.

The rest of this paper is organized as follows: in Section 2, we review the *Component Interaction Automata* formalism and demonstrate its expressive power on a tailored version of a simple e-commerce application. We proceed by developing the core ingredients of observable equivalence and partition refinement for component interaction automata in Section 3 and construct, in Section 4, the machinery to distill action prefixes from synchronization cliques. We discuss possible implications of the existence of synchronization cliques in Section 5 and conclude with a brief summary of our main observations and an outlook to future work in Section 6.

2 The Component Interaction Automata Modeling Language

I/O Automata [14], *Interface Automata* [6], and *Team Automata* [27] have all emerged as light-weight contenders for capturing concisely the *temporal* aspects of component-based software systems. These formalisms use an *automata-based language* to represent both the assumptions about a system's environment and the order in which interactions with the environment can occur. However, none of these models cater directly for multiple instantiations of the same component within a single system or allow for a more fine-grained specification of hierarchical relationships between organizational entities in a system.

These restrictions have been relaxed in *Component Interaction Automata* [2,5]. In this approach we find two new concepts: a *hierarchy of component names* and *structured labels*. The former provides us with a means to record the architecture of a composite system, whereas the latter paves the way to specify the *action*, the *originating component*, and the *target component* in the transitions of component interaction automata, a feature that allows us to disambiguate multiple occurrences of the same component (or action) within a single system. Specifically, the *Component Interaction Automata* formalism supports three forms of structured labels: $(-, a, n)$, receive a from the environment as *input* at component n, $(n, a, -)$, send a from component n as *output* to the environment, and (n_1, a, n_2), components n_1 and n_2 *synchronize* internally through action a.

The *Component Interaction Automata* formalism uses *component identifiers* to uniquely identify specific component instances in a given system. However, a given component identifier can occur at most once in a composite component interaction automaton. This requirement addresses a frequent difficulty in the specification of component-based systems – the difference between components and component instances [12]. The *I/O Automata* and *Interface Automata* formalisms, for example, do not distinguish between components and their instances. Every specification involves only instances. It is for this reason that all actions of composed components have to be pairwise disjoined [6,14] (*i.e.*, a single component can occur at most once in a composite system). In contrast, the *Component Interaction Automata* formalism distinguishes between components and their instances. Each component is instantiated with a unique identifier that we use also to disambiguate the corresponding component transitions. Consider, for example, a component $\mathcal{C}$ that defines an input via action a and two instances of $\mathcal{C}$, named A and B. Then the structured labels for the input transitions of A and B are $(-, a, A)$ and $(-, a, B)$, respectively. The unique component identifiers A and B are what allows for the safe coexistence of multiple instances of the same component (or action) in a given system.

We presuppose an infinite set $\mathcal{A}$ of *component identifiers*. A hierarchy of component names is defined as follows [5]:

Definition 2.1 A hierarchy of component names H is defined recursively by

- $H = (a_1, ..., a_n)$, where $a_1, ..., a_n \in \mathcal{A}$ are pairwise disjoint component identifiers and $S(H) = \cup_{i=1}^{n}\{a_i\}$ denoting the set of component identifiers of H;
- $H = (H_1, ..., H_m)$, where $H_1, ..., H_m$ are hierarchies of component identifiers satisfying $\forall\, 1 \leq i, j \leq m, i \neq j :\ S(H_i) \cap S(H_j) = \emptyset$ and $S(H) = \cup_{i=1}^{m} S(H_i)$ denoting the set of component identifiers of H.

Definition 2.2 A component interaction automaton $\mathcal{C}$ is a quintuple (Q, Act, δ, I, H) where:

- Q is a finite set of states,
- Act is a finite set of actions,
- $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of labeled transitions, where $\Sigma \subseteq \{(S(H) \cup \{-\} \times Act \times S(H) \cup \{-\})\} \setminus \{(\{-\} \times Act \times \{-\})\}$ is the set of structured labels induced by $\mathcal{C}$,
- $I \subseteq Q$ is a non empty set of initial states, and
- H is a tuple denoting $\mathcal{C}$'s hierarchical composition structure.

Each component interaction automaton is further characterized by two sets of $P \subseteq Act$, the provided actions, and $R \subseteq Act$, the required actions, which capture the automaton's enabled interface with an environment. We write $\mathcal{C}_R^P$ to denote an automaton $\mathcal{C}$ that is input-enabled in P and output-enabled in R.

In the original definition [2,5], the set of *provided actions* P and the set of *required actions* R originate from a secondary specification outside the *Component Interaction Automata* formalism. Incorporating these *architectural constraints* into the specification of component interaction automata does not affect the underlying composition rules, but rather make the relationship with the associated automata more explicit and ease the computation of composition [13]. We abbreviate, however, the annotation in a natural way if an automaton is enabled in all actions and omit the corresponding specification.

As motivating example, consider a simple electronic commerce system with three participants [10]: a *Customer*, a *Store*, and a *Bank*. The behavior of the composite system is as follows. The customer may initiate a transaction by passing a voucher to the store. The store will then redeem this voucher with the bank (*i.e.*, the bank will eventually deposit money into the store's account) and, through a third party, ship the ordered goods. In addition, the customer may cancel the order before the store had a chance to redeem the

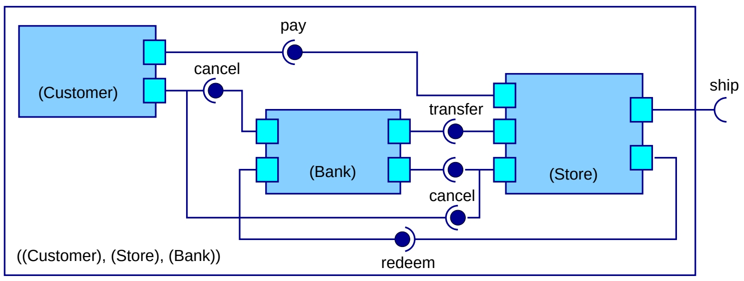


Fig. 1. A simple e-commerce system.

voucher, in which case the voucher will be returned to the customer immediately. We allow the customer to cancel an order with either the store or the bank. The high-level interaction protocol for this system is shown in Figure 1.

We can model *Customer*, *Store*, and *Bank* as component interaction automata as follows. We write $(Customer)$, $(Store)$, and $(Bank)$ to denote the architecture of the component interaction automata *Customer*, *Store*, and *Bank*. More precisely, the three automata are *primitive* (or plain) components with an opaque structure (*i.e.*, no explicit hierarchical relationships):

$$\begin{aligned} Customer = (&\{c_0, c_1\}, \{pay, cancel\},\\ &\{(c_0, (Customer, pay, -), c_1), (c_1, (Customer, cancel, -), c_0)\},\\ &\{c_0\}, (Customer)) \end{aligned}$$

$$\begin{aligned} Store = (&\{s_0, s_1, s_2, s_3, s_4, s_5\}, \{pay, redeem, transfer, ship\},\\ &\{(s_0, (-, pay, Store), s_1), (s_1, (Store, redeem, -), s_2), (s_1, (-, cancel, Store), s_0),\\ &\ (s_2, (-, transfer, Store), s_3), (s_2, (Store, ship, -), s_4), (s_3, (Store, ship, -), s_5),\\ &\ (s_4, (-, transfer, Store), s_5)\}, \{s_0\}, (Store)) \end{aligned}$$

$$\begin{aligned} Bank = (&\{b_0, b_1, b_2, b_3\}, \{cancel, redeem, transfer\},\\ &\{(b_0, (-, cancel, Bank), b_1), (b_0, (-, redeem, Bank), b_2),\\ &\ (b_1, (Bank, cancel, -), b_0), (b_2, (Bank, transfer, -), b_3)\},\\ &\{b_0\}, (Bank)) \end{aligned}$$

The *Customer* automaton has two states and two output transitions (*i.e.*, customer requests). The *Store* automaton, on the other hand, defines six states and seven transitions and guarantees that orders will only be shipped, if the payment voucher has been redeemed successfully. The *Store* receives a voucher (*i.e.*, action pay), money (*i.e.*, action $transfer$), or a cancelation as input and issues as output the shipment of goods and the request to redeem the voucher. Finally, the *Bank* automaton, defining four states and four transitions, mediates *Customer* and *Store*. If the *Store* has not yet cashed the voucher, then the *Customer* can still cancel the order and receive a refund. The *Bank* will forward a cancelation notice to the *Store*. If the *Store* has already submitted the voucher, then the *Bank* will eventually transfer funds to the *Store*. At this point, the *Customer* cannot cancel the order anymore.

The composition of component interaction automata is defined as the *cross-product* over their state spaces. Furthermore, the sets P and R determine, which input and output transitions occur in the composite system (*i.e.*, interface with the environment). By convention, if any state is rendered inaccessible in the composite automaton, then we remove

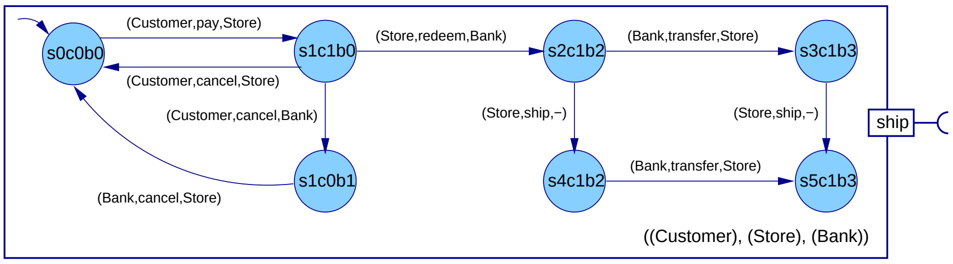


Fig. 2. The composite e-commerce component interaction automaton.

it immediately from the state space in order to obtain the most concise result. The behavior of the composite automaton is *completely* captured by its reachable states.

Definition 2.3 Let $\mathcal{S}_R^P = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$ be a system of pairwise disjoint component interaction automata, where $\mathcal{I}$ is a finite indexing set and P, R are the provided and required actions. Then $\mathcal{C}_R^P = (\prod_{i\in\mathcal{I}} Q_i, \cup_{i\in\mathcal{I}} Act_i, \delta_{\mathcal{S}_R^P}, \prod_{i\in\mathcal{I}} I_i, (H_i)_{i\in\mathcal{I}})$ is a composite component interaction automaton where q_j denotes a function $\prod_{i\in\mathcal{I}} Q_i \to Q_j$, the projection from product state q to jth component state q and

$$\delta_{\mathcal{S}_R^P} = \delta_{OldSync} \cup \delta_{NewSync} \cup \delta_{Input} \cup \delta_{Output}$$

with

$$\begin{aligned}
\delta_{OldSync} &= \{(q,(n_1,a,n_2),q') \mid \exists i \in \mathcal{I} : (q_i,(n_1,a,n_2),q_i') \in \delta_i \;\wedge \\
&\qquad \forall j \in \mathcal{I}, j \neq i :\; q_j = q_j'\},\\
\delta_{NewSync} &= \{(q,(n_1,a,n_2),q') \mid \exists i_1, i_2 \in \mathcal{I}, i_1 \neq i_2 : (q_{i_1},(n_1,a,-),q_{i_1}') \in \delta_{i_1} \wedge \\
&\qquad (q_{i_2},(-,a,n_2),q_{i_2}') \in \delta_{i_2} \wedge \forall j \in \mathcal{I}, i_1 \neq j \neq i_2 : q_j = q_j'\},\\
\delta_{Input} &= \{(q,(-,a,n),q') \mid a \in R \;\wedge\; \exists i \in \mathcal{I} : (q_i,(-,a,n),q_i') \in \delta_i \;\wedge \\
&\qquad \forall j \in \mathcal{I}, j \neq i : q_j = q_j'\},\\
\delta_{Output} &= \{(q,(n,a,-),q') \mid a \in P \;\wedge\; \exists i \in \mathcal{I} : (q_i,(n,a,-),q_i') \in \delta_i \;\wedge \\
&\qquad \forall j \in \mathcal{I}, j \neq i : q_j = q_j'\}.
\end{aligned}$$

The composition rule builds the product automaton for a given system $\mathcal{S}_R^P$. It does so by simultaneously recombining the behavior of all individual component interaction automata $\{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$. The transitions of the composite automaton result from four sets: the transposed preexisting internal synchronizations $\delta_{OldSync}$ of the individual component interaction automata, the newly formed internal synchronizations $\delta_{NewSync}$ due to interactions between the individual component interaction automata, and the sets δ_{Input} and δ_{Output}, the transposed remaining interactions of the product automaton with the environment.

Applied to our e-commerce system, we can denote the composition of the three components *Customer*, *Store*, and *Bank* using the following expression:

$$\mathcal{S}_\emptyset^{\{ship\}} = \{Customer, Store, Bank\},$$

which yields a composite automaton with 7 reachable states (out of 48 product states). Moreover, due to the architectural constraints $P = \{ship\}$ and $R = \emptyset$ the composite system can only interact with its environment by emitting a $ship$ action. A graphical representation of the composite system is shown in Figure 2.

3 Observable Equivalence and Partition Refinement

The problem of *combinatorial state explosion* does not only occur when constructing new composite components or systems, but also when we wish to study their inherent properties [13]. A measure to alleviate state explosion is *partition refinement* [13,9,20,18,8], which allows, by means of some equivalence relation, for the identification of states that exhibit the same interactive behavior with respect to an external observer. Partition refinement merges equivalent states into one and removes the remaining superfluous states and their transitions from the system. We use *bisimulation* [19], in particular a notion of *weak bisimulation* [13,16], as the desired observable equivalence relation for the reduction of component interaction automata. From an external observer's point of view, weak bisimulation yields a *co-inductive* testing strategy in which two component interface automata cannot be distinguished, if they only differ in their internal synchronizations.

However, the *Component Interaction Automata* formalism requires an additional criterion to be met: two component interaction automata A and B are considered equivalent, if and only if they are bisimular and adhere to the same underlying composition structure [13]. In other words, any technique to reduce the complexity of a given component interaction automaton has also to retain its underlying hierarchical composition structure. This means, two states q, p with transitions $(q, (-, a, A), r)$ and $(p, (-, a, B), r)$ must not be equated, as the target components in the transition labels differ.

An important element in the definition of an observable equivalence relation over component interaction automata is the notion of *synchronization path*.

Definition 3.1 If $(n_1, a_1, n'_1) \cdots (n_k, a_k, n'_k) \in \Sigma$ are internal synchronizations of a component interaction automaton $\mathcal{C}$, then we write $q \overset{*}{\Longrightarrow} p$ to denote the reflexive transitive closure of

$$q \xrightarrow{(n_1,a_1,n'_1)} r_1 \xrightarrow{(n_2,a_2,n'_2)} \cdots \xrightarrow{(n_{k-1},a_{k-1},n'_{k-1})} r_{k-1} \xrightarrow{(n_k,a_k,n'_k)} p,$$

called synchronization path between q and p.

Synchronization paths give rise to *weak transitions*.

Definition 3.2 If $l \in \Sigma$ is a structured label, then $q \overset{l}{\Longrightarrow} p$ is a weak transition from q to p over label l, if there exists r, r' such that

$$q \overset{*}{\Longrightarrow} r \xrightarrow{l} r' \overset{*}{\Longrightarrow} p.$$

Using the concept of weak transitions, we can define now a *weak bisimulation* over component interaction automata.

Definition 3.3 Given $A = (Q_A, Act_A, \delta_A, I_A, H)$ and $B = (Q_B, Act_B, \delta_B, H)$, two component interaction automata with an identical composition hierarchy H, a binary relation $\mathcal{R} \subseteq Q \times Q$ with $Q = Q_A \cup Q_B$ is a weak bisimulation, if it is symmetric and $(q, p) \in \mathcal{R}$ implies, for all $l \in \Sigma$, $\Sigma = \Sigma_A \cup \Sigma_B$ being the set of structured labels induced by A and B,

- whenever $q \xrightarrow{l} q'$, then $\exists p'$ such that $p \overset{l}{\Longrightarrow} p'$ and $(q', p') \in \mathcal{R}$.

Two component interaction automata A and B are weakly bisimilar, written $A \approx B$, if they are related by some weak bisimulation.

Using the preceding definition, we can find a new automaton, $\mathcal{R}_{\emptyset}^{\{ship\}}$, capable of reproducing the interactive behavior of our e-commerce systems up to weak bisimulation. $\mathcal{R}_{\emptyset}^{\{ship\}}$ (cf. Figure 3) satisfies two requirements: (i) it interacts with the environment through the structured label $(Store, ship, -)$, and (ii) it adheres to the hierarchical composition structure $((Customer), (Store), (Bank))$.

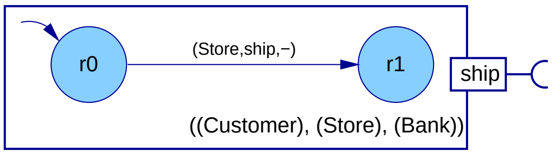


Fig. 3. The weakly-bisimular e-commerce component interaction automaton $\mathcal{R}_{\emptyset}^{\{ship\}}$.

To show that $\mathcal{R}_{\emptyset}^{\{ship\}}$ and $\mathcal{S}_{\emptyset}^{\{ship\}} = \{Customer, Store, Bank\}$ are indeed observable equivalent with respect to an external observer, we have to find a weak bisimulation $\mathcal{R}$ such that $\mathcal{R}_{\emptyset}^{\{ship\}} \approx \mathcal{S}_{\emptyset}^{\{ship\}}$. Such a relation exists and is defined as $\mathcal{R} = r \cup r^{-1}$ with

$$\begin{aligned} r = \{&(s0c0b0, r0), (s1c1b0, r0), (s1c0b1, r0), (s2c1b2, r0), (s3c1b3, r0), \\ &(s4c1b2, r1), (s5c1b3, r1)\}. \end{aligned}$$

There are only two states in $\mathcal{S}_{\emptyset}^{\{ship\}}$, $s2c1b2$ and $s3c1b3$, that require the automaton $\mathcal{R}_{\emptyset}^{\{ship\}}$ to move. Consider, for example, state $s2c1b2$ of $\mathcal{S}_{\emptyset}^{\{ship\}}$. Since $(s2c1b2, r0) \in \mathcal{R}$ and $\mathcal{S}_{\emptyset}^{\{ship\}}$ can perform $(s2c1b2, (Store, ship, -), s4c1b2)$, we select $(r0, (Store, ship, -), r1)$ as a matching transition of $\mathcal{R}_{\emptyset}^{\{ship\}}$ that yields the pair $(s4c1b2, r1) \in \mathcal{R}$, as required. For all states in $\mathcal{S}_{\emptyset}^{\{ship\}}$ other than $s2c1b2$ and $s3c1b3$, $\mathcal{R}_{\emptyset}^{\{ship\}}$ pauses, since all internal synchronization have been factored out in $\mathcal{R}_{\emptyset}^{\{ship\}}$.

The global tactic for the computation of bisimularity is *partition refinement*, which factorizes a given state space into equivalence classes [9,20,18,8]. The result of partition refinement is a surjective function that maps the elements of the original state space to its corresponding representatives of the computed equivalence classes. Partition refinement always yields a minimal automaton.

In the heart of partition refinement is a *splitter function* that determines the granularity of the computed equivalence classes. A splitter for component interaction automata is a boolean predicate $\gamma : Q \times \Sigma \times \mathcal{S} \mapsto \{\texttt{true}, \texttt{false}\}$, where $\mathcal{S} \subseteq 2^Q$ is a set of candidate equivalence classes for $\mathcal{C} = (Q, Act, \delta, I, H)$, the component interaction automaton in question.

Definition 3.4 Let $q \in Q$ be a state, $P' \in \mathcal{S}$ be candidate equivalence class, and $l \in \Sigma$ be a structured label for a component interaction automaton $\mathcal{C} = (Q, Act, \delta, I, H)$. Then

$$\gamma(q, l, P') := \begin{cases} \texttt{true} & \textit{if there is } p' \in P' \textit{ such that } q \overset{l}{\Longrightarrow} p', \\ \texttt{false} & \textit{otherwise} \end{cases}$$

We obtain with this definition a means of expressing the computation of a weakly-bisimular component interaction automaton as the possibility of a set of its states, P, to evolve into another set of states, P', with the same observable behavior, where P' is the equivalence class of P.

Definition 3.5 Let γ be a splitter function generating weakly-bisimular equivalence classes for a component interaction automaton $\mathcal{C} = (Q, Act, \delta, I, H)$. Then

$$refine(X, l, P') := \cup_{P \in X}(\cup_{v \in \{\mathtt{true},\mathtt{false}\}}\{q \mid \forall q \in P.\ \gamma(q, l, P') = v\}) - \{\emptyset\}$$

The actual refinement process is defined by a procedure, *refine* : $\mathcal{X} \times \Sigma \times \mathcal{S} \times \mathcal{X}$, that takes a set of partitions $X \in \mathcal{X}$, a structured label $l \in \Sigma$, and a candicate equivalence class $P' \in \mathcal{S}$ to yield, possibly new, candidate equivalence classes. Partition refinement, starting with $X = \{Q\}$ as initial partition set, repeatedly applies *refine* to X and its derivatives for all $l \in \Sigma$ until a fixed-point is reached [9].

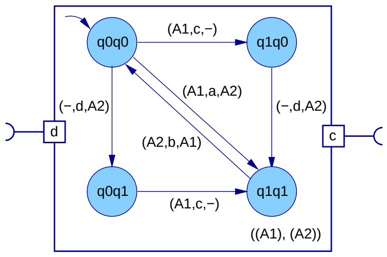
When applied to our composite e-commerce system $\mathcal{S}_{\emptyset}^{\{ship\}}$, partition refinement computes the following equivalence classes:

$$\{r0 = \{s0c0b0, s1c1b0, s1c0b1, s2c1b2, s3c1b3\}, r1 = \{s4c1b2, s5c1b3\}\},$$

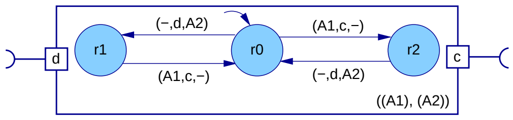
which correspond exactly to the weak bisimulation $\mathcal{R}$, shown earlier. More precisely, we can use these equivalence classes to construct the automaton $\mathcal{R}_{\emptyset}^{\{ship\}}$.

4 Action Prefixes

Partition refinement, up to weak bisimulation, can eliminate most if not all, as in case of $\mathcal{R}_{\emptyset}^{\{ship\}}$, internal synchronizations from a given component interaction automaton. It provides, therefore, a suitable abstraction method that lets system designers focus on the essence of the behavioral protocol defined by a given component interaction automaton. As shown in Section 3, when using the perspective of an external observer, only the output $(Store, ship, -)$ remains in the interface of $\mathcal{R}_{\emptyset}^{\{ship\}}$, a significant improvement with respect to the original complexity of $\mathcal{S}_{\emptyset}^{\{ship\}}$.



(a) $A_{\{d\}}^{\{c\}}$



(b) $A\text{-}R_{\{d\}}^{\{c\}}$

Fig. 4. The weakly-bisimular component interaction automata $A_{\{d\}}^{\{c\}}$ and $A\text{-}R_{\{d\}}^{\{c\}}$.

Unfortunately, there are also situations in which partition refinement can eliminate information, which, in itself, can be viewed vital for the understanding of the interactive behavior of a component interaction automaton. Consider, for example, the two automata $A_{\{d\}}^{\{c\}}$ and $A\text{-}R_{\{d\}}^{\{c\}}$, shown in Figure 4. Both are weakly-bisimular, $A_{\{d\}}^{\{c\}}$, however, contains a subgraph that produces a condition similar to the *small-world effect* [17]. In particular, the states $q0q0$ and $q1q1$ in automaton $A_{\{d\}}^{\{c\}}$ form a *synchronization clique* generating two distinct regular sublanguages, $L_{q0q0} = \{(ab)^n | n \geq 0\} \cup \{b(ab)^m | m \geq 0\}$ and $L_{q1q1} = \{a(ba)^n | n \geq 0\} \cup \{(ba)^m | m \geq 0\}$, of synchronization paths, which can originate from any clique state and terminate in the designated state $q0q0$ and $q1q1$, respectively.

The prefix strings emerging from these sublanguages define pre-conditions that determine, when the transitions $(r0, (-, d, A2), r1)$ and $(r0, (A1, c, -), r2)$ can actually occur in the reduced automaton $A\text{-}R^{\{c\}}_{\{d\}}$.

Definition 4.1 Let $\mathcal{C} = (Q, Act, \delta, I, H)$ be a component interaction automaton and $X \in \mathcal{X}$ be a set of equivalence classes up to weak bisimulation for $\mathcal{C}$. Then a synchronization clique is a non-empty directed graph (V, E), where $V \subseteq Q$ is a set of clique states and $E \subseteq \delta$ is a set of internal synchronizations $(q, (n, a, n'), p)$ with $q, p \in V$ and $q \neq p$, if there exists $P \in X$ such that $q, p \in P$.

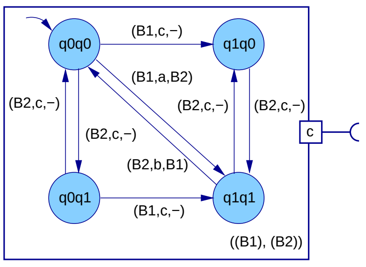
A synchronization clique appears, when partition refinement creates new *reflexive* internal synchronizations due to mapping the endpoints of these transitions onto the same equivalence class. By default, we can ignore preexisting reflexive internal synchronizations, as they can occur, interleaving, in any order. However, the newly formed reflexive internal synchronizations are of a different kind, as their non-reflexive originals encode a specific partial order over internal synchronizations. This property is lost in the refinement process. We can, however, recover it through the notion of action prefixes. For example, in $A\text{-}R^{\{c\}}_{\{d\}}$ the transition $(r0, (-, d, A2), r1)$ may occur only after a, possibly empty, sequence of internal synchronizations over actions b and a, captured by the prefix $[b](ab)^*$ that reifies the required synchronization path to arrive in state $q0q0$ from synchronization clique $\{q0q0, q1q1\}$. In other words, the internal synchronizations have disappeared in $A\text{-}R^{\{c\}}_{\{d\}}$, but we can use the prefix $[b](ab)^*$ to reenforce the existing pre-condition for the occurrence of transition $(r0, (-, d, A2), r1)$ in $A\text{-}R^{\{c\}}_{\{d\}}$. That is, $(r0, (-, d, A2), r1)$ can occur without any proceeding internal synchronizations, only after a single b, or only after a sequence of paired a's and b's.

The presence of synchronization cliques can cause even more worries, as illustrated in Figure 5. The automaton $B^{\{c\}}_{\emptyset}$ is defined as the composition of the following two automata $B1$ and $B2$[2]:

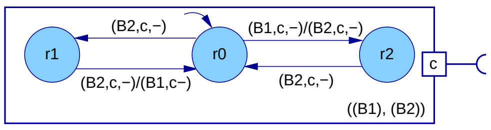
$$
\begin{aligned}
B1 = (&\{q_0, q_1\}, \{a, b, c\},\\
&\{(q_0, (B1, a, -), q_1), (q_0, (B1, c, -), q_1), (q_1, (-, b, B1), q_0)\},\\
&\{q_0\}, (B1))\\
B2 = (&\{q_0, q_1\}, \{a, b, c\},\\
&\{(q_0, (-, a, B2), q_1), (q_0, (B2, c, -), q_1), (q_1, (B2, c, -), q_0), (q_1, (B2, b, -), q_0)\},\\
&\{q_0\}, (B2))
\end{aligned}
$$

The composition of $B1$ and $B2$, the automaton $B^{\{c\}}_{\emptyset}$, yields also a synchronization clique generating the corresponding sublanguages $L_{q0q0} = \{(ab)^n | n \geq 0\} \cup \{b(ab)^m | m \geq 0\}$ and $L_{q1q1} = \{a(ba)^n | n \geq 0\} \cup \{(ba)^m | m \geq 0\}$. Unlike $A^{\{c\}}_{\{d\}}$, however, the reduction of $B^{\{c\}}_{\emptyset}$ results in a non-deterministic specification (cf. Figure 5(b)). It is transition $(q_0, (B1, c, -), q_1)$ of $B1$ that enables this phenomenon, not the flip-flop between $B2$'s states over c. Fortunately, action prefixes provide us with the means to disambiguate the conflicting transitions. In particular, $(r0, (B2, c, -), r1)$ can only occur after a synchro-

[2] These automata have been especially designed to reproduce an effect, which we have observed in many system specifications that we have analyzed over time [13].



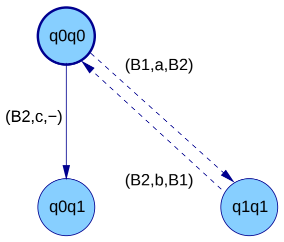
(a) $B_{\emptyset}^{\{c\}}$



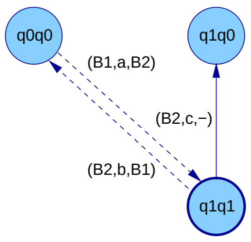
(b) $B\text{-}R_{\emptyset}^{\{c\}}$

Fig. 5. The weakly-bisimular component interaction automata $B_{\emptyset}^{\{c\}}$ and $B\text{-}R_{\emptyset}^{\{c\}}$.

nization sequence $[b](ab)^*$, whereas $(r0, (B2, c, -), r2)$ is enabled by $[a](ba)^*$. The prefixes $[b](ab)^*$ and $[a](ba)^*$ capture the possible corresponding reified synchronization paths within automaton $B_{\emptyset}^{\{c\}}$ induced by synchronization clique $\{q0q0, q1q1\}$, as illustrated in Figure 6.



(a) Prefix $[b](ab)^*$.



(b) Prefix $[a](ba)^*$.

Fig. 6. The prefix-giving interaction sequences in $B_{\emptyset}^{\{c\}}$.

Definition 4.2 Let $\mathcal{C} = (Q, Act, \delta, I, H)$ be a component interaction automaton, Σ be the induced alphabet of $\mathcal{C}$, and (V, E) be a synchronization clique in $\mathcal{C}$. Then a finite action prefix generator is a quadruple $\mathcal{C}_P = (V, \overline{Act}, \overline{E}, q_P)$, where

- V is the set of clique states,
- $\overline{Act} = \{a | (q, (n, a, n'), p) \in E\}$ is the prefix alphabet,
- $\overline{E} = \{(q, a, p) | (q, (n, a, n')p) \in E\}$ is the prefix transition function,
- $q_P \in V$ is a prefix state, if there is $l \in \Sigma$ such that $q_P \xrightarrow{l} p' \in \delta$, $p' \notin V$, and

all states in V are start states. We write $A_P[q]$ to denote the action prefix generator for prefix state q. If W_P is the set of all prefix strings that $A_P[q]$ accepts in q, we say that W_P is the action prefix language of $A_P[q]$ and write $L(A_P[q]) = W_P$.

An action prefix generator simultaneously explores all possible paths in a synchronization clique in order to distill the required action prefixes for a given prefix state. From a technical point of view, a prefix generator iterates over all clique states in (V, E) and constructs for each a finite state machine whose language is the union of all accepted action prefixes for a given prefix state. For example, in automaton $B_{\emptyset}^{\{c\}}$, both states in the synchronization clique $\{q0q0, q1q1\}$ are prefix states and the generated languages are $L(A_P[q0q0]) = \{b^{0:1}(ab)^n | n \geq 0\}$ and $L(A_P[q1q1]) = \{a^{0:1}(ba)^m | m \geq 0\}$, which we denote by the action prefixes $[b](ab)^*$ and $[a](ba)^*$. On the other hand, state $q1q1$ in automaton $A^{\{c\}}_{\{d\}}$ (cf. Figure 4(a)) is not a prefix state and we, therefore, obtain only a prefix for state $q0q0$ (*i.e.*, $[b](ab)^*$).

Definition 4.3 Let $\mathcal{C} = (Q, Act, \delta, I, H)$ be component interaction automaton, $q \xrightarrow{l} q' \in \delta$, and $A_P[q] = \alpha$ be an action prefix, then $q \xrightarrow{/\alpha/l} q'$ is a α-prefixed transition, with

$$/\alpha/l = \begin{cases} (-, /\alpha/a, n) & \textit{if } l = (-, a, n), \\ (n, /\alpha/a, -) & \textit{if } l = (n, a, -), \textit{ and} \\ (n_1, /\alpha/a, n_2) & \textit{if } l = (n_1, a, n_2). \end{cases}$$

Returning to the reduced automaton $B\text{-}R^{\{c\}}_{\emptyset}$ (cf. Figure 5(b)), we can obtain a deterministic version $B\text{-}R'^{\{c\}}_{\emptyset}$ by applying the generated prefixes to the conflicting transitions:

$$\begin{aligned} B\text{-}R'^{\{c\}}_{\emptyset} = (&\{r_0, r_1, r2\}, \{a, b, c\}, \\ &\{(r_0, (B2, /[b](ab)^*/c, -), r_1), (r_0, (B2, /[a](ba)^*/c, -), r_2), (r_0, (B1, /[b](ab)^*/c, -), r_2), \\ &(r1, (B2, c-), r0), (r_1, (B1, c, -), r_0), (r2, (B2, c, -), r0)\}, \\ &\{r_0\}, ((B1), (B2)) \end{aligned}$$

$B\text{-}R'^{\{c\}}_{\emptyset}$ is, naturally, not weakly-bisimular to $B\text{-}R^{\{c\}}_{\emptyset}$, as prefixed transitions produce a different behavior. However, we can restore bisimularity by erasing the added prefixes. We can think of prefixes as *types* [4], or more precisely *sequence types* [23], that explicitly record interaction constraints of the reduced component interaction automaton.

The composition of action prefixes is defined in the usual way. The composition of a refined automaton, containing prefixed transitions, with another automaton and the successive refinement may yield new action prefixes that have to be incorporated into the final result. We use the regular *concatenation* operation to built composite action prefixes. For example, if an existing action prefix $[f](ef)^*$ needs to be prefixed by $[b](ab)^*$, then the newly composed action prefix becomes $[f](ef)^*[b](ab)^*$. That is, before an interaction prefixed with $[f](ef)^*[b](ab)^*$ can occur, the corresponding component interaction automata must have performed a, possibly empty, sequence of internal synchronizations over actions f and e, followed by a, possibly empty, sequence of internal synchronizations over actions b and a.

Finally, we need to incorporate the prefix mechanism into the composition rule (cf. Definition 2.3) for new synchronizations, $\delta_{NewSync}$. We use the tags '?' to indicate a prefix associated with an input action and '!' to denote a prefix originating from an output action.

Definition 4.4 Let $q_1 \xrightarrow{(n_1,/\alpha_1/a,-)} q_1'$ and $q_2 \xrightarrow{(-,/\alpha_2/a,n_2)} q_2'$ be two prefixed transitions that synchronize according to Definition 2.3. Then $q \xrightarrow{(n_1,/?\alpha_1!\alpha_2/a,n_2)} q'$ is the resulting prefixed synchronization transition, where $?\alpha_1!\alpha_2$ is an atomic directional prefix.

Two component interaction automata can synchronize through matching complementary structured labels. These labels may, in turn, occur prefixed as result of a previous refinement of the underlying automata. These prefixes, however, cannot simply be concatenated as regular prefixes. Each prefix encodes either an input constraint or an output constraint, which we both must retain. On the surface, this appears cumbersome, but we facilitate the use of directional prefixes by considering them *atomic*, as if they were plain actions. We only require, as for all prefixes, that they are well-formed, that is, they are regularly composed of elements from the set, Act, of actions.

5 Discussion

We cannot underestimate the computational needs for the construction of a composite component interaction automaton. For example, even for a relatively small system consisting of components with no more than 4 states, the resulting product automaton requires easily in excess of 16,000 states with approx. 880,000 transitions and can take more than 6 hours to compute on a PC equipped with a 2.2 GHz dual-core processor and 2GB of main memory [13].

To study different means for an effective specification and construction of component interaction automata, we have developed an experimental composition framework for *Component Interaction Automata* in PLT-Scheme [21] that provides modular support for the specification, composition, and refinement of component interaction automata [13]. All analysis and transformation functions in the system are timed and can be controlled by a variety of parameters to fine-tune the induced operational semantics of an operation. The system also generates information about frequencies and distributions of states and transitions within composite automata, data that allows for an independent statistical analysis of the effects of composition and refinement.

One of the rather unexpected findings, while conducting experiments in our composition framework, is the existence of synchronization cliques in component interaction automata. To explain their presence, we have adopted some of the terminology that has been developed in network theory in order to characterize properties of complex networks [17]. Of particular interest are small-world networks that have been discovered in an astonishing number of natural phenomena, but also in software systems. Potanin et al. [22] have studied Java programs and detected *power laws* in object graphs indicating that object-oriented systems form *scale-free networks* [17]. A consequence of the existence of power laws in object-oriented systems is that there is no *typical* size to objects [22]. We find a similar property, both, in component interaction automata and emerging synchronization cliques.

But there remains a curiosity as to why synchronization cliques exist. We do not find a similar phenomenon in process-based models [24,9,16,12,15]. There is, however, a difference in the way internal sysnchronizations are represented. Process-based formalisms use a special symbol, τ, to denote the handshake between two matching, interacting processes. The synchronization of processes takes place internally. From an external observer's point of view, we notice the occurrence of a process synchronization through a delay between adjacent interactions with the environment. Milner [16] calls τ a *perfect action*, which arises from a pair of complementary input- and output-actions. What makes τ special is the observable equivalence between a sequence $P_1 \xrightarrow{\tau} P_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_n$ of process synchronizations and a single synchronization $P_1 \xrightarrow{\tau} P_n$. A similar concept does not exist in *Component Interaction Automata* and its predecessors *I/O Automata* and *Interface Automata*. We cannot equate a sequence of internal synchronizations in a component interaction automaton with a single action. First, such an abstraction would ignore the inherent partial order defined by a synchronization sequence and second, there exists no designated action in the *Component Interaction Automata* formalism that can subsume interaction sequences under one umbrella. Moreover, the precise sequence of internal synchronizations conveys a valuable information. For example, the *Store* will only issue the action $(Store, ship, -)$ after a successful interaction with the *Bank* to redeem the payment voucher (cf. Figure 2). This knowledge is vital for the understanding of the behavior of the whole e-commerce system $\mathcal{S}_{\emptyset}^{\{ship\}}$.

We have chosen regular expressions like $[b](ab)^*$ rather than fresh action labels to denote action prefixes in order to make pre-conditions to interactions as explicit as possible. This works well for simple prefixes. Experiments have shown, however, that the prefixes can grow in complexity rather quickly, rendering this structural technique unwieldy. We can envision a *nominal* approach to the specification of action prefixes in which we assign each action prefix a unique identifier and add a corresponding lookup table to the specification of the component interaction automata in question.

Finally, the outcome of partition refinement can be improved even further, if we erase the information about the underlying composition hierarchy by making the analyzed component *primitive* before refinement [13]. The composition of multiple instances of the same component can produce identical sub-structures in the resulting composite automaton. However, the unique component identifiers used to disambiguate shared actions prevent partition refinement from simplifying common sub-structures into a single, unifying one. We can overcome this difficulty by creating a fresh image of a given component interaction automaton in which all component names are the same. We will lose, though, the information, which particular sub-component participates in an actual occurring interaction with the environment.

6 Conclusion and Future Work

In this paper we have discussed some of the effects that partition refinement can produce when we apply this state space reduction technique to *Component Interaction Automata* specifications. We use weak bisimulation as underlying equivalence relation to drive the refinement process. From an external observer's point of view, weak bisimulation yields a means to hide internal intra-component synchronizations.

While a corresponding implementation of partition refinement for *Component Interaction Automata* specifications is feasible and effective, its application has revealed a specific property of component interaction automata that mandates an additional analysis to recover pre-conditions encoded in so-called *synchronization cliques*. A synchronization clique is a subgraph of internal intra-component synchronizations that define guards for component interactions with the environment. Partition refinement removes synchronization cliques from the specification of given component interaction automaton. But, in this paper, we have presented a workable solution to restore pre-conditions in reduced automata, when necessary.

We are only beginning to understand the emerging properties of software systems in general and component-based software systems in particular. There is sufficient evidence for the existence of small-world networks in software. To further our knowledge in this area, in future work we aim at studying network effects in component interaction automata specifications. In particular, we seek to explore possibilities to (i) predict the presence of synchronization cliques, (ii) estimate the reduction ratio, and (iii) use frequency distributions to monitor evolutionary changes in component interaction automata specifications.

References

[1] Beugnard, A., J.-M. Jézéquel, N. Plouzeau and D. Watkins, *Making Components Contract Aware*, IEEE Computer **32** (1999), pp. 38–45.

[2] Brim, L., I. Černá, P. Vařeková and B. Zimmerova, *Component-Interaction Automata as a Verification-Oriented Component-Based System Specification*, SIGSOFT Software Engineering Notes **31** (2006), pp. 1–8.

[3] Broy, M., *A Core Theory of Interfaces and Architecture and Its Impact on Object Orientation*, in: R. H. Reussner, J. A. Stafford and C. A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, LNCS 3938 (2004), pp. 26–47.

[4] Cardelli, L., *Type Systems*, in: *Handbook of Computer Science and Engineering*, CRC Press, 1997 pp. 2208–2236.

[5] Černá, I., P. Vařeková and B. Zimmerova, *Component Substitutability via Equivalencies of Component-Interaction Automata*, Electronic Notes in Theoretical Computer Science **182** (2007), pp. 39–55.

[6] de Alfaro, L. and T. A. Henzinger, *Interface Automata*, in: V. Gruhn and A. M. Tjoa, editors, *Proceedings ESEC/FSE 2001* (2001), pp. 109–120.

[7] de Alfaro, L., T. A. Henzinger and M. Stoelinga, *Timed Interfaces*, in: S.-V. A. L. and J. Sifakis, editors, *Proceedings of 2nd International Conference on Embedded Softare (EMSOFT 2002)*, LNCS 2491 (2002), pp. 108–122.

[8] Habib, M., C. Paul and L. Viennot, *Partition Refinement Techniques: An Interesting Algorithmic Tool Kit*, International Journal of Foundations of Computer Science **10** (1999), pp. 147–170.

[9] Hermanns, H., "Interactive Markov Chains: The Quest for Quantified Quality," LNCS 2428, Springer, Heidelberg, Germany, 2002.

[10] Hopcroft, J. E., R. Motwani and J. D. Ullman, "Automata Theory, Languages, and Computation," Pearson Education, 2007, 3rd edition.

[11] Lee, E. A. and Y. Xiong, *System-Level Types for Component-Based Design*, in: T. A. Henzinger and C. M. Kirsch, editors, *Proceedings of 1st International Workshop on Embedded Software (EMSOFT 2001)*, LNCS 2211 (2001), pp. 237–253.

[12] Lumpe, M., "A π-Calculus Based Approach to Software Composition," Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics (1999).

[13] Lumpe, M., L. Grunske and J.-G. Schneider, *State Space Reduction Techniques for Component Interfaces*, in: M. R. V. Chaudron and C. Szyperski, editors, *CBSE 2008*, LNCS 5282 (2008), pp. 130–145.

[14] Lunch, N. A. and M. R. T. Tuttle, *Hierarchical Correctness Proofs for Distributed Algorithms*, in: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, 1987, pp. 137–151.

[15] Mateescu, R., P. Poizat and G. Salaün, *Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques*, in: *Proceedings of ICSOC 2008*, LNCS 5364 (2008), pp. 84–99.

[16] Milner, R., "Communication and Concurrency," Prentice Hall, 1989.

[17] Newman, M. E. J., *The Structure and Function of Complex Networks*, SIAM Review **45** (2003), pp. 167–256.

[18] Paige, R. and R. E. Tarjan, *Three Partition Refinement Algorithms*, SIAM Journal on Computing **16** (1987), pp. 973–989.

[19] Park, D., *Concurrency and Automata on Infinite Sequences*, in: P. Deussen, editor, *5th GI Conference on Theoretical Computer Science*, LNCS 104 (1981), pp. 167–183.

[20] Pistore, M. and D. Sangiorgi, *A Partition Refinement Algorithm for the π-Calculus*, Information and Computation **164** (2001), pp. 264–321.

[21] PLT Scheme, "v372," http://www.plt-scheme.org (2008).

[22] Potanin, A., J. Noble, M. R. Frean and R. Biddle, *Scale-Free Geometry in OO Programs*, Commun. ACM **48** (2005), pp. 99–103.

[23] Puntigam, F., *Coordination Requirements Expressed in Types for Active Objects*, in: M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, LNCS 1241 (1997), pp. 367–388.

[24] Schneider, J.-G. and O. Nierstrasz, *Components, Scripts and Glue*, in: L. Barroca, J. Hall and P. Hall, editors, *Software Architectures – Advances and Applications*, Springer, 1999 pp. 13–25.

[25] Seco, J. C. and L. Caires, *A Basic Model of Typed Components*, in: E. Bertino, editor, *Proceedings of ECOOP 2000*, LNCS 1850 (2000), pp. 108–128.

[26] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," Addison-Wesley / ACM Press, 2002, Second edition.

[27] ter Beek, M. H., C. A. Ellis, J. Kleijn and G. Rozenberg, *Synchronizations in Team Automata for Groupware Systems*, Computer Supported Cooperative Work **12** (2003), pp. 21–69.

Behaviour Protocols for Interacting Stateful Components with Ports [1]

Sebastian S. Bauer, Rolf Hennicker, Stephan Janisch

Institut für Informatik
Ludwig-Maximilians-Universität München
{bauerse, hennicker, janisch}@pst.ifi.lmu.de

Abstract

We propose a formal foundation for behaviour protocols of interacting components with (encapsulated) data states. Formally, behaviour protocols are given by labelled transition systems which specify the order of operation invocations as well as the allowed changes of data states of components in terms of pre- and postconditions. We study the compatibility of protocols and we consider their composition which yields a behaviour protocol for a component assembly. Behaviour protocols are equipped with a model-theoretic semantics which describes the class of all correct component or assembly implementations. Implementation models are again formalised in terms of labelled transition systems and the correctness notion is based on an alternating simulation relation between protocol and implementation which takes into account control and data states. As a major result we show that our approach is compositional, i.e. that correct implementation models of compatible protocols compose to a correct implementation of the resulting assembly protocol.

Keywords: Behaviour protocol, pre- and postcondition, stateful component, compositionality

1 Introduction

Component-based software development has received much attention not only in practice but also in theory during the last decade. Thereby an important role is played by formal specifications of component behaviours which are usually based on a control-flow oriented perspective describing the sequences of actions a component can perform when interacting with its environment; cf., e.g., [12]. Some approaches also consider data that can be transmitted by value passing messages but less attention has been directed towards the integration of *data states* that a component can possess and which are typically specified by invariants and pre- and postconditions in sequential systems. Frameworks like CSP-OZ [6] or rCOS [8] support this facility but they do not distinguish (at least semantically) between input and output actions and their expressive power is limited to cases where the effect of an operation is specified independently of the control-flow behaviour of

[1] This research has been partially supported by the GLOWA-Danube project 01LW0602A2 sponsored by the German Federal Ministry of Education and Research.

a component. Other approaches are based on symbolic transition systems with a strong focus on the generation of abstractions for model checking [1,5] but they do not study formal correctness notions to relate specifications and implementations which is needed in a top-down development process. The goal of this work is to provide a formal foundation for interacting *stateful* components with a clear separation between specification and implementation such that the latter can still choose "among the possibilities left by the specification"; see [1]. Of crucial importance is, of course, the study of component composition and preservation of local correctness in global contexts. In this paper we propose an integrated framework which takes into account control flow and the evolution of component data states as well as the discrimination of input and output for distinguishing operation reception and operation calls. In particular, we are interested in a contract-based specification style where the caller of an operation can formulate its assumptions and guarantees with respect to the data state of the called component. On this basis we study compatibility of specifications, which are called behaviour protocols hereafter.

As a roadmap for this paper we follow the ideas of de Alfaro and Henzinger [3] who have studied interface automata, their refinement, their compatibility and their composition and we aim at an extension of these ideas by taking into account stateful components, where components are equipped with local data states to allow for an evaluation of pre- and postconditions attached to the labels of a behaviour protocol. In the context of data states we cannot use the same formalism for specification and refinement (implementation) of component behaviour which both have been formalised in terms of interface automata in [3,4]. Though labelled transition systems still provide an appropriate basis, the crucial difference here is that for the purpose of behaviour specification labels must include logical predicates describing pre- and postconditions while for modelling component implementations labels must represent concrete actions like operation invocations which can occur in concrete states of an implementation. As a consequence, we must adjust the alternating simulation relation used by de Alfaro and Henzinger in an appropriate way. Thereby an important role is played by so-called observers which determine the visible states of components.

After providing the necessary technical preliminaries we introduce the main structural elements of our component model in Sect. 2. We study compatibility of behaviour protocols of different components (and assemblies) which are supposed to be composed via connectors between open ports of components (and assemblies) in Sect. 3. Our notion of compatibility extends the corresponding notion in [4] by requiring that if an operation can be called by one component such that the required precondition is satisfied then the connected component must be able to treat the operation invocation in the specified way. We define the (synchronous) composition of behaviour protocols which turns out to have the expected properties (see below) if the protocols are compatible.

In the next step, in Sect. 4, we focus on component (and assembly) implementations which are formalised as labelled transition systems whose states are divided into a control part and a data part and whose transitions carry labels which represent sending or receiving of concrete operation invocations for particular actual parameters. We define a correctness notion for implementations in the spirit of the alternating simulation relation used for refinements in [4] which requires that any specified input must be accepted by an implementation and that, conversely, any possible output of an implementation must be admitted by the behaviour protocol. In the context of data states the alternating simulation relation between a behaviour protocol and an implementation model is, however, more involved since it has to take into account pre- and postconditions in protocols which must be related to the (visible) data states of component implementations when transitions are performed. Our main theorem shows that the proposed concepts for protocol compatibility and implementation correctness work smoothly together. This means, two implementation models which are locally correct w.r.t. compatible behaviour protocols compose to a correct implementation model of the composed behaviour protocol.

Preliminaries. Our approach is based on labelled transition systems to formalise the control-flow aspects of component behaviours. A *labelled transition system* (LTS) $M = (Q, q_0, L, \Delta)$ consists of a set Q of states, an initial state $q_0 \in Q$, a set L of labels, and a transition relation $\Delta \subseteq Q \times L \times Q$. If $(q, l, q') \in \Delta$, l is *enabled* at q. M is called L'*-enabled*, for a set $L' \subseteq L$ of labels, if for any reachable state $q \in Q$ and label $l \in L'$ there is at least one transition in Δ such that l is enabled at q.

To deal with the specification and implementation of the externally visible data states of components we use observer signatures. An *observer signature* Σ_{Obs} consists of a set of (visible) state variables. A Σ_{Obs}*-data state* $\sigma : \Sigma_{\mathrm{Obs}} \to \mathcal{V}$ assigns to each state variable in Σ_{Obs} a value in some predefined data universe $\mathcal{V}$. The class of all Σ_{Obs}-data states is denoted by $\mathcal{D}(\Sigma_{\mathrm{Obs}})$. For any observer signature Σ_{Obs}, we assume given a set $\mathcal{S}(\Sigma_{\mathrm{Obs}})$ of *state predicates* φ and a set $\mathcal{T}(\Sigma_{\mathrm{Obs}})$ of *transition predicates* π with associated sets $\mathrm{var}(\varphi)$ and $\mathrm{var}(\pi)$ of variables. State predicates refer to single states and transition predicates refer to pre- and poststates. We assume that state predicates $\varphi \in \mathcal{S}(\Sigma_{\mathrm{Obs}})$ are equipped with a *satisfaction relation* $\sigma; \rho \vDash \varphi$ for states $\sigma \in \mathcal{D}(\Sigma_{\mathrm{Obs}})$ and valuations $\rho : \mathrm{var}(\varphi) \to \mathcal{V}$. Similarly, for transition predicates $\pi \in \mathcal{T}(\Sigma_{\mathrm{Obs}})$ we assume a satisfaction relation $\sigma, \sigma'; \rho \vDash \pi$, for two states $\sigma, \sigma' \in \mathcal{D}(\Sigma_{\mathrm{Obs}})$ and valuation $\rho : \mathrm{var}(\pi) \to \mathcal{V}$. For $\varphi \in \mathcal{S}(\Sigma_{\mathrm{Obs}})$, we write $\vDash \varphi$ to express that φ is universally valid, i.e. $\sigma; \rho \vDash \varphi$ for all states $\sigma \in \mathcal{D}(\Sigma_{\mathrm{Obs}})$ and for all valuations $\rho : \mathrm{var}(\varphi) \to \mathcal{V}$. Universal validity of transition predicates is defined in the analogous way. We assume that state predicates (transition predicates resp.) are closed under the usual logical connectives (like $\wedge$, $\Rightarrow$, etc.) with the usual interpretation. In some occasions we will use state predicates in combination with transition predicates. Then a state predicate $\varphi \in \mathcal{S}(\Sigma_{\mathrm{Obs}})$ is considered as a special transition predicate where $\sigma, \sigma'; \rho \vDash \varphi$ is defined by $\sigma; \rho \vDash \varphi$. We do not fix a particular syntax for observer signatures, observers, and predicates here. In the examples unprimed symbols refer to the prestate and primed symbols to the poststate of a transition.

2 Component Model

In this section we summarize the structural concepts of our component model which extends the one in [7] by introducing observer signatures for ports and components. We do, however, not consider hierarchical components here and we make the simplifying assumption that names of ports and components are globally unique.

Components interact with each other by using operations which belong to the provided and required interfaces of their ports. An *operation* op is of the form $opname(X_{\text{in}})$ where X_{in} is a (possibly empty) sequence of input variables.[2] We write $\text{var}_{\text{in}}(op)$ to refer to the input variables of an operation op. An *interface* is a pair $(\Sigma_{\text{Obs}}, Op)$ consisting of an observer signature Σ_{Obs} and a set Op of operations. A *port signature* $(I_{\text{prov}}, I_{\text{req}})$ consists of a provided interface I_{prov} and a required interface I_{req}. Throughout this paper when we talk about a port P, we always assume given a *port declaration* $P : \Sigma$ where Σ is a port signature and P is a globally unique port name. We write $prv(P)$ for the provided interface of Σ, $obs_{prv}(P)$ for the observer signature and $opns_{prv}(P)$ for the operations of $prv(P)$. The operations in $opns_{prv}(P)$ are offered at port P and the observer signature $obs_{prv}(P)$ defines the possible obervations that can be made at this port (about the data state of its owning component). Symmetrically, we write $req(P)$ for the required interface of Σ, $obs_{req}(P)$ refers to the observer signature and $opns_{req}(P)$ to the operations of the required interface of $req(P)$. In this case, the operations in $opns_{req}(P)$ are required from components which are connected to P and the observer signature $obs_{prv}(P)$ defines which observations are required about the data states of connected components.

Components encapsulate data states and interact with their environment via ports. The data states of a component can only be observed via observers which are determined by the component's observer signature. The access points of a component are given by ports. Formally, a *component signature* $(\Sigma_{\text{Obs}}, (P : \Sigma_P)_{P \in I})$ consists of an observer signature Σ_{Obs} and a finite family of port declarations $P : \Sigma_P$. In the following when we talk about a component C, we always assume given a *component declaration* $C : \Sigma$ where Σ is a component signature and C is a globally unique component name. We write $obs(C)$ to refer to the observer signature and $ports(C)$ to refer to the ports declared in Σ. We require that for all ports $P \in ports(C)$, $obs_{prv}(P) = obs(C)$, i.e. on each port all observers of its owning component are visible.[3] For a port P, $cmp(P)$ denotes its owning component.

For building assemblies we connect ports of components. If P_1 and P_2 are ports such that $req(P_1) = prv(P_2)$ and $req(P_2) = prv(P_1)$, then $K : \{P_1, P_2\}$ is a (binary) *connector declaration* where K is a globally unique connector name. In the following when we talk about a connector K, we always assume given a connector declaration of the form $K : \{P_1, P_2\}$ and we write $ports(K)$ to refer to its set of ports $\{P_1, P_2\}$.

An assembly $A = \langle (C : \Sigma_C)_{C \in I}; (K : \{P_1^K, P_2^K\})_{K \in I'} \rangle$ consists of a finite family of component declarations $C : \Sigma_C$ and a finite family of connector declarations $K : \{P_1^K, P_2^K\}$. We write $cmps(A)$ to refer to the components of the assembly, $conns(A)$ to refer to its connectors and we define $ports(A) = \bigcup\{ports(C) \mid C \in cmps(A)\}$. The open ports of A, i.e., the ports which are not connected, are given by $open(A) = ports(A) \setminus \bigcup\{ports(K) \mid K \in conns(A)\}$. For the above definitions to make sense, we require that (i) only ports of components inside A are connected,

[2] For the sake of simplicity we do not consider output variables here which, however, could be easily integrated in our framework.

[3] In general, the observers of a port could be a subset of the component's observers or should be related by an abstraction function. This could be methodologically desirable to emphasize the difference between port and component protocols which, however, goes beyond the scope of this paper.

i.e., for all $K \in \mathit{conns}(A)$ we have that $\mathit{ports}(K) \subseteq \mathit{ports}(A)$; and (ii) there is at most one connector for each port, i.e., if $P \in \mathit{ports}(A)$ and $K, K' \in \mathit{conns}(A)$ with $P \in \mathit{ports}(K) \cap \mathit{ports}(K')$, then $K = K'$. Finally, composition of two assemblies A_1 and A_2 via a connector $K : \{P_1, P_2\}$ with $P_i \in \mathit{open}(A_i)$ for $i = 1, 2$ is denoted by $A = A_1 +_K A_2$ and defined by $\mathit{cmps}(A) = \mathit{cmps}(A_1) \cup \mathit{cmps}(A_2)$ and $\mathit{conns}(A) = \mathit{conns}(A_1) \cup \mathit{conns}(A_2) \cup \{K : \{P_1, P_2\}\}$.

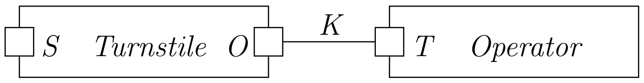


Figure 1. Components of the turnstile system.

Example 2.1 Our running example models a turnstile located at the entrance of a subway. The static structure of the system is given by the assembly depicted in Fig. 1. It consists of two components, *Turnstile* and *Operator*, which are connected via their ports O and T. The port S of the turnstile is left open. The observer signatures and the provided and required operations on each port will be explained later when the behaviour protocols of the two components are discussed.

3 Behaviour Protocols and their Compatibility

Behaviour protocols are a means to specify the observable behaviour of components and assemblies. For components they specify the legal sequences of operation invocations on the ports of a component, their invocation conditions and their effect with respect to the (visible) data state of a component. For assemblies behaviour protocols specify on the one hand, the legal interactions between connected components (taking into account the components' states) and, on the other hand, the legal sequences of invocations on those ports which are left open in the assembly. We start by introducing the syntax of behaviour protocols which are given by appropriate labelled transition systems.

Protocol labels are divided into labels $\mathcal{L}^{\mathcal{P}}(P)$ for ports P and labels $\mathcal{L}^{\mathcal{P}}(K)$ for connectors K; see Fig. 2. Labels for ports model either receiving (?) or sending (!) of a message. Messages which are supposed to be received must correspond to operations of the provided interface of a port while messages which can be sent must correspond to operations of the required interface of a port. Protocol labels can be equipped with pre- and postconditions represented by state and transition predicates of the respective observer signatures. A label $[\varphi]?P.m[\pi]$ for a port P expresses that port P is ready to receive an operation invocation of m under the assumption that the precondition φ holds, and after the execution of m the postcondition π is ensured. In this case φ must be a state predicate and π a transition predicate over the observer signature of the provided interface of port P. A label $[\varphi]!P.m[\pi]$ describes the sending of an invocation of m on port P with the guarantee of the precondition φ upon operation call and with the expectation that π holds when the operation is finished. Here φ must be a state predicate and π a transition predicate over the observer signature of the required interface of port P. For a connector K which connects two ports P_i and P_j, a label $[\varphi]P_i \triangleright_K P_j.m[\pi]$ stands for the synchronised sending, reception and execution of an operation m via

1. Labels $\mathcal{L}^{\mathcal{P}}(P)$ for a port P, $m \in opns_{prv}(P)$, $n \in opns_{req}(P)$:
   - $[\varphi]?P.m[\pi]$ where $\varphi \in \mathcal{S}(obs_{prv}(P))$, $\mathrm{var}(\varphi) \subseteq \mathrm{var}_{\mathrm{in}}(m)$, $\pi \in \mathcal{T}(obs_{prv}(P))$, $\mathrm{var}(\pi) \subseteq \mathrm{var}_{\mathrm{in}}(m)$
   - $[\varphi]!P.n[\pi]$ where $\varphi \in \mathcal{S}(obs_{req}(P))$, $\mathrm{var}(\varphi) \subseteq \mathrm{var}_{\mathrm{in}}(n)$, $\pi \in \mathcal{T}(obs_{req}(P))$, $\mathrm{var}(\pi) \subseteq \mathrm{var}_{\mathrm{in}}(n)$
2. Labels $\mathcal{L}^{\mathcal{P}}(K)$ for a connector $K : \{P_1, P_2\}$, $m \in opns_{prv}(P_j)$, $i, j \in \{1, 2\}$, $i \neq j$:
   - $[\varphi]P_i \triangleright_K P_j.m[\pi]$ where $\varphi \in \mathcal{S}(obs_{prv}(P_j))$, $\mathrm{var}(\varphi) \subseteq \mathrm{var}_{\mathrm{in}}(m)$, $\pi \in \mathcal{T}(obs_{prv}(P_j))$, and $\mathrm{var}(\pi) \subseteq \mathrm{var}_{\mathrm{in}}(m)$
3. Labels $\mathcal{L}^{\mathcal{P}}(A)$ for an assembly A: $\mathcal{L}^{\mathcal{P}}(A) = \bigcup_{P \in open(A)} \mathcal{L}^{\mathcal{P}}(P) \cup \bigcup_{K \in conns(A)} \mathcal{L}^{\mathcal{P}}(K)$

Figure 2. Labels for protocols.

the connected ports P_i and P_j. In this case the pre- and postconditions must be predicates over the observer signature $obs_{req}(P_i)$ (which is the same as $obs_{prv}(P_j)$ since required and provided interfaces of connected ports coincide). For an assembly A the protocol labels in $\mathcal{L}^{\mathcal{P}}(A)$ are those labels which correspond to connectors or to open ports of A.

Definition 3.1 [Behaviour Protocol]
Let A be an assembly. A *behaviour protocol* for A, also called *A-protocol*, is an LTS $F = (S, s_0, \mathcal{L}^{\mathcal{P}}(A), \Delta)$ where S is a finite set of *protocol states*, $s_0 \in S$ is the *initial protocol state*, $\mathcal{L}^{\mathcal{P}}(A)$ is the set of *protocol labels*, and Δ is a finite *protocol transition relation*. The class of all A-protocols is denoted by $Prot(A)$.

Here and in the following all definitions and results are provided for assemblies but they carry over to components since a component C can be identified with a degenerated assembly $\langle\{C\}; \emptyset\rangle$ which contains only the component C and no connectors. From the methodological point of view behaviour protocols for a proper assembly A correspond to architecture protocols in [11] while behaviour protocols for components C correspond to frame protocols in [11] and to interface automata in [3].

Example 3.2 Let us now come back to the turnstile example with the assembly shown in Fig. 1. For the observer signature of the *Turnstile* component we use two visible state variables: *fare* for the actual costs of a trip, and *passed* for the number of persons that have already passed the turnstile. The turnstile has two ports S and O. At port S no operation is required and two operations are provided: $coin(x : int)$ for dropping a coin with amount x into the turnstile's slot, and $pass()$ for passing through the turnstile. At port O the turnstile requires an operation $alarm()$ to inform the operator that a client has tried to pass the turnstile without paying and an operation $ready()$ is provided to switch off the alarm mode.

Fig. 3 presents the frame protocol F_T of component *Turnstile*. If in the initial state *LOCKED* a coin is deposited whose value is at least the required fare the turnstile becomes unlocked. In the state *UNLOCKED* a person can pass through the turnstile with the effect that the number of passed persons is increased by one and the state *LOCKED* is reached again. If a person tries to pass the turnstile without dropping a coin into its slot this causes the turnstile to send out an alarm on its port O. On the same port the alarm can be shut off by invoking ready. The frame protocol F_O of component *Operator* is shown in Fig. 4. It says that an operator can invoke ready whenever an alarm has been received. If a protocol label shows no explicit pre- or postcondition we implicitly assume the trivial predicate *true*. For instance, in the turnstile protocol of Fig. 3 the transition with label $?S.pass()$ between the states *LOCKED* and *ON_ALERT* has the implicit pre- and postcondition *true* (and hence is underspecified) while for the same operation called in state *UNLOCKED* the postcondition is $passed' = passed + 1$. This shows that the effect of an invocation of a particular operation may indeed depend on the source state where the operation is called which can be conveniently specified by the behaviour protocols as introduced here.

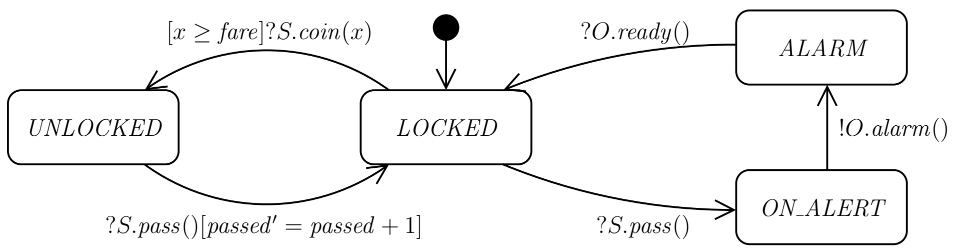


Figure 3. Protocol F_T of component *Turnstile*.

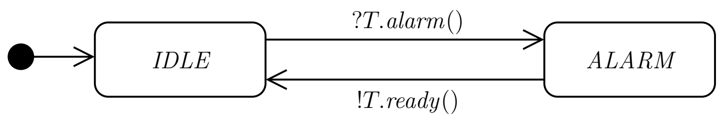


Figure 4. Protocol F_O of component *Operator*.

Two behaviour protocols can be combined to an assembly protocol that describes the behaviour of a system with interacting components. For this purpose we introduce a composition operator $\boxtimes_K$ which composes two protocols in accordance with a connector K between ports P_1 and P_2. The composition synchronises transitions whose labels match on corresponding inputs and outputs on P_1 and P_2 and vice versa. For instance, a transition with label $[\varphi_1]!P_1.op[\pi_1]$ of the first protocol is synchronised with a transition with label $[\varphi_2]?P_2.op[\pi_2]$ of the second protocol which yields a transition with label $[\varphi_1 \wedge \varphi_2]P_1 \triangleright_K P_2.op[\pi_1 \wedge \pi_2]$. The resulting transition expresses a correct communication which can only occur if both preconditions and both postconditions are satisfied. Thus by protocol composition via a connector K with $ports(K) = \{P_1, P_2\}$, two transitions with matching labels in $\mathcal{L}^{\mathcal{P}}(P_1)$ and $\mathcal{L}^{\mathcal{P}}(P_2)$ are synchronised to a single transition with label in $\mathcal{L}^{\mathcal{P}}(K)$. Transitions with labels in $\mathcal{L}^{\mathcal{P}}(P_1) \cup \mathcal{L}^{\mathcal{P}}(P_2)$ which can not be matched are deleted and all other transitions are interleaved.

Definition 3.3 [Protocol Composition]
For $i \in \{1, 2\}$, let A_i be assemblies, $P_i \in open(A_i)$, $F_i = (S_i, s_{0,i}, \mathcal{L}^{\mathcal{P}}(A_i), \Delta_i) \in Prot(A_i)$, and let $K : \{P_1, P_2\}$ be a connector. The *protocol composition of* F_1 *and* F_2 *via* K is defined by

$$F_1 \boxtimes_K F_2 = (S_1 \times S_2, (s_{0,1}, s_{0,2}), \mathcal{L}^{\mathcal{P}}(A_1 +_K A_2), \Delta)$$

where Δ is the least relation satisfying: if $(s_1, s_2) \in S_1 \times S_2$ then

(1) for $i, j \in \{1, 2\}$, $i \neq j$, for all $op \in opns_{req}(P_i)$, $l_i = [\varphi_i]!P_i.op[\pi_i] \in \mathcal{L}^{\mathcal{P}}(P_i)$, and $l_j = [\varphi_j]?P_j.op[\pi_j] \in \mathcal{L}^{\mathcal{P}}(P_j)$, if $(s_i, l_i, s'_i) \in \Delta_i$ and $(s_j, l_j, s'_j) \in \Delta_j$ then $((s_1, s_2), [\varphi_i \wedge \varphi_j]P_i \triangleright_K P_j.op[\pi_i \wedge \pi_j], (s'_1, s'_2)) \in \Delta$;

(2) for all $l_1 \in \mathcal{L}^{\mathcal{P}}(A_1) \setminus \mathcal{L}^{\mathcal{P}}(P_1)$ and $s_2 \in S_2$, if $(s_1, l_1, s'_1) \in \Delta_1$ then $((s_1, s_2), l_1, (s'_1, s_2)) \in \Delta$;

(3) for all $l_2 \in \mathcal{L}^{\mathcal{P}}(A_2) \setminus \mathcal{L}^{\mathcal{P}}(P_2)$ and $s_1 \in S_1$, if $(s_2, l_2, s'_2) \in \Delta_2$ then $((s_1, s_2), l_2, (s_1, s'_2)) \in \Delta$.

The composition operator for protocols is associative (up to equivalence of preconditions and of postconditions). It is related to the synchronous product of symbolic transition systems defined in [5] where synchronisation vectors are used instead of corresponding input/output labels. While in [5] predicates occurring in labels always refer to the data states of the owning component, in our approach we distinguish between send labels, whose predicates refer to the data states of the connected component, and receive labels whose predicates refer to the data states of the owning component.

Apparently, the conjunctions of the preconditions and of the postconditions used in the protocol composition do, in general, not allow for independent implementability which can only be ensured if the protocols to be composed are compatible. Two protocols are compatible w.r.t. a connector K if their transitions with send labels on a port of K can be mutually simulated by corresponding transitions with a matching input label on the opposite port. In particular, the pre- and postconditions of matching labels must subsume each other in the usual covariant manner where it is allowed to weaken preconditions and to strengthen postconditions; cf., e.g. [9,13]. Formally, we define protocol compatibility by using the following compatibility relation (which is a particular kind of a mutual simulation relation).

Definition 3.4 [Compatibility Relation]
For $i \in \{1, 2\}$, let A_i be assemblies, $P_i \in open(A_i)$, $F_i = (S_i, s_{0,i}, \mathcal{L}^{\mathcal{P}}(A_i), \Delta_i) \in Prot(A_i)$, and $K : \{P_1, P_2\}$ be a connector. A *K-compatibility relation for F_1 and F_2* is a relation $R \subseteq S_1 \times S_2$ such that for all $(s_1, s_2) \in R$ the following holds:

(1) for $i, j \in \{1, 2\}$, $i \neq j$, for all $l_i = [\varphi_i]!P_i.op[\pi_i] \in \mathcal{L}^{\mathcal{P}}(P_i)$, if $(s_i, l_i, s'_i) \in \Delta_i$, then there exists a transition $(s_j, l_j, s'_j) \in \Delta_j$ with $l_j = [\varphi_j]?P_j.op[\pi_j] \in \mathcal{L}^{\mathcal{P}}(P_j)$ such that $(s'_i, s'_j) \in R$, $\models (\varphi_i \Rightarrow \varphi_j)$, and $\models (\varphi_i \wedge \pi_j \Rightarrow \pi_i)$;

(2) for all $l_1 \in \mathcal{L}^{\mathcal{P}}(A_1) \setminus \mathcal{L}^{\mathcal{P}}(P_1)$, if $(s_1, l_1, s'_1) \in \Delta_1$ then $(s'_1, s_2) \in R$;

(3) for all $l_2 \in \mathcal{L}^{\mathcal{P}}(A_2) \setminus \mathcal{L}^{\mathcal{P}}(P_2)$, if $(s_2, l_2, s'_2) \in \Delta_2$ then $(s_1, s'_2) \in R$.

Condition (1) formalises the requirements that for every operation call specified by a protocol there must exists a corresponding reception specified by the other protocol, with the conditions on pre- and postconditions as explained above. By conditions (2) and (3) compatibility must be independent of other transitions not involved in communications on K.

Definition 3.5 [Protocol Compatibility]
Let F_1, F_2, K be as in Def. 3.4. F_1 and F_2 are *K-compatible* if there exists a K-compatibility relation R for F_1 and F_2 such that $(s_{0,1}, s_{0,2}) \in R$.

1. Labels $\mathcal{L}(P)$ for a port P, $m \in opns_{prv}(P)$, $n \in opns_{req}(P)$:
   - $?P.(m,\rho)$ where $\rho : \mathrm{var}_{\mathrm{in}}(m) \to \mathcal{V}$
   - $(T, !P.(n,\rho))$ where $T \subseteq \mathcal{D}(obs_{req}(P))$, $\rho : \mathrm{var}_{\mathrm{in}}(n) \to \mathcal{V}$
2. Labels $\mathcal{L}(K)$ for a connector $K : \{P_1, P_2\}$, $m \in opns_{prv}(P_j)$, $i, j \in \{1,2\}$, $i \neq j$:
   - $P_i \triangleright_K P_j.(m,\rho)$ where $\rho : \mathrm{var}_{\mathrm{in}}(m) \to \mathcal{V}$
3. Labels $\mathcal{L}(A)$ for assembly A: $\mathcal{L}(A) = \bigcup_{P \in open(A)} \mathcal{L}(P) \cup \bigcup_{K \in conns(A)} \mathcal{L}(K)$

Figure 5. Labels for implementation models.

4 Protocol Implementation and Compositionality

In this section we define a formal semantics for behaviour protocols in terms of their correct implementations and we study compositionality of implementations and protocols. For the formalisation of implementations we follow a model-theoretic approach where an implementation is represented by an input-enabled labelled transition system, also called implementation model.

The states of an implementation model must carry information for both, the control flow and the evolution of data states. To discriminate the two aspects we distinguish between control states and data states. As already explained in the preliminaries a (visible) data state is determined by the values of the observers of a given observer signature. Hence, for a component C, a (visible) data state is an element $\sigma \in \mathcal{D}(obs(C))$. The underlying state space of C in an implementation model is then formed by the cartesian product $Ctrl_C \times Q_C$ of a set of control states $Ctrl_C$ and a set of data states $Q_C \subseteq \mathcal{D}(obs(C))$. The state space of an assembly is formed by the cartesian product of the state spaces of all contained components.

We will now define the different labels occurring in implementation models. Implementation labels are divided, like protocol labels, into labels $\mathcal{L}(P)$ for ports P and labels $\mathcal{L}(K)$ for connectors K; see Fig. 5. The labels in $\mathcal{L}(P)$ describe the actions of receiving or sending an operation invocation on port P. A label of the form $?P.(m,\rho)$ expresses the reception of an invocation of a provided operation m on port P with actual parameters determined by a valuation $\rho : \mathrm{var}_{\mathrm{in}}(m) \to \mathcal{V}$. A transition labelled with $?P.(m,\rho)$ connects the state where the operation is called with the state after execution of the operation. Hence the implementation models considered here assume atomic operation executions. We do not model non terminating programs here which could, however, be easily integrated by using partial transitions which lead to an undefined state $\perp$. A label of the form $(T, !P.(m,\rho))$ expresses that the implementation sends out an operation call of m with actual parameters ρ provided that the receiver is in some state determined by T. More precisely, T is a set of data states over the observer signature of the connected component which models the fact that the implementation only invokes m if the visible data state of the receiver component belongs to T. In a concrete implementation this would mean that the sender component performs in an atomic action a test on the data state of the receiver component (by means of the observers) and, depending on the result, invokes the required operation. Implementation models for assemblies must also include labels for communication via connectors. For a connector K between ports P_1 and P_2, the set $\mathcal{L}(K)$ consists of labels of the form $P_1 \triangleright_K P_2.(m, \rho)$ which express the synchronised operation invocation (m, ρ) sent on P_1 and received on P_2. The target state of a transition labelled by $P_1 \triangleright_K P_2.(m, \rho)$ is reached when the operation has finished its execution. For an assembly A the implementation labels in $\mathcal{L}(A)$ are those labels which correspond to connectors or to open ports of A. Moreover, the set of input labels $\mathcal{L}_?(A)$ of A is given by all labels in $\mathcal{L}(A)$ of the form $?P.(m, \rho)$ where $P \in \mathit{open}(A)$. Implementation models are required to be $\mathcal{L}_?(A)$-enabled, i.e. all provided operations on the open ports of an assembly can be called in each reachable state.

Definition 4.1 [Implementation Model]
For an assembly A, an *A-implementation model* (A-implementation for short) is an $\mathcal{L}_?(A)$-enabled LTS $M = (Q, \boldsymbol{q}_0, \mathcal{L}(A), \Delta)$ with state space

$$Q = \prod_{C \in \mathit{cmps}(A)} \mathit{Ctrl}_C \times Q_C$$

where for each component $C \in \mathit{cmps}(A)$, Ctrl_C is a set of control states and $Q_C \subseteq \mathcal{D}(\mathit{obs}(C))$ is a set of (visible) data states of C. The class of all A-implementations is denoted by $\mathit{Impl}(A)$. For a state $\boldsymbol{q} \in Q$ and a component $C \in \mathit{cmps}(A)$ we write q_C for the projection of $\boldsymbol{q}$ to the state of the component C which is a pair $q_C = (c, \sigma) \in \mathit{Ctrl}_C \times Q_C$. We write $\delta(q_C)$ to refer to the data state part σ of q_C.

We require that implementation models are *well-formed*: A component's state may only be changed by a transition whose label involves a port of the component. More specifically, the data state of a component C can only be changed by a transition whose label has the form $?P.(m, \rho)$ or $P_1 \triangleright_K P.(m, \rho)$ where P is a port of C. The formalisation of well-formedness of implementation models is straightforward and omitted here.

Let us now discuss implementation correctness for an implementation model M w.r.t. a given protocol F. The behaviour protocol F can be considered as a contract between the implementor and the users of provided and required operations. From the implementor's point of view this means that it can be assumed that, first, a provided operation is only called in a state where the call is admissible according to the protocol and, secondly, that the given precondition for the invoked operation is satisfied. Under these assumptions the implementation must ensure that after the execution of the operation the given postcondition is satisfied and that one can proceed as specified by the protocol. If the user does not meet the implementor's assumptions then the implementation can have an arbitrary behaviour. From the user's point of view the contract principle imposes the obligation that a required operation is only invoked in accordance with the protocol; in particular the given precondition must be satisfied. Then the user can assume that the given postcondition holds after execution of the operation and that she/he can proceed as specified by the protocol. It may still be useful to remark that an implementation model plays, in general, the role of an implementor *and* the role of a user. The implementor's role is shown by transitions with labels of the form $?P.(m, \rho)$ while the user's role is shown by transitions with labels of the form $(T, !P.(m, \rho))$. The above considerations can be formalised by adapting the concept of an alternating simulation relation for interface automata in [3] to our needs where we have to deal with predicates on the specification level and with data states on the implementation level. In this context the alternating simulation relation contains pairs $(s, \boldsymbol{q})$, where s is a protocol state and $\boldsymbol{q}$ denotes an assembly state; i.e. $\boldsymbol{q}$ determines, for each component C in the assembly, the component's control and data state $q_C = (c, \sigma)$. The simulation relation is alternating, because reception of messages as specified in the protocol must be simulated in the implementation model, and conversely, every sending of a message in the model must be simulated by the protocol.

Definition 4.2 [Alternating Simulation Relation]
Let A be an assembly, $F = (S, s_0, \mathcal{L}^{\mathcal{P}}(A), \Delta_F, I) \in \mathit{Prot}(A)$ be an A-protocol, and $M = (Q, \boldsymbol{q}_0, \mathcal{L}(A), \Delta_M) \in \mathit{Impl}(A)$. An *alternating simulation relation between F and M* is a relation $R \subseteq S \times Q$ such that for all $(s, \boldsymbol{q}) \in R$,

(1) for all $P \in \mathit{open}(A)$, $C = \mathit{cmp}(P)$,
  (a) for all labels $l = [\varphi]?P.op[\pi] \in \mathcal{L}^{\mathcal{P}}(P)$ and for all $\rho : \mathrm{var}_{\mathrm{in}}(op) \to \mathcal{V}$, if $(s, l, s') \in \Delta_F$ and $\delta(q_C); \rho \models \varphi$ then for all transitions $(\boldsymbol{q}, ?P.(op, \rho), \boldsymbol{q}') \in \Delta_M$ it holds $(s', \boldsymbol{q}') \in R$ and $\delta(q_C), \delta(q'_C); \rho \models \pi$;
  (b) for all labels $l = (T, !P.(op, \rho)) \in \mathcal{L}(P)$, if $(\boldsymbol{q}, l, \boldsymbol{q}') \in \Delta_M$ then there exists a transition $(s, [\varphi]!P.op[\pi], s') \in \Delta_F$ such that for all $\sigma \in T$ it holds $\sigma; \rho \models \varphi$ and $(s', \boldsymbol{q}') \in R$;

(2) for all $K : \{P_1, P_2\} \in \mathit{conns}(A)$, $C_1 = \mathit{cmp}(P_1)$, $C_2 = \mathit{cmp}(P_2)$ it holds:
for $i, j \in \{1, 2\}$, $i \neq j$, for all labels $l = P_i \triangleright_K P_j.(op, \rho) \in \mathcal{L}(K)$, if $(\boldsymbol{q}, l, \boldsymbol{q}') \in \Delta_M$ then there exists a transition $(s, [\varphi]P_i \triangleright_K P_j.op[\pi], s') \in \Delta_F$ such that $(s', \boldsymbol{q}') \in R$, $\delta(q_{C_j}); \rho \models \varphi$ and $\delta(q_{C_j}), \delta(q'_{C_j}); \rho \models \pi$.

Conditions (1)(a) and (1)(b) formalise the contract principle behind protocols as described above. Condition (2) expresses that communications between components on the implementation level must be simulated by corresponding protocol transitions. Thus we do not treat communications as invisible actions which are abstracted in a refinement relation because for assembly implementations it is still important that communications conform to the protocol. At a later stage one can still abstract from communications and build a composite component arround an assembly which, however, goes beyond the scope of this paper.

We can now define correctness of assembly models w.r.t. assembly protocols in terms of alternating simulation relations.

Definition 4.3 [Implementation Correctness]
Let A be an assembly, $F = (S, s_0, \mathcal{L}^{\mathcal{P}}(A), \Delta_F) \in \mathit{Prot}(A)$ be an A-protocol, and $M = (Q, \boldsymbol{q}_0, \mathcal{L}(A), \Delta_M) \in \mathit{Impl}(A)$. M is a *correct A-implementation* of F, if there exists an alternating simulation relation R between F and M such that $(s_0, \boldsymbol{q}_0) \in R$. The class of all correct A-implementations of F is denoted by $[\![F]\!]$.

Implementation models can be composed to build larger systems from smaller ones. For this purpose we introduce the operator $\otimes_K$ which composes two implementation models in accordance with a connector K between ports P_1 and P_2. The composition of two models M_1, M_2 synchronises transitions concerning the connected ports if their labels, e.g. $(T, !P_1.(op, \rho))$ and $?P_2.(op, \rho)$, express the same operation invocation (op, ρ) and if M_1 meets M_2 in a state whose visible data part (of the connected component) lies in the set T which has guarded the send message. The remaining messages on the connected ports P_1 and P_2 are removed. Transitions with labels not in $\mathcal{L}(P_1) \cup \mathcal{L}(P_2)$ are interleaved in the model composition.

Definition 4.4 [Model Composition]
For $i \in \{1,2\}$, let A_i be assemblies, $M_i = (Q_i, \boldsymbol{q}_{0,i}, \mathcal{L}(A_i), \Delta_i) \in \mathit{Impl}(A_i)$, $P_i \in \mathit{open}(A_i)$, and let $K : \{P_1, P_2\}$ be a connector. The *model composition of* M_1 *and* M_2 *via* K is defined by $M_1 \otimes_K M_2 = (Q, \boldsymbol{q}_0, \mathcal{L}(A), \Delta)$ where $A = A_1 +_K A_2$, $Q = Q_1 \times Q_2$, $\boldsymbol{q}_0 = (\boldsymbol{q}_{0,1}, \boldsymbol{q}_{0,2})$, and the transition relation Δ is the least relation satisfying

(1) for $i, j \in \{1,2\}$, $i \neq j$, for all $op \in \mathit{opns}_{\mathit{req}}(P_i)$, $\rho : \mathrm{var}_{\mathrm{in}}(op) \to \mathcal{V}$,
if $(\boldsymbol{q}_i, (T, !P_i.(op, \rho)), \boldsymbol{q}'_i) \in \Delta_i$ and $(\boldsymbol{q}_j, ?P_j.(op, \rho), \boldsymbol{q}'_j) \in \Delta_j$
and $\delta(q_{j,\mathit{cmp}(P_j)}) \in T$, then $((\boldsymbol{q}_1, \boldsymbol{q}_2), P_i \triangleright_K P_j.(op, \rho), (\boldsymbol{q}'_1, \boldsymbol{q}'_2)) \in \Delta$;

(2) for all $l_1 \in \mathcal{L}(A_1) \setminus \mathcal{L}(P_1)$ and $\boldsymbol{q}_2 \in Q_2$,
if $(\boldsymbol{q}_1, l_1, \boldsymbol{q}'_1) \in \Delta_1$ then $((\boldsymbol{q}_1, \boldsymbol{q}_2), l_1, (\boldsymbol{q}'_1, \boldsymbol{q}_2)) \in \Delta$;

(3) for all $l_2 \in \mathcal{L}(A_2) \setminus \mathcal{L}(P_2)$ and $\boldsymbol{q}_1 \in Q_1$,
if $(\boldsymbol{q}_2, l_2, \boldsymbol{q}'_2) \in \Delta_2$ then $((\boldsymbol{q}_1, \boldsymbol{q}_2), l_2, (\boldsymbol{q}_1, \boldsymbol{q}'_2)) \in \Delta$.

The composition operator $\otimes_K$ is associative and is straightforward to show that it preserves well-formedness of implementation models.

We are now able to present our final compositionality result which says that two correct implementation models with compatible protocols can be composed to a correct implementation model of the composed protocol. Often this is also called *independent implementability* [3]: Under the assumption of compatible protocols, *any* correct implementation of the first protocol composed with *any* correct implementation of the second protocol yields a correct implementation of the composed specification, i.e., protocols can be independently implemented.

Theorem 4.5 *For $i \in \{1,2\}$, let F_i be A_i-protocols, $P_i \in \mathit{ports}(A_i)$, $M_i \in \mathit{Impl}(A_i)$, and let $K : \{P_1, P_2\}$ be a connector. If $M_1 \in [\![F_1]\!]$, $M_2 \in [\![F_2]\!]$, and F_1, F_2 are K-compatible, then $M_1 \otimes_K M_2 \in [\![F_1 \boxtimes_K F_2]\!]$.*

Proof For $i \in \{1,2\}$, let S_i be the state space of F_i and Q_i be the state space of M_i. By assumption there exist alternating simulation relations $R_1 \subseteq S_1 \times Q_1$ and $R_2 \subseteq S_2 \times Q_2$, and a K-compatibility relation $R^{\mathrm{compat}} \subseteq S_1 \times S_2$ for F_1 and F_2. For proving the correctness of $M_1 \otimes_K M_2$ we must find an alternating simulation relation R between $F_1 \boxtimes_K F_2$ and $M_1 \otimes_K M_2$ such that $((s_{0,1}, s_{0,2}), (q_{0,1}, q_{0,2})) \in R$. We define R by

$$((s_1, s_2), (q_1, q_2)) \in R :\Longleftrightarrow (s_1, q_1) \in R_1 \text{ and } (s_2, q_2) \in R_2 \text{ and } (s_1, s_2) \in R^{\mathrm{compat}}.$$

It is straightforward to show that R is indeed an alternating simulation relation as required. □

The following example illustrates that protocol compatibility is indeed mandatory for the validity of Thm. 4.5.

Example 4.6 Fig. 6 shows two connected components C and D which are specified by the protocols F_C and F_D resp. We assume that the state predicate φ is not equivalent to *true*. Then F_C and F_D are obviously *not* compatible because the label $!P.a$ in F_C has the (implicit) precondition *true*. Moreover, extracts of two correct implementations $M_C \in [\![F_C]\!]$ and $M_D \in [\![F_D]\!]$ are given which show the relevant transitions for this example. In the graphical presentation of M_D we use the following “relaxed” notations: The indicated initial state denotes a set of reachable states which contains the concrete initial state which we assume does not satisfy φ; the transition labelled with $\{\varphi\}?Q.a$ stands for a set of transitions $\{(q, ?Q.(a, \rho), q') \mid \delta(q); \rho \models \varphi\}$ and, similarly, the transition labelled with $\{\neg\varphi\}?Q.a$ stands for a set of transitions. The implementation model M_D is correct since for any call of a on port Q which meets M_D in a data state satisfying φ it shows the specified behaviour, otherwise it may show arbitrary behaviour; in our case, M_D sends out b on the open port R. In the model composition $M_C \otimes_K M_D$, the message b will indeed be sent out but this is not an admissible action according to the protocol composition $F_C \boxtimes_K F_D$. Thus the theorem does not hold without the assumption of compatible protocols.

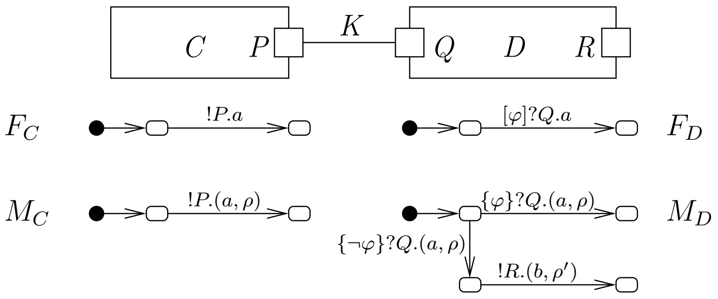


Figure 6. Incompatible protocols.

Theorem 4.5 is strongly related to the compositionality result of de Alfaro and Henzinger which was formulated for refinements of interface automata; cf. [4]. As already mentioned before a crucial difference in our approach is that, caused by the treatment of data states, we use two different formalisms for specification and for implementation: finite behaviour protocols and implementation models (with possibly infinitely many states). On the specification level we have required compatibility of protocols while on the implementation level this requirement is trivially satisfied since implementation models are input-enabled. This is also the reason why it was not necessary to restrict the parallel composition operator to compatible states as done in [4].

5 Concluding Remarks

We have proposed a formalism for specification and implementation of component and assembly behaviour which integrates the aspects of control flow and evolving data states. A simpler form of the current approach has been presented in [2] where components and assemblies were not considered and where a program can only play one role, either being a user or being the provider of a set of operations. So two different correctness notions were considered in [2], user correctness and implementation correctness, and no alternating simulation was used. Moreover, we have focused on a sequential approach in [2] where the correctness notion for implementations allows to utilise postconditions to infer properties of the connected component's data states. If, for instance, in two subsequent operation calls the postcondition of the first operation implies the precondition of the second one, the invocation of the latter would be safe in a sequential system even without querying the data state again. Obviously, this would, in general, not work for systems of concurrently running components as considered here, because there might be some interfering operation executions (invoked on different ports) in between possibly changing the component's data state in an unexpected way. There are essentially three ways out of this problem:

- Blocking of operation execution until the precondition is satisfied (which is suggested in [10] for asynchronous communication but which requires possibly costly deadlock and liveness analysis).
- Querying the data state of the target component to check the precondition before an operation is called; in this case the query and the operation execution must be combined to an atomic action (which is the model supported here at the cost of flexibility and performance).
- Architectural constraints or protocol constraints on access to shared observers (which is still an issue for further investigation).

There are several directions in which the current approach can be further developed. First, it is straightforward to integrate state invariants for components. It should also not be difficult to take into account hierarchically structured components as considered in [7]. A real challenge will be to study to what extent the ideas presented here can be transferred to asynchronously communicating components which use FIFO buffers for transmitting messages.

Acknowledgment. We would like to thank Moritz Hammer for careful reading of a draft of this paper and the anonymous reviewers for many valuable hints and suggestions on the previous, submitted version of this paper. In particular, their comments have shown that we needed a simplification of the presentation. Therefore we have decided to simplify behaviour protocols and implementation models such that, differently to the pure form of behaviour protocols in [11], the events of emitting an operation invocation and getting the corresponding response are not distinguished anymore; symmetrically, also the events of receiving an operation invocation and sending the response are not distinguished here. As a consequence, the current formalisation assumes that operation executions are atomic. The more general case would allow for an arbitrary interleaving of operation executions; but then to get the compositionality theorem one needs to impose appropriate restrictions to ensure that the change of data states is consistent to the single views of each user of a component.

References

[1] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In David de Frutos-Escrig and Manuel Núñez, editors, *Formal Techniques for Networked and Distributed Systems (FORTE 2004), 24th IFIP WG 6.1 International Conference*, volume 3235 of *Lect. Notes Comp. Sci.*, pages 43–60. Springer, 2004.

[2] S. S. Bauer and R. Hennicker. Views on behaviour protocols and their semantic foundation. In A. Kurz and A. Tarlecki, editors, *Proc. 3rd Conference on Algebra and Coalgebra in Computer Science (CALCO'09)*, to appear 2009.

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

[4] L. de Alfaro and T. A. Henzinger. Interface-based Design. In Manfred Broy, Johannes Grünbauer, David Harel, and C. A. R. Hoare, editors, *Engineering Theories of Software-intensive Systems*, volume 195 of *NATO Science Series: Mathematics, Physics, and Chemistry*, pages 83–104. Springer, 2005.

[5] F. Fernandes and J.-C. Royer. The STSLib project: Towards a formal component model based on STS. *Electr. Notes Theor. Comput. Sci.*, 215:131–149, 2008.

[6] C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 423–438, Canterbury, UK, 1997. Chapman and Hall, London.

[7] R. Hennicker, S. Janisch, and A. Knapp. On the Observable Behaviour of Composite Components. In Carlos Canal and Corina Pasareanu, editors, *Proc. 5th Int. Wsh. Formal Aspects of Component Software (FACS'08)*, to appear 2009.

[8] Z. Liu, J. He, and X. Li. rCOS: Refinement of Component and Object Systems. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, Third International Symposium (FMCO 2004)*, volume 3657 of *Lect. Notes Comp. Sci.*, pages 183–221. Springer, 2004.

[9] B. Meyer. *Object-Oriented Software Construction.* Prentice–Hall, Upper Saddle River, New Jersey, 2nd edition, 1997.

[10] P. Nienaltowski, B. Meyer, and J. S. Ostroff. Contracts for concurrency. *Form. Asp. Comput.*, 21(4):305–318, 2009.

[11] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Software Eng.*, 28(11):1056–1076, 2002.

[12] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lect. Notes Comp. Sci.* Springer, 2008.

[13] H. Wehrheim. *Behavioural Subtyping in Object-Oriented Specification Formalisms.* Habilitation thesis, Universität Oldenburg, 2002.

A Boolean algebra of contracts for assume-guarantee reasoning

Yann Glouche[1] Paul Le Guernic Jean-Pierre Talpin Thierry Gautier

*INRIA, Unité de Recherche Rennes-Bretagne-Atlantique
Campus de Beaulieu, 35042 Rennes Cedex, France*

Abstract

Contract-based design is an expressive paradigm for a modular and compositional specification of programs. It is in turn becoming a fundamental concept in mainstream industrial computer-aided design tools for embedded system design. In this paper, we elaborate new foundations for contract-based embedded system design by proposing a general-purpose algebra of assume/guarantee contracts based on two simple concepts: first, the assumption or guarantee of a component is defined as a filter and, second, filters enjoy the structure of a Boolean algebra. This yields a structure of contracts that is a Heyting algebra.

1 Introduction

Common methodological guidelines for attacking the design of large embedded architectures advise the validation of specifications as early as possible and an iterative validation of each refinement or modification made to the initial specification, until the implementation of the system is finalized. Additionally, cooperative component-based development requires to use and to assemble components, which have been developed by different suppliers, in a safe and consistent way [10,15]. These components have to be provided with their conditions of use and guarantees that they have been validated when these conditions are satisfied. This represents a notion of *contract*. Contracts are now often required as a useful mechanism for validation in robust software design. Design by Contract, as advocated in [26], is being made available for usual languages like C++ or Java. Assertion-based contracts express program invariants, pre- and post-conditions, as Boolean type expressions that have to be true for the contract being validated. We adopt a different paradigm of contract to define a component-based validation technique in the context of a synchronous modeling framework. In our model, a component is represented by an abstract view of its behaviors. It has a finite set of input/output variables to

[1] Partially funded by the EADS Foundation.

cooperate with its environment. Behaviors are viewed as multi-set traces on the variables of the component. The abstract model of a component is thus a *process*, defined as a set of such behaviors.

A *contract* is a pair (*assumptions, guarantees*). *Assumptions* describe properties expected by a component to be satisfied by the context (the environment) in which this component is used; on the opposite *guarantees* describe properties that are satisfied by the component itself when the context satisfies the *assumptions*. Such a contract may be documentary; however, when a suitable formal model exists, contracts can be subject to some formal verification tool. We want to provide designers with such a formal model allowing "simple" but powerful and efficient computation on contracts. Thus, we define a novel algebraic framework to enable formal reasoning on contracts. It is based on two simple concepts.

First, the assumptions and guarantees of a component are defined as *process-filters*: assumptions filter the processes (sets of behaviors) a component may accept and guarantees filter the processes a component provides. A *process-filter* is the set of processes, whatever their input and output variables are, that are compatible with some property (or constraint), expressed on the variables of the component. Second and foremost, we define a Boolean algebra to manipulate process-filters. This yields an algebraically rich structure which allows us to reason on contracts (to abstract, refine, combine and normalize them). This algebraic model is based on a minimalist model of execution traces, allowing one to adapt it easily to a particular design framework.

A characteristic of this model is that it allows one to precisely handle the variables of components and their possible behaviors. This is a key point. Indeed, assumptions and guarantees are expressed, as usual, by properties constraining or relating the behaviors of some variables. What has to be considered very carefully is thus the "compatibility" of such constraints with the possible behaviors of other variables. This is the reason why we introduce partial order relations on processes and on process-filters. Moreover, having a Boolean algebra on process-filters allows one to formally, unambiguously and finitely express complementation within the algebra. This is, in turn, a real advantage compared to related formalisms and models.

Plan

The article is organized as follows. Section 2 introduces a suitably general algebra of processes which borrows its notations and concepts from domain theory [1]. A contract (**A**,**G**) is viewed as a pair of logical devices filtering processes: the assumption **A** filters processes to select (accept or conversely reject) those that are asserted (accepted or conversely rejected) by the guarantee **G**. Process-filters are defined in Section 3 and contracts in Section 4. Section 5 discusses application of our model to the synchronous Signal language. Related works are further discussed in Section 6. Section 7 concludes the presentation. Detailed proofs of all properties presented in this article are available in a technical report [12].

2 An algebra of processes

We start with the definition of a suitable algebra for behaviors and processes. Usually, a behavior describes the trace of a discrete process (a Mazurkiewicz trace [24] or a tuple of signals in Lee's tagged signal model [18]). We deliberately choose a more abstract definition in order to encompass not only sequences of Boolean, integer, real variables but also behaviors of more complex systems such as hybrid systems or, on the contrary, simpler "behaviors" associating scalar values with variables, to represent execution cost, memory size, etc. In this paper we focus the presentation on usual process behaviors.

Definition 2.1 [Behavior] Let $\mathcal{V}$ be an infinite, countable set of variables, and $\mathcal{D}$ a set of values; for $\mathbf{Y}$, a finite set of variables included in $\mathcal{V}$ (written $\mathbf{Y} \subset \mathcal{V}$), $\mathbf{Y}$ nonempty, a $\mathbf{Y}$-*behavior* is a function $b : \mathbf{Y} \rightarrow \mathcal{D}$.

The set of $\mathbf{Y}$-behaviors is denoted by $\mathbb{B}_{\mathbf{Y}} =_\Delta \mathbf{Y} \rightarrow \mathcal{D}$. Definition 2.1 is extended to the empty variable domain: $\mathbb{B}_\emptyset =_\Delta \emptyset$ (there is no behavior associated with the empty set of variables). For $\mathbf{Y}$, a finite set of variables included in $\mathcal{V}$, $\mathbf{Y}$ nonempty, c a $\mathbf{Y}$-behavior, $\mathbf{X}$ a (possibly empty) subset of $\mathbf{Y}$, $c_{|\mathbf{X}}$ is the restriction of c on $\mathbf{X}$, $c_{|\mathbf{X}} =_\Delta \{(x,c(x))|x \in \mathbf{X}\}$, $c_{|\emptyset} =_\Delta \emptyset$; then $c_{|\mathbf{Y}} = c$.

In Figure 1, the x, y-behaviors b_1 and b_2 are functions from the variables x, y to functions that denote signals. Behavior b_1 is a discrete sampling mapping a domain of time represented by natural numbers to values. Behavior b_2 associates x, y to continuous functions of time.

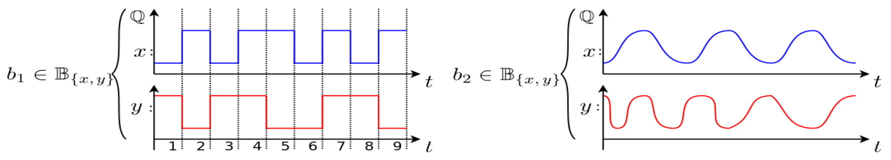


Figure 1. Examples of behaviors.

We define a *process* as a set of behaviors on a given set of variables.

Definition 2.2 [Process] For $\mathbf{X}$, a finite set of variables ($\mathbf{X} \subset \mathcal{V}$), an $\mathbf{X}$-*process* p is a nonempty set of $\mathbf{X}$-behaviors.

Thus, since $\mathbb{B}_\emptyset = \emptyset$, there is a unique $\emptyset$-process, designated by $\Omega =_\Delta \{\emptyset\}$; Ω has the empty behavior as unique behavior. The *empty process* is denoted by $\mho =_\Delta \emptyset$.

Since Ω does not have any variable, it has no effect when composed (intersected) with other processes. It can be seen as the universal process, for constraint conjunction, in contrast with $\mho$, the empty set of behaviors, the use of which in constraint conjunction always results in the empty set. $\mho$ can be seen as the null process.

For $\mathbf{X}$, a finite set of variables ($\mathbf{X} \subset \mathcal{V}$), we denote by $\mathbb{P}_{\mathbf{X}} =_\Delta \mathcal{P}(\mathbb{B}_{\mathbf{X}}) \setminus \{\mho\}$ the set of $\mathbf{X}$-processes ($\mathbb{P}_\emptyset = \{\Omega\}$). $\mathbb{P} =_\Delta \cup_{(\mathbf{X} \subset \mathcal{V})} \mathbb{P}_{\mathbf{X}}$ denotes the set of all processes. The domain of behaviors in an $\mathbf{X}$-process p is denoted by $var(p) =_\Delta \mathbf{X}$.

A process is a nonempty set of behaviors. Then we extend $\mathbb{P}$ to $\mathbb{P}^\star =_\Delta \mathbb{P} \cup \{\mho\}$ and $\forall \mathbf{X} \subset \mathcal{V}$, $\mathbb{P}_{\mathbf{X}}$ to $\mathbb{P}^\star{}_{\mathbf{X}} =_\Delta \mathbb{P}_{\mathbf{X}} \cup \{\mho\}$. Moreover, we extend the definition of $var(p)$ to $var(\mho) =_\Delta \mathcal{V}$.

The following operators will be used to define filters and contracts: the complementary of a process p in $\mathbb{P}_{\mathbf{X}}$ is a process in $\mathbb{P}^\star{}_{\mathbf{X}}$; the restriction of a process p in $\mathbb{P}_{\mathbf{X}}$ to $\mathbf{Y} \subseteq \mathbf{X} \subset \mathcal{V}$ is the abstraction (projection) of p to $\mathbf{Y}$; finally, the extension of p in $\mathbb{P}_{\mathbf{X}}$ to $\mathbf{Y} \subset \mathcal{V}$, $\mathbf{Y}$ finite, is the process on $\mathbf{Y}$ that has the same constraints as p.

Definition 2.3 [Complementary, restriction and extension] For $\mathbf{X}$, a finite set of variables ($\mathbf{X} \subset \mathcal{V}$), the complementary $\widetilde{p}$ of a process $p \in \mathbb{P}_\mathbf{X}$ is defined by $\widetilde{p} =_\Delta (\mathbb{B}_\mathbf{X} \setminus p)$. Also, $\widetilde{\mathbb{B}_\mathbf{X}} = \mho$. When $\mathbf{X}$, $\mathbf{Y}$ are finite sets of variables such that $\mathbf{X} \subseteq \mathbf{Y} \subset \mathcal{V}$, we define by $q_{|\mathbf{X}} =_\Delta \{c_{|\mathbf{X}} | c \in q\}$ the restriction $q_{|\mathbf{X}} \in \mathbb{P}_\mathbf{X}$ of $q \in \mathbb{P}_\mathbf{Y}$ and by $p^{|\mathbf{Y}} =_\Delta \{c \in \mathbb{B}_\mathbf{Y} | c_{|\mathbf{X}} \in p\}$ the extension $p^{|\mathbf{Y}} \in \mathbb{P}_\mathbf{Y}$ of $p \in \mathbb{P}_\mathbf{X}$. Hence, we have $q_{|\emptyset} = \Omega$, $q_{|var(q)} = q$, $\Omega^{|\mathbf{Y}} = \mathbb{B}_\mathbf{Y}$ and $p^{|var(p)} = p$.

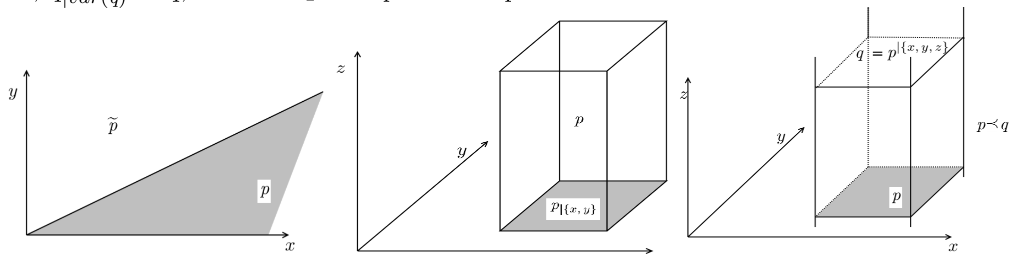


Figure 2. Complementary, restriction and extension of a process.

The complementary $\widetilde{p}$ of a process p defined on the variables x and y, Figure 2, consists of all behaviors defined on x, y not belonging to p. The restriction $p_{|\{x,y\}}$ of a process p defined on x, y, z, consists of its projection on the restricted domain; right, the extension $p^{|\{x,y,z\}}$ of a process p defined on x, y is the largest process defined on x, y, z whose restriction on x, y is equal to p.

The set $\mathbb{P}^\star{}_\mathbf{X}$, equipped with union, intersection and complementary, extended with $\widetilde{\mho} = \mathbb{B}_\mathbf{X}$, is a Boolean algebra with supremum $\mathbb{P}^\star{}_\mathbf{X}$ and infimum $\mho$. The definition of restriction is extended to $\mho$, the null process, with $\mho_{|\mathbf{X}} =_\Delta \{c_{|\mathbf{X}} | c \in \emptyset\} = \mho$. Since $\mathcal{V}$ is the set of all variables, the definition of extension is simply extended to $\mho$, with $\mho^{|\mathcal{V}} =_\Delta \mho$. The process extension operator induces a partial order $\preceq$, such that $p \preceq q$ if q is an extension of p to variables of q; the relation $\preceq$, used to define filters, is studied below.

Definition 2.4 [Process extension relation] The process extension relation $\preceq$ is defined by: $(\forall\ p \in \mathbb{P})\ (\forall\ q \in \mathbb{P})\ \ (p \preceq q) =_\Delta ((var(p) \subseteq var(q)) \wedge (p^{|var(q)} = q))$

Thus, if $(p \preceq q)$, q is defined on more variables than p; on the variables of p, q has the same constraints as p; its other variables are free. This relation extends to $\mathbb{P}^\star$ with $(\mho \preceq \mho)$.

Property 2.5 $(\mathbb{P}^\star, \preceq)$ is a poset.

Checking transitivity, antisymmetry and reflexivity is immediate. In this poset, the upper set of a process p, called *extension upper set*, is the set of all its extensions; it is denoted by $p{\uparrow}_\preceq =_\Delta \{q \in \mathbb{P} | p \preceq q\}$. The extension upper set is illustrated in Figure 3.

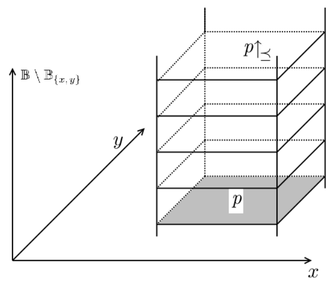


Figure 3. Extension upper set.

To study properties of *extension upper sets*, we characterize the set of variables that are constrained by a given process: we write that a process $q \in \mathbb{P}$ *controls* some variable y, if y belongs to $var(q)$ and q is not equal to the extension on $var(q)$ of its projection on $(var(q) \setminus \{y\})$.

This is illustrated in Figure 4, there is some behavior b in q that has the same restriction on $(var(q)\setminus\{y\})$ as some behavior c in $\mathbb{B}_{var(q)}$ such that c does not belong to q; thus q is strictly included in $(q_{|(var(q)\setminus\{y\})})^{|var(q)}$.

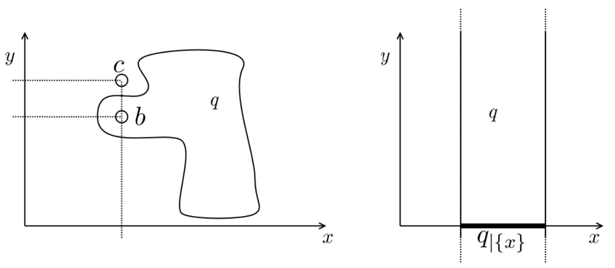


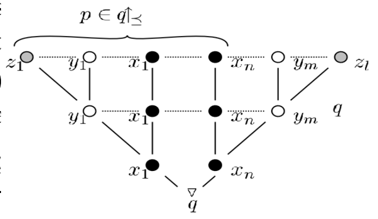
Figure 4. Controlled (left) and non-controlled (right) variable y in a process q.

Formally, a process $q \in \mathbb{P}$ controls a variable y, written $(q \rhd y)$, iff $(y \in var(q))$ and $q \neq ((q_{|(var(q)\setminus\{y\})})^{|var(q)})$. A process $q \in \mathbb{P}$ controls a variable set $\mathbf{X}$, written $(q \rhd \mathbf{X})$, iff $(\forall\ x \in \mathbf{X})\ (q \rhd x)$.
Moreover, $\rhd$ is extended to $\mathbb{P}^\star$ with $\mho \rhd \mathcal{V}$. Note that if a process p controls $\mathbf{X}$, this does not imply that, for all $x \in \mathbf{X}$, $y \in \mathbf{X}$, $x \neq y$, $(p_{|(\mathbf{X}\setminus\{x\})})$ controls y: it may be the case that x is constrained in p by y; then if x is "removed" (by the projection on other controlled variables), y may be free in this projection. We define a *reduced process* (the key concept to define filters) as being a process that controls all of its variables.

Definition 2.6 [Reduced process] A process $p \in \mathbb{P}^\star$ is *reduced* iff $p \rhd var(p)$.

For instance, Ω is reduced. On the contrary, $\mathbb{B}_\mathbf{X}$ is never reduced when $\mathbf{X}$ is not empty. *Reduced processes* are minimal in $(\mathbb{P},\preceq)$. We denote by $\overset{\triangledown}{q}$, called *reduction of* q, the (minimal) process such that $\overset{\triangledown}{q} \preceq q$ (p is reduced iff $\overset{\triangledown}{p} = p$). For all $\mathbf{X}$, we have $\overset{\triangledown}{\mathbb{B}_\mathbf{X}} = \Omega$.

Figure 5 illustrates the reduction $\overset{\triangledown}{q}$ of a process q and a process p, in the extension upper set $\overset{\triangledown}{q}\uparrow_\preceq$. Assuming that $var(q) = (\{x_{1\ldots n}\} \cup \{y_{1\ldots m}\})$ and that q controls the variables $\{x_{1\ldots n}\}$, we have $var(\overset{\triangledown}{q}) = \{x_{1\ldots n}\}$; the process p is such that $p \in \overset{\triangledown}{q}\uparrow_\preceq$ with $var(p) \subseteq (\{x_{1\ldots n}\} \cup \{y_{1\ldots m}\} \cup \{z_{1\ldots l}\})$; it controls the variables $\{x_{1\ldots n}\}$, and $\{y_{1\ldots m}\} \cup \{z_{1\ldots l}\}$ is a set of free variables, such that $\overset{\triangledown}{q} = \overset{\triangledown}{p}$.



$p \in \overset{\triangledown}{q}\uparrow_\preceq$
z_1
y_1
x_1
x_n
y_m
z_l
y_1
x_1
x_n
y_m
q
x_1
x_n
$\overset{\triangledown}{q}$

Figure 5. Reduction of a process.

Property 2.7 The complementary $\widetilde{p}$ of a nonempty process p strictly included in $\mathbb{B}_{var(p)}$ is reduced iff p is reduced; then $\widetilde{p}$ and p control the same set of variables $var(p)$.

From the above, the extension upper set $\overset{\triangledown}{p}\uparrow_\preceq$ of the reduction of p is a (principal) filtered set [1]: it is nonempty and each pair of elements has a lower bound. Then $\overset{\triangledown}{p}\uparrow_\preceq$ is composed of all the sets of behaviors, defined on variable sets that include the variables controlled by p, as maximal processes (for union of sets of behaviors) that have exactly the same constraints as p (variables that are not controlled by p are also not controlled in the processes of $\overset{\triangledown}{p}\uparrow_\preceq$). We also observe that $var(\overset{\triangledown}{q})$ is the greatest subset of variables such that $q \rhd var(\overset{\triangledown}{q})$. For a process $q \in \mathbb{P}^\star$, we extend the definition of $var()$ to the extension upper set of its reduction by $var(\overset{\triangledown}{q}\uparrow_\preceq) =_\Delta var(\overset{\triangledown}{q})$. Notice that $\mho\uparrow_\preceq = \{\mho\}$.

We define the *inclusion lower set* of a set of processes to capture all the subsets of behaviors of these processes. Let $\mathbf{R} \subseteq \mathbb{P}^\star$, $\mathbf{R}{\downarrow_\subseteq}$ is the inclusion lower set of $\mathbf{R}$ for $\subseteq$ defined by $\mathbf{R}{\downarrow_\subseteq} =_\Delta \{p \in \mathbb{P}^\star | (\exists\ q \in \mathbf{R})\ (p \subseteq q)\}$. Hence, none of the processes but $\mho$ belongs to the *inclusion lower set* of the *extension upper set* of $\mho$; on the contrary, all processes belong to the *inclusion lower set* of the *extension upper set* of Ω: $[\overset{\triangledown}{\mho}{\uparrow_\preceq}]{\downarrow_\subseteq} = \{\mho\}$ and $[\overset{\triangledown}{\Omega}{\uparrow_\preceq}]{\downarrow_\subseteq} = \mathbb{P}^\star$.

3 An algebra of filters

In this section, we define a *process-filter* by the set of processes that satisfy a given property. We propose an order relation ($\sqsubseteq$) on the set of process-filters Φ. We establish that (Φ,$\sqsubseteq$) is a Boolean algebra. A *process-filter* $\mathbf{R}$ is a subset of $\mathbb{P}^\star$ that filters processes. It contains all the processes that are "equivalent" with respect to some constraint or property, so that all processes in $\mathbf{R}$ are accepted or all of them but $\mho$ are rejected. A process-filter is built from a unique process *generator* by extending it to larger sets of variables and then by including subprocesses of these "maximal allowed behavior sets".

Definition 3.1 [Process-filter] A set of processes $\mathbf{R}$ is a *process-filter* iff $(\exists\ r \in \mathbb{P}^\star)$ $((r = \overset{\triangledown}{r}) \wedge (\mathbf{R} = [r{\uparrow_\preceq}]{\downarrow_\subseteq}))$. The process r is a *generator* of $\mathbf{R}$ ($\mathbf{R}$ is generated by r).

The process-filter generated by the reduction of a process p is denoted by $\widehat{p} =_\Delta [\overset{\triangledown}{p}{\uparrow_\preceq}]{\downarrow_\subseteq}$. The generator of a process-filter $\mathbf{R}$ is unique, we refer to it as $\overset{\triangledown}{\mathbf{R}}$. Ω generates the set of all processes (including $\mho$) and $\mho$ belongs to all filters. Formally, $(\forall\ p,r,s \in \mathbb{P}^\star)$, we have:

$$(p \in \widehat{r}) \Longrightarrow (var(\overset{\triangledown}{r}) \subseteq var(p)) \qquad \widehat{r} = \widehat{s} \Longleftrightarrow \overset{\triangledown}{r} = \overset{\triangledown}{s} \qquad \Omega \in \widehat{r} \Longleftrightarrow \widehat{r} = \mathbb{P}^\star$$

Figure 6 illustrates how a process-filter is generated from a process p (depicted by the bold line) in two successive operations. The first operation consists of building the extension upper set of the process: it takes all the processes that are compatible with p and that are defined on a larger set of variables. The second operation proceeds using the inclusion lower set of this set of processes: it takes all the processes that are defined by subsets of behaviors from processes in the extension upper set (in other words, those processes that remain compatible when adding constraints, since adding constraints removes behaviors).

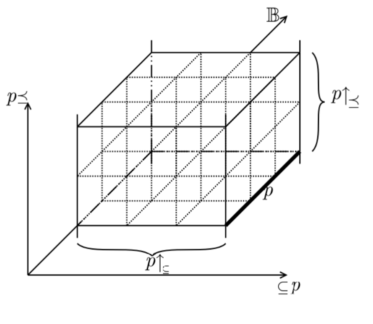


Figure 6. Example of process-filter.

We denote by Φ the set of process-filters. We call *strict process-filters* the process-filters that are neither $\mathbb{P}^\star$ nor $\{\mho\}$. The filtered variable set of a process-filter $\mathbf{R}$ is $var(\mathbf{R})$ defined by $var(\mathbf{R}) =_\Delta var(\overset{\triangledown}{\mathbf{R}})$.

We define an order relation on process-filters, which we call relaxation, and write $\mathbf{R} \sqsubseteq \mathbf{S}$ to mean that $\mathbf{R}$ is less wide than $\mathbf{S}$.

Definition 3.2 [Process-filter relaxation] For $\mathbf{R}$ and $\mathbf{S}$, two process-filters, let $\mathbf{Z}$ $= var(\mathbf{R}) \cup var(\mathbf{S})$. The relation $\mathbf{S}$ *relaxes* $\mathbf{R}$, written $\mathbf{R} \sqsubseteq \mathbf{S}$, is defined by: $\{\mho\} \sqsubseteq \mathbf{S} \qquad (\mathbf{R} \sqsubseteq \{\mho\}) \Longleftrightarrow \{\mho\} = \mathbf{R} \qquad (\mathbf{R} \sqsubseteq \mathbf{S} \Longleftrightarrow \overset{\triangledown}{\mathbf{R}}^{|\mathbf{Z}} \subseteq \overset{\triangledown}{\mathbf{S}}^{|\mathbf{Z}})$

The relaxation relation defines the structure of process-filters, which is shown to be a lattice.

Lemma 3.3 *(Φ,$\sqsubseteq$) is a lattice of supremum $\mathbb{P}^\star$ and infimum $\{\mho\}$. Let $\mathbf{R}$ and $\mathbf{S}$ be two process-filters, $\mathbf{V} = var(\mathbf{R}) \cup var(\mathbf{S})$, $\mathbf{R_V} = \overset{\triangledown}{\mathbf{R}}^{|\mathbf{V}}$ and $\mathbf{S_V} = \overset{\triangledown}{\mathbf{S}}^{|\mathbf{V}}$. Conjunction $\mathbf{R} \sqcap \mathbf{S}$, disjunction $\mathbf{R} \sqcup \mathbf{S}$ and complementary $\widetilde{\mathbf{R}}$ are defined by:*

$$\{\mho\} \sqcap \mathbf{R} =_\Delta \{\mho\} \qquad \mathbf{R} \sqcap \mathbf{S} =_\Delta [\overset{\triangledown}{(\mathbf{R_V} \cap \mathbf{S_V})}\Uparrow_{\preceq}]\downarrow_{\subseteq} \qquad \widetilde{\mathbf{R}} =_\Delta [\widetilde{\overset{\triangledown}{\mathbf{R}}}\Uparrow_{\preceq}]\downarrow_{\subseteq}$$

$$\{\mho\} \sqcup \mathbf{R} =_\Delta \mathbf{R} \qquad \mathbf{R} \sqcup \mathbf{S} =_\Delta [\overset{\triangledown}{(\mathbf{R_V} \cup \mathbf{S_V})}\Uparrow_{\preceq}]\downarrow_{\subseteq} \qquad \widetilde{\mathbb{P}^\star} =_\Delta \{\mho\}$$

$$\widetilde{\{\mho\}} =_\Delta \mathbb{P}^\star$$

If $\mathbf{R} \neq \{\mho\}$ then $\widetilde{\mathbf{R}} \neq \{\mho\}$ and $\widetilde{\overset{\triangledown}{\mathbf{R}}} = (\mathbb{B}_{var(\mathbf{R})} \setminus \overset{\triangledown}{\mathbf{R}})$ is reduced and $var(\mathbf{R}) = var(\widetilde{\mathbf{R}})$.

Let us comment the definitions of these operators. Conjunction of two strict process-filters $\mathbf{R}$ and $\mathbf{S}$, for instance, is obtained by first building the extension of the generators $\overset{\triangledown}{\mathbf{R}}$ and $\overset{\triangledown}{\mathbf{S}}$ on the union of the sets of their controlled variables; then the intersection of these processes, which is also a process (set of behaviors) is considered; since this operation may result in some variables becoming free (not controlled), the reduction of this process is taken; and finally, the result is the process-filter generated by this reduction. The same mechanism, with union, is used to define disjunction. And the complementary of a strict process-filter $\mathbf{R}$ is the process-filter generated by the complementary of its generator $\overset{\triangledown}{\mathbf{R}}$.

The process-filter conjunction $\mathbf{R}\sqcap\mathbf{S}$ of two strict process-filters $\mathbf{R}$ and $\mathbf{S}$ is the greatest process-filter $\mathbf{T} = \mathbf{R}\sqcap\mathbf{S}$ that accepts all processes that are accepted by $\mathbf{R}$ and by $\mathbf{S}$.

Example 3.4 Let x, a variable taking values in {0,1,2,3} and u, y, v three variables taking values in {0,1}; let $r \in \mathbb{P}_{\{u,\ x,\ y\}}$, $s \in \mathbb{P}_{\{x,\ y,\ v\}}$, two reduced processes defined by

$$r = \{b | b(u) \in \{0,1\} \wedge b(x) \in \{0,1\} \wedge b(y) \in \{0,1\}\} \cup \{\{(u,1),(x,2),(y,0)\}\}$$
$$s = \{b | b(x) \in \{0,1\} \wedge b(y) \in \{0,1\} \wedge b(v) \in \{0,1\}\} \cup \{\{(x,3),(y,1),(v,0)\}\}$$

We observe that $r \rhd \{u,x,y\}$; u and y are free in r when x is 0 or 1; v is free whatever the value of x is in r. We also have $s \rhd \{x,y,v\}$; y and v are free in s when x is 0 or 1; thus u is free whatever the value of x is in s. From the above definitions, we have that $p =_\Delta r \cap s = \{\mathrm{b} | b(u) \in \{0,1\} \wedge b(x) \in \{0,1\} \wedge b(y) \in \{0,1\} \wedge b(v) \in \{0,1\}\}$ and $\overset{\triangledown}{p} = \{\mathrm{b} | b(x) \in \{0,1\}\}$.

The process-filter disjunction $\mathbf{R}\sqcup\mathbf{S}$ of two strict process-filters $\mathbf{R}$ and $\mathbf{S}$ is the smallest process-filter $\mathbf{T} = \mathbf{R}\sqcup\mathbf{S}$ that accepts all processes that are accepted by $\mathbf{R}$ or by $\mathbf{S}$.

Example 3.5 Let x, a variable taking values in {0,1,2,3} and u, y, v three variables taking values in {0,1}; let $r \in \mathbb{P}_{\{u,\ x,\ y\}}$, $s \in \mathbb{P}_{\{x,\ y,\ v\}}$, two reduced processes such that

$$r = \{b | b(u) \in \{0,1\} \wedge b(x) \in \{0,1\} \wedge b(y) = 0\}$$
$$s = \{b | b(x) \in \{0,1\} \wedge b(y) = 1 \wedge b(v) \in \{0,1\}\}$$

Hence, $p =_\Delta r \cup s = \{\mathrm{b} | b(u) \in \{0,1\} \wedge b(x) \in \{0,1\} \wedge b(y) \in \{0,1\} \wedge b(v) \in \{0,1\}\}$ and $\overset{\triangledown}{p} = \{\mathrm{b} | b(x) \in \{0,1\}\}$.

Now we can state a first main result, which is that process-filters form a Boolean algebra.

Theorem 3.6 *$(\Phi,\sqsubseteq)$ is a Boolean algebra with $\mathbb{P}^\star$ as 1, $\{\mho\}$ as 0 and the complementary $\widetilde{\mathbf{R}}$.*

Variable elimination operators are defined on process-filters.

Definition 3.7 [Variable elimination in process-filter] Let x be a variable, $\mathbf{R}$ a process-filter, and $\mathbf{X} =_\Delta var(\mathbf{R})$. The E-elimination of x in $\mathbf{R}$, noted $\mathbf{R}_{|\exists x}$, is the projection of $\mathbf{R}$ on controlled variables other than x. The generator of the U-elimination of x in $\mathbf{R}$ (U-elimination of x in $\mathbf{R}$ is noted $\mathbf{R}_{|\forall x}$) contains the behaviors of $\overset{\triangledown}{\mathbf{R}}$ restricted on $\mathbf{X}\backslash\{x\}$ for which x is free in $\overset{\triangledown}{\mathbf{R}}$.

$$\mathbf{R}_{|\exists x} =_\Delta \begin{cases} \widehat{(\overset{\triangledown}{\mathbf{R}})_{|\mathbf{X}\backslash\{x\}}}, & x \in \mathbf{X} \\ \mathbf{R}, & \text{otherwise} \end{cases} \qquad \mathbf{R}_{|\forall x} =_\Delta \widetilde{\widetilde{\mathbf{R}}_{|\exists x}}$$

Notice that $\mathbf{R}_{|\forall x} \sqsubseteq \mathbf{R} \sqsubseteq \mathbf{R}_{|\exists x}$.

Example 3.8 Let $\mathbf{R}$, a process-filter generated by $((x > 0) \Rightarrow (y > 0)) \wedge ((y > 0) \Rightarrow (z > 0))$. Then $\mathbf{R}_{|\exists x}$ is generated by $((y > 0) \Rightarrow (z > 0))$ and $\mathbf{R}_{|\forall x}$ is generated by $((y > 0) \wedge (z > 0))$.

4 An algebra of contracts

We define the notion of assume/guarantee contract and propose a refinement relation on contracts.

Definition 4.1 [Contract] A *contract* $\mathbf{C} = (\mathbf{A},\mathbf{G})$ is a pair of process-filters. $var(\mathbf{C})$, the variable set of $\mathbf{C} = (\mathbf{A},\mathbf{G})$, is defined by $var(\mathbf{C}) =_\Delta var(\mathbf{A}) \cup var(\mathbf{G})$. $\mathbb{C} =_\Delta \Phi\times\Phi$ is the set of contracts.

Usually, an assumption $\mathbf{A}$ is an assertion on the behavior of the environment (it is typically expressed on the inputs of a process) and thus defines the set of behaviors that the process has to take into account. The guarantee $\mathbf{G}$ defines properties that should be guaranteed by a process running in an environment where behaviors satisfy $\mathbf{A}$.

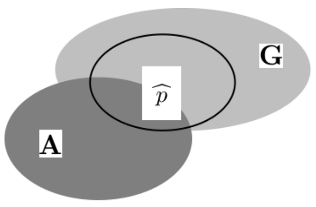


Figure 7. A process p satisfying a contract $(\mathbf{A},\mathbf{G})$.

Figure 7 depicts a process p *satisfying* the contract $(\mathbf{A},\mathbf{G})$ ($\widehat{p}$ is the process-filter generated by the reduction of p).

A process p *satisfies* a contract $\mathbf{C} = (\mathbf{A},\mathbf{G})$ if all its behaviors that are accepted by $\mathbf{A}$ (i.e., that are behaviors of some process in $\mathbf{A}$), are also accepted by $\mathbf{G}$; this is made more precise and formal by the following definition.

Definition 4.2 [Satisfaction] Let $\mathbf{C} = (\mathbf{A},\mathbf{G})$ a contract; a process p satisfies $\mathbf{C}$, written $p \vDash \mathbf{C}$, iff $(\widehat{p} \sqcap \mathbf{A}) \sqsubseteq \mathbf{G}$.

Property 4.3 $p \vDash \mathbf{C} \iff \widehat{p} \sqsubseteq (\widetilde{\mathbf{A}} \sqcup \mathbf{G})$

We define a preorder relation that allows to compare contracts. A contract $(\mathbf{A}_1,\mathbf{G}_1)$ is *finer* than a contract $(\mathbf{A}_2,\mathbf{G}_2)$, written $(\mathbf{A}_1,\mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2,\mathbf{G}_2)$, iff all processes that satisfy the contract $(\mathbf{A}_1,\mathbf{G}_1)$ also satisfy the contract $(\mathbf{A}_2,\mathbf{G}_2)$.

Definition 4.4 [Satisfaction preorder] $(\mathbf{A}_1,\mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2,\mathbf{G}_2)$ iff $(\forall\ p \in \mathbb{P})((p \models \mathbf{(A_1,G_1)}) \implies (p \models \mathbf{(A_2,G_2)}))$

The preorder on contracts satisfies the following property:

Property 4.5 $(\mathbf{A}_1,\mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2,\mathbf{G}_2)$ iff $(\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqsubseteq (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)$

Definition 4.6 [Refinement of contracts] A contract $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ *refines* a contract $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$, written $\mathbf{C}_1 \preccurlyeq \mathbf{C}_2$, iff $(\mathbf{A}_1,\mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2,\mathbf{G}_2)$, $(\mathbf{A}_2 \sqsubseteq \mathbf{A}_1)$ and $\mathbf{G}_1 \sqsubseteq (\mathbf{A}_1 \sqcup \mathbf{G}_2)$.

Refinement of contracts amounts to relaxing assumptions and reinforcing promises under the initial assumptions. The intuitive meaning is that for any p that satisfies a contract $\mathbf{C}$, if $\mathbf{C}$ refines $\mathbf{D}$ then p satisfies $\mathbf{D}$. Our relation of refinement formalizes substitutability.

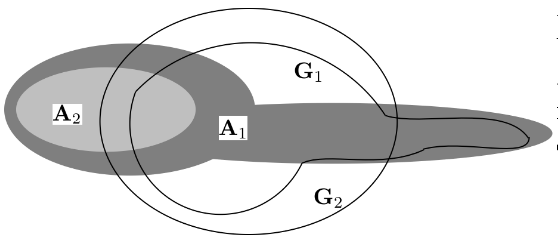


Figure 8. Refinement of contracts.

Figure 8 depicts a contract $(\mathbf{A}_1,\mathbf{G}_1)$ that refines a contract $(\mathbf{A}_2,\mathbf{G}_2)$. Among contracts that can be used to refine an existing contract $(\mathbf{A}_2,\mathbf{G}_2)$, we choose those contracts $(\mathbf{A}_1,\mathbf{G}_1)$ that "scan" more processes than $(\mathbf{A}_2,\mathbf{G}_2)$ $(\mathbf{A}_2 \sqsubseteq \mathbf{A}_1)$ and that guarantee less processes than those of $\mathbf{A}_1 \sqcup \mathbf{G}_2$. But other choices could have been made.

By definition of the satisfaction pre-order, we can express the refinement relation in the algebra of process-filters as follows:

Property 4.7 $(\mathbf{A}_1,\mathbf{G}_1) \preccurlyeq (\mathbf{A}_2,\mathbf{G}_2)$ iff $\mathbf{A}_2 \sqsubseteq \mathbf{A}_1$, $(\mathbf{A}_2 \sqcap \mathbf{G}_1) \sqsubseteq \mathbf{G}_2$ and $\mathbf{G}_1 \sqsubseteq (\mathbf{A}_1 \sqcup \mathbf{G}_2)$.

The refinement relation ($\preccurlyeq$) defines the poset of contracts, which is shown to be a lattice. In this lattice, the union or disjunction of contracts is defined by their least upper bound and the intersection or conjunction of contracts is defined by their greatest lower bound.

Lemma 4.8 (Composition of contracts) *Two contracts* $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ *and* $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ *have a greatest lower bound* $\mathbf{C} = (\mathbf{A},\mathbf{G})$ *, written* $(\mathbf{C}_1 \Downarrow \mathbf{C}_2)$*, defined by:*

$$\mathbf{A} = \mathbf{A}_1 \sqcup \mathbf{A}_2 \text{ and } \mathbf{G} = ((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2))$$

and a least upper bound $\mathbf{D} = (\mathbf{B},\mathbf{H})$*, written* $(\mathbf{C}_1 \Uparrow \mathbf{C}_2)$*, defined by:*

$$\mathbf{A} = \mathbf{A}_1 \sqcap \mathbf{A}_2 \text{ and } \mathbf{G} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_2 \sqcap \mathbf{G}_1)$$

A Heyting algebra H is a bounded lattice such that for all a and b in H there is a greatest element x of H such that the greatest lower bound of a and x refines b [6]. For all contracts $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$, $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$, there is a greatest element $\mathbf{X} = (\mathbf{I},\mathbf{J})$ of $\mathbb{C}$ such that the greatest lower bound of $\mathbf{C}_1$ and $\mathbf{X}$ refines $\mathbf{C}_2$, with:

$\mathbf{I} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_1 \sqcap \widetilde{\mathbf{G}_2})$ $\quad$ $\mathbf{J} = \mathbf{G}_2 \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{G}_1}) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \widetilde{\mathbf{G}_1})$

Then our contract algebra is a Heyting algebra. In particular, it is distributive.

Theorem 4.9 *$(\mathbb{C}, \preccurlyeq)$ is a Heyting algebra with supremum $(\{\mho\},\mathbb{P}^\star)$ and infimum $(\mathbb{P}^\star,\{\mho\})$.*

Let x a variable, $\mathbf{C} = (\mathbf{A},\mathbf{G})$ a contract, the elimination of x in $\mathbf{C}$ is the contract $\mathbf{C}_{\backslash x}$ defined by $\mathbf{C}_{\backslash x} =_\Delta (\mathbf{A}_{|\forall x},\mathbf{G}_{|\exists x})$.

Property 4.10 A contract $\mathbf{C}$ refines the elimination of a variable in $\mathbf{C}$: $\mathbf{C} \preccurlyeq \mathbf{C}_{\backslash x}$

5 Application to the Signal language

In the synchronous multiclocked model of Signal [19], a process (noted p), consists of the synchronous composition (noted $p \,|\, q$) of equations on signals (noted $x = y\ f\ z$). A signal x consists of an infinite flow of values that is discretely sampled according to the pace of its clock. A set of tags t is to denote symbolic periods in time during which transitions take place. It samples a signal over a countable series of causally related tags. Then the events, signals, behaviors and processes are defined as follows:
- an *event* e is a pair consisting of a tag t and a value v,
- a *signal* s is a function from a *chain* of tags to a set of values,
- a *behavior* b is a function from a set of names to signals,
- a *process* $p \in \mathbb{P}$ is a set of behaviors that have the same domain.

Synchronous composition $p \,|\, q$ consists of the simultaneous solution of the equations in p and q at all times.

Definition 5.1 [Synchronous composition of processes] The synchronous composition of two processes $p,q \in \mathbb{P}$ is defined by: $p \,|\, q =_\Delta \{\ b \cup c \mid (b,c) \in p \times q \wedge b_{|var(p)\,\cap\,var(q)} = c_{|var(p)\,\cap\,var(q)}\ \}$

In the context of component-based or contract-based engineering, refinement and substitutability are recognized as being fundamental requirements [9]. Refinement allows one to replace a component by a finer version of it. Substitutability allows one to implement every contract independently of its context of use. These properties are essential for considering an implementation as a succession of steps of refinement, until final implementation. As noticed in [27], other aspects might be considered in a design methodology. In particular, shared implementation for different specifications, multiple viewpoints and conjunctive requirements for a given component.

Considering the synchronous compositon of Signal processes and the greatest lower bound as a composition operator for contracts, we have:

Property 5.2 Let two processes $p,q \in \mathbb{P}$, and contracts $\mathbf{C}_1$, $\mathbf{C}_2$, $\mathbf{C'}_1$, $\mathbf{C'}_2 \in \mathbb{C}$.
(1) $\mathbf{C}_1 \preccurlyeq \mathbf{C}_2 \implies ((p \vDash \mathbf{C}_1) \implies (p \vDash \mathbf{C}_2))$ (4) $((p \vDash \mathbf{C}_1) \wedge (q \vDash \mathbf{C}_2)) \implies ((p \,|\, q) \vDash (\mathbf{C}_1 \Downarrow \mathbf{C}_2))$
(2) $\mathbf{C}_1 \rightsquigarrow \mathbf{C}_2 \iff ((p \vDash \mathbf{C}_1) \implies (p \vDash \mathbf{C}_2))$ (5) $((p \vDash \mathbf{C}_1) \wedge (p \vDash \mathbf{C}_2)) \iff (p \vDash (\mathbf{C}_1 \Downarrow \mathbf{C}_2))$
(3) If $\mathbf{C'}_1 \preccurlyeq \mathbf{C}_1$ and $\mathbf{C'}_2 \preccurlyeq \mathbf{C}_2$ then $\mathbf{C'}_1 \Downarrow \mathbf{C'}_2 \preccurlyeq \mathbf{C}_1 \Downarrow \mathbf{C}_2$

(1) and (2) relate to refinement and implementation; (3) and (4) allow for subtitutability in composition; (5) addresses multiple viewpoints.

In [13], we develop a module system based on our paradigm of contract for the Signal formalism, and applied it to the specification of a component-based design process. This module system, embedding data-flow equations defined in Signal syntax, has been implemented in OCaml. It produces a proof tree that consists of

1/ an elaborated Signal program, that hierarchically renders the structure of the system described in the original module expressions, 2/ a static type assignment, that is sound and complete with respect to the module type inference system, 3/ a proof obligation consisting of refinement constraints, that are compiled as an observer or a temporal property in Signal.

The property is then tended to Signal's model-checker, Sigali [21], which allows to prove or disprove that it is satisfied by the generated program. Satisfaction implies that the type assignment and produced Signal program are correct with the initially intended specification. The generated property may however be used for other purposes. One is to use the controller synthesis services of Sigali [20] to automatically generate a Signal program that enforces the property on the generated program. Another, in the case of infinite state system (e.g. on numbers), would be to generate defensive simulation code in order to produce a trace if the property is violated.

We now illustrate the distinctive features of our contract algebra by considering the specification of a four-stroke engine and its translation into observers in the synchronous language Signal.

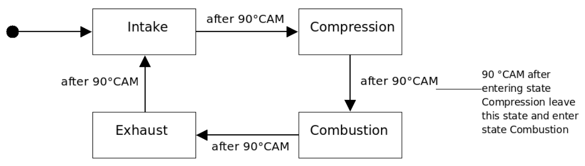

Intake
after 90°CAM
Compression
after 90°CAM
after 90°CAM
90 °CAM after entering state Compression leave this state and enter state Combustion
Exhaust
after 90°CAM
Combustion
Figure 9. State machine of 4-stroke engine cycle.

Figure 9 represents a state machine that denotes the successive operation modes of a 4-stroke engine: *Intake*, *Compression*, *Combustion* and *Exhaust*. They are driven by the camshaft whose position is measured in degrees.

The angle of the camshaft defines a discrete timing reference, the *clock cam*, measured in degrees $CAM°$, of initial value 0. Transitions in the state machine are triggered by measures of the camshaft angle. The variables cam, $Intake$, $Compression$, $Combustion$, $Exhaust$ model the behavior of the engine. We wish to define a contract to stipulate that intake always takes place in the first quarter on the camshaft revolution. To do this, we define the *generator* of a process-filter for the assumption. It should be a measure of the environmental variable cam. Namely cam should be in the first quarter. Under these assumptions, the state machine should be guaranteed to be in the intake mode, hence the generator of the process-filter for the guarantee: $\mathbf{A}_{Intake} =_{\Delta} (cam\ mod\ 360\ <\ 90)$ $\mathbf{G}_{Intake} =_{\Delta} Intake$

The generic structure of processes in contracts finds a direct instance and compositional translation into the synchronous multi-clocked model of computation of Signal. Using Signal equations:

$$\mathbf{A}_{Intake} = true\ when (cam\ mod 360 < 90) \mathbf{G}_{Intake} = true\ when\ intake\ default\ false$$

A subtlety of the Signal language is that the contract not only talks about the value, true or false, of the signals, but also about the status of the signal names, present or absent. Hence, the signal $\mathbf{A}_{Intake}$ is present and true iff cam is present and less than 90. Hence, in Signal, the complementary of the assumptions is simply defined by $\widetilde{\mathbf{A}}_{Intake} = false\ when\ \mathbf{A}_{Intake}\ default\ true$ to mean that it is true iff cam is absent or bigger than 90. Notice that, for a trace of the assumptions $\mathbf{A}_{Intake}$, the set of possible traces corresponding to $\widetilde{\mathbf{A}}_{Intake}$ is infinite (and dense) since it is not defined on the same clock as $\mathbf{A}_{Intake}$.

$\mathbf{A}_{Intake} = 1\_0\_1\_0\_1\_0\_1\_0\_1\_$ and $\widetilde{\mathbf{A}_{Intake}} = 0\_\_0\_\_0\_\_\_0\_\_0\_$ or $0111011\_01\_10\_\_101\ldots$
It is also worth noticing that the clock of $\widetilde{\mathbf{A}_{Intake}}$ (its reference in time) need not be explicitly related to or ordered with $\mathbf{A}_{Intake}$ or $\mathbf{G}_{Intake}$: it implicitly and partially relates to the *cam* clock.

Notice that our model of contract is agnostic as to a particular model of computation and accepts a generic domain of behaviors. Had we instead considered executable specifications, such as synchronous observers [22], it would have been more difficult to compositionally define the complementary of a proposition without referring to a global reference of time in the environment (e.g., a clock for the camshaft), hence abstracting every assumption or guarantee with respect to that global clock. This does not need to be the case in the present example. Beside its Boolean structure, our algebra supports the capability to compositionally refine contracts (without altering or abstracting individual properties) and accepts both synchronous and asynchronous specifications.

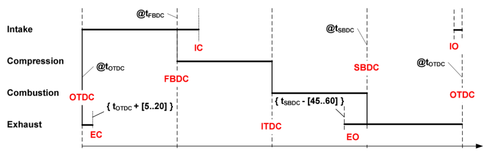


Figure 10. Model of 4-stroke engine cycle.

For instance, consider a more precise model of the 4-stroke engine found in [3](Figure 10). To additionally require that, while in the intake mode, the engine should reach the EC state (Exhaust closes) between 5 and 20 degrees, one will simply compose (greatest lower bound) the intake contract with the additional one:

$$A_{EC} = true\ when(4 < cam\ mod\ 360 < 21)\ G_{EC} = true\ when\ EC\ default\ false$$

Contracts can be used to express exclusion properties. For instance, when the engine is in the intake mode, one should not start compression.

$$\mathbf{A}_{excl} =_{\triangle} OTDC \quad \mathbf{G}_{excl} =_{\triangle} \neg FBDC$$

In addition to the above safety properties, contracts can also be used to express liveness properties. For instance, consider the protocol for starting the engine. A battery is used to initiate its rotation. When the engine has successfully started, the battery can be turned off. We can specify a contract to guarantee that engine cycles are properly counted. We write *cycle′* for the next value of the variable *cycle*.

$$\mathbf{A}_{count} =_{\triangle} Exhaust \quad \mathbf{G}_{count} =_{\triangle} cycle' = cycle + 1$$

Another contract is defined to specify that the starter battery (*starter*) will eventually be turned off after a few cycles. We write $F()$ for the future property of LTL.

$$\mathbf{A}_{live} =_{\triangle} (cycle > 0) \quad \mathbf{G}_{live} =_{\triangle} F(\neg starter)$$

6 Related work

The use of contracts has been advocated for a long time in computer science [23,14] and, more recently, has been successfully applied in object-oriented software engineering [25]. In object-oriented programming, the basic idea of design-by-contract is to consider the services provided by a class as a contract between the class and its caller. The contract is composed of two parts: requirements made by the class upon its caller and promises made by the class to its caller.

In the context of software engineering, the notion of assertion-based contract has been adapted for a wide variety of languages and formalisms but the central notion of time and/or trace needed for reactive system design is not always taken into account. For instance, extensions of OCL with linear or branching-time temporal logics have been proposed in [28,11], focusing on the expressivity of the proposed constraint language (the way constraints may talk about the internals of classes and objects), and considering a fixed "sequence of states". This is a serious limitation for concurrent system design, as this sequence becomes an interleaving of that of individual objects.

In the theory of interface automata [2], the notion of interface offers benefits similar to our notion of contracts and for the purpose of checking interface compatibility between reactive modules. In that context, it is irrelevant to separate the assumptions from guarantees and only one contract needs to be and is associated with a module. Separation and multiple views become of importance in a more general-purpose software engineering context. Separation allows more flexibility in finding (contra-variant) compatibility relations between components. Multiple views allows better isolation between modules and hence favor compositionality. In our contract algebra as in interface automata, a contract can be expressed with only one filter. To this end, a filtering equivalence relation [12] (that defines the equivalence class of contracts that accept the same set of processes) may be used to express a contract with only one guarantee filter and with its hypothesis filter accepting all the processes (or, conversely, with only one hypothesis filter and a guarantee filter that accepts no process).

In [7], a system of assume-guarantee contracts with similar aims of genericity is proposed. By contrast to our domain-theoretical approach, the EC Speeds project considers an automata-based approach, which is indeed dual but makes notions such as the complementary of a contract more difficult to express from within the model. The proposed approach also leaves the role of variables in contracts unspecified, at the cost of some algebraic relations such as inclusion.

In [16], the authors show that the framework of interface automata may be embedded into that of modal I/O automata. [27] further develops this approach by considering modal specifications. This consists of labelling transitions that *may* be fired and other that *must*. Modal specifications are equipped with a parallel composition operator and refinement order which induces a greatest lower bound. The GLB allows addressing multiple-viewpoint and conjunctive requirements. With the experience of [7], the authors notice the difficulty in handling interfaces having different alphabets. Thanks to modalities, they propose different alphabet equalizations depending on whether parallel composition or conjunction is considered. Then they consider contracts as residuations G/A (the residuation is the adjoint of parallel composition), where assumptions A and guarantees G are both specified as modal specifications. The objectives of this approach are quite close to ours. Our model deals carefully alphabet equalization. Moreover, using synchronous composition for processes and greatest lower bound for process-filters and for contracts, our model captures both substitutability and multiple-viewpoint.

In [22], a notion of synchronous contracts is proposed for the programming language Lustre. In this approach, contracts are executable specifications (synchronous observers) timely paced by a clock (the clock of the environment). This yields an approach which is satisfactory to verify safety properties of individual modules (which have a clock) but can hardly scale to the modeling of globally asynchronous architectures (which have multiple clocks).

In [8], a compositonal notion of refinement is proposed for a simpler stream-processing data-flow language. By contrast to our algebra, which encompasses the expression of temporal properties, it is limited to reasoning on input-output types and input-output causality graph.

The system JASS [5] is somewhat closer to our motivations and solution. It proposes a notion of *trace*, and a language to talk about traces. However, it seems that it evolves mainly towards debugging and defensive code generation. For embedded systems, we prefer to use contracts for validating composition and hope to use formal tools once we have a dedicated language for contracts. Like in JML [17], the notion of agent with inputs/outputs does not exist in JASS, the language is based on class invariants, and pre/post-conditions associated with methods.

Our main contribution is to define a complete domain theoretical framework for assume-guarantee reasoning. Starting from a domain-theoretical characterization of behaviors and processes, we build a Boolean algebra of process-filters and a Heyting algebra of contracts. This yields a rich structure which is: 1/ generic, in the way it can be implemented or instantiated to specific models of computation; 2/ flexible, in the way it can help structuring and normalizing expressions; 3/ complete, in the sense that all combinations of propositions can be expressed *within* the model.

Finally, a temporal logic that is consistent with our model, such as for instance the ATL (Alternating-time Temporal Logic [4]) can directly be used to express assumptions about the context of a process and guarantees provided by that process.

7 Conclusion

Starting from the choice of an abstract characterization of behaviors as functions from variables to a domain of values (Booleans, integers, series, sets of tagged values, continuous functions), we introduced the notion of process-filters to formally characterize the logical device that filters behaviors from process much like the assumption and guarantee of a contract do. In our model, a process p fulfils its requirements (or satisfies) $(\mathbf{A},\mathbf{G})$ if either it is rejected by $\mathbf{A}$ (it is then out of the scope of the contract $(\mathbf{A},\mathbf{G})$), or it is accepted by $\mathbf{G}$.

Our main results are that the structure of process-filters is a Boolean algebra and that the structure of contracts is a Heyting algebra, respectively. This rich structure allows for reasoning on contracts with great flexibility to abstract, refine and combine them. Moreover, contracts are not limited to expressing safety properties, as is the case in most related frameworks, but encompass the expression of liveness properties. This is all again due to the central notion of process-filter.

In the aim of assessing the generality and scalability of our approach, we are designing a module system based on the paradigm of contract for Signal and applying it to the specification of a component-based design process. The paradigm we are putting forward is to regard a contract as the behavioral type of a module or component and to use it for the elaboration of the functional architecture of a system together with a proof obligation that validates the correctness of assumptions and guarantees made while constructing that architecture.

References

[1] Abramsky S. and Jung A. Domain theory. In *Handbook of logic in computer science (vol. 3): semantic structures*, pages 1–168. Oxford University Press, 1997.

[2] Alfaro L. and Henzinger T. A. Interface automata. In *ESEC / SIGSOFT FSE*, pp. 109–120, 2001.

[3] André C., Mallet F., and Peraldi-Frati M.-A. A multi-form time approach to real-time system modeling. Research Report RR 2007-14-FR, I3S, 2007.

[4] Alur, R., Henzinger, T., and Kupferman, O. Alternating-time Temporal Logic. In *Journal of the ACM, v. 49*, ACM Press, 2002.

[5] Bartetzko D., Fischer C., Möller M., and Wehrheim H. Jass - Java with assertions. *In Havelund, K., Rosu, G. eds : Runtime Verification*, Volume 55 of ENTCS(1):91–108, 2001.

[6] Bell John L. Boolean Algebras and Distributive Lattices Treated Constructively, Math. Logic Quarterly 45, 1999.

[7] Benveniste A., Caillaud B., and Passerone R. A generic model of contracts for embedded systems. In *Research report N° 6214, INRIA-IRISA, S4 Team*, 2007.

[8] Broy, M. Compositional refinement of interactive systems. Journal of the ACM, v. 44. ACM Press, 1997.

[9] Doyen L., Henzinger T. A., Jobstmann B., and Petrov T., Interface theories with component reuse, In *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT'08*, 2008, pp. 79-88.

[10] Edwards S., Lavagno L., Lee E.A., and Sangiovanni-Vincentelli A. Design of Embedded Systems: Formal Models, Validation, and Synthesis. In *Proceedings of the IEEE, 85(3)*, pages 366–390, 1997.

[11] Flake S. and Mueller W. An OCL extension for realtime constraints. In *Lecture Notes in Computer Science 2263*, pp. 150–171, 2001.

[12] Glouche Y., Le Guernic P., Talpin J.-P., and Gautier T. A boolean algebra of contracts for logical assume-guarantee reasoning. Research Report RR 6570, INRIA, 2008.

[13] Glouche Y., Talpin J.-P., Le Guernic P., and Gautier T. A module language for typing by contracts. In *Nasa Formal Methods Symposium*, Springer, 2009.

[14] Hoare C.A.R. An axiomatic basis for computer programming. In *Communications of the ACM*, pp. 576–583, 1969.

[15] Kopetz H. Component-Based Design of Large Distributed Real-Time Systems. In *Control Engineering Practice, 6(1)*, pages 53–60, 1997.

[16] Larsen K.G., Nyman U., and Wasowski A. Modal I/O automata for interface and product line theories. In *Programmming Languages and Systems, 16th European Symposium on Programming, ESOP'07*, ser. Lecture Notes in Computer Science, vol. 4421. Springer, 2007, pp. 64-79.

[17] Leavens G. T., Baker A. L., and Ruby C. JML: A notation for detailed design. In Kilov H., Rumpe B., and Simmonds I., editors, *Behavioral Specifications of Businesses and Systems*, pp. 175–188. Kluwer Academic Publishers, 1999.

[18] Lee E.A., Sangionanni-Vincentelli A., A framework for comparing models of computation. In IEEE transaction on computer-aided design, v. 17, 1998.

[19] Le Guernic P., Talpin J.-P., and Le Lann J.-C. Polychrony for system design. *Journal for Circuits, Systems and Computers*, Special Issue on Application Specific Hardware Design, 2003.

[20] Marchand, H., Bournai, P., Le Borgne, M., Le Guernic, P. Synthesis of Discrete-Event Controllers based on the Signal Environment, Discrete Event Dynamic System: Theory and Applications, v. 10(4), 2000.

[21] Marchand, H., Rutten, E., Le Borgne, M., Samaan, M. Formal Verification of programs specified with Signal: Application to a Power Transformer Station Controller. Science of Computer Programming, v. 41(1). Elsevier, 2001.

[22] Maraninchi F. and Morel L. Logical-time contracts for reactive embedded components. In *In 30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04, Rennes, France*, 2004.

[23] Martin A. and Lamport L. Composing specifications. In *ACM Trans. Program. Lang. Syst. 15*, pp. 73–132, 1993.

[24] Mazurkiewicz A. Basic notions of trace theory. In *Lecture Notes In Computer Science*, v. 354, pp. 285 - 363, 1989.

[25] Meyer B. *Object-Oriented Software Construction, Second Edition.* ISE Inc., Santa Barbara, 1997.

[26] Mitchell R. and McKim J. *Design by Contract, by Example*, Addison-Wesley, 2002.

[27] Raclet J.-B., Badouel E., Benveniste A., Caillaud B., Passerone R. Why are modalites good for Interface Theories? In *9th International Conference on Application of Concurrency*, 2009.

[28] Ziemann P. and Gogolla M. An extension of OCL with temporal logic. In *Critical Systems Development with UML-Proceedings of the UML'02 workshop, Technische Universität München, Institut für Informatik*, 2002.

Automated formalisation for verification of diagrammatic models

James R. Williams [1,2]

Department of Computer Science
University of York, York, UK

Fiona A. C. Polack [3]

Department of Computer Science
University of York, York, UK

Abstract

Software engineering uses models to design and analyse systems. The current state-of-the-art, various forms of model-driven development, uses diagrams with defined abstract syntax but relatively-lose translational approaches to semantics, which makes it difficult to perform rigorous analysis and verification of models. Here, we present work-in-progress on tool support for formal verification of diagrammatic models. The work builds on Amálio's rigorous template-based approach to formalisation, which formally expresses the intended semantics of both the diagram notation and modelled system, along with standard correctness conjectures and, in many cases, proof of these conjectures.

Keywords: formal verification, model-driven development, tool-support

1 Introduction

In practical software engineering, diagrammatic approaches are widely used for sketching, specifying and designing systems. There are challenges in applying formal analysis to these approaches, either because the semantics is inadequately defined, or because the level of detail of the semantics does not admit interesting or useful formal analysis. Conversely, the formal notations used in verification of critical systems and in academia are considered inaccessible by many practical engineers [21,12,27]. If it were possible to provide access to formal analysis without great

[1] We would like to thank: Dimitrios Kolovos, without whose help and Epsilon tool-suite we could not have produced this tool; Nuno Amálio, Faculté des Sciences, de la Technologie et de la Communication, University of Luxembourg, whose doctoral work on GeFoRME and FTL form the basis of the tool; Leo Freitas and Ian Toyn, for advice on CZT and CADiZ; Richard Paige for comments on earlier drafts; and the UK National Research and Training Initiative: Large-Scale Complex IT Systems for part sponsorship.

[2] Email: james@jamesrobertwilliams.co.uk

[3] Email: fiona@cs.york.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

cost, the rigour and reliability of diagrammatic models could be significantly enhanced. For many years, people have sought to combine formal and diagrammatic approaches so that formal type-checking and proof of properties can be used to explore the correctness and internal consistency of diagrams. However, few integrations have found favour, either because formalisation does not meet engineering needs, or because engineers are exposed to the formalisation and formal model.

In an attempt to bring formalisms into practical engineering, Amálio [3] proposes a generative framework for rigorous model engineering (GeFoRME) that applies template-based translation to diagrammatic models. Formal templates capture the intended semantics of diagram concepts as well as the modelled system. A well-founded template language (the formal template language, FTL [3,8]) makes it possible to generate not only formal models but also standard consistency conjectures and, in many cases, their proofs. Amálio demonstrates an instance of GeFoRME called UML+Z, comprising a library of templates that capture the object semantics of UML. Instantiating the templates with details from UML diagrams results in a Z model that conforms to Amálio's object-oriented (OO) Z structuring, ZOO [4,7,3,10]. UML+Z has been used on small case studies [3,9] of conventional object-oriented models including UML class and state diagrams and Catalysis [19] snapshots. It has also been used in an attempt to formalise part of the meta-object facility [2] and on a model of autonomous objects [10]. The GeFoRME approach provides ease of construction, strong semantic support, and traceability between the formal and diagrammatic models. If templates adequately capture the semantics of source-model concepts, and the source model is syntactically correct, then the transformation produces a type-correct formal model, on which the conjectures constructed by template translation are true by construction [6,8].

In all the work with UML+Z, however, a major inhibiting factor is the lack of tool support for management and instantiation of templates. The GeFoRME approach employs a formal *translation*. Translation is a conventional way of considering "integrated methods" (reviewed in, for example, [17]), but it is essentially the same as the *model transformations* that are characteristic of model-driven development (MDD) – the family of approaches to tool-supported practical software development that focuses on the construction and manipulation of primarily-diagrammatic models [29]. As in formal translation, MDD model transformation defines a mapping from a source model to a target model [11]. The mapping, or transformation definition, comprises transformation rules which specify how each modelling construct is transformed. In transforming diagrammatic models to formal models, we apply a model-to-text (as opposed to model-to-model) transformation. Model-to-text transformation is most commonly used for code generation [1], so is appropriate for generating, for instance, the LaTeX (or any other) markup for a formal notation.

This paper presents a tool for the management and instantiation of FTL templates. The tool currently supports the UML+Z translation, as well as a means to extend the template base; the tool and GeFoRME are sufficiently generic that the approach could also support other diagrammatic source and formal target languages. The tool provides the interaction necessary for a naïve user to generate a formal model and automatically verify it using existing Z tools. In addition, the tool supports expert users who wish to extend the template repertoire or provide new specialisations of the tool.

In the next section, we introduce Amálio's template-based formalisation. In section 3 we describe the tool support, and in section 4 we illustrate the capabilities of UML+Z, the translation tool and support for different user levels. In section 5, we discuss the advantages and limitation of the work so far, and consider the extensions to the tool and the template library.

2 Amálio's Template-based Formalisation

In [9], Amálio motivates his work by identifying some shortcomings of diagrammatic approaches to software engineering that can be addressed by practical formalisation. He notes the need to model concepts that cannot be expressed diagrammatically – constraints and model properties – and considers the general problem of semantics. Diagrammatic approaches such as entity-relationship modelling and UML are widely used in practical software engineering, and the notations have well-defined abstract syntax (for instance, UML is defined using metamodels, based on the concept of reflectivity). However, the semantics of concepts such as *class*, *association* and *generalisation*, are under-specified. In most cases, the semantics is clarified only when a diagrammatic model is converted to code – thus, a class diagram that forms the basis for a Java program assumes a Java semantics, but the same diagram used to create a C++ program assumes a C++ semantics. In the case of UML (and related domain-specific modelling notations), annotations are added to diagrams using notations such as the object constraint language, OCL. The relationship between the modelling notations and the constraint language is defined at the abstract syntax level. An obvious connotation of this situation is that whilst properties of models can be demonstrated informally (for example, by animation), analysis of consistency or model properties is not consistent across modellers or modelling tools. In practice, little attention is paid to consistency across model-views or between diagrams and constraint expressions. Note that this problem is not unique to diagrammatic models: insufficient semantic underpinning was also a problem for Z, ultimately addressed in its ISO standardisation.

Most attempts to associate formal semantics to diagrammatic modelling assign a specific formal meaning to each diagrammatic concept. Whilst this significantly reduces ambiguity, and admits formal analysis, the formalisation assumes a single, fixed semantics for each concept [9] – and the semantics that is assumed is often only apparent to the formalist. One well-known example is the UML to Object-Z translation, which imposes the semantics of Object-Z inheritance on UML generalisation [15,14]. By contrast, Amálio's approach [3] builds on ideas of pattern-based development [20] and problem-driven methods [22] to advocate a framework for rigorous, but practical MDD [9]; concept semantics are captured explicitly in the templates, so a different semantics simply requires use of a different set of templates. The traceability provided by the template transformation approach allows engineers to work with the diagrams, which, in effect, form a graphical interface for the formality that lies beneath. In many cases, Amálio's approach allows the formalism to be completely hidden from the developer.

Amálio devised the Formal Template Language (FTL) [3,8] as the rigorous underpinning to the GeFoRME framework, supporting proof with template representations. Having captured patterns of formal development (e.g. a model structure) in FTL, reasoning can be applied at the pattern level using meta-proof. For example, a precondition of an operation can be calculated or an initialisation conjecture proved to establish meta-theorems that apply when the concept templates are instantiated [3,8]. The FTL templates and translation process are illustrated in Figure 1 (which also shows the subsequent automation, covered in section 3).

The existing UML+Z templates and meta-theorems relate to Amálio's ZOO structuring for Z [7,3]. ZOO uses standard Z – and can easily adapt to pre-standardisation dialects such as the Z/Eves variant. Unlike Object-Z [30], ZOO does not require any extension to the Z language or its tool support. Amálio shows how a ZOO model is built incrementally using template instantiation. Structural components are views representing the main OO concepts: objects, classes, associations and system. An object is an atom, a member of the set of all possible objects and of the set of possible objects of its class [10]. The class structure uses a *promoted Z abstract data type* [16]; it is represented by an intensional structure that defines the common properties of the class's objects, and an extensional structure, that defines the class as the set of its objects. The formal association structure forms tuples of the classes' objects using a Z relation. A system is an ensemble of classes and associations. Additional properties (constraints, conjectures) are expressed on the appropriate views.

2.1 Template Instantiation: Amálio's Bank Case Study

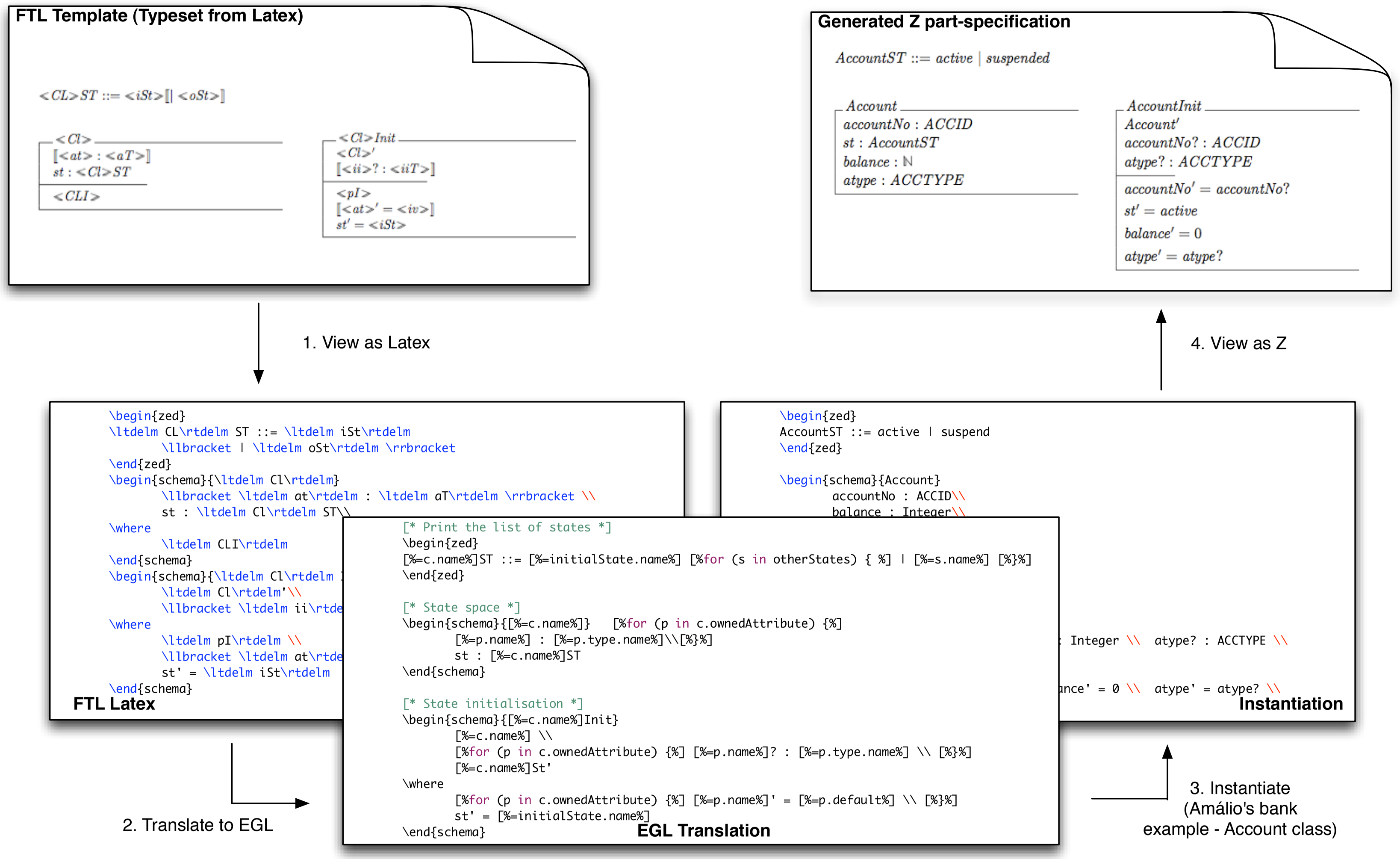
One of Amálio's case studies is a banking system. Amálio's formalisation of two classes from his example demonstrates in outline how template instantiation works. Amálio gives further descriptions in [3,9,8,6,4].

The class diagram, figure 2, shows two classes – *Customer* and *Account*. The association, *holds*, allows an instance of *Customer* to have zero or more accounts, and an instance of *Account* to have exactly one customer. The *Account* class has operations to *withdraw* money, to *deposit* money, to *getBalance* for an account; to *suspend* and to *reactivate* an account. The state changes caused by the *Account* operations are described in figure 3. The formalisation is demonstrated here for the *Account* class, illustrated in figure 1.

Select FTL templates The templates needed to instantiate the *Account* class comprise those for the intensional and extensional definition of a class, and those for the initialisation of the class (initialisation is a formal technique that captures a potential start-state for the system, and then proves that this is a valid state of the system) – see "FTL Template (Typeset from Latex)" in figure 1.

Select class diagram concepts and instantiate templates Each template is instantiated by replacing placeholders (e.g. $\lessdot x \gtrdot : \lessdot t \gtrdot$) with relevant concept and type names from the UML diagram. Where a template includes a FTL list (e.g. $[\![\ldots]\!]_{(sep,empty)}$), the statement is instantiated once for each element – shown in figure 1 as the instantiation of attributes of the *Account* class.

Select operation templates and identify operation effects For each operation on a class, the template instantiation depends on the type of operation.


FTL Template (Typeset from Latex)
Generated Z part-specification
1. View as Latex
2. Translate to EGL
3. Instantiate (Amálio's bank example - Account class)
4. View as Z
FTL Latex
EGL Translation
Instantiation

FTL Template (Typeset from Latex):

$<CL>ST ::= <iSt> \llbracket \mid <oSt> \rrbracket$

```
<Cl>
[<at> : <aT>]
st : <Cl>ST
----
<CLI>
```

```
<Cl>Init
<Cl>'
[<ii>? : <iiT>]
----
<pI>
[<at>' = <iv>]
st' = <iSt>
```

FTL Latex:

```
\begin{zed}
\ltdelm CL\rtdelm ST ::= \ltdelm iSt\rtdelm
        \llbracket | \ltdelm oSt\rtdelm \rrbracket
\end{zed}
\begin{schema}{\ltdelm Cl\rtdelm}
        \llbracket \ltdelm at\rtdelm : \ltdelm aT\rtdelm \rrbracket \\
        st : \ltdelm Cl\rtdelm ST\\
\where
        \ltdelm CLI\rtdelm
\end{schema}
\begin{schema}{\ltdelm Cl\rtdelm
        \ltdelm Cl\rtdelm'\\
        \llbracket \ltdelm ii\rtde
\where
        \ltdelm pI\rtdelm \\
        \llbracket \ltdelm at\rtde
        st' = \ltdelm iSt\rtdelm
\end{schema}
```

EGL Translation:

```
[* Print the list of states *]
\begin{zed}
[%=c.name%]ST ::= [%=initialState.name%] [%for (s in otherStates) { %] | [%=s.name%] [%}%]
\end{zed}

[* State space *]
\begin{schema}{[%=c.name%]}    [%for (p in c.ownedAttribute) {%]
        [%=p.name%] : [%=p.type.name%]\\[%}%]
        st : [%=c.name%]ST
\end{schema}

[* State initialisation *]
\begin{schema}{[%=c.name%]Init}
        [%=c.name%] \\
        [%for (p in c.ownedAttribute) {%] [%=p.name%]? : [%=p.type.name%] \\ [%}%]
        [%=c.name%]St'
\where
        [%for (p in c.ownedAttribute) {%] [%=p.name%]' = [%=p.default%] \\ [%}%]
        st' = [%=initialState.name%]
\end{schema}
```

Instantiation:

```
\begin{zed}
AccountST ::= active | suspend
\end{zed}

\begin{schema}{Account}
        accountNo : ACCID\\
        balance : Integer\\
: Integer \\  atype? : ACCTYPE \\
nce' = 0 \\  atype' = atype? \\
```

Generated Z part-specification:

$AccountST ::= active \mid suspended$

```
Account
accountNo : ACCID
st : AccountST
balance : ℕ
atype : ACCTYPE
```

```
AccountInit
Account'
accountNo? : ACCID
atype? : ACCTYPE
----
accountNo' = accountNo?
st' = active
balance' = 0
atype' = atype?
```

Fig. 1. The steps of a UML+Z Translation, with examples of the type-set and LaTeX formats involved

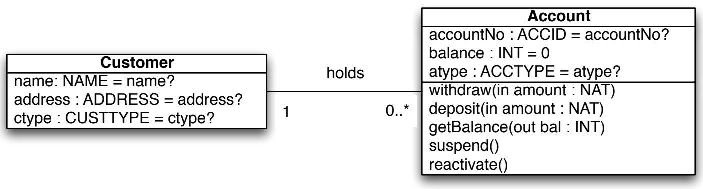

Customer
name: NAME = name?
address : ADDRESS = address?
ctype : CUSTTYPE = ctype?
holds
1
0..*
Account
accountNo : ACCID = accountNo?
balance : INT = 0
atype : ACCTYPE = atype?
withdraw(in amount : NAT)
deposit(in amount : NAT)
getBalance(out bal : INT)
suspend()
reactivate()

Fig. 2. Extract from Amálio's bank system class diagram [3]

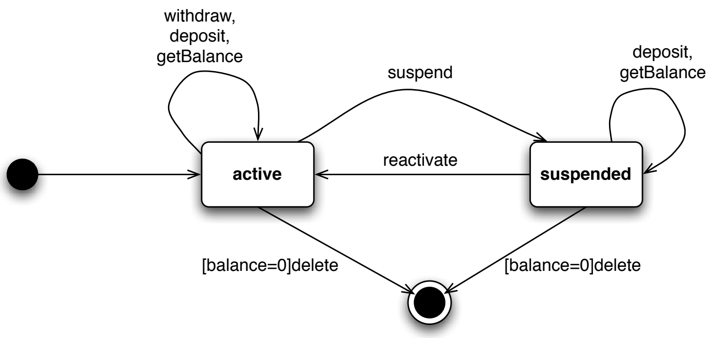


Fig. 3. Extract from Amálio's bank system state diagram [3]

Details of the operation are not included in class diagrams, but can be extracted from other UML diagrams or from diagram annotations, or can be entered by the user. For instance, the *withdraw* operation is identified as an `update` operation that changes the state of the system (the alternative is `observe`). Here, the attribute value that changes is *balance*, and the operation uses the formula, $balance - amount?$, where $amount?$ is the input to the operation.

Identify states from state diagram The state diagram shows which system states need to be represented, but these have to be interpreted in terms of the classes and attributes. For instance, in figure 3, the two states are *active* and *suspended*. The appropriate template instantiation adds an "attribute", *st: accountST*, to the formal representation of *Account*. Template instantiation then constructs the formal definition of the attribute type *accountST*, which here results in the expression, $accountST ::= active|suspended$, shown in figure 1.

Instantiate initialisation templates Initialisation in Z is an operation that has only an after-state, represented in post-condition predicates. Unlike other operations, it can be generated automatically by setting all attributes to a default value or an input. The initialisation template is shown in "FTL Template (Typeset from Latex)" of figure 1, and its instantiation is shown in the "Generated Z part-specification". Note that primed components (e.g. *Account'*) are the after-states of an operation, and queried components (e.g. *atype?*) are inputs. The initialisation of the state attribute *st'* is derived from the state diagram, since the start event in the state diagram points to the first state of the system – in the *Account* example, *st' = active*. In addition to the initialisation specification shown in figure 1, a Z conjecture is generated, that the initialisation state is a valid state of the system. This is associated to one of the meta-theorems (discussed above), and is true by construction if the Z model is type-correct.

3 The AUtoZ Tool

The AUtoZ tool (www.jamesrobertwilliams.co.uk/autoz.php) provides automation for Amálio's UML+Z process, outlined in section 2.1. This section describes design criteria for the tool. It then outlines the design of the *Automatic formalisation of UML to Z* tool framework, AUtoZ.

3.1 Tool Rationale

The motivation of enhancing the rigour of practical software engineering means that the tool has to be readily accessible to software engineers. The tool should integrate with existing software engineering tools, and should be able to inter-operate directly with existing diagramming and formal support tools.

Amálio's GeFoRME is a generic framework, which can be specialised for different diagrammatic and formal notations, so the tool needs to be modifiable to other notations.

The tool needs to catalogue the existing UML+Z templates efficiently and support the addition of new templates and meta-proofs: there is potential to extend the existing UML+Z templates with a wide range of alternative concept semantics. This requirement highlights the need to provide support for both basic transformation-and-analysis use, and expert maintenance of the template libraries.

3.2 Tool Design

The AUtoZ tool is a framework on which different instance tools can be built. The two instances that currently exist are AUtoCADiZ and AUtoZ/Eves. AUtoZ is a plug-in for the Eclipse development environment, which is used as the basis for many modelling and model-management activities in software engineering.

The tool requires serialised input of a diagram whose concepts (abstract syntax) can be selected. Common MDD diagramming tools provide serialised XMI output, and are underpinned by a metamodel that defines the abstract syntax (concepts) of the notation. This allows concepts and their labels to be automatically extracted from diagrams, to instantiate the FTL templates. Here, we use the existing Eclipse modelling plug-in, UML2 (www.eclipse.org/uml2/), since it supports UML 2.x modelling, uses a standard metamodel-based approach to abstract syntax, and sits within the Eclipse development environment. However, in principle AUtoZ could use XMI output from many other modelling tools.

To represent the FTL templates as model transformation mappings, we convert the FTL format to Epsilon EGL. Epsilon (www.eclipse.org/gmt/epsilon) provides a suite of integrated model-management tools [25,24], as part of the Eclipse-GMT project. EGL, the Epsilon Generation Language, supports model-to-text transformation. An EGL *run configuration* specifies a file containing the XMI source model, and a file containing EGL transformation rules to be executed on the source model, as described in [26]. The AUtoZ Eclipse plug-in is a customised EGL run configuration that executes the transformations on a UML2 XMI source model to

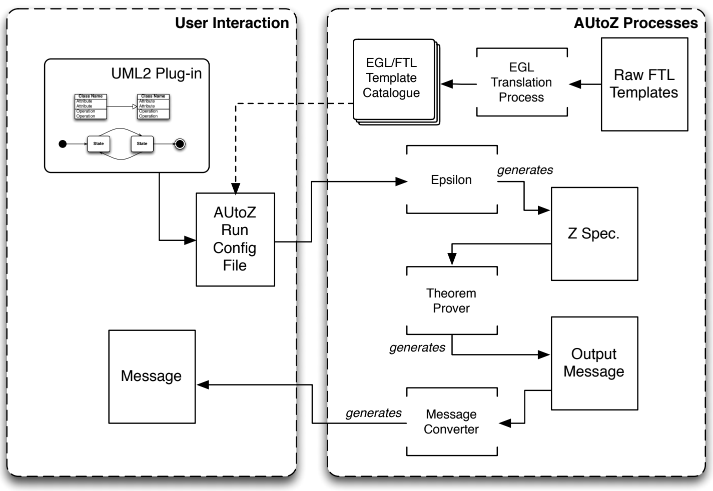


Fig. 4. AUtoZ workflow. UML2 and Epsilon are existing Eclipse facilities. The theorem prover can be any suitable formal analysis tool, and is called automatically by the AUtoZ tool instance.

Table 1
Some of the FTL-to-EGL conversions

FREE TEXT:	FTL (typeset)	$subCl : CLASS \rightarrow CLASS$
	FTL (LaTeX)	`subCl : CLASS \to CLASS`
	EGL	`subCl : CLASS \to CLASS`
PLACEHOLDER:	FTL (typeset)	$\ll x \gg : \ll t \gg$
	FTL (LaTeX)	`\ltdelm x \rtdelm : \ltdelm t \rtdelm`
	EGL	`[%=x.name%] : [%=x.type.name%]`
LIST:	FTL (typeset)	$[\![\cdots]\!]_{(sep,empty)}$
	FTL (LaTeX)	`\llbracket ...\rrbracket_{sep,empty}`
	EGL	`[% var append : Boolean := false;`
		`for (...){`
		`if (append){%]` *sep* `[%}`
		`...;`
		`append := true;`
		`} %]`

generate Z LaTeX markup, which is then input directly to the formal support tool for analysis. Figure 4 summarises the workflow of AUtoZ. A developer provides the UML model; the tool calls an AUtoZ run configuration file; Eclipse plug-ins convert UML to Z LaTeX via EGL, and call the relevant Z tool. The result of the formal analysis is returned to the user.

The EGL transformation rules are also templates, in a format that is very similar to FTL (though without any formal underpinning). Three typical FTL templates are shown in Table 1 with their LaTeX source, and the corresponding EGL rules. Many of the FTL concepts have a direct equivalent in EGL, and others, such as FTL lists, follow a common pattern in EGL.

The component architecture of AUtoZ (figure 5) facilitates development of tool specialisations and instances integrated with different formal analysis tools. The *AUtoZ framework* component is the framework common to any UML+Z tool instantiated from the AUtoZ framework, and comprises two Eclipse plug-ins: *common* provides the necessary Eclipse features (dialogues, wizards, tools and launchers),

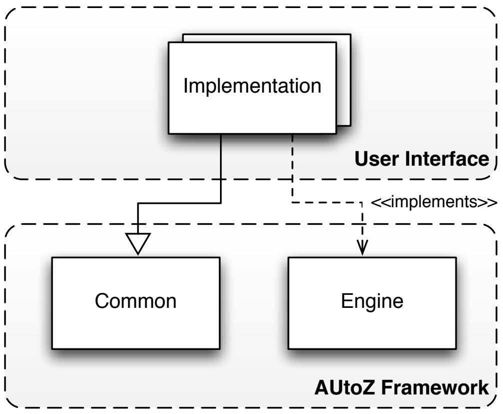


Fig. 5. The generic architecture layers of AUtoZ

and *engine* defines the interfaces that specialisations must deploy. The *user interface* component represents the specific tool implementation, also developed as Eclipse plug-ins. This allows different versions of AUtoZ using different formal tools to co-exist in the same Eclipse installation.

The activities required to transform UML to Z LaTeX markup, and to present the LaTeX to the formal tool are designed to be completely transparent. This is necessary to support the *basic user*, a developer who is well-versed in diagrammatic modelling but has little interest in formal methods. For such a user, the AUtoZ tools just extend conventional modelling tools with press-button validation of models, with no exposure to the formal underpinning.

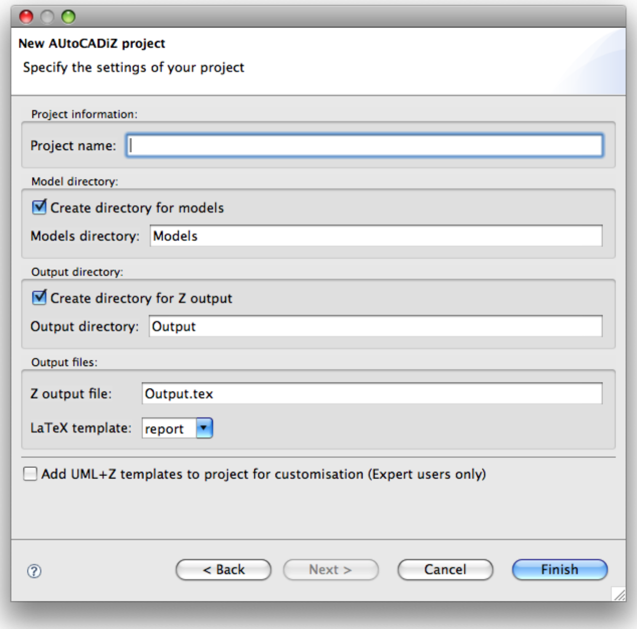
For the *expert user*, the AUtoZ tool creates an AUtoZ Eclipse project to manage files and support creation and editing of new FTL (EGL) templates. The expert user needs to be familiar with formal methods and able to interact directly with formal notations and tools, in order to develop new FTL templates and meta-theorems and to write and edit EGL transformation rules corresponding to the FTL templates – perhaps to capture the variant semantics of particular source models. In some circumstances, an expert user may be needed to complete validation, for instance where a particular model has a property that is not captured in a meta-theorem or cannot be discharged automatically.

4 Using AUtoCADiZ

AUtoZ is a generic tool framework. To illustrate its use, we consider an instance of AUtoZ that uses the CADiZ Z tool. We describe both the basic and expert uses. Space does not allow a detailed analysis, so we simply present a usage scenario for both the basic and the expert user; further examples are considered in [31].

4.1 Basic Use to Validate a UML2 Model

AUtoZ implementations are installed in the Eclipse IDE, and can be selected from the AUtoZ folder. A basic user wishing to validate the bank model shown in figures 2 and 3 would use the AUtoCADiZ implementation of the tool as described in the following workflow.



New AUtoCADiZ project
Specify the settings of your project

Project information:
Project name:

Model directory:
☑ Create directory for models
Models directory: Models

Output directory:
☑ Create directory for Z output
Output directory: Output

Output files:
Z output file: Output.tex
LaTeX template: report

☐ Add UML+Z templates to project for customisation (Expert users only)

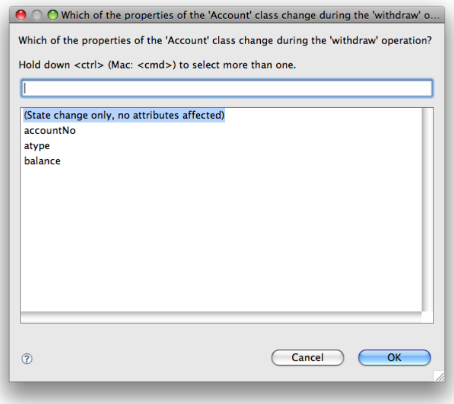
Fig. 6. Part of the dialogue used to create a new AUtoCADiZ project.

Start a new project The user opens the Eclipse IDE, and uses the `File` menu to open a `new project` dialogue. An AUtoCADiZ project is set up and named, in normal Eclipse style. For example, the user might name the project `Spec1`, and save the generated Z LaTeX markup to file `Spec1.tex`. Completion of the set-up (clicking `Finish`) results in the new project appearing in the Eclipse `Project Explorer` view. A screenshot from this dialogue is shown in figure 6.

Diagram creation AUtoCADiZ is linked to the Eclipse UML2 plug-in, which the user can use to create diagrams such as figures 2 and 3. Some design conventions have to be followed to facilitate the transformation, notably the definition of attribute types (used to generate Z given sets and other user-defined types). At this stage, the state diagram is associated to the class diagram *Account* class by naming conventions only, and the transition labelling is simply a list of the operations responsible for each change of state. On completion, the user saves the diagrams to the `Models` directory.

Create a run configuration Using the Eclipse `Run Configurations` dialogue, the user selects the `AUtoCADiZ Template` option and `New launch configuration`, associating these to the `Spec1` project and `Spec1.tex` output file. The user opts to print the CADiZ output to the console. This run configuration is saved.

Execute the formalisation Transformation is initiated by running the saved run configuration. Almost all of the target Z model is generated directly from the serialised output of the diagramming tool, with no user interaction. However, as in GeFoRME, the Z model of class operations requires detail not in the prepared diagrams. Currently, the tool presents a simple dialogue (figure 7) to guide



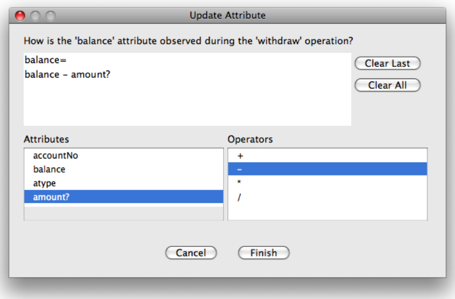
Which of the properties of the 'Account' class change during the 'withdraw' o...

Which of the properties of the 'Account' class change during the 'withdraw' operation?

Hold down <ctrl> (Mac: <cmd>) to select more than one.

(State change only, no attributes affected)
accountNo
atype
balance

(a) Specifying which attributes change



Update Attribute

How is the 'balance' attribute observed during the 'withdraw' operation?

balance=
balance – amount?

Attributes: accountNo, balance, atype, amount?

Operators: +, –, *, /

(b) Specifying how an attribute changes

Fig. 7. Part of the user dialogue for input of details for operations. Attributes are taken from the class diagram. The change formula options reflect the Z types of the selected attributes.

the user in adding the required operation details (see section 5.1 for proposed extensions).

The tool then completes the generation of the Z LaTeX markup, which is passed to CADiZ for type-checking and analysis. The user receives the output from CADiZ at the console, as requested. Amálio has shown that his templates and meta-theorems produce type-correct Z when run on a syntactically-correct and consistent UML model [3]. In most cases, therefore, the basic user can expect to get a simple confirmation message from CADiZ. We return to this in section 5.1 below.

4.2 Expert Use: Working with Templates

An expert user needs some understanding of the template languages (EGL, FTL), and the formalisation process. The AUtoZ tools then provide an interface for adding and changing templates. The dialogue to create a project (outlined for AUtoCADiZ, above) provides the appropriate options, which result in a `Templates` directory being made available for the project, in the Eclipse `Project Explorer` view. Templates can be edited and added to subfolders containing EGL transformation rules.

For example, consider the addition of a new template which lists all of the classes in the system, named `AllClasses()`. When creating a new AUtoZ project, the expert user selects an option to include the template catalogue in the project directory. A file is added to the catalogue containing the new EGL template:

```
[* Template to list all classes *] [% operation AllClasses
      (){
for (c in Class.allOfType){ %]
   [%=c.name%] \\
   [%]}
}%]
```

This template can be used in an instantiation by editing the EGL logic file in the template catalogue (`Logic.egl`) to include the following:

```
...
    [* import new operation *]
    import 'AllClasses.egl';
...
```

```
    ...
    \chapter{[%=p.name%] Package}

      [* Call the new operation *]
      \section{List of Classes in [%=p.name%] Package}
        AllClasses(); ...
```

To use the modified logic file, the expert uses the `run configuration` dialogue to select the customised file, in place of the default template catalogue (stored in the Eclipse plug-in bundle).

Further examples of expert use, including template creation, tool customisation, and tool instance creation can be found in [31].

5 Discussion

Development of the AUtoZ tool is work in progress. To date, the template instantiation of Amálio's bank case study has been completely replicated in the AUtoCADiZ tool, and parts have been replicated in AUtoZ/Eves, including the verification of instantiated conjectures and meta-theorems. In this section, we outline a number of future directions for this work.

5.1 Research Areas

Two aspects of the tool support require further research, namely the formal specification of operations from UML diagrams, and the handling of messages from the formal tools.

Operations Formally specifying UML operations in Z is not a straightforward task, because the details needed to instantiate templates are scattered across UML diagrams, require interpretation from diagrams, or are simply not amenable to recording in diagrams. The current approach using user dialogues does not scale – even modest UML diagrams often have very large numbers of operations. Further research is needed to find patterns and commonalities in operation construction that can be used to automatically incorporate operation details from other diagrams and minimise user interaction. Furthermore, the tools, and Amálio's UML+Z approach, need extending to exploit pre- and post-conditions written in OCL – an existing body of work on the use of OCL in formalisation (with B or Object-Z target models) [13,28,23] provides a starting point.

Error Handling As noted above, the AUtoCADiZ tool currently outputs CADiZ error messages direct to the user. This is unsatisfactory, as the interpretation of typical CADiZ messages requires some expertise both in CADiZ and in Z. Clearly, the tools cannot truly accessible to modellers and software engineers until the error messages generated by the Z tools can be related back to the templates, and thus to the inconsistent parts of the UML models.

In general, relating the error messages to diagrammatic models is the most important hurdle to overcome if non-expert practitioners are to be able to use formally-underpinned tools to analyse diagrammatic models. One direction would be to develop another intermediate language into which formal tool messages can be translated, and have the intermediate language manage the mapping back to diagrammatic model components. This task is complicated by the diverse ways in which formal tools present messages. There is some new work on traceability and message generation using Epsilon, which may offer a way forward [18].

5.2 Extensions to the Tool

Automatic template translation In describing the tool, we have shown that FTL maps readily on to EGL, and we note that an expert user involved in template maintenance would need to be familiar with both languages. FTL is also needed for meta-theorems and proof work, because of its formal underpinnings [3]. We have the groundwork for an automatic translation from FTL to EGL; when implemented it will allow the expert user to maintain templates using either FTL or EGL.

Template catalogue Williams [31] discussed various necessary improvements to the cataloguing of Amálio's FTL templates, but the tool currently relies on copying the templates to the AUtoZ project. A better solution would be to have a central, physically browsable catalogue, and annotated templates, as well as ways to add custom templates to the general catalogue. Note that it is not enough to provide an interactive catalogue; we need to provide support for template-validation, using Amálio's approach to meta-theorems and the relevant formal analysis tools.

Examples and help Whilst the AUtoZ tools present a solution to the shortage of formally-trained software engineers, potential basic users of the tool still need to understand how to validate models with the tools. We would like to add example models and Z specifications as training and reference literature. The Eclipse Examples Project (`www.eclipse.org/examples`) could host such examples and help for new tool users.

Improved run configuration The current run configuration does not provide suitable checkpoints in the process of generation and analysis of formal models. For example, the user has to generate a complete Z specification and pass it to the formal tool for analysis; if the `run configuration` has not been set up properly, and the connection to the formal tool is not achieved (for example, an environment variable has not been set), then the user has to start again and re-generate the Z specification. This could be addressed by providing some checking of the `run configuration` file before execution. Furthermore, generated Z models could easily be saved to file before being passed to the formal analysis tool. A logical next step would then be lazy re-generation, which only re-generates parts of models that have changed. Epsilon tools can be used to compare models, though some further work is needed to identify the FTL template connotations of changes to diagrams.

Inclusion of Amálio's templates for inheritance and composition Several parts of UML+Z are not yet included in the AUtoZ tool template repertoire. In particular, Amálio considers various semantics of inheritance [3] and proposes templates for the formalisation of composition relations [5], including the proof that models remain consistent on deletion of a composed class.

Linkage to CZT The Community Z Tools (CZT) initiative (`czt.sourceforge.net`) offers a range of support tools for Z, including parsers, translators, type-checking and a Unicode markup for Z. It would be useful to investigate integration of AUtoZ tools into this formal project.

Generalising tool support Amálio [3] developed a generic formalisation framework, GeFoRME, of which UML+Z is one specialisation. The AUtoZ tool could be made more generic by converting the components of the *AUtoZ framework* layer to a *GeFoRME Framework*, upon which any GeFoRME Framework application could be built. Under this development, AUtoZ would be constructed as a specific instance of the GeFoRME framework, with AUtoCADiZ, AUtoZ/Eves as implementations. Other GeFoRME frameworks, such as a UML+B Framework, could be built upon the same structures. Such a product-line architecture would allow massive reuse of components and powerful tools. There is also the potential to generalise the handling of source models.

6 Conclusion

The AUtoZ tools are a practical contribution to the verification, through formal analysis, of diagrammatic models. The work exploits the component-based UML+Z formalisation of Amálio, itself motivated by the desire to make formal analysis accessible to diagrammatic modellers. We have described the tools, the levels of use that they support, and the potential for generalisation and extension. Working within the Eclipse IDE and an established MDD tool suite, makes this style of formalisation both practical and usable in state-of-the-art diagram-based software modelling, and in model-driven engineering.

References

[1] Aagedal, J., A. Berre, R. Gronmo, J. Neple and T. Oldevik, *Toward standardised model to text transformations*, in: *Model Driven Architecture Foundations and Applications*, LNCS **3748** (2005), pp. 239–253.

[2] Abdul Sani, A., "Formal Analysis of Metamodel: an evaluation of an approach using ZOO templates to derive a Z model of part of EMOF," Master's thesis, Computer Science, University of York (2007).

[3] Amálio, N., "Generative frameworks for rigorous model-driven development," Ph.D. thesis, Dept. Computer Science, Univ. of York (2007).

[4] Amálio, N. and F. Polack, *Comparison of formalisation approaches of UML class constructs in Z and Object-Z*, in: *ZB 2003*, LNCS **2651** (2003), pp. 339–359.

[5] Amálio, N., F. Polack and S. Stepney, *Modular UML semantics: Interpretations in Z based on templates and generics*, in: *FACS'03 Workshop* (2003), pp. 81–100, technical Report 284: `www.iist.unu.edu/newrh/III/1/docs/techreports/report284.html`.

[6] Amálio, N., F. Polack and S. Stepney, *Formal proof from UML models*, in: *ICFEM04: Formal Methods and Software Engineering*, LNCS **3308** (2004), pp. 418–433.

[7] Amálio, N., F. Polack and S. Stepney, *An object-oriented structuring for Z based on views*, in: *ZB 2005*, LNCS **3455** (2005), pp. 262–278.

[8] Amálio, N., F. Polack and S. Stepney, *A formal template language enabling metaproof*, in: *FM 2006: Formal Methods*, LNCS **4085** (2006), pp. 252–267.

[9] Amálio, N., F. Polack and S. Stepney, *Frameworks based on templates for rigorous model-driven development*, in: *IFM 2005 Doctoral Symposium*, ENTCS **191** (2007), pp. 3–23.

[10] Amálio, N., F. Polack and J. Zhang, *Autonomous objects and bottom-up composition in ZOO applied to a case study of biological reactivity*, in: *ABZ2008*, LNCS **5238** (2008), pp. 323–336.

[11] Bast, W., A. Kleppe and J. Warmer, "MDA explained: the model driven architecture: Practice and promise," Addison-Wesley, 2003.

[12] Bowen, J. P. and M. G. Hinchey, *Seven more myths of formal methods*, IEEE Software **12** (1995), pp. 34–41.

[13] Broda, K., D. Roe and A. Russo, *Mapping UML models incorporating OCL constraints into Object-Z*, Technical report, Imperial College London (2003).

[14] Carrington, D. and S.-K. Kim, *Using integrated metamodeling to define OO design patterns with Object-Z and UML*, in: *Asia-Pacific Software Engineering Conference* (2004), pp. 528–537.

[15] Carrington, D. and S.-K. Kim, *A tool for a formal pattern modeling language*, in: *ICFEM*, LNCS **4260** (2006), pp. 568–587.

[16] Davies, J. and J. Woodcock, "Using Z: Specification, Refinement, and Proof," Prentice-Hall, 1996.

[17] Docker, T. W. G., R. B. France and L. T. Semmens, *Integrated structured analysis and formal specification techniques*, The Computer Journal **35** (1992), pp. 600–610.

[18] Drivalos, N., K. J. Fernandes, D. S. Kolovos and R. F. Paige, *Engineering a DSL for software traceability*, in: *Software Languages Engineering 2008*, LNCS **5452** (2009), pp. 151–167.

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

[20] Gamma, E., R. Helm, R. Johnson and J.Vlissides, "Design Patterns: Elements of Resusable Object-Oriented Software," Addison-Wesley, 1995.

[21] Hall, A., *Seven myths of formal methods*, IEEE Software **7** (1990), pp. 11–19.

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

[23] Kleppe, A. and J. Warmer, *Object-Z to OCL dictionary*, `www.klasse.nl/ocl/oz-ocl-mapping.pdf`.

[24] Kolovos, D. S., "An Extensible Platform for Specification of Integrated Languages for Model Management," Ph.D. thesis, Computer Science, University of York (2008).

[25] Kolovos, D. S., R. F. Paige and F. A. C. Polack, *Epsilon development tools for Eclipse*, in: *Eclipse Summit* (2006).

[26] Kolovos, D. S., R. F. Paige, F. A. C. Polack and L. M. Rose, *The Epsilon Generation Language*, in: *EC-MDA*, LNCS **5095** (2008), pp. 1–16.

[27] Le Charlier, B. and P. Flener, *Specifications are necessarily informal or: Some more myths of formal methods*, The Journal of Systems and Software **40** (1998), pp. 275–296.

[28] Ledang, H. and J. Souquières, *Integration of UML and B specification techniques: Systematic transformation from OCL expressions into B*, in: *APSEC '02* (2002), p. 495.

[29] Schmidt, D. C., *Model-driven engineering*, IEEE Computer (39), pp. 25–31.

[30] Smith, G., "The Object-Z Specification Language," Kluwer, 2000.

[31] Williams, J. R., "AUtoZ: Automatic formalisation of UML to Z," MEng thesis, Computer Science, University of York (2009), `www.jamesrobertwilliams.co.uk/publications/WilliamsMEng.pdf`.

Model Checking of Component Protocol Conformance – Optimizations by Reducing False Negatives

Andreas Both and Wolf Zimmermann [1]

Institute of Computer Science
University of Halle
Halle (Saale), Germany

Abstract

In previous work we defined a conservative representation, capturing the behavior of an abstraction of a component-based system and a protocol (based on interactions) to be verified. We have achieved to modelling unbound concurrency and unbound recursion within this abstraction. It turned out that the protocol checking problem is undecidable. Therefore an overapproximation of this protocol checking problem is used. This overapproximation is checked for counterexamples. Due to the overapproximation, some of these counterexamples are spurious, i. e. they cannot occur. In this work we introduce an a priori approach to avoid spurious counterexamples. More concrete, when searching for a counterexample in the overapproximation we cut branches in the search space that will definitely lead to spurious counterexamples.

Keywords: verification, protocol conformance, component-based systems, components, model checking, static analysis, process rewrite systems

1 Motivation

Developing software contains nowadays a big share of reusing previously developed software called components. Often these components were developed a long time ago by different developers, third party companies, in different programming languages, supplied as binary code, or as distributed components (e. g., web services). Thus, there is a big interest in components which can be composed to reliable systems. We support this with a verification process [2]. Here, we will improve the verification. This reduces the additional costs of the verification and raises the quality of the verification results.

Creating reliable stateful components is not easy, because the developer has to prevent and catch every possible error, which might possibly be caused by a different sequence of interactions with this component. We bypass this problem by allowing the developer to define sequences of allowed interactions (with the

[1] Email: andreas.both@informatik.uni-halle.de and wolf.zimmermann@informatik.uni-halle.de

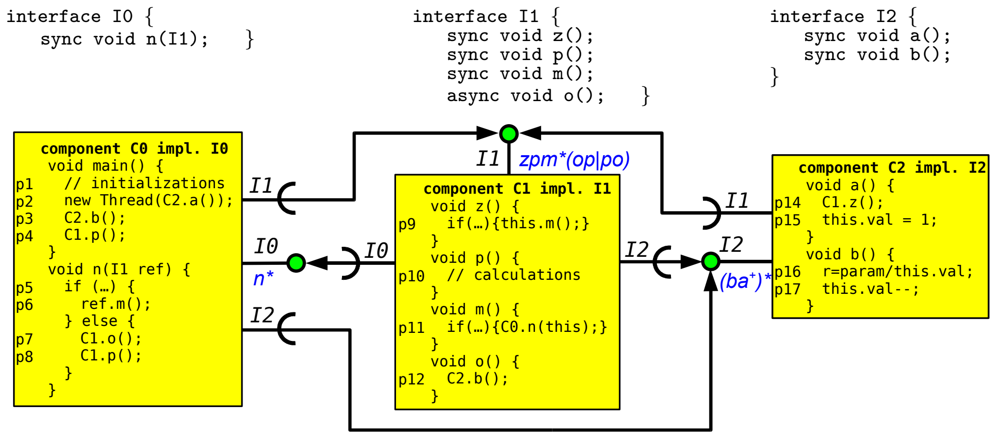
considered component) called protocol. The user of a component is responsible for the protocol being obeyed (protocol conformance). A tool should support the user of components to verify the protocol conformance. Hence, an error caused by an unexpected interaction can be discovered statically. Moreover, only the abstractions of the used components are required. It is not necessary to know the source code. This ensures a better (re)usability of previously developed components.

In [3,2] we have shown how we can use Process Rewrite Systems (PRS) [14] to capture the behavior of source code conservatively. PRS unify Petri nets and Pushdown Automata. Hence they allow recursion and parallelism without restrictions on the calling depth of procedures and the number of concurrent threads, respectively. In the rest of this article, we call this *unbound recursion and unbound parallelism*. Moreover, in these works we defined a process implementing the static verification of the interactions of applications.

It turned out that the protocol conformance checking problem is undecidable [3]. Therefore we overapproximated the protocol conformance checking by representing at least the behavior of the forbidden interactions and the behavior of the considered program as a PRS, named *Combined Abstraction*. If there is no derivation from the initial to a final state in the Combined Abstraction, then there is no forbidden interaction according to the protocol, i. e., the protocol conformance is guaranteed. However, if such a derivation is found, it might be a counterexample. Due to the undecidability of the protocol conformance checking problem, this counterexample does not necessarily lead to a forbidden interaction. We call such counterexamples *spurious*. In a previous work [4] we defined a counterexample-guided abstraction refinement loop (CEGAR-loop) improving the verification speed. For this approach we took the advantages of the PRS-hierarchy, which classifies the PRS by the allowed operators. Since a counterexample is checked after it has been found, we call such approaches *a posteriori*. Because of the fact that all counterexamples of a conservative approximation have to be verified in the real software (manually or tool supported), every spurious counterexample causes an expensive checking in detail. Thus, a large number of spurious counterexamples reduces the practical applicability of a verification.

In this work, we improve the search for counterexamples by cutting off branches in the search tree that definitely lead to spurious counterexamples. The problem of finding a derivation from an initial to final state is at least EXPSPACE-hard (because of the subsumed Petri net, cf. [11]).

The paper is organized as follows. In Section 2 we describe the protocol conformance problem, introduce PRS, and we shortly review our verification process. Also we further discuss the idea behind the construction of the Combined Abstraction. In Section 3 we analyze reasons for spurious counterexamples, while in Section 4 we show how to avoid these spurious counterexamples by using a specialized search strategy. Moreover, we present an algorithm implementing this search. Thereby branches are cut off by precomputing infeasible paths. The paper finishes with a consideration of the related work in Section 5 and the conclusion and future work in Section 6.



```
interface I0 {
   sync void n(I1);   }
```

```
interface I1 {
   sync void z();
   sync void p();
   sync void m();
   async void o();   }
```

```
interface I2 {
   sync void a();
   sync void b();
}
```

```
   component C0 impl. I0
    void main() {
p1    // initializations
p2    new Thread(C2.a());
p3    C2.b();
p4    C1.p();
    }
    void n(I1 ref) {
p5    if (…) {
p6      ref.m();
      } else {
p7      C1.o();
p8      C1.p();
      }
    }
```

```
   component C1 impl. I1
    void z() {
p9    if(…){this.m();}
    }
    void p() {
p10   // calculations
    }
    void m() {
p11   if(…){C0.n(this);}
    }
    void o() {
p12   C2.b();
    }
```

```
   component C2 impl. I2
    void a() {
p14   C1.z();
p15   this.val = 1;
    }
    void b() {
p16   r=param/this.val;
p17   this.val--;
    }
```

Calculations influencing only the dataflow and result types are omitted. Relevant program points are marked with p_i. Synchronous remote method calls are performed blocking, asynchronous remote method calls are performed non-blocking.

Fig. 1. Example with three interfaces and derived components.

2 Foundations

2.1 Protocols

A *protocol* describes the allowed use of all callable operations (cf. interfaces) of a single component or instance [2], respectively. It can be used to verify (incoming remote) invocations dynamically, and also to verify statically, if the component is always used in the manner specified by the protocol. In this work, we consider static verification. Among other scopes of applications, protocols can be used to avoid uncaught exceptions (cf. [3]) during the execution of a component or to obey business rules (cf. [2]).

Creating and verifying service protocols can help to ensure the restrictions of business rules. For example, we use an SSO-component [3] with the following actions: register and sign in (action a), sign in (b), optionally change password (c), logout (d). The component could have the following protocol (business rule) formulated as regular expression: $((a|b)c^*d)^*$. This protocol should be obeyed by every caller of the component. We will check automatically, using the mentioned SSO-component protocol, whether an application (assembled from components) obeys the defined constraints.

In accordance with other works [22,19,7] we use finite state machines (short: FSM) to represent the protocol P_C of a component C. The FSM $P_C \hat{=} (Q_{P_C}, \Sigma_{P_C}, \rightarrow_{P_C}, I_{P_C}, F_{P_C})$ is defined as usual, i. e., Q_{P_C} is a finite set of states, Σ_{P_C} is a finite set of atomic actions, $\rightarrow_{P_C} \subseteq Q_{P_C} \times \Sigma_{P_C} \times Q_{P_C}$ is a finite set of transition rules, $I_{P_C} \in Q_{P_C}$ is the initial state, $F_{P_C} \subseteq Q_{P_C}$ is the set of final states.

The protocol we use here contains only the callable operations (not the abstraction of the component as in e. g., [18]). Thus the protocol can be fixed, while the component implementation is exchanged.

Figure 1 shows an example of a component system implementing different in-

[2] We use the term component for simplification.

[3] Single Sign On. A component which provides the functionality of a login/logout/session management, so different applications can use this mechanism to verify a user.

$((p_2||((p_6||p_3).p_7))||((p_3||p_5).p_9.p_4)).p_1$

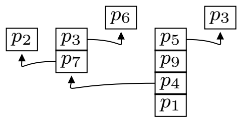


Fig. 2. Example of process-algebraic expression and corresponding cactus stack.

terfaces. The protocols of the interfaces are shown as regular expressions. We will consider only the protocol of the component `C2`[4]. This component might crash on a division by zero exception, if it is called e. g., with the sequence $a\,b\,b$. If the protocol $(ba^+)^*$ is obeyed, this crash can be avoided.

The *use of a component* C in a system S assembled from components is the set of all possible sequences of method invocations to C (described in the following subsection). Thus, this can also be modeled as a language $L(\Pi_{S,C})$. Hence the protocol conformance checking is equivalent to check whether $L(\Pi_{S,C}) \subseteq L(P_C)$, where $L(P_C)$ is a regular language defined by the protocol P_C of the component C.

2.2 Process Rewrite Systems (PRS)

Many approaches (e. g., [19,7,16]) model the use of required interfaces in component-based systems by regular languages obtained by finite transducers. In [22] it is shown that this approach leads to false positives if recursion is present. Thus we need a representation for the abstraction of the behavior of the source code that can capture every control-flow path conservatively to find all possible errors.

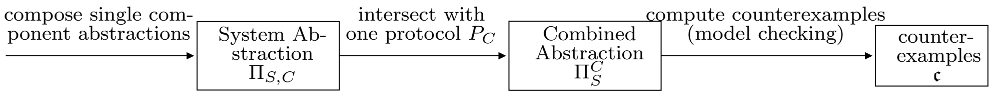
The natural execution model for capturing unbound recursion and unbound concurrency uses states that are represented as a cactus stack (tree of stacks). If a procedure call in a process is executed, a stack frame is pushed on a stack. If a new parallel process is started, a new stack grows for this process (like in a saguaro stack, cf. Figure 2). Hence an execution transforms a cactus stack into a cactus stack.

A cactus stack naturally represents a process-algebraic expression (and vice versa). If a procedure frame p_1 is called on a stack p, this is represented by $p_1.p$. If a fork to a process p_2 on a stack p with the frame p_1 happens, this is represented by $(p_1||p_2).p$. Thus, we can model state transformations by transforming process-algebraic expressions into process-algebraic expressions. Process Rewrite Systems [14] are a descriptive technique for such transformations. Let $PEX(Q)$ be the set of process-algebraic expressions over a finite set Q of atomic processes and the binary operators "." (for sequential composition, associative) and "||" (for parallel composition, commutative and associative).

A *Process Rewrite System* is defined as $\Pi \hat{=} (Q, \Sigma, I, \rightarrow, F)$, where Q is a finite set of atomic processes, Σ is a finite alphabet over actions, $I \in Q$ is the initial process, $\rightarrow \subseteq PEX(Q) \times (\Sigma \uplus \{\lambda\}) \times PEX(Q)$ is a set of process rewrite rules, $F \subseteq PEX(Q)$ is a finite set of final processes. The process ε denotes the empty process. It is the identity on "||" and the left identity on ".".

We introduce a special action $\lambda \notin \Sigma$, denoting *no relevant interaction* or *empty word*. The process rewrite rules define a derivation relation

[4] All other protocols can be checked using the similar way.



compose single component abstractions
System Abstraction $\Pi_{S,C}$
intersect with one protocol P_C
Combined Abstraction Π_S^C
compute counterexamples (model checking)
counterexamples $\mathfrak{c}$

Fig. 3. Verification process.

$p_1 \xrightarrow{\lambda} p_3||p_2$, $p_2 \xrightarrow{a} p_{14}.p_2{}'$, $p_2{}' \xrightarrow{\lambda} \varepsilon$, $p_5 \xrightarrow{\lambda} p_6$,
$p_3 \xrightarrow{b} p_{16}.p_4$, $p_4 \xrightarrow{p} p_{10}.\varepsilon$, $p_5 \xrightarrow{\lambda} p_7$, $p_6 \xrightarrow{m} p_{11}.\varepsilon$,
$p_7 \xrightarrow{o} p_{12}||p_8$, $p_8 \xrightarrow{p} p_{10}.\varepsilon$, $p_9 \xrightarrow{m} p_{11}.\varepsilon$, $p_9 \xrightarrow{\lambda} \varepsilon$,
$p_{10} \xrightarrow{\lambda} \varepsilon$, $p_{11} \xrightarrow{n} p_5.\varepsilon$, $p_{11} \xrightarrow{\lambda} \varepsilon$, $p_{12} \xrightarrow{b} p_{14}.\varepsilon$,
$p_{14} \xrightarrow{z} p_9.p_{15}$, $p_{15} \xrightarrow{\lambda} \varepsilon$, $p_{16} \xrightarrow{\lambda} p_{17}$, $p_{17} \xrightarrow{\lambda} \varepsilon$

(a) Constructed system abstraction Π_S of the example in Figure 1.

$p_1 \xrightarrow{\lambda} p_3||p_2$, $p_2 \xrightarrow{a} p_{14}.p_2{}'$, $p_2{}' \xrightarrow{\lambda} \varepsilon$, $p_5 \xrightarrow{\lambda} p_6$,
$p_3 \xrightarrow{b} p_{16}.p_4$, $p_4 \xrightarrow{\lambda} p_{10}.\varepsilon$, $p_5 \xrightarrow{\lambda} p_7$, $p_6 \xrightarrow{\lambda} p_{11}.\varepsilon$,
$p_7 \xrightarrow{\lambda} p_{12}||p_8$, $p_8 \xrightarrow{\lambda} p_{10}.\varepsilon$, $p_9 \xrightarrow{\lambda} p_{11}.\varepsilon$, $p_9 \xrightarrow{\lambda} \varepsilon$,
$p_{10} \xrightarrow{\lambda} \varepsilon$, $p_{11} \xrightarrow{\lambda} p_5.\varepsilon$, $p_{11} \xrightarrow{\lambda} \varepsilon$, $p_{12} \xrightarrow{b} p_{14}.\varepsilon$,
$p_{14} \xrightarrow{\lambda} p_9.p_{15}$, $p_{15} \xrightarrow{\lambda} \varepsilon$, $p_{16} \xrightarrow{\lambda} p_{17}$, $p_{17} \xrightarrow{\lambda} \varepsilon$

(b) $\Pi_{S,\texttt{C2}}$, considers only the interactions of component `C2` in Figure 4a.

Fig. 4. Example: Constructed system abstraction Π_S of the example in Figure 1 and $\Pi_{S,\texttt{C2}}$.

$\Rightarrow \in PEX(Q) \times \Sigma^* \times PEX(Q)$ by the following inference rules $(a \in \Sigma, x \in \Sigma^*)$:

$$\frac{(t_1 \xrightarrow{a} t_2) \in \Pi}{t_1 \overset{a}{\Rightarrow} t_2},\quad \frac{t_1 \overset{a}{\Rightarrow} t_2}{t_1.t_3 \overset{a}{\Rightarrow} t_2.t_3},\quad \frac{t_1 \overset{a}{\Rightarrow} t_2}{t_1||t_3 \overset{a}{\Rightarrow} t_2||t_3},\quad \frac{t_1 \overset{a}{\Rightarrow} t_2}{t_3||t_1 \overset{a}{\Rightarrow} t_3||t_2},\quad \frac{t_1 \overset{x}{\Rightarrow} t_2 \quad t_2 \overset{a}{\Rightarrow} t_3}{t_1 \overset{x\,a}{\Longrightarrow} t_3}$$

The second rule formalizes that only the frames on the top of the stacks of a cactus stack can be considered for transformations. The third and forth rule specify that any stack in a cactus stack can be considered (i. e., each of the processes currently being executed can be selected for state transformations). Thus, these two rules model interleaving semantics.

$L(\Pi) \hat{=} \{w : \exists f \in F | I \overset{w}{\Rightarrow} f\}$ is the *language accepted* by Π.

Pushdown automata are represented by the class of PRS, which allow no parallel operator. In contrast Petri nets are represented by the class of PRS, which allow no sequential operator. Thus PRS allowing the use of both operators unify the behavior of pushdown automata and Petri nets.

Mayr shows in [14] that the rules of any PRS can be transformed into a *normal form*, i. e., the LHS and the RHS has one of the forms t_1, $t_1.t_2$ or $t_1||t_2$, where t_1 and t_2 are atomic processes. We assume in this work that every PRS has been transformed into a PRS in normal form, and that the LHS contains at most the $||$-operator. In [14] this restriction is called Process Algebra Nets (PAN). It can handle unbound recursion and unbound parallelism including synchronization. In this work, we use PAN as representation of the behavior of the considered components.

The verification process (cf. Figure 3) creates an abstraction Π_S of the system behavior. Each possible interaction between components is represented by Π_S. In Figure 4a we see the abstraction of the system composed by the abstractions of the components `C1`, `C2` and `C3` (cf. Figure 1). The transformation takes the control-flow into account. Thus every execution path is captured by the abstraction. The abstraction can be made compositional by adding a variation of PRS to the components description. The process of obtaining this description is fully automatic. For details we refer to [3,2]. For checking protocol conformance to a component C, only interactions to C of Π_S are considered. This can be achieved by replacing all interactions by λ, that do not belong to the interfaces of C by λ, cf. Figure 4b. We denote this PRS by $\Pi_{S,C}$.

Now, a PRS Π_S^C is constructed such that $L(\Pi_S^C) \supseteq \overline{L(P_C)} \cap L(\Pi_{S,C})$, i. e., Π_S^C contains all sequences of interactions that are forbidden [5] by the protocol P_C of one

[5] These sequences are represented by the inverted protocol, i. e., the inverted FSM $\overline{P_C}$ accepting the complement $\overline{L(P_C)} \hat{=} \Sigma^* \setminus L(P_C)$.

$$\begin{array}{rlll}
\mathcal{R}^1_C = \{(v,p,v'') \xrightarrow{a} (v',p',v''): & (v \xrightarrow{a}_{P_C} v') \wedge (p \xrightarrow{a}_{\Pi_{S,C}} p') & \text{with} & v'' \in Q_{P_C}\} \\
\mathcal{R}^\lambda_C = \{(v,p,v') \xrightarrow{\lambda} (v,p',v'): & (p \xrightarrow{\lambda}_{\Pi_{S,C}} p'); & \text{with} & v,v' \in Q_{P_C}\} \\
\mathcal{R}^1_{Seq} = \{(v,p,v''') \xrightarrow{a} (v',p',v'').(v'',p'',v'''): & (v \xrightarrow{a}_{P_C} v') \wedge (p \xrightarrow{a}_{\Pi_{S,C}} p'.p'') & \text{with} & v'',v''' \in Q_{P_C}\} \\
\mathcal{R}^\lambda_{Seq} = \{(v,p,v'') \xrightarrow{\lambda} (v,p',v').(v',p'',v''): & (p \xrightarrow{\lambda}_{\Pi_{S,C}} p'.p'') & \text{with} & v',v'' \in Q_{P_C}\} \\
\mathcal{R}^1_{PFork} = \{(v,p,v'') \xrightarrow{a} (v',p',v'')||(v',p'',v''): & (v \xrightarrow{a}_{P_C} v') \wedge (p \xrightarrow{a}_{\Pi_{S,C}} p'||p'') & \text{with} & v'' \in Q_{P_C}\} \\
\mathcal{R}^\lambda_{PFork} = \{(v,p,v') \xrightarrow{\lambda} (v,p',v')||(v,p'',v'): & (p \xrightarrow{\lambda}_{\Pi_{S,C}} p'||p'') & \text{with} & v,v' \in Q_{P_C}\} \\
\mathcal{R}^1_{PSync} = \{(v,p,v'')||(v,p',v'') \xrightarrow{a} (v',p'',v''): & (v \xrightarrow{a}_{P_C} v') \wedge (p||p' \xrightarrow{a}_{\Pi_{S,C}} p'') & \text{with} & v'' \in Q_{P_C}\} \\
\mathcal{R}^\lambda_{PSync} = \{(v,p,v')||(v,p',v') \xrightarrow{\lambda} (v,p'',v'): & (p||p' \xrightarrow{\lambda}_{\Pi_{S,C}} p'') & \text{with} & v,v' \in Q_{P_C}\} \\
\mathcal{R}^0 = \{(v,p,v'') \xrightarrow{\lambda} (v',p,v''): & (v \xrightarrow{a}_{P_C} v') & \text{with} & p \in Q_{\Pi_{S,C}}\} \\
\mathcal{R}^\varepsilon = \{(v,\varepsilon,v) \xrightarrow{\lambda} \varepsilon: & & \text{with} & v \in Q_{P_C}\}
\end{array}$$

Fig. 5. Rules for creating the Combined Abstraction Π^C_S.

component C (i. e., $\overline{P_C}$) but nevertheless exists in the program S. It is not possible to construct a PAN such that $L(\Pi^C_S) = \overline{L(P_C)} \cap L(\Pi_{S,C})$ since it is undecidable whether $L(\Pi_{S,C}) \subseteq \overline{L(P_C)}$ [3], but it is decidable whether $L(\Pi^C_S) = \emptyset$ [14]. The *Combined Abstraction* Π^C_S is also a PRS in the same class of the PRS-hierarchy (cf. [14]) as Π_S. Thus every error can be found, because the Combined Abstraction Π^C_S contains all paths from the initial program point to the final state: $I \overset{*}{\Rightarrow} \varepsilon$.

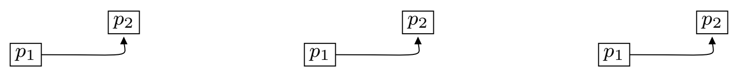
The transition rules $\rightarrow_{\Pi^C_S} = \mathcal{R}^1_C \cup \mathcal{R}^\lambda_C \cup \mathcal{R}^1_{Seq} \cup \mathcal{R}^\lambda_{Seq} \cup \mathcal{R}^1_{PFork} \cup \mathcal{R}^\lambda_{PFork} \cup \mathcal{R}^1_{PSync} \cup \mathcal{R}^\lambda_{PSync} \cup \mathcal{R}^0 \cup \mathcal{R}^\varepsilon$ of the Combined Abstraction Π^C_S are computed by using the directives in Figure 5. The construction is similar to the intersection of PDA and FSM (cf. [9]). All process constants are triples (v_i, p_k, v_j) where $v_i, v_j \in Q_{P_C}$ and $p_k \in Q_S$. A triple (v_i, p_k, v_j) encodes the situation, where p_k should be rewritten to the empty process $p_k \overset{w}{\Rightarrow}_{\Pi_S} \varepsilon$, while the state v_i of the FSM P_C is transformed into v_j accepting the same word w: $v_i \overset{w}{\Rightarrow}_{P_C} v_j$. A constant (v_j, ε, v_j) is equivalent to the empty word, because the targeted protocol state is reached and process constants have been eliminated (cf. $\mathcal{R}^\varepsilon$ in Figure 5). Technical details of the construction of the Combined Abstraction Π^C_S can be found in [2].

Finally, the Π^C_S is model checked. The result contains counterexamples $\mathfrak{c}$. These derivation paths $\mathfrak{c} : I_0 \overset{w}{\Rightarrow} \varepsilon$ of Π^C_S are equivalent to possible sequences of interactions w. In Figure 7 a counterexample $\mathfrak{c}_0$ is constructed while model checking a given Combined Abstraction. In Step 1 and Step 2 of the derivation a rule is used included in $\mathcal{R}^1_{PFork}$ and $\mathcal{R}^1_C$, respectively. The resulting term $(v_1, p_1, v_2)||(v_2, \varepsilon, v_2)$ contains two subterms with a different first entry of the triple, but the protocol can only be in one state (because it is defined globally). To synchronize these entries in step 3 the rule $((v_1, p_1, v_2) \xrightarrow{\lambda} (v_2, p_1, v_2)) \in \mathcal{R}^0$ is used.

The transition rules of the sets $\mathcal{R}^1_C$, $\mathcal{R}^1_{Seq}$, $\mathcal{R}^1_{PFork}$ and $\mathcal{R}^1_{PSync}$ are called *action rules*, because the protocol states and the process constants are changed while performing an interaction between components. On the other hand $\mathcal{R}^\lambda_C$, $\mathcal{R}^\lambda_{Seq}$, $\mathcal{R}^\lambda_{PFork}$, $\mathcal{R}^\lambda_{PSync}$ perform no protocol action. They are called *λ-rules*. The set $\mathcal{R}^0$ contains rules, which we call *sleep rules*. These are needed to anticipate the interactions performed in a parallel thread (interleaving semantics) and to synchronize the protocol states of the transition rules.

process-algebraic expressions: $(v_1,p_1,v_3)||\underline{(v_1,p_2,v_3)} \overset{b}{\Rightarrow} \underline{(v_1,p_1,v_3)}||(v_2,p_3,v_3) \overset{\lambda}{\Rightarrow} (v_2,p_1,v_3)||(v_2,p_3,v_3)$

cactus stacks:



protocol states: $v_1 = v_1$ $\quad v_1 \neq v_2$ $\quad v_1 = v_1$

With the transition rules $(v_1,p_2,v_3) \xrightarrow{b} (v_2,p_3,v_3) \in \mathcal{R}^1_C$ and $(v_1,p_1,v_3) \xrightarrow{\lambda} (v_2,p_1,v_3) \in \mathcal{R}^0$.

Fig. 6. Example: Synchronization of cactus stack in Combined Abstraction.

Given system abstraction (initial constant p_0): $p_0 \xrightarrow{a} p_1||p_1 \quad p_1 \xrightarrow{b} \varepsilon \quad p_1 \xrightarrow{\lambda} \varepsilon$

Given inverted protocol (initial state v_0, final state v_2): $v_0 \xrightarrow{a} v_1 \quad v_1 \xrightarrow{b} v_2$

Generated Combined Abstraction (initial constant (v_0,p_0,v_2), only rules required for counterexamples):

$$(v_0,p_0,v_2) \xrightarrow{a} (v_1,p_1,v_2)||(v_1,p_1,v_2) \quad (v_1,p_1,v_2) \xrightarrow{b} (v_2,\varepsilon,v_2) \quad (v_1,p_1,v_2) \xrightarrow{\lambda} (v_2,p_1,v_2)$$
$$(v_1,p_1,v_2) \xrightarrow{\lambda} (v_1,\varepsilon,v_2) \quad (v_1,\varepsilon,v_2) \xrightarrow{\lambda} (v_2,\varepsilon,v_2) \quad (v_2,p_1,v_2) \xrightarrow{\lambda} (v_2,\varepsilon,v_2)$$
$$(v_0,p_0,v_2) \xrightarrow{\lambda} (v_1,p_0,v_2) \quad (v_1,p_0,v_2) \xrightarrow{\lambda} (v_2,p_0,v_2)$$

Computed counterexamples:

$$\mathfrak{c}_0 \hat{=} \underline{(v_0,p_0,v_2)} \overset{a}{\Rightarrow} (v_1,p_1,v_2)||\underline{(v_1,p_1,v_2)} \overset{b}{\Rightarrow} \underline{(v_1,p_1,v_2)}||(v_2,\varepsilon,v_2) \overset{\lambda}{\Rightarrow} \underline{(v_2,p_1,v_2)}||(v_2,\varepsilon,v_2)$$
$$\overset{\lambda}{\Rightarrow} (v_2,\varepsilon,v_2)||\underline{(v_2,\varepsilon,v_2)} \overset{\lambda}{\Rightarrow} \underline{(v_2,\varepsilon,v_2)} \overset{\lambda}{\Rightarrow} \varepsilon$$
$$\mathfrak{c}_1 \hat{=} \underline{(v_0,p_0,v_2)} \overset{a}{\Rightarrow} (v_1,p_1,v_2)||\underline{(v_1,p_1,v_2)} \overset{\lambda}{\Rightarrow} (v_1,p_1,v_2)||\underline{(v_1,\varepsilon,v_2)} \overset{\lambda}{\Rightarrow} \underline{(v_1,p_1,v_2)}||(v_2,\varepsilon,v_2)$$
$$\overset{\lambda}{\Rightarrow} \underline{(v_1,\varepsilon,v_2)}||(v_2,\varepsilon,v_2) \overset{\lambda}{\Rightarrow} (v_2,\varepsilon,v_2)||\underline{(v_2,\varepsilon,v_2)} \overset{\lambda}{\Rightarrow} \underline{(v_2,\varepsilon,v_2)} \overset{\lambda}{\Rightarrow} \varepsilon$$
$$\mathfrak{c}_2 \hat{=} \underline{(v_0,p_0,v_2)} \overset{a}{\Rightarrow} (v_1,p_1,v_2)||\underline{(v_1,p_1,v_2)} \overset{b}{\Rightarrow} (v_1,p_1,v_2)||\underline{(v_2,\varepsilon,v_2)} \overset{\lambda}{\Rightarrow} \underline{(v_1,p_1,v_2)} \overset{b}{\Rightarrow} \underline{(v_2,\varepsilon,v_2)} \overset{\lambda}{\Rightarrow} \varepsilon$$

Fig. 7. Example: Spurious False Negatives.

An example can be seen in Figure 6. There we represent a part of the derivation $\mathfrak{c}_0$ in Figure 7. The first applied transition rule (action rule) changes the protocol state in the second triple only. Thus two protocol states v_1 and v_2 exist in the term. This is not possible, because the protocol state is global. To translate the protocol state v_1 into v_2 a sleep rule is applied in the second step.

However, some of the counterexamples might not occur (e. g., because of over-approximation). We call these counterexamples *false negatives*. False negatives cannot generally be avoided, since the protocol conformance checking problem is undecidable, as soon as unbound parallelism and unbound recursion are taken into account [3].

Table 1 summarizes the notations in this paper.

3 Structure of False Negatives

While solving the model checking problem for protocol conformance checking, a big issue is to ensure that not too many false negatives are created. This is important, because every reported counterexample has to be validated in the verified source code because of the conservative approximation of the behavior. This is an expensive task, thus the effort should be reduced as much as possible.

False negatives can be classified into the following categories:

- *Real false negatives*: Because the source code abstractions are created without any data flow or detailed control flow analysis, it is possible that a trace could be

Notation	Description
P_C	protocol of a component C (represented as FSM or regular expression)
Π_S	system abstraction of a full component-based system
$\Pi_{S,C}$	system abstraction, considering only the interactions of a component C
Π_S^C	Combined Abstraction of a component C
$PEX(Q)$	set of process-algebraic expressions
s, t	process-algebraic terms, $s, t \in PEX(Q)$
v	states of an (inverted) protocol, $v \in P_C$
a, b, λ	interactions in the considered system, $a, b \in \Sigma$, no interaction $\lambda \notin \Sigma$
w, x, y, λ	sequences of interactions, $w, x, y \in \Sigma^*$, empty sequence of interactions λ
p, ε	atomic processes, $p \in Q$, empty process, $\varepsilon \in Q$

Table 1
Notations used in this paper.

contained in the component abstraction that does not correspond to any execution path of the implementation.

- *Spurious false negatives*: We construct an approximated intersection of the languages described by the system and by the considered inverted protocol. Therefore it is possible to get false negatives.

If a precise behavior abstraction is assumed, spurious counterexamples caused by an inappropriate number of interactions are likely to occur. While computing counterexamples it is possible that these contain too many or too few interactions. The reachability analysis for PRS may compute counterexamples, e. g., in Figure 7:

(i) The counterexample $\mathfrak{c}_1$ has too few interactions. The sleep rules are used for reaching the final state of the protocol without interactions. Due to state changes in the system abstraction without interactions, the final state ε can be reached.

(ii) The counterexample $\mathfrak{c}_2$ has too many interactions. Consecutive applications of action rules lead to a process-algebraic expression that can be reduced to ε if additional action rules are applied.

Remark: The full setting of the example in Figure 7 can be found in the appendix.

4 Reducing the Number of False Negatives

In this section we will describe an extension of the reachability analysis of PAN, while using the information about interactions of the considered software encoded in the Combined Abstraction. This will eliminate the false negatives caused by an inappropriate number of interactions.

4.1 Basic Idea

A counterexample $\mathfrak{c}$ is a sequence of interactions w used to create a derivation from the initial constant I to the final constant ε: $I \overset{w}{\Rightarrow} \varepsilon$. This sequence has to be contained in the language of the inverted protocol $\overline{P_C}$ (inverted protocol FSM), thus we can check $w \in \overline{L(P_C)}$. Using this approach we can eliminate the counterexamples $\mathfrak{c}_1$ and $\mathfrak{c}_2$ of Figure 7.

However, this check requires the explicit construction of a spurious counterexample. Here, our goal is to avoid this explicit construction. First, this leads to a

Given system abstraction rules (initial constant p_0):	$p_0 \xrightarrow{a} p_1 \| p_1$,	$p_1 \xrightarrow{b} \varepsilon$,	$p_1 \xrightarrow{\lambda} \varepsilon$
Given inverted protocol (initial state v_0, final state v_2):	$v_0 \xrightarrow{a} v_1$,	$v_1 \xrightarrow{b} v_2$,	$v_2 \xrightarrow{b} v_2$
A computable counterexample:	$\mathfrak{c}_2$ of Figure 7		

Fig. 8. Adapted example of Figure 7: language inclusion results in an impractical output (impractical, because it is difficult to capture the invalidity of the computed counterexample).

more efficient search for counterexamples. Second, the above might give false hints to how counterexamples can be derived. The latter is demonstrated by the example of Figure 8: The counterexample $\mathfrak{c}_2$ can be constructed in the same way as in Figure 7. However, the language inclusion check does not fail, because the word w of $\mathfrak{c}_2$ is contained in the language of the inverted protocol: $w = abb \in L(abb^*)$. A closer look shows, that $\mathfrak{c}_2$ contains two transitions from the protocol state v_1 to v_2 using the interaction b. As we know this is not possible, because only one interaction is allowed at one point in time (interleaving semantics). This situation is confusing, since developers might conclude erroneously from $\mathfrak{c}_2$, that it is a false negative. Hence this approach looses some pieces of information about the generated counterexamples.

The idea is now, when a protocol state change happens in one of the terms where a rule is applicable (head terms), then we first change the protocol state in the other head terms before we continue the search. We call this heuristic the Round-robin reachability. The derivation of $\mathfrak{c}_0$ in Figure 7 follows this strategy.

4.2 The Round-robin Reachability Algorithm

In this section we prove that this heuristic does not exclude non-spurious counterexamples. Figure 9 shows a backtracking algorithm implementing this strategy.

It uses the following notations:

- As can be seen from the inference rules defining the derivations, the derivation relation in a term $(p_1.u_1||\ldots||p_n.u_n).u_{n+1}$, where $n > 0$, $u_1, \ldots, u_{n+1} \in PEX(Q)$, $p_1, \ldots, p_n \in Q$ is only applied to one of the terms p_i. Formally, we define the notion of the *set of heads* $H(t)$ for a process-algebraic term $t \in PEX(Q)$ inductively as follows: The multiset of (atomic) heads $H(t)$ and the multiset of synchronization possibilities $S(t)$ of a PRS term t is defined as follows:

$$H(p) \hat{=} \{\{p\}\} \qquad S(t) \hat{=} \{\{p_i||p_j : p_i, p_j \in H(t), p_i \neq p_j \vee \xi_{A(t)}(p_i) \geq 2\}\}$$
$$H(t_1.t_2) \hat{=} H(t_1)$$
$$H(t_1||t_2) \hat{=} H(t_1)||H(t_2)$$

Where $\xi_{A(t)}$ denotes the number of elements in a multiset A and $p, p_1, p_2 \in Q, t_1, t_2 \in PEX(Q)$

Intuitively the heads are the top stack frames in a cactus stack. E. g., the heads in Figure 2 are $\{\{p_2, p_3, p_3, p_5, p_6\}\}$ and the synchronization possibilities are $\{\{p_2||p_3, p_2||p_3, p_2||p_5, p_2||p_6, p_3||p_3, p_3||p_5, p_3||p_6, p_3||p_5, p_3||p_6, p_5||p_6\}\}$. As we can see the synchronization properties are an overapproximation – e. g., $p_5||p_6$ can not be transformed using PRS rules into the given cactus stack – but this definition satisfies our needs.

Property 1 *If in a PRS* Π *a rule* $\delta : t_1 \xrightarrow{b} t_2$ *is applied to process-algebraic term* t *(i. e.,* $t \overset{b}{\Rightarrow} t'$*), then there is a head* $h \in H(t)$ *or a synchronization possibility* $h \in S(t)$*, s. t.* t' *is obtained from* t *by replacing* h *by* h' *by the same rule* δ*, where*

```
(1)  Algorithm newReach :
(2)  Input:    PRS Term $t$
(3)  Output: Derivation $t \overset{w}{\Rightarrow} \varepsilon$ if it exists, false otherwise
(4)  if $t = \varepsilon$ then return $\varepsilon$;
(5)  let $\mathcal{R}$ be the set of rules applicable to $t$;
(6)  foreach $\delta \in \mathcal{R}$ do
(7)      $t' := t$;
(8)      case $\delta$ of
(9)          $t_1 \xrightarrow{\lambda} t_2$:
(10)             $t' :=$ apply $\delta$ to $t'$;
(11)             if newReach$(t') = t' \overset{w}{\Rightarrow} \varepsilon$
(12)             then return $t \overset{\lambda}{\Rightarrow} t' \overset{w}{\Rightarrow} \varepsilon$ ;
(13)         $t_1 \xrightarrow{a} t_2$:
(14)             choose $h \in H(t') \cup S(t')$
(15)             $t' :=$ apply $\delta$ at $h$ of $t'$ ;
(16)             $d := t \overset{a}{\Rightarrow} t'$;
(17)             foreach $h' \in H(t') \wedge \mathrm{ps}(h') \neq \mathrm{ps}(t_2)$
(18)                 let $\delta'$ be the corresponding sleep rule of $\delta$;
(19)                 $t' :=$ apply $\delta'$ at $h'$ of $t'$
(20)                 $d := d \overset{\lambda}{\Rightarrow} t'$;
(21)             od;
(22)     esac
(23)     if newReach$(t') = t' \overset{w}{\Rightarrow} \varepsilon$ return $d \overset{aw}{\Rightarrow} \varepsilon$;
(24) od;
(25) return false;
```

Fig. 9. Round-robin reachability algorithm in pseudo code.

$h \overset{b}{\Rightarrow} h'$.

Property 1 states that any action rule is either applied to a top frame of a cactus stack or it synchronizes two threads (i. e., it merges two stacks into one).

- *Corresponding sleep rule*: A rule $\delta \in \mathcal{R}^0$ of the considered Combined Abstraction Π_S^C which is created by using a rule $v \xrightarrow{a} v'$ of the protocol P_C is called the sleep rule corresponding to an action rule $t_1 \xrightarrow{a} t_2$.
- A *protocol state of a process-algebraic term* is the set of protocol states. The protocol state of a triple (v, p, v') is v. Formally we define the function $\mathrm{ps}(t)$ for a process-algebraic term $t \in PEX(Q)$ as follows: $\mathrm{ps}(t) = \{v : (v, p, v') \in H(t)\}$.

In the algorithm (cf. Figure 9) it is valid: if an action rule δ is applied to one head, all other heads are rewritten to the corresponding protocol state by using sleep rules corresponding to δ (cf. line 17-21).

At line 14 of the algorithm we choose one head $h \in H(t)$ or one synchronization possibility $h \in S(t)$ of the current term t to apply δ: $h \overset{a}{\Rightarrow} h'$. Following the definition of action rules this means that the protocol state of h could differ from the one of h'. At line 17 we ensure that all other heads are imitating rewriting of the protocol state in h' while applying a corresponding sleep rule.

Thus in every derivation step, if one action rule δ has been applied, then all heads are translated into the same protocol state by applying the corresponding sleep rules of δ to extend the derivation d. Hence the problems of the language inclusion are eliminated:

- It is ensured that only one protocol transition is performed in one derivation step of the Combined Abstraction.
- It is checked in every derivation step, whether such a derivation will create spurious false negatives.

The following theorem states that each non-spurious counterexample can be constructed by the Round-robin reachability.

Theorem 4.1 *Let $\Pi_{S,C}$ be a system abstraction (considering component C), P_C be the inverted protocol of component C, and $w \in \Sigma^*$, such that $t \overset{w}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ for a term $t \in PEX(Q_{\Pi_{S,C}})$ and $v \overset{w}{\Rightarrow}_{P_C} f$ for a protocol state $v \in Q_{P_C}$ and a final protocol state $f \in F_{P_C}$. Then there is a process-algebraic expression $s \in Q_{\Pi_S^C}$ over the Combined Abstraction such that $s \overset{w}{\Rightarrow} \varepsilon$ and $|ps(s)| = 1$, i. e., the protocol states in the heads are equal.*

The proof of Theorem 4.1 requires some technical definitions and lemmas. Let $s \in PEX(Q_{\Pi_S^C})$ be a process-algebraic expression over the Combined Abstraction Π_S^C. The process-algebraic expression $\mathrm{F}(s) \in PEX(Q_{\Pi_{S,C}})$ over the system abstraction $\Pi_{S,C}$ "forgets" the protocol states in s. Formally:

$$\begin{aligned} \mathrm{F}((v,p,v')) &= p & \forall (v,p,v') \in Q_{\Pi_S^C} \\ \mathrm{F}(s.s') &= \mathrm{F}(s).\mathrm{F}(s') & \forall s,s' \in PEX(Q_{\Pi_S^C}) \\ \mathrm{F}(s||s') &= \mathrm{F}(s)||\mathrm{F}(s') & \forall s,s' \in PEX(Q_{\Pi_S^C}) \end{aligned}$$

E. g., $\mathrm{F}((v_1,p_2,v_5)||(v_1,p_3,v_2).(v_2,p_4,v_5)) = p_2||p_3.p_4$. Furthermore let $\mathrm{F}^{-1}(t) = \{s|\mathrm{F}(s) = t\}$.

The following lemma states that "λ-derivations" $t \overset{\lambda}{\Rightarrow}_{\Pi_{S,C}} t'$ in the system abstraction $\Pi_{S,C}$ can be transformed into corresponding λ-derivations $s \overset{\lambda}{\Rightarrow}_{\Pi_S^C} s'$ in the Combined Abstraction Π_S^C:

Lemma 4.2 *Let $t, t' \in PEX(Q_{\Pi_{S,C}})$ such that $t \overset{\lambda}{\Rightarrow}_{\Pi_{S,C}} t'$. Then, for all $s' \in F^{-1}(t')$, there is an $s \in F^{-1}(t)$ such that the following properties are satisfied.*

(i) *$s \overset{\lambda}{\Rightarrow}_{\Pi_S^C} s'$ using only rules $\delta \in \mathcal{R}_C^\lambda \cup \mathcal{R}_{Seq}^\lambda \cup \mathcal{R}_{PFork}^\lambda \cup \mathcal{R}_{PSync}^\lambda$.*

(ii) *$(v,p,v') \in H(s)$ iff there is a $p' \in Q_{\Pi_{S,C}}$ such that either $(v,p',v') \in H(s')$ or $(v,p,v) \in H(s')$.*

<u>Remark:</u> 2. implies that $ps(s) = ps(s')$, i. e., no state change in the protocol happens.

Proof. Straightforward induction on the construction of $t \overset{\lambda}{\Rightarrow}_{\Pi_{S,C}} t'$. $\square$

Lemma 4.3 states that the Round-robin derivation can always be constructed, if there is exactly one protocol state change, i. e., an action rule is applied (cf. line 14-21 in Figure 9).

Lemma 4.3 *Let $t_1, t_2 \in PEX(Q_{\Pi_{S,C}})$ two process-algebraic expressions over the system abstraction $\Pi_{S,C}$ such that $t_1 \overset{a}{\Rightarrow}_{\Pi_{S,C}} t_2$ for an $a \in \Sigma$ by application of a single rule δ. If in the protocol P_C, there is a state transition $v \overset{a}{\rightarrow}_{P_C} v'$, then for any $s_2 \in F^{-1}(t_2)$ satisfying $ps(s_2) = \{v'\}$, there is a $s_1 \in F^{-1}(t_1)$ such that the following properties are satisfied:*

(i) *$ps(s_1) = \{v\}$*

(ii) *$s_1 \overset{a}{\Rightarrow}_{\Pi_S^C} s' \overset{\lambda}{\Rightarrow}_{\Pi_S^C} s_2$ where $s_1 \overset{a}{\Rightarrow}_{\Pi_S^C} s'$ uses a single rule $\delta \in \mathcal{R}_C^1 \cup \mathcal{R}_{Seq}^1 \cup \mathcal{R}_{PFork}^1 \cup$*

$\mathcal{R}^1_{PSync}$ and $s' \overset{\lambda}{\Rightarrow}_{\Pi^C_S} s_2$ only uses rules δ which are corresponding sleep rules of $v \xrightarrow{a}_{P_C} v'$.

Proof. (of Lemma 4.3)
<u>Case 1:</u> $\delta\hat{=}(p \xrightarrow{a}_{\Pi_{S,C}} p')$ or $\delta\hat{=}(p \xrightarrow{a}_{\Pi_{S,C}} p'||p'')$ or $\delta\hat{=}(p||p'' \xrightarrow{a}_{\Pi_{S,C}} p')$. Then, there must be a protocol state $v'' \in Q_{P_C}$ such that $(v', p', v'') \in H(s_2)$. Suppose that $H(s_2) \setminus \{(v', p', v'')\} = \{(v', p_1, v_1), \ldots, (v', p_n, v_n)\}$. Then a derivation $s'_n \overset{\lambda}{\Rightarrow}_{\Pi^C_S} s'_{n-1} \overset{\lambda}{\Rightarrow}_{\Pi^C_S} \ldots \overset{\lambda}{\Rightarrow}_{\Pi^C_S} s'_1 \overset{\lambda}{\Rightarrow}_{\Pi^C_S} s'_0 = s_2$ where s'_{i-1} is the result of the application of the rule $(v, p_i, v_i) \xrightarrow{\lambda}_{\Pi^C_S} (v', p_i, v_i)$ at head (v, p_i, v_i) of term s'_i, $i = n, \ldots, 1$. Thus, only corresponding sleep rules of $v \xrightarrow{a}_{\Pi^C_S} v'$ are applied, and $(v', p', v'') \in H(s'_n)$ is the only atomic process with a protocol state different from v. By induction on the construction of $t_1 \overset{a}{\Rightarrow}_{\Pi_{S,C}} t_2$, one can prove that $s \overset{a}{\Rightarrow}_{\Pi^C_S} s'_n$ using $(v, p, v'') \xrightarrow{a}_{\Pi^C_S} (v', p, v'')$, $(v, p, v'')||(v, p', v'') \xrightarrow{a}_{\Pi^C_S} (v', p'', v'')$ for a $v'' \in Q_{P_C}$, or $(v, p, v'') \xrightarrow{a}_{\Pi^C_S} (v', p', v'')||(v', r'', v'')$, respectively.
<u>Case 2:</u> $\delta = p \xrightarrow{a}_{\Pi_{S,C}} p'||p''$. Then there is a protocol state $v'' \in Q_{P_C}$ such that $(v', p', v'') \in H(s_2)$, and $(v', p'', v'') \in H(s_2)$. The rest of the proof is analogous except that $H(s_2) \setminus \{\{(v', p', v''), (v', p'', v'')\}\}$ is used. □

Proof. (of Theorem 4.1)
Induction on w:
<u>$w = \lambda$:</u> By Lemma 4.2, there is a $s \in \mathrm{F}^{-1}(t)$ such that $s \overset{\lambda}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ and, since $v \overset{\lambda}{\Rightarrow}_{P_C} f$ implies $v = f$, $H(s) = \{\{f\}\}$.
<u>$w = ax$:</u> for an $a \in \Sigma$ and $x \in \Sigma^*$.
Then, $t \overset{ax}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$ has the form $t \overset{\lambda}{\Rightarrow}_{\Pi_{S,C}} t_1 \overset{a}{\Rightarrow}_{\Pi_{S,C}} t_2 \overset{x}{\Rightarrow}_{\Pi_{S,C}} \varepsilon$, and $v \overset{w}{\Rightarrow}_{P_C} f$ has the form $v \overset{a}{\Rightarrow}_{P_C} v' \overset{x}{\Rightarrow}_{P_C} f$.
By induction hypothesis, there is an $s_2 \in PEX(Q_{\Pi^C_S})$ such that $s_2 \overset{a}{\Rightarrow}_{\Pi^C_S} \varepsilon$ and $|ps(s_2)| = 1$. By Lemma 4.3, there is an $s_1 \in PEX(Q_{\Pi^C_S})$ such that $s_1 \overset{a}{\Rightarrow}_{\Pi^C_S} s_2$, where $\Rightarrow_{\Pi^C_S}$ is constructed according to Round-robin reachability. By Lemma 4.2, there is an $s \in PEX(Q_{\Pi^C_S})$ such that $s \overset{\lambda}{\Rightarrow}_{\Pi^C_S} s_1$ only using rules of $\mathcal{R}^0$ and $|\mathrm{ps(s)}| = 1$. Thus $s \overset{\lambda}{\Rightarrow}_{\Pi^C_S} s_1 \overset{a}{\Rightarrow}_{\Pi^C_S} s_2 \overset{x}{\Rightarrow}_{\Pi^C_S} \varepsilon$. □

Corollary 4.4 *If $w \in L(\Pi_{S,C}) \cap L(P_C)$, then there is a Round-robin reachability derivation $t \overset{w}{\Rightarrow}_{\Pi^C_S} \varepsilon$ in the Combined Abstraction Π^C_S, where $t \in Q_{\Pi^C_S}$ is the initial state of Π^C_S.*

Hence the reachability problem can be solved by using the Round-robin reachability and will create fewer false negatives. This leads to a better applicability because a component developer or quality management representative has to check a lower number of counterexamples. Moreover, because we cut branches during the verification, it will probably be finished faster.

Finally, considering the example in Figure 7, the counterexamples $\mathfrak{c}_1$ and $\mathfrak{c}_2$ are not created while using the Round-robin reachability. This will also reduce the number of false negatives in Figure 1, because Figure 7 was a simplification of the behavior of the system abstraction shown in Figure 4b.

5 Related Work

Many works on static protocol-checking of components consider local protocol checking on FSMs. The same approach can also be applied to check protocols of objects in object-oriented systems. The idea of static type checking by using FSMs goes back to Nierstrasz [15]. His approach uses regular languages to model the dynamic behavior of objects, which is less powerful than context-free grammars (CFG). Therefore, the approach cannot handle recursive call-backs. In [12] object-life cycles for the dynamic exchange of implementations of classes and methods using a combination of the bridge/strategy pattern are considered. This approach also bases on FSMs. The approach comprises dynamic as well as static conformance checking. Tenzer and Stevens [21] investigate approaches for checking object-life cycles. They assume that object-life cycles of UML-classes are described using UML state-charts and that for each method of a client, there is a FSM that describes the calling sequence from that method. In order to deal with recursion, Tenzer and Stevens add a rather complicated recursion mechanism to FSMs. It is not clear whether this recursion mechanism is as powerful as pushdown automata and therefore could accept general context-free languages. All these works are for sequential systems. Schmidt et al. [8] propose an approach for protocol checking of concurrent component-based systems. Their approach is also FSM-based and unable to deal with recursive call-backs.

An alternative approach for an investigation of protocol conformance is the use of process algebras such as CSP (e. g., [1]). These approaches are more powerful than FSMs and context-free grammars. However, mechanized checking requires some restrictions on the specification language. For example, [1] uses a subset of CSP that allows only the specification of finite processes. At the end the conformance checking is reduced to checking FSMs similar to [8].

In [16] behavioral protocol conformance is used to describe a problem similar to ours. In contrast to our approach the developer has to define not only the allowed receivable calls but also the calls of the component. This approach can not handle recursive callbacks, since the verification is reduced to finite state model checking.

Many works use process algebras as abstractions for the formal (behavioral) analysis of e. g., BPEL applications. In [6] FSP were used, while in [20] CCS was used. Process algebras are similar to PAN considered in our work. These works do not verify behavior in our sense (protocol conformance).

The approach improved in this paper is represented in [2]. It is a generalization of [22,3]. In [22] recursion is modeled by CFG, so only sequential behavior is considered. Moreover, recursive callbacks are already respected. The model checking problem is discussed in [5].

In our previous work [3,2] $(1, G)$-PRS (named PA-processes) and (P, G)-PRS (named PAN) are used to model the abstraction of components and we show the applicability of this process for verifying component-based systems. This requires another model checking algorithm, which is already defined [13], but to our knowledge not implemented yet.

Other works discuss the model checking of PRS. While [17] generates overapproximations of the execution paths, underapproximations of the reachable configurations are computed in [10] (bounded model checking). In contrast to these works we only focus on the interpretation of the information included in our representation model.

To our knowledge no other work exists, where an a priori approach is implemented to reduce the number of false negatives during the model checking.

6 Conclusions and Future Work

Our verification approach ensures properties based on interaction protocols of programs which include unbound recursion and unbound parallelism. This leads to a higher quality of software, because the protocol conformance can be checked before the deployment. Every error will be found.

In this paper we have shown how we can use the special properties of the representation defined in earlier works to reduce the number of false negatives. We call this approach Round-robin reachability, because it balances the derivation steps applied on each parallel term. This improvement reduces the costs of the quality check, because fewer counterexamples have to be reviewed to find the real errors. We assume that, most spurious counterexamples might be removed in an industrial environment.

Although this improved verification process can not reduce the complexity of the verification in a general case, we assume that in an industrial setting the verification will be finished much faster. To check this assumption is part of our future work.

Currently, we validate our approach in an industrial case study of component-based systems written in Python and C/C++. We can create abstractions of source codes written in Python, already. Presently, we are creating generator of abstractions for C/C++ source code. Early results show that our approach is capable of finding errors or unexpected behavior in real programs. In future work we will create abstractions of BPEL and PHP source codes, considering Java is also planned.

We thank the anonymous referees for their helpful comments.

References

[1] R. Allen and S. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[2] Andreas Both and Wolf Zimmermann. Automatic protocol conformance checking of recursive and parallel bpel systems. In *IEEE Sixth European Conference on Web Services (ECOWS '08)*, pages 81–91, November 2008.

[3] Andreas Both and Wolf Zimmermann. Automatic protocol conformance checking of recursive and parallel component-based systems. In Michel R. V. Chaudron, Clemens A. Szyperski, and Ralf Reussner, editors, *Component-Based Software Engineering, 11th International Symposium (CBSE 2008)*, volume 5282 of *LNCS*, pages 163–179. Springer, October 2008.

[4] Andreas Both and Wolf Zimmermann. A step towards a more practical protocol conformance checking algorithm. In *35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2009*. IEEE Computer Society, August 2009.

[5] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR'97: Proc. of the 8th Int. Conf. on Concurrency Theory*, volume 1243 of *LNCS*, pages 135–150. Springer, 1997.

[6] Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. Model-based analysis of obligations in web service choreography. In *AICT/ICIW*, page 149, 2006.

[7] J. Freudig, W. Löwe, R. Neumann, and M. Trapp. Subtyping of context-free classes. In *Proc. 3rd White Object Oriented Nights*, 1998.

[8] H. W. Schmidt, B. J. Krämer, I. Poernemo, and R. Reussner. Predictable component architectures using dependent finite state machines. In *Proc. of the NATO Workshop Radical Innovations of Software and Systems Engineering in the Future*, volume 2941 of *LNCS*, pages 310–324. Springer, 2002.

[9] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[10] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas Reps. Interprocedural analysis of concurrent programs under a context bound. 2007.

[11] R. J. Lipton. The reachability problem requires exponential space. 62, New Haven, Connecticut: Yale University, Department of Computer Science, Research, January 1976.

[12] W. Löwe, R. Neumann, M. Trapp, and W. Zimmermann. Robust dynamic exchange of implementation aspects. In *TOOLS 29 – Technology of Object-Oriented Languages and Systems*, pages 351–360. IEEE, 1999.

[13] Richard Mayr. Combining petri nets and pa-processes. In *TACS'97: Proc. of the Third Int. Symposium on Theoretical Aspects of Computer Software*, pages 547–561, London, UK, 1997. Springer.

[14] Richard Mayr. Process rewrite systems. *Information and Computation*, 156(1-2):264–286, 2000.

[15] Oscar Nierstrasz. Regular types for active objects. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

[16] Pavel Parizek and Frantisek Plasil. Modeling of component environment in presence of callbacks and autonomous activities. In Richard F. Paige and Bertrand Meyer, editors, *TOOLS (46)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 2–21. Springer, 2008.

[17] Gaël Patin, Mihaela Sighireanu, and Tayssir Touili. Spade: Verification of multithreaded dynamic and recursive programs. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 254–257. Springer, 2007.

[18] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[19] R. H. Reussner. Counter-constraint finite state machines: A new model for resource-bounded component protocols. In *Proc. of the 29th Annual Conf. in Current Trends in Theory and Practice of Informatics*, volume 2540 of *LNCS*, pages 20–40. Springer, 2002.

[20] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. *International Conference on Web Services*, 0:43, 2004.

[21] J. Tenzer and P. Stevens. Modelling recursive calls with uml state diagrams. In *6th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'03)*, volume 2621 of *LNCS*, pages 135–149. Springer, 2003.

[22] Wolf Zimmermann and Michael Schaarschmidt. Automatic checking of component protocols in component-based systems. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *LNCS*, pages 1–17. Springer, 2006.

Complete Example of Figure 7

System abstraction: $\Pi_{S,C} = (\{p_0, p_1, p_2\}, \{a, b, \lambda\}, p_0, \{p_0 \xrightarrow{a} p_1, p_1 \xrightarrow{b} \varepsilon, p_1 \xrightarrow{\lambda} \varepsilon\}, \{\varepsilon\})$

Inverted protocol: $\overline{P_C} = (\{v_0, v_1, v_2\}, \{a, b\}, \{v_0 \xrightarrow{a} v_1, v_1 \xrightarrow{b} v_2\}, v_0, \{v_2\})$

Combined Abstraction: $\Pi_S^C = (\{v_0, v_1, v_2\} \times \{p_0, p_1, p_2\} \times \{v_0, v_1, v_2\}, \{a, b, \lambda\},$
$(v_0, p_0, v_2), \mathcal{R}_C^1 \cup \mathcal{R}_C^\lambda \cup \mathcal{R}_{PFork}^1 \cup \mathcal{R}^0 \cup \mathcal{R}^\varepsilon, \{\varepsilon\}\ )$

Non-optimized sets of transition rules of Π_S^C:

$$\begin{aligned}
\mathcal{R}_C^1 &= \{(v_1, p_1, v_0) \xrightarrow{b} (v_2, \varepsilon, v_0), (v_1, p_1, v_1) \xrightarrow{b} (v_2, \varepsilon, v_1), (v_1, p_1, v_2) \xrightarrow{b} (v_2, \varepsilon, v_2)\} \\
\mathcal{R}_C^\lambda &= \{(v_0, p_1, v_0) \xrightarrow{\lambda} (v_0, \varepsilon, v_0), (v_0, p_1, v_1) \xrightarrow{\lambda} (v_0, \varepsilon, v_1), (v_0, p_1, v_2) \xrightarrow{\lambda} (v_0, \varepsilon, v_2), \\
&\quad (v_1, p_1, v_0) \xrightarrow{\lambda} (v_1, \varepsilon, v_0), (v_1, p_1, v_1) \xrightarrow{\lambda} (v_1, \varepsilon, v_1), (v_1, p_1, v_2) \xrightarrow{\lambda} (v_1, \varepsilon, v_2), \\
&\quad (v_2, p_1, v_0) \xrightarrow{\lambda} (v_2, \varepsilon, v_0), (v_2, p_1, v_1) \xrightarrow{\lambda} (v_2, \varepsilon, v_1), (v_2, p_1, v_2) \xrightarrow{\lambda} (v_2, \varepsilon, v_2)\} \\
\mathcal{R}_{PFork}^1 &= \{(v_0, p_0, v_0) \xrightarrow{a} (v_1, p_1, v_0)||(v_1, p_1, v_0), (v_0, p_0, v_1) \xrightarrow{a} (v_1, p_1, v_1)||(v_1, p_1, v_1), \\
&\quad (v_0, p_0, v_2) \xrightarrow{a} (v_1, p_1, v_2)||(v_1, p_1, v_2)\} \\
\mathcal{R}^0 &= \{(v_0, p_0, v_0) \xrightarrow{\lambda} (v_1, p_0, v_0), (v_0, p_0, v_1) \xrightarrow{\lambda} (v_1, p_0, v_1), (v_0, p_0, v_2) \xrightarrow{\lambda} (v_1, p_1, v_2), (v_0, p_1, v_0) \xrightarrow{\lambda} \\
&\quad (v_1, p_1, v_0), (v_0, p_1, v_1) \xrightarrow{\lambda} (v_1, p_1, v_1), (v_0, p_1, v_2) \xrightarrow{\lambda} (v_1, p_1, v_2), (v_1, p_0, v_0) \xrightarrow{\lambda} (v_2, p_0, v_0), \\
&\quad (v_1, p_0, v_1) \xrightarrow{\lambda} (v_2, p_0, v_1), (v_1, p_2, v_2) \xrightarrow{\lambda} (v_2, p_1, v_2), (v_1, p_1, v_0) \xrightarrow{\lambda} (v_2, p_1, v_0), (v_1, p_1, v_1) \xrightarrow{\lambda} \\
&\quad (v_2, p_1, v_1), (v_1, p_1, v_2) \xrightarrow{\lambda} (v_2, p_2, v_2), (v_0, \varepsilon, v_0) \xrightarrow{\lambda} (v_1, \varepsilon, v_0), (v_0, \varepsilon, v_1) \xrightarrow{\lambda} (v_1, \varepsilon, v_1), (v_0, \varepsilon, v_2) \\
&\quad \xrightarrow{\lambda} (v_1, \varepsilon, v_2), (v_1, \varepsilon, v_0) \xrightarrow{\lambda} (v_2, \varepsilon, v_0), (v_1, \varepsilon, v_1) \xrightarrow{\lambda} (v_2, \varepsilon, v_1), (v_1, \varepsilon, v_2) \xrightarrow{\lambda} (v_2, \varepsilon, v_2)\} \\
\mathcal{R}^\varepsilon &= \{(v_0, \varepsilon, v_0) \xrightarrow{\lambda} \varepsilon, (v_1, \varepsilon, v_1) \xrightarrow{\lambda} \varepsilon, (v_2, \varepsilon, v_2) \xrightarrow{\lambda} \varepsilon\}
\end{aligned}$$

Reachability in Tree-Like Component Systems is PSPACE-Complete

Mila Majster-Cederbaum[1]

Department of Computer Science
University Mannheim
Mannheim, Germany

Nils Semmelrock [2]

Department of Computer Science
University Mannheim
Mannheim, Germany

Abstract

The reachability problem in component systems is PSPACE-complete. We show here that even the reachability problem in the subclass of component systems with "tree-like" communication is PSPACE-complete. For this purpose we reduce the question if a Quantified Boolean Formula (QBF) is true to the reachability problem in "tree-like" component systems.

Keywords: Component-Based Modeling, Architectural Constraints, Reachability, PSPACE-Completeness

1 Introduction

In component-based modeling techniques the size of the global state space of a system is in the worst case exponential in the number of its components. This problem is often referred to as the effect of state space explosion. Thus, checking properties of a component-based system by exploring the state space very quickly becomes inefficient. Here we explore the complexity-theoretical classification of reachability.

As a formal model for component-based systems we consider here interaction systems [8], a formalism for component-based modeling which offers in general an arbitrary degree of synchronization. Reachability in general interaction systems was proved to be PSPACE-complete [15] similar to results in 1-safe Petri nets [6].

[1] Email:mcb@informatik.uni-mannheim.de

[2] Email:nsemmelr@informatik.uni-mannheim.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Tree-like component systems are component systems where the communication structure forms a tree. This is an important class of systems which has been early studied e.g. in [9,5] and more recently e.g. in [4,12].

Here we show that even in the subclass of tree-like interaction systems the reachability problem (and therefore proving deadlock-freedom as well) is PSPACE-complete. We also sketch that deciding progress in tree-like interaction systems is PSPACE-complete.

2 Interaction Systems

Interaction systems are a component-based formalism, i.e. a system is composed of subsystems called components. Components are put together by some kind of glue-code. Interaction systems are defined in two layers. The first layer, the interaction model, describes the interfaces of the components and the glue-code of a system by connecting the interfaces of the components. The second layer describes the behavior of the components, which is here given in form of labeled transition systems.

Definition 2.1 Let $K = \{1, \ldots, n\}$ be a finite set of components. For each $i \in K$ let A_i be a finite set of ports such that $\underset{i,j \in K}{\forall} i \neq j \Rightarrow A_i \cap A_j = \emptyset$. An **interaction model** is a tuple $IM := (K, \{A_i\}_{i \in K}, C)$, where C is a set such that

a) $\forall c \in C : c \subseteq \bigcup_{i \in K} A_i$,

b) $\forall c \in C \forall i \in K : |c \cap A_i| \leq 1$ and

c) $\bigcup_{c \in C} c = \bigcup_{i \in K} A_i$.

The elements of C are called **connectors**. Let for $c \in C$ and $i \in K$ $i(c) := c \cap A_i$ be the set of ports of i which participate in c, i.e. $|i(c)| \leq 1$.

Let $T_i = (Q_i, A_i, \rightarrow_i, q_i^0)$ for $i \in K$ be a labeled transition system with a finite set of states Q_i, a transition relation $\rightarrow_i \subseteq Q_i \times A_i \times Q_i$ and an initial state $q_i^0 \in Q_i$. A transition system T_i for $i \in K$ models the behavior of component i. We will write $q_i \xrightarrow{a_i}_i q_i'$ instead of $(q_i, a_i, q_i') \in \rightarrow_i$.

Definition 2.2 An **interaction system** is a tuple $Sys := (IM, \{T_i\}_{i \in K})$. The behavior of Sys is given by the transition system
$T = (Q_{Sys}, C, \rightarrow, q^0)$ where

a) $Q_{Sys} := \prod_{i \in K} Q_i$ is the state space,

b) $q^0 = (q_1^0, \ldots, q_n^0) \in Q_{Sys}$ is the initial state and

c) $\rightarrow \subseteq Q_{Sys} \times C \times Q_{Sys}$ is the transition relation with $q \xrightarrow{c} q'$ iff for all $i \in K$ $q_i \xrightarrow{i(c)}_i q_i'$ if $i(c) \neq \emptyset$ and $q_i = q_i'$ otherwise.

Definition 2.3 Let Sys be an interaction system and $T = (Q_{Sys}, C, \rightarrow, q^0)$ the associated global transition system. A global state $q \in Q_{Sys}$ is called **reachable** iff there is a path that leads from the initial state q^0 to q in T.

As mentioned we focus on a structural constraint on interaction systems. More precisely we look at the important class of interaction systems such that the glue-code describes a tree-like communication pattern, i.e. components never form a cycle with respect to their connectors.

A tree-like communication structure induces an important class of component-based systems. Many interesting systems belong to this class, e.g. hierarchical systems or networks built by a master-slave operator [9]. For these reasons, this class of component systems has been studied intensely e.g. in [3,4,5,12].

Definition 2.4 Let $IM = (K, \{A_i\}_{i\in K}, C)$ be an interaction model. The **interaction graph** $G^* = (K, E)$ of IM is an undirected graph with $\{i, j\} \in E$ iff there is a connector $c \in C$ with $i(c) \neq \emptyset$ and $j(c) \neq \emptyset$.

An interaction model IM is called **tree-like** iff the associated interaction graph G^* is a tree. An interaction system Sys is called tree-like if its associated interaction model is tree-like.

Remark 2.5 Note, that a tree-like interaction system with a set C of connectors implies that on all $c \in C$ $|c| \leq 2$.

Example 2.6 Consider the dining philosophers problem with n philosophers and n forks. The philosophers, respectively the forks, are labeled with $0, \dots, n-1$. The philosophers are placed in order around a table such that between philosopher i and $i+1$ (we assume modulo n arithmetics) fork i is placed. We construct an interaction system such that each philosopher and each fork corresponds to a component. Let $i \in \{0, \dots, n-1\}$ then philosopher i is modeled by component $Phil_i$ with the set of ports $A_{Phil_i} := \{t\_left_i, t\_right_i, p\_left_i, p\_right_i\}$ such that the ports are modeling, from left to right: "take left fork", "take right fork", "put left fork back on the table" and "put right fork back on the table". Fork i is modeled by component $Fork_i$ with the set of ports $A_{Fork_i} := \{take_i, release_i\}$.

The following connectors describe the synchronization between the philosophers and the forks, corresponding to the seating order.

$$\begin{aligned} take\_left_i &:= \{t\_left_i, take_i\} & take\_right_i &:= \{t\_right_i, take_{i-1}\} \\ put\_left_i &:= \{p\_left_i, release_i\} & put\_right_i &= \{p\_right_i, release_{i-1}\} \end{aligned}$$

Consider the problem for $n = 3$ philosophers, then the set K of components is given by $K = \{Phil_0, Phil_1, Phil_2, Fork_0, Fork_1, Fork_2\}$ and the set C of connectors by $C := \{take\_left_i, take\_right_i, put\_left_i, put\_right_i | i = 0, \dots, 2\}$. The interaction model is given by $IM = (K, \{A_i\}_{i\in K}, C)$. The corresponding interaction graph G^* for IM is given in Figure 1 and forms a cycle.

3 QBF Reduction to Tree-Like Interaction Systems

We will show that reachability in tree-like interaction systems is PSPACE-complete. The PSPACE-hardness will be proved by a reduction from QBF [7]. PSPACE-hardness of reachability in general interaction systems was shown by a reduction

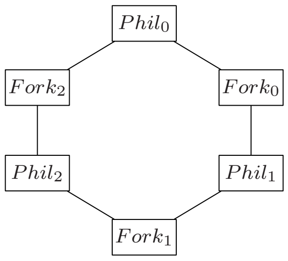


Fig. 1. Interaction graph G^* for the interaction model IM.

from reachability in 1-safe Petri nets [15]. To show the PSPACE-hardness of reachability in 1-safe Petri nets a reduction from QBF was used [6].

3.1 Reduction

Reachability in tree-like interaction systems is in PSPACE. Given a tree-like interaction system and a global state q one can guess a sequence of connectors (because PSPACE=NPSPACE) and check in linear space if it leads from the initial state q^0 to q. At any time we story exactly one global state from which we guess a valid successor state. To prove the PSPACE-hardness we present a reduction from the validity problem for Quantified Boolean Formulas (QBF) to the reachability problem in tree-like interaction systems.

3.1.1 QBF

An instance of QBF [7] is given as a well-formed quantified Boolean formula where its variables $x_1, \ldots, x_n$ are all bound and distinct. Without loss of generality we look at QBF instances over the grammar

$$P ::= x|\neg P|P \wedge P|\exists x.P.$$

In the following we will assume that a QBF formula is built over this grammar. Let H be a QBF then the question is if H is true. The language TQBF is defined as the set of true QBF instances and is well known to be PSPACE-complete.

There is a straightforward, recursive algorithm called *eval* to determine whether a QBF P given over the grammar above is in TQBF.

Algorithm 1

```
1 $eval(P)$
2     $if(P = x)$
3         $return\ value(x)$
4     $if(P = \neg P')$
5         $return\ \neg eval(P')$
6     $if(P = P' \wedge P'')$
7         $return\ eval(P') \wedge eval(P'')$
8     $//\ P = \exists x.P'$ is the only remaining possibility
9     $return\ eval(P'_{x=true}) \vee eval(P'_{x=false})$
```

In line 9 $P'_{x=true}$ denotes the subformula P' with *true* assigned to the variable x.

In line 3 $value(x)$ returns the truth value that is assigned to x. This is possible because every variable x in H is bound by an existential quantifier and therefore a truth value is assigned in line 9. Obviously, $H \in TQBF \Leftrightarrow eval(H) = true$.

3.1.2 RIST

Let IST be the class of tree-like interaction systems. For $Sys \in IST$ let $Reach(Sys) \subseteq Q_{Sys}$ be the set of reachable states. Let

$$RIST := \bigcup_{Sys \in IST} \left(\{Sys\} \times Q_{Sys}\right).$$

For $(Sys, q) \in RIST$ we want to decide if q is reachable in Sys. Let $TRIST \subseteq RIST$ be the set of true $RIST$ instances, i.e.

$$TRIST = \bigcup_{Sys \in IST} \left(\{Sys\} \times Reach(Sys)\right).$$

In the following we will introduce for a QBF H a tree-like interaction system Sys_H and a global state q^t such that

i) $H \in TQBF \Leftrightarrow (Sys_H, q^t) \in TRIST$ and

ii) the size of Sys_H is polynomial in the size of H.

The idea for the construction of Sys_H can be sketched as follows: the interaction system basically simulates the evaluation of the formula H, as in algorithm $eval$ (see Algorithm 1), based on the syntax tree of H. The subformulas of H are the components of the system, and the interaction model describes the propagation of truth values between the nodes of the syntax tree.

Example 3.1 Consider the formula $H = \neg\exists x_1.(x_1 \wedge \neg x_1)$. The associated interaction graph G^*_H of IM_H is given in Figure 2 where components with highlighted frames denote components that do not model subformulas of H. IM_H is constructed accordingly to the following reduction.

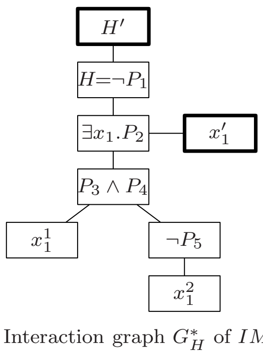


Fig. 2. Interaction graph G^*_H of IM_H.

We now describe in detail how Sys_H is constructed:

3.1.3 Components

Let H be a QBF with variables $x_1, \ldots, x_n$ and $K_2 = \{x_i' | x_i \text{ is a variable in } H\}$. The set of components K_2 is needed to avoid cycles in the interaction graph. Generally, there may be several occurrences of a variable x_i in H. Let x_i occur k_i times for $i = 1, \ldots, n$ as a subformula in H, then we assume that the jth occurrence of variable x_i is renamed in H as x_i^j for $j \in \{1, \ldots, k_i\}$.

Let $K_H = K_1 \cup K_2 \cup \{H'\}$ be a set of components such that
$K_1 = \{P | P \text{ is a subformula of } H\}$. The component H' is an auxiliary component which simplifies the definition of the behavior of the components in K_1.

Given a truth assignment to the variables, subformulas are assigned true or false. Therefore, when we mention an assignment to a component in $K_1 \cup K_2$ we refer to the assignment of the subformula that is modeled by this component.

In the following we will give the port sets of the components. Many ports, in different components, serve the same purpose and only differ in their subscripts. Once such a port is introduced it will not be explicitly explained again.

Port sets of components modeling variables
For $i = 1, \ldots, n$ and $j = 1, \ldots, k_i$ the component $P = x_i^j \in K_1$ represents the jth occurrence of variable x_i in H. The set A_P of ports is given by

$$A_P := \{\mathfrak{a}_P, t_P, f_P, r_P t\} \cup \{r_P x_l t, r_P x_l f | l = 1, \ldots, n\}.$$

- $\mathfrak{a}_P$ abbreviates "activate P" and starts the evaluation of P.
- t_P respectively f_P confirm that currently true respectively false is assigned to P.
- $r_P x_l t$ abbreviates "P receives instruction to set x_l true". If $l = i$ then true is assigned to P. For $i \neq l$ $r_P x_l t$ has no effect on P. The same applies to $r_P x_l f$ setting x_l to false.
- $r_P t$ has the function to assign true to P.

Port sets for negated formulas
A component modeling a negation, i.e. a subformula of the form $P = \neg P_1$ has the following set of ports A_P

$$\begin{aligned} A_P := \{&\mathfrak{e}_P^1, \mathfrak{a}_P, sub_P^1 t, sub_P^1 f, t_P, f_P, r_P t, s_P^1 t\} \cup \\ &\{r_P x_l t, r_P x_l f, s_P^1 x_l t, s_P^1 x_l f | l = 1, \ldots, n\}. \end{aligned}$$

- $\mathfrak{e}_P^1$ abbreviates "evaluate the first subformula of P" and evaluates the subformula P_1.
- $sub_P^1 t$ (abbreviates "subformula 1 is true") respectively $sub_P^1 f$ affirm that P_1 was evaluated true respectively false.
- According to the structure of a negation f_P (abbreviates "$P = \neg P_1$ is false") is enabled if P_1 was evaluated true. Conversely t_P is enabled if P_1 was evaluated false.
- As above $r_P x_l t$ models that P receives the instruction to set x_l true. On the other hand $s_P^1 x_l t$ ("set x_l true in the first subformula of P") models that P itself sends the instruction to set x_l to true to P_1. The same applies to $s_P^1 x_l f$ if x_l needs to be set to false.
- $s_P^1 t$ has the function to set the truth assignment of P's subformula P_1 to true.

Port sets for conjunctions
The component that models a conjunction, i.e. a subformula of the form $P = P_1 \wedge P_2$ has the set of ports

$$\begin{aligned} A_P := \{&\mathfrak{a}_P, \mathfrak{e}^1_P, \mathfrak{e}^2_P, sub^1_P t, sub^1_P f, sub^2_P t, sub^2_P f, t_P, f_P, r_P t, s^1_P t, s^2_P t\} \cup \\ &\{r_P x_l t, r_P x_l f, s^1_P x_l t, s^1_P x_l f, s^2_P x_l t, s^2_P x_l f | l = 1, \ldots, n\}. \end{aligned}$$

This is the only formula that has two direct subformulas. $P = P_1 \wedge P_2$ needs to evaluate P_1 and P_2, therefore there are ports $\mathfrak{e}^1_P$ and $\mathfrak{e}^2_P$. Similarly there are $sub^1_P t$, $sub^1_P f$, $sub^2_P t$, $sub^2_P f$ for actually receiving the truth values of P_1 and P_2. Likewise, $s^1_P x_l t$ and $s^2_P x_l t$ model that P needs to set x_l to true in its first and second subformula and respectively $s^1_P x_l f$ and $s^2_P x_l f$ to false.

Port sets for existentially quantified formulas and associated component x'_i
In the interaction system Sys_H a component for a subformula of the form $P = \exists x_i.P_1$ with $i = 1, \ldots, n$ needs to have access to the current truth assignment of the variable x_i. For this purpose the set of components K_2 was introduced. Let x_i be the variable that is quantified by the subformula $P = \exists x_i.P_1$. The component x'_i models the truth assignment of x_i. The set of ports $A_{x'_i}$ is given by

$$A_{x'_i} := \{r x_i t, r x_i f, t_{x_i}, f_{x_i}\}.$$

t_{x_i} respectively f_{x_i} affirm that the current state of x'_i is true respectively false. $r x_i t$ assigns x'_i true. Analogously $r x_i f$ switches the assignment to false.

The port set A_P for $P = \exists x_i.P_1$ is given by

$$\begin{aligned} A_P := \{&\mathfrak{a}_P, \mathfrak{e}^1_P, sub^1_P t, sub^1_P f, t_P, f_P, x_i t, x_i f, s x_i t, s x_i f, r_P t, s^1_P t\} \cup \\ &\{r_P x_l t, r_P x_l f, s^1_P x_l t, s^1_P x_l f | l = 1, \ldots, n\}. \end{aligned}$$

$\mathfrak{a}_P$, $\mathfrak{e}^1_P$, $sub^1_P t$, $sub^1_P f$, t_P and f_P act similarly to the corresponding ports of the other components specified above. $x_i t$ confirms that true is assigned to x_i, and $s x_i t$ sets x_i to true if the current assignment is false. On the other hand $x_i f$ confirms that false is assigned to x_i, and $s x_i f$ assigns false to x_i if that is not the case.

Port set for the auxiliary component H'
Given the syntax tree for H, whose root is labeled H, H' can be interpreted as a direct dummy predecessor formula of H without any logical operator. The set of ports $A_{H'}$ is given by

$$A_{H'} := \{\mathfrak{e}^1_{H'}, sub^1_{H'} t, sub^1_{H'} f, s^1_{H'} t, end_{H'}\}.$$

All ports but $end_{H'}$ act exactly as the ports described above. It will be shown that the formula H is in TQBF iff the component associated with H is evaluated true, i.e. $sub^1_{H'} t$ can interact eventually. When the evaluation of the QBF H has been simulated, i.e. H' reached a state that represent the fact that H was evaluated true or false, then the port $end_{H'}$ becomes enabled. This only assures that the behavior of H' does not deadlock.

3.1.4 Connectors

We will now define a set C of connectors. Let $P \in K_1 \cup \{H'\}$ be a subformula which is not an occurrence of a variable. P can have one direct subformula which is P_1 or two direct subformulas P_1 and P_2. If P needs the truth value of P_k, $k \in \{1,2\}$, to be evaluated then the evaluation in P_k needs to be activated. This is realized by the synchronization of $\mathfrak{e}_P^k$ and $\mathfrak{a}_{P_k}$. Furthermore P can ask P_k for its current truth value. These interactions are realized by

$$eval\_P \rightarrow P_k := \{\mathfrak{e}_P^k, \mathfrak{a}_{P_k}\} \qquad P\_ask\_P_k\_true := \{sub_P^k t, t_{P_k}\}$$
$$P\_ask\_P_k\_false := \{sub_P^k f, f_{P_k}\}$$

for $k \in \{1,2\}$. These connectors already connect all components in $K_1 \cup \{H'\}$ and result in an interaction graph that is related to the syntax tree of the QBF H.

If P needs all occurrences of variable x_i to be set to true or false a direct interaction with the components that model these variables would lead to a cycle in the associated interaction graph. Therefore, P passes this information to its subformulas, i.e. $s_P^k x_i t$ in P has to synchronize with $r_{P_k} x_i t$ in P_k where P_k is a direct subformula of P. Let $i \in \{1,\dots,n\}$. The following connectors, for $k \in \{1,2\}$, realize the synchronizations needed to propagate the information to switch a variable.

$$set\_x_i\_true\_P \rightarrow P_k := \{s_P^k x_i t, r_{P_k} x_i t\}$$
$$set\_x_i\_false\_P \rightarrow P_k := \{s_P^k x_i f, r_{P_k} x_i f\}$$

If the QBF H is true, we need all components to be in one fixed state – this will be a state that models the assignment true. In fact, the component H' will observe if H is true and reach a fixed state. To assure that all components can reach a fixed state, a similar technique as above is used. A component can set the truth assignment of the components that represent its subformulas to true by the following connector for $k \in \{1,2\}$.

$$set\_P_k\_true\_P \rightarrow P_k := \{s_P^k t, r_{P_k} t\}$$

Consider a subformula of the form $P = \exists x_i.P_1 \in K_1$ and the associated component $x_i' \in K_2$. The component representing P can assign x_i' the truth value true or false and can ask x_i' whether the current truth assignment is true or false. This is realized by

$$set\_x_i'\_true := \{sx_i t, rx_i t\} \qquad ask\_true_{x_i'} := \{x_i t, t_{x_i}\}$$
$$set\_x_i'\_false := \{sx_i f, rx_i f\} \qquad ask\_false_{x_i'} := \{x_i f, f_{x_i}\}$$

IF H' reaches a state that indicates that H was evaluated true or false, i.e. the simulation of the evaluation of H is finished, then the unary connector $evaluated := \{end_{H'}\}$ becomes enabled.

Let C be the set of connectors given by

$$\{eval\_P \to P_k, P\_ask\_P_k\_true, P\_ask\_P_k\_false | P \in K_1 \cup \{H'\} \text{ with succ. } P_k\} \cup$$
$$\{set\_x'_i\_true, set\_x'_i\_false, ask\_true_{x'_i}, ask\_false_{x'_i} | x'_i \in K_2\} \cup$$
$$\{set\_P_k\_true\_P \to P_k | P \in K_1 \cup \{H'\} \text{ with succ. } P_k\} \cup$$
$$\{set\_x_i\_true\_P \to P_k, set\_x_i\_false\_P \to P_k | P \in K_1 \text{with succ. } P_k, i \in \{1, \dots, n\}\} \cup$$
$$\{evaluated\}.$$

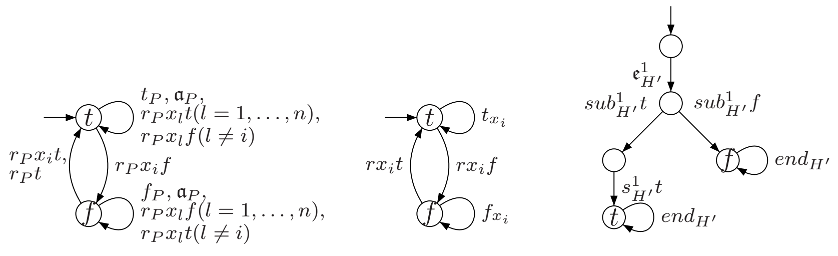
So far we have the interaction model $IM_H := (K_H, \{A_P\}_{P \in K_H}, C)$. This way any QBF formula H over the grammar, given above, can be mapped to an interaction model IM_H.

Remark 3.2 The interaction graph G^*_H, associated to IM_H, is a tree, as it is constructed along the syntax tree and augmented with the components H' and x'_i for $i = 1, \dots, n$ without forming cycles.

3.1.5 Local Behavior

The local behavior of the components is given by labeled transition systems. Every system has one state labeled t and one labeled f. These states model the fact that either true respectively false was assigned to this component or it was evaluated true respectively false. The initial state will be denoted by an ingoing arrow.

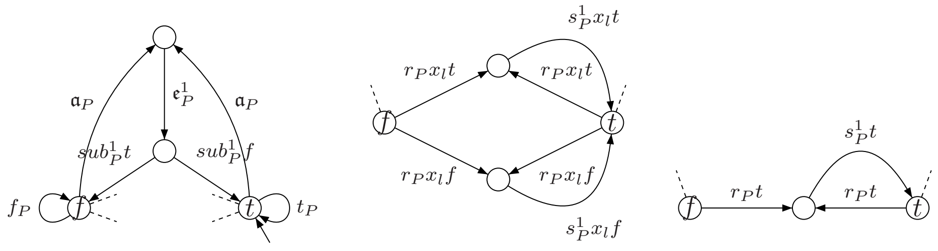
Figure 3(a) depicts the transition system of the component modeling the jth occurrence of variable x_i. Figure 3(b) gives the local behavior of a component $x'_i \in K_2$. The behavior of H' is given in 3(c). The transition systems for a variable x^j_i and a $x'_i \in K_2$ are self-explanatory. If in $T_{H'}$ the port $\mathfrak{e}^1_{H'}$ is performed, i.e. component H needs to be evaluated, then $T_{H'}$ waits to perform either $sub^1_{H'}t$ or $sub^1_{H'}f$. This ports can only be performed if T_H reaches its state labeled t respectively f. It will be shown that this indicates whether the associated QBF is true or false.



(a) $P = x^j_i$ (b) x'_i (c) H'

Fig. 3. Transition systems $T_{x^j_i}$ for a component x^j_i (a), $T_{x'_i}$ for x'_i (b) and $T_{H'}$ for the component H' (c).

In Figure 4 the transition system for a component of the form $P = \neg P_1$ is pictured. Note, that for better readability, the transition system in Figure 4(a) is not completely displayed. In system 4(a) the transitions and states pictured in Figure 4(b) and 4(c) have to be included between the states labeled t and f for $l = 1, \dots, n$.



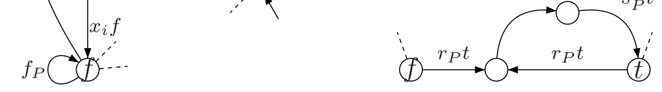
(a) $P = \neg P_1$

(b) Section of 4(a) and 5(a), for $l \in \{1, \ldots, n\}$

(c) Section of 4(a)

Fig. 4. Main section of the transition systems $T_{\neg P_1}$ (a), part of $T_{\neg P_1}$ for $l \in \{1, \ldots, n\}$ (b), part of $T_{\neg P_1}$ (c).

In Figure 5 the transition system for a component of the form $P = \exists x_i.P_1$ is pictured. For better readability, the transition system in Figure 5(a) is not completely displayed. In system 5(a) the transitions and states pictured in Figure 4(b) and 5(b) have to be included between the states labeled t and f for $l = 1, \ldots, n$.

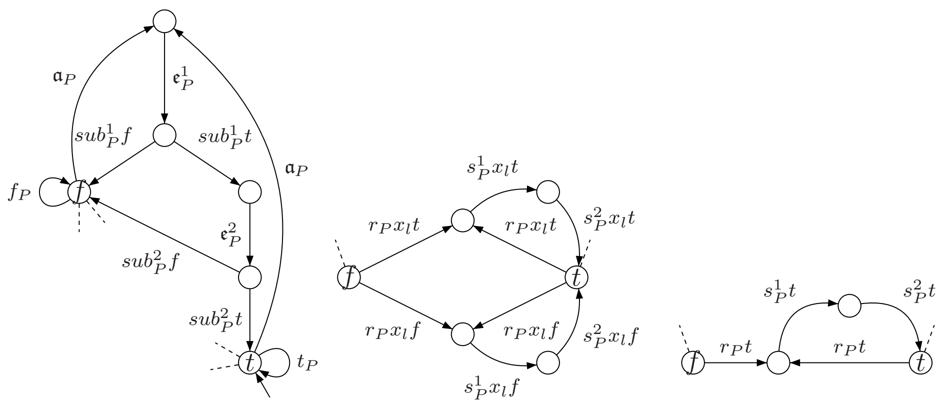


(a) $P = \exists x_i.P_1$

(b) Section of 5(a)

Fig. 5. Main sections of the transition system $T_{\exists x_i.P_1}$ (a), part of $T_{\exists x_i.P_1}$ (b).

In Figure 6 the transition system for a component of the form $P = P_1 \wedge P_2$ is pictured. Note, that the transition system in Figure 6(a) is not completely displayed. The transitions and states pictured in Figure 6(b) and 6(c) have to be included between the states labeled t and f for $l = 1, \ldots, n$.



(a) $P = P_1 \wedge P_2$ (b) Section of 6(a), for $l \in \{1, \dots, n\}$ (c) Section of 6(a)

Fig. 6. Transition system $T_{P_1 \wedge P_2}$.

The resulting interaction system is denoted by $Sys_H := (IM_H, \{T_P\}_{P \in K_H})$.

Theorem 3.3 *Let H be a QBF over the grammar $P ::= x|\neg P|P \wedge P|\exists x.P$ and Sys_H the associated interaction system obtained from the reduction. Let q^t be the global state in which all components are in their state labeled t, then*

$$H \in TQBF \Leftrightarrow (Sys_H, q^t) \in TRIST.$$

The proof of Theorem 3.3 can be found in the Appendix of [14].

4 QBF Reduction to Progress in Tree-Like Interaction Systems

By minor modification of the reduction given above it is possible to show the PSPACE-completeness of the progress property in tree-like interaction systems. At first we give some definitions to introduce progress in interaction systems and then give an overview why it is PSPACE-complete to decide this property in tree-like interaction systems. In general interaction systems progress is PSPACE-complete [15], so progress in tree-like interaction systems is in PSPACE.

Definition 4.1 Let Sys be an interaction system and $T = (Q_{Sys}, C, \rightarrow, q^0)$ the associated global transition system. A global state $q \in Q_{Sys}$ is called a **deadlock** if no connector is enabled in q, i.e. there is no $c \in C$ and $q' \in Q_{Sys}$ such that $q \xrightarrow{c} q'$. A system Sys is **free of deadlocks** if there is no reachable state $q \in Q_{Sys}$ such that q is a deadlock.

Definition 4.2 Let Sys be a deadlock-free interaction system. A **run of** Sys is an infinite sequence σ

$$q^0 \xrightarrow{c_1} q^1 \xrightarrow{c_2} q^2 \dots,$$

with $q^l \in Q_{Sys}$ and $c_l \in C$ for $l \geq 1$.

Definition 4.3 Let Sys be a deadlock-free interaction system with components K. $k \in K$ **may progress** in Sys if for every run σ k participates infinitely often in σ.

An instance of the progress problem in interaction systems is given by a tuple (Sys, k) where Sys is a deadlock-free interaction system with components K and $k \in K$. The question is if k may progress in Sys.

We modify Sys_H as follows:

We introduce an additional component called *pro* with the set of ports $A_{pro} := \{t_{pro}\}$ and the behavior given by the transition system T_{pro} in Figure 7. The idea is to embed *pro* in Sys_H such that t_{pro} will participate infinitely often in every run σ iff H is true.

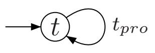


Fig. 7. Transition system T_{pro} for the component *pro*.

In addition we modify the component H' as follows. The set of ports $A_{H'}$ of the component H' is now given by

$$A_{H'} := \{\mathfrak{e}^1_{H'}, sub^1_{H'}t, sub^1_{H'}f, s^1_{H'}t, end\_true_{H'}, end\_false_{H'}\},$$

i.e. $end_{H'}$ is removed and the ports $end\_true_{H'}$ and $end\_false_{H'}$ are added. The modified behavior of H' is given by the transition system $T_{H'}$ in Figure 8.

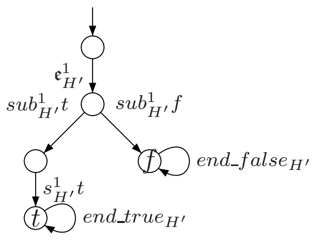


Fig. 8. Modified transition system $T_{H'}$ for the component H'.

In addition, the connector *evaluated* is removed von the set C of connectors, and the two following connectors are added.

$$evaluated\_true := \{end\_true_{H'}, t_{pro}\},$$
$$evaluated\_false := \{end\_false_{H'}\}.$$

It is easy to see that the connector *evaluated_true* is the only connector that is enabled if the state q^t is reached. In this case, *evaluated_true* will perform infinitely often, i.e. the component *pro* will participate infinitely often. Therefore the component *pro* may progress iff H is true.

5 Conclusion and Related Work

We investigated a complexity issue for component-based systems. In [6] the reachability in 1-safe Petri nets was proven to be PSPACE-complete and [15] used this result to show the PSPACE-completeness of the reachability problem in component-based systems. Here we restricted ourselves to tree-like systems and showed that even in this class deciding reachability is PSPACE-complete.

We conjecture that this result can be still strengthened such that it should be possible to show that even for the class of linear interaction systems, where the interaction graph forms a sequence of components, the reachability problem is PSPACE-complete. Given these complexity issues it makes sense to look for conditions that can be tested in polynomial time and guarantee a desired property.

For general component systems this is pursued e.g. in [1,2,10,11,13,16]. For tree-like component systems [3,4,5,12] have followed this approach and in particular established conditions that ensure deadlock-freedom.

References

[1] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6:213–249, 1997.

[2] Paul Attie and Hana Chockler. Efficiently Verifiable Conditions for Deadlock-Freedom of Large Concurrent Programs. In *Proceedings of VMCAI'05*, LNCS 3385, pages 465–481, 2005.

[3] H. Baumeister, F. Hacklinger, R. Hennicker, A. Knapp, and M. Wirsing. A Component Model for Architectural Programming. In *Proceedings of FACS'05*, volume 160 of *ENTCS*, pages 75–96. Elsevier, 2006.

[4] Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. Architecting Families of Software Systems with Process Algebras. *ACM Trans. on Software Engineering and Methodology*, 11:386 – 426, October 2002.

[5] Stephen D. Brookes and A. W. Roscoe. Deadlock Analysis in Networks of Communicating Processes. *Distributed Computing*, 4:209–230, 1991.

[6] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity Results for 1-safe Nets. In *Theoretical Computer Science*, pages 326–337. Springer-Verlag, 1995.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.

[8] Gregor Gössler and Joseph Sifakis. Composition for Component-Based Modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005.

[9] Charles A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International series in computer science. Prentice-Hall International, Englewood Cliffs, NJ [u.a.], 1985.

[10] Paola Inverardi and Sebastián Uchitel. Proving Deadlock Freedom in Component-Based Programming. In *FASE '01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 60–75, London, UK, 2001. Springer-Verlag.

[11] M. Majster-Cederbaum, M. Martens, and C. Minnameier. A Polynomial-Time Checkable Sufficient Condition for Deadlock-Freedom of Component-Based Systems. *Lecture Notes in Computer Science*, 4362/2007:888–899, 2007.

[12] Mila Majster-Cederbaum and Moritz Martens. Compositional Analysis of Deadlock-Freedom for Tree-Like Component Architectures. In *EMSOFT '08: Proceedings of the 7th ACM international conference on Embedded software*, pages 199–206, New York, NY, USA, 2008. ACM.

[13] Mila Majster-Cederbaum, Moritz Martens, and Christoph Minnameier. Liveness in Interaction Systems. *Electron. Notes Theor. Comput. Sci.*, 215:57–74, 2008.

[14] Mila Majster-Cederbaum and Nils Semmelrock. Reachability in Tree-Like Component Systems is PSPACE-Complete. Technical Report TR-2009-004, University of Mannheim, Germany, 2009.

[15] Mila E. Majster-Cederbaum and Christoph Minnameier. Everything Is PSPACE-Complete in Interaction Systems. In *ICTAC*, pages 216–227, 2008.

[16] Christoph Minnameier and Mila Majster-Cederbaum. Cross-Checking – Enhanced Over-Approximation of the Reachable Global State Space of Component-Based Systems, 2009. submitted for puplication.

Composition of Services with Constraints

Balbiani, Cheikh [1],[2]

IRIT-CNRS, Toulouse, France

Héam[3]

LSV-CNRS-INRIA, Cachan, France

Kouchnarenko[4]

INRIA-CASSIS-LIFC, Besançon, France

Abstract

Composition of Web services consists of the interleaving of the sequence of actions executed by the elementary services in accordance with a client specification. We model Web services as automata executing actions and also sending and receiving messages. This paper provides a theoretical study for three service composition problems, and in particular for the problem of computing a Boolean formula which exactly characterises the conditions required for services to answer the client's request. New complexity results are established for these problems within the framework of service composition with constraints.

Keywords: Composition, Décidability.

1 Introduction and Related Work

Service oriented computing [23] is a programming paradigm which considers services as elementary components. From these components, distributed applications are realised in accordance with a client specification. To realise some distributed applications, elementary components have to be composed. The composition problem has been investigated since the 2000's with many solutions proposed [3,2,17]. Often, services are seen as finite automata. In this case, the client specification is given by a finite automaton which represents all computations that a client wants the services to execute. By executing their transitions, services modify their environment and that of the client. The problem of combining services becomes that of

[1] Email: Philippe.Balbiani@irit.fr

[2] Email: Cheick@irit.fr

[3] Email: pcheam@lsv.ens-cachan.fr

[4] Email: Olga.Kouchnarenko@lifc.univ-fcomte.fr

composing automata, like in [3,2]. In other cases, services are able to send and to receive messages. Client specification is then given by a logical formula which represents client's goals he wants services to reach. By communicating together, services modify their knowledge and those of their client (see, e.g., [17]). This paper follows the line of reasoning suggested in [5,6,7,8], which consists in giving the semantics of services by means of automata. The paper particularly focuses on the following problem: Given services $\mathcal{A}_1$, ..., $\mathcal{A}_n$ and the request $\mathcal{A}$ of a client, can $\mathcal{A}_1$, ..., $\mathcal{A}_n$ be organised as to answer $\mathcal{A}$? The originality of our approach consists in modeling the services by Boolean automata, i.e. finite automata extended with parametric Boolean conditions. The main motivation for using this model is to manage conditional actions or communications of $\mathcal{A}_1$, ..., $\mathcal{A}_n$. For instance, a conditional action may be, for some $i, j \in \{1, \ldots, n\}$, that $\mathcal{A}_i$ accepts to communicate with $\mathcal{A}_j$ if and only if $\mathcal{A}_j$ has a security certificate given by some authority. A conditional action may also be, for some $i, j \in \{1, \ldots, n\}$, that $\mathcal{A}_i$ accepts to answer $\mathcal{A}_j$ only if the IP address of $\mathcal{A}_j$ is in a selected area. This kind of conditions frequently appear when specifying services. As far as we know, the composition problems studied in the literature do not handle this kind of conditions. This paper provides a theoretical study for three service composition problems, and in particular for the problem of computing a Boolean formula ϕ which exactly characterises the conditions required for $\mathcal{A}_1$, ..., $\mathcal{A}_n$ to answer the client's request. The paper is organised as follows. Section 2 introduces the formal background. In Section 3, we formally define the valuation decision problem, the Boolean formula decision problem and the Boolean formula synthesis problem for both simulation-based relations and trace-based relations. Sections 4 and 5 contains our complexity results. Finally we conclude in section 6.

Related Work In the context of finite automata, many research works on complexity results for various finite automata compositions have been done.

In [10], the authors investigated the following problem: given $n + m$ finite automata $\mathcal{A}_1, \ldots, \mathcal{A}_{n+m}$, what is the complexity of deciding whether $\mathcal{A}_1 \times \ldots \times \mathcal{A}_n$ is equivalent to $\mathcal{A}_{n+1} \times \ldots \times \mathcal{A}_{n+m}$. The class of problems considered in [10] is for non flat systems, i.e. one requires that $n \geq 2$ and $m \geq 2$. Another crucial difference w.r.t. our work concerns the product. In [10], the product is partially synchronised but, contrarily to our work, synchronisation does not produce ε-transition but labeled transitions. Their main result shows that the above decision problem is EXPTIME-hard for any relation between the simulation preorder and bisimulation, and that it is EXSPACE-hard for any relation between trace inclusion and the intersection of ready trace equivalence.

A more general problem is considered in [20] where the product is closer to ours since some actions can be *hidden*, i.e. replaced by an ε-transition. The author proved that for non flat systems the equivalence checking is PSPACE-hard for any relation between bisimilarity and trace equivalence, and that the problem is EXPSPACE-complete for trace equivalence, and EXPTIME-complete for bisimilarity. It was also conjectured in [20] that the problem is EXPTIME-hard for any relation between bisimilarity and trace equivalence. This conjecture was enhanced and proved in [22]: the problem is EXPTIME-hard for any relation between bisimilarity and trace preorder.

In [15], it is shown that deciding whether $\mathcal{A}$ is simulated by $\mathcal{A}_1 \times \ldots \times \mathcal{A}_n$ is still EXPTIME-hard. In this work, the considered product is asynchronous.

Several recent works focus on the use of finite automata based models to address Web services composition problems. In [18,17], the authors propose a model where Web services compositions expressed in BPEL are formally defined by state transition systems with communicating and internal (unobservable) actions. In [17], this model was enriched using a knowledge base, and in [9] using a theory. These approaches were applied to practical applications in [24,12,11]. In [1,21], Web services are defined by PWL-S documents and modelled by guarded finite automata. A similar approach is investigated in [16] where Web services are defined in BPEL. In [14], the authors model by Input/Output automata Web services interfaces described in BPEL, OWLS and WSDL.

2 Preliminaries

Let P be a finite set of Boolean variables (with typical members denoted p, p', ...), Σ_a be a countable set of asynchronous actions (with typical members denoted α, β, ...) and Σ_s be a countable set of synchronous actions (with typical members denoted σ, τ, ...). We will assume that P, Σ_a and Σ_s are disjoint.

Finite automata

A finite automaton is a tuple $\mathcal{A} = (Q, E, I, F)$ where

- Q is a finite set of states,
- E is a function from $Q \times Q$ into the set of all finite subsets of $\Sigma_a \cup (\{?, !\} \times \Sigma_s) \cup \{\epsilon\}$,
- $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states.

For all $x \in \{?, !\}$, for all $\sigma \in \Sigma_s$, (x, σ) will be denoted $x\sigma$.

$\mathcal{A}$ is said to be ϵ-free iff E is a function from $Q \times Q$ into the set of all finite subsets of $\Sigma_a \cup (\{?, !\} \times \Sigma_s)$. We shall say that $\mathcal{A}$ is weakly asynchronous iff E is a function from $Q \times Q$ into the set of all finite subsets of $\Sigma_a \cup \{\epsilon\}$. $\mathcal{A}$ is said to be strongly asynchronous iff E is a function from $Q \times Q$ into the set of all finite subsets of Σ_a. Given a weakly asynchronous automaton $\mathcal{A}$, let E^* be the function from $Q \times Q$ into the set of all finite subsets of Σ_a such that for all $q, r \in Q$, for all $\alpha \in \Sigma_a$, $\alpha \in E^*(q, r)$ iff there are sequences $(q_0, q_1, \ldots, q_m), (r_0, r_1, \ldots, r_n) \in Q^+$ such that

- for all positive integers i, if $i \leq m$ then $\epsilon \in E(q_{i-1}, q_i)$,
- for all positive integers j, if $j \leq n$ then $\epsilon \in E(r_{j-1}, r_j)$,
- $\alpha \in E(q_m, r_0)$,
- $q_0 = q$ and $r_n = r$.

In this case, a run of $\mathcal{A}$ on a sequence $(\alpha_1, \ldots, \alpha_k) \in \Sigma_a^*$ is a sequence $(q_0, q_1, \ldots, q_k) \in Q^+$ such that for all positive integers i, if $i \leq k$ then $\alpha_i \in E^*(q_{i-1}, q_i)$, $q_0 \in I$ and $q_k \in F$. Moreover, the traces of $\mathcal{A}$, denoted $tr(\mathcal{A})$, is the set of all sequences $(\alpha_1, \ldots, \alpha_k) \in \Sigma_a^*$ such that there is a run of $\mathcal{A}$ on $(\alpha_1, \ldots, \alpha_k)$.

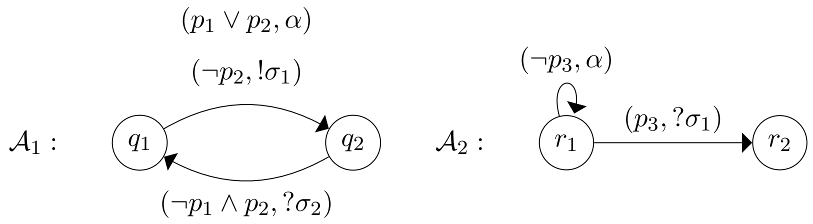


Figure 1. Boolean automata $\mathcal{A}_1$ and $\mathcal{A}_2$.

Boolean automata

The set $\mathcal{B}(P)$ of all Boolean formulas (with typical members denoted ϕ, ψ, ...) is defined by: $\phi ::= p \mid \bot \mid \neg\phi \mid (\phi \vee \phi)$, with $p \in P$. The other constructs are defined as usual. In particular, $\top = \neg\bot$ and $(\phi \wedge \psi) = \neg(\neg\phi \vee \neg\psi)$. A valuation is a function from P into $\{0, 1\}$. Every valuation V gives rise to a function $\widehat{V}$ from $\mathcal{B}(P)$ into $\{0, 1\}$ in the usual way. A Boolean automaton is a tuple $\mathcal{A} = (Q, E, I, F)$ where

- Q is a finite set of states,
- E is a function from $Q \times Q$ into the set of all finite subsets of $\mathcal{B}(P) \times (\Sigma_a \cup (\{?, !\} \times \Sigma_s) \cup \{\epsilon\})$,
- $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states.

The notions of ϵ-freeness, weak asynchronicity and strong asynchronicity are defined for Boolean automata in the same way as they are defined for finite automata. As example, take the case of the Boolean automata $\mathcal{A}_1$ and $\mathcal{A}_2$ from Fig. 1, with $\Sigma_a = \{\alpha\}$ and $\Sigma_s = \{\sigma_1, \sigma_2\}$.

Synchronisation

Let $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1)$, ..., $\mathcal{A}_n = (Q_n, E_n, I_n, F_n)$ be ϵ-free Boolean automata. Their synchronisation, denoted $\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n$, is the weakly asynchronous Boolean automaton $\mathcal{A} = (Q, E, I, F)$ defined by:

- $Q = Q_1 \times \ldots \times Q_n$,
- E is the function from $Q \times Q$ into the set of all finite subsets of $\mathcal{B}(P) \times (\Sigma_a \cup \{\epsilon\})$ such that for all $\boldsymbol{q}, \boldsymbol{r} \in Q$, for all $\phi \in \mathcal{B}(P)$,
  · for all $\alpha \in \Sigma_a$, $(\phi, \alpha) \in E(\boldsymbol{q}, \boldsymbol{r})$ iff there is $i \in \{1, \ldots, n\}$ such that $\boldsymbol{q} \equiv_i \boldsymbol{r}$ and $(\phi, \alpha) \in E_i(q_i, r_i)$,
  · $(\phi, \epsilon) \in E(\boldsymbol{q}, \boldsymbol{r})$ iff there are $i_1, i_2 \in \{1, \ldots, n\}$, there are $\phi_{i_1}, \phi_{i_2} \in \mathcal{B}(P)$, there is $\sigma \in \Sigma_s$ such that $i_1 \neq i_2$, $\boldsymbol{q} \equiv_{i_1,i_2} \boldsymbol{r}$, either $(\phi_{i_1}, (?, \sigma)) \in E_{i_1}(q_{i_1}, r_{i_1})$ and $(\phi_{i_2}, (!, \sigma)) \in E_{i_2}(q_{i_2}, r_{i_2})$ or $(\phi_{i_1}, (!, \sigma)) \in E_{i_1}(q_{i_1}, r_{i_1})$ and $(\phi_{i_2}, (?, \sigma)) \in E_{i_2}(q_{i_2}, r_{i_2})$ and $\phi = \phi_{i_1} \wedge \phi_{i_2}$.
- $I = I_1 \times \ldots \times I_n$ and $F = F_1 \times \ldots \times F_n$,

the binary relations $\equiv_i, \equiv_{i_1,i_2} \subseteq Q \times Q$ being such that for all $\boldsymbol{q}, \boldsymbol{r} \in Q$, $\boldsymbol{q} \equiv_i \boldsymbol{r}$ iff for all $j \in \{1, \ldots, n\}$, if $i \neq j$ then $q_j = r_j$ and $\boldsymbol{q} \equiv_{i_1,i_2} \boldsymbol{r}$ iff for all $j \in \{1, \ldots, n\}$, if $i_1 \neq j$ and $i_2 \neq j$ then $q_j = r_j$. Consider, as example, the Boolean automata A_1 and $\mathcal{A}_2$ from Fig. 1 and $\mathcal{A}_1 \otimes \mathcal{A}_2$ from Fig. 2.

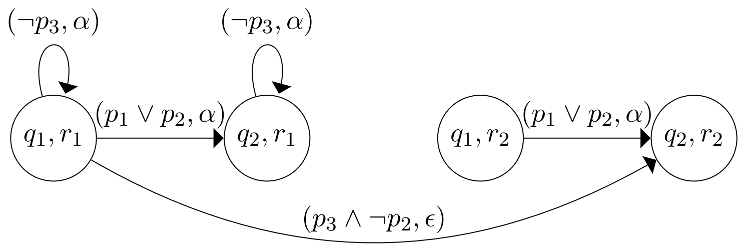


Figure 2. Boolean automata $\mathcal{A}_1 \otimes \mathcal{A}_2$.

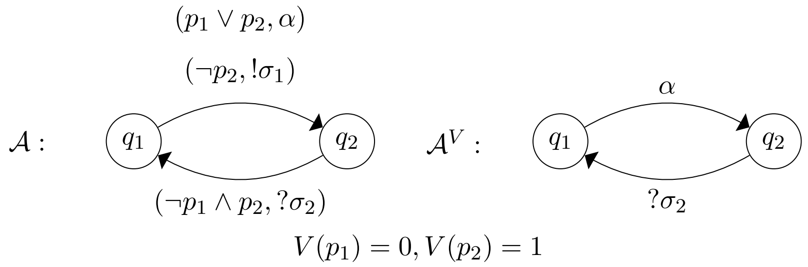


Figure 3. Boolean automaton $\mathcal{A}$ and the associated automaton $\mathcal{A}^V$.

From Boolean automata to finite automata

Let $\mathcal{A} = (Q, E, I, F)$ be a Boolean automaton and V be a valuation. The interpretation of $\mathcal{A}$ through V, denoted $\mathcal{A}^V$, is the finite automaton $\mathcal{A}' = (Q', E', I', F')$ defined by:

- $Q' = Q$,
- E' is the function from $Q' \times Q'$ into the set of all finite subsets of $\Sigma_a \cup (\{?, !\} \times \Sigma_s) \cup \{\epsilon\}$ such that for all $q, r \in Q'$,
  - for all $\alpha \in \Sigma_a$, $\alpha \in E'(q, r)$ iff there is $\phi \in \mathcal{B}(P)$ such that $(\phi, \alpha) \in E(q, r)$ and $\widehat{V}(\phi) = 1$,
  - for all $x \in \{?, !\}$, for all $\sigma \in \Sigma_s$, $x\sigma \in E'(q, r)$ iff there is $\phi \in \mathcal{B}(P)$ such that $(\phi, x\sigma) \in E(q, r)$ and $\widehat{V}(\phi) = 1$,
  - $\epsilon \in E'(q, r)$ iff there is $\phi \in \mathcal{B}(P)$ such that $(\phi, \epsilon) \in E(q, r)$ and $\widehat{V}(\phi) = 1$,
- $I' = I$ and $F' = F$.

As example, take the case of $\mathcal{A}$ and $\mathcal{A}^V$ from Fig. 3.

Trace inclusion and trace equivalence

Let $\mathcal{A} = (Q, E, I, F)$, $\mathcal{A}' = (Q', E', I', F')$ be weakly asynchronous finite automata. We shall say that $\mathcal{A}$ is trace-included in $\mathcal{A}'$, denoted $\mathcal{A} \sqsubseteq \mathcal{A}'$, iff $tr(\mathcal{A}) \subseteq tr(\mathcal{A}')$. $\mathcal{A}$ is said to be trace-equivalent to $\mathcal{A}'$, denoted $\mathcal{A} \equiv \mathcal{A}'$, iff $\mathcal{A} \sqsubseteq \mathcal{A}'$ and $\mathcal{A}' \sqsubseteq \mathcal{A}$.

Simulation and bisimulation

Let $\mathcal{A} = (Q, E, I, F)$, $\mathcal{A}' = (Q', E', I', F')$ be weakly asynchronous finite automata. We define a binary relation $Z \subseteq Q \times Q'$ such that $dom(Z) \cap I \neq \emptyset$ and $ran(Z) \cap I' \neq \emptyset$ to be a simulation of $\mathcal{A}$ by $\mathcal{A}'$, denoted Z: $\mathcal{A} \longleftarrow \mathcal{A}'$, iff for all $q \in$

Q, for all $q' \in Q'$, if $q\ Z\ q'$ then

- for all $\alpha \in \Sigma_a$, for all $r \in Q$, if $\alpha \in E^*(q, r)$ then there is $r' \in Q'$ such that $r\ Z\ r'$ and $\alpha \in E'^*(q', r')$,
- if $q \in I$ then $q' \in I'$ and if $q \in F$ then $q' \in F'$.

Note: $dom(Z)$ and $ran(Z)$ respectively denote the domain of Z and the range of Z. If there is a simulation Z of $\mathcal{A}$ by $\mathcal{A}'$ then we write $\mathcal{A} \longleftarrow \mathcal{A}'$. We define a binary relation $Z \subseteq Q \times Q'$ to be a bisimulation between $\mathcal{A}$ and $\mathcal{A}'$, denoted Z: $\mathcal{A} \longleftrightarrow \mathcal{A}'$, iff Z: $\mathcal{A} \longleftarrow \mathcal{A}'$ and Z^{-1}: $\mathcal{A}' \longleftarrow \mathcal{A}$. If there is a bisimulation between $\mathcal{A}$ and $\mathcal{A}'$ then we write $\mathcal{A} \longleftrightarrow \mathcal{A}'$.

3 Composition of Services

3.1 Formal definitions

Let $R \in \{\sqsubseteq, \equiv, \longleftarrow, \longleftrightarrow\}$. The **valuation decision (VD)** problem for R is defined by:

- input: a strongly asynchronous finite automaton $\mathcal{A} = (Q, E, I, F)$ and ϵ-free Boolean automata $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1)$, ..., $\mathcal{A}_n = (Q_n, E_n, I_n, F_n)$,
- output: check if there is a valuation V such that $\mathcal{A}\ R\ (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$.

The **Boolean formula decision (BFD)** problem for R is defined by:

- input: a strongly asynchronous finite automaton $\mathcal{A} = (Q, E, I, F)$, ϵ-free Boolean automata $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1)$, ..., $\mathcal{A}_n = (Q_n, E_n, I_n, F_n)$ and a Boolean formula ϕ,
- output: check if for all valuations V, $\mathcal{A}\ R\ (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ iff $\widehat{V}(\phi) = 1$.

The **Boolean formula synthesis (BFS)** problem for R is defined by:

- input: a strongly asynchronous finite automaton $\mathcal{A} = (Q, E, I, F)$ and ϵ-free Boolean automata $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1)$, ..., $\mathcal{A}_n = (Q_n, E_n, I_n, F_n)$,
- output: find out a Boolean formula ϕ such that for all valuations V, $\mathcal{A}\ R\ (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ iff $\widehat{V}(\phi) = 1$.

The questions we are interested in are: Is there a valuation of these services such that the desired composition is possible (VD problem)? How to compute a boolean formula ϕ over these predicates that is true iff the desired composition is possible (BFS problem)?

3.2 Motivating Example

Consider three services S_1, S_2, S_3.

Service S_1 provides an on-line booking for European football match places that will take place in Madrid. This service is modeled by the automaton $\mathcal{A}_1$ in Fig. 4.

When S_1 receives a request for a football place (*ticket_req*) booking, it checks whether there is available places having the desired price (action *c_v*). This check can be done if and only if the request is provided by a non black listed supporter (predicate $\neg p_{black}$). If there is no available place, S_1 informs the requester by the

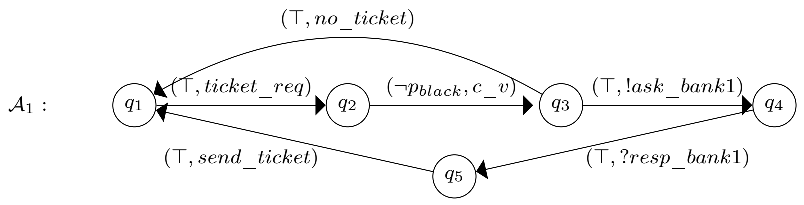


Figure 4. Boolean automaton $\mathcal{A}_1$

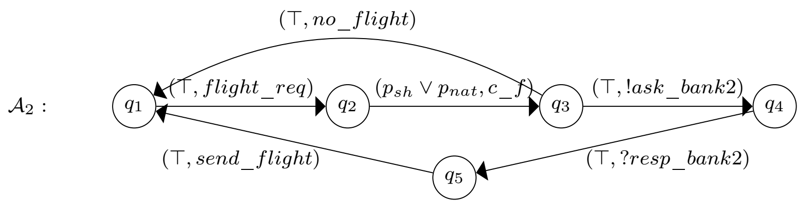


Figure 5. Boolean automaton $\mathcal{A}_2$

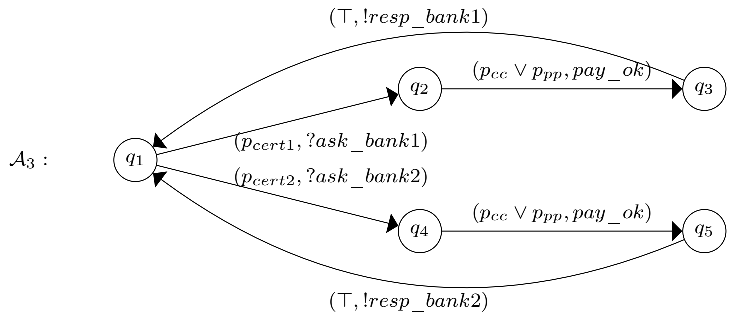

$\mathcal{A}_3$:
$(\top, !resp\_bank1)$
$(p_{cc} \vee p_{pp}, pay\_ok)$
$(p_{cert1}, ?ask\_bank1)$
$(p_{cert2}, ?ask\_bank2)$
$(p_{cc} \vee p_{pp}, pay\_ok)$
$(\top, !resp\_bank2)$
q_1
q_2
q_3
q_4
q_5

Figure 6. Boolean automaton $\mathcal{A}_3$

action no_ticket. If there is a ticket, S_1 asks the bank service (S_3) for the ticket payment (actions $!ask\_bank1$). If the bank answers the payment is Ok (message $?resp\_bank1$), the ticket is sent to the requester (action $send\_ticket$).

Service S_2 sells flight tickets. It is modeled by the automaton $\mathcal{A}_2$ in Fig. 5.

S_2 works like S_1 but sells tickets only for people who do not need a Visa for coming to Spain, thus either people whose nationality is in a given list (predicate p_{nat}), or if the starting point of the flight is in the Shengen Space (predicate p_{sh}).

The **service** S_3 is the bank service modeled by the automaton $\mathcal{A}_3$ in Fig. 6.

S_3 accepts request from services having a security certificate (predicate p_{cert_i} means that S_i has such a certificate). Then, if the buyer has either a credit card (predicate p_{cc}) or a paypal account (predicate p_{pp}), and if the payment is OK (in order to not overload the example, we do not model this communicating point; we just encode it by the action pay_ok), the bank validates the payment to the related service.

As the reader can see, services can be composed to provide the intented service S (depicted in Fig. 7), with several conditions on the value of predicates $p_{nat}, p_{cert1},$, etc.

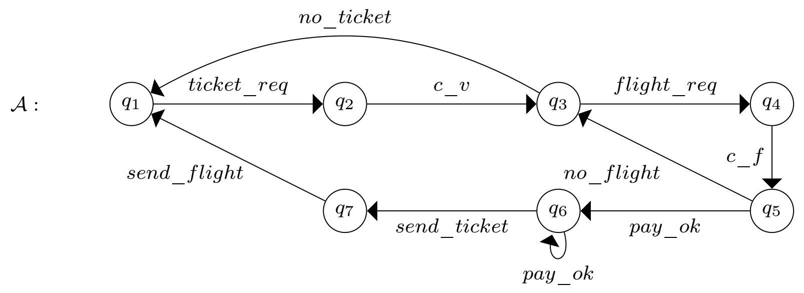


Figure 7. Automaton $\mathcal{A}$

4 Trace Inclusion and Trace Equivalence

Let us recall that given two finite automata $\mathcal{A}_1$ and $\mathcal{A}_2$, testing whether $\mathcal{A}_1 \sqsubseteq (\mathcal{A}_2$ can be done in $PSPACE$.

Upper bound

Let $\mathcal{A} = (Q, E, I, F)$ be a strongly asynchronous finite automaton and $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1)$, ..., $\mathcal{A}_n = (Q_n, E_n, I_n, F_n)$ be ϵ-free Boolean automata. We now define a deterministic algorithm which returns the value "accept" iff there is a valuation V such that $\mathcal{A} \sqsubseteq (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$:

```
(i) for all valuations $V$
    (a) compute $(\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$;
    (b) check if $\mathcal{A} \sqsubseteq (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$;
(ii) if one of these calls returns the value "accept" then return the value "accept"
     else return the value "reject";
```

Obviously, the deterministic algorithm above is exponential-space-bounded. Similarly, an exponential-space-bounded deterministic algorithm which returns the value "accept" iff there is a valuation V such that $\mathcal{A} \equiv (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ can be defined. As a result,

Proposition 4.1 *Let $R \in \{\sqsubseteq, \equiv\}$. The VD problem for R is in $EXPSPACE$.*

Similarly,

Proposition 4.2 *Let $R \in \{\sqsubseteq, \equiv\}$. The BFD problem for R is in $EXPSPACE$.*

Let $\mathcal{A} = (Q, E, I, F)$ be a strongly asynchronous finite automaton and $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1)$, ..., $\mathcal{A}_n = (Q_n, E_n, I_n, F_n)$ be ϵ-free Boolean automata. We now define a deterministic algorithm which returns a Boolean formula ϕ such that for all valuations V, $\mathcal{A} \sqsubseteq (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ iff $\widehat{V}(\phi) = 1$:

```
(i) $\phi := \bot$;
(ii) for all maximal consistent conjunctions $\psi$ of $P$-literals
    (a) compute the unique valuation $V$ such that $\widehat{V}(\psi) = 1$;
    (b) compute $(\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$;
    (c) if $\mathcal{A} \sqsubseteq (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ then $\phi := \phi \vee \psi$;
```

Obviously, the deterministic algorithm above is exponential-space-bounded. Similarly, an exponential-space-bounded deterministic algorithm which returns a Boolean formula ϕ such that for all valuations V, $\mathcal{A} \equiv (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ iff $\widehat{V}(\phi) = 1$ can be defined. As a result,

Proposition 4.3 *Let $R \in \{\sqsubseteq, \equiv\}$. The BFS problem for R is solvable by means of an exponential-space-bounded deterministic algorithm.*

Lower bound

To prove that the VD problem for $\sqsubseteq$ is $EXPSPACE$-hard, we give a polynomial time reduction of the universality problem for regular expressions with squaring, which is known to be $EXPSPACE$-hard [13], to the VD problem for $\sqsubseteq$. The set of all regular expressions with squaring (with typical members denoted exp, exp', ...) is defined by:

$$exp ::= \alpha \mid \epsilon \mid (exp \circ exp) \mid (exp \cup exp) \mid exp^+ \mid exp^2.$$

The number of occurrences of operators ϵ, $\circ$, $\cup$, $^+$ and ^{2} in regular expression exp with squaring is denoted $op(exp)$. Every regular expression exp with squaring gives rise to a language denoted $lang(exp)$ in the usual way. Let σ_1, σ_2, ... be an enumeration of Σ_s. Given a regular expression exp with squaring, let $n = 2 \times op(exp)$. Let $\mathcal{A} = (Q, E, I, F)$ be the strongly asynchronous finite automaton defined as follows: $Q = \{q\}$, E is the function from $Q \times Q$ into the set of all finite subsets of $\mathcal{B}(P) \times \Sigma_a$ such that for all $q, r \in Q$, for all $\phi \in \mathcal{B}(P)$, for all $\alpha \in \Sigma_a$, $(\phi, \alpha) \in E(q, r)$ iff $\phi = \top$, $I = \{q\}$ and $F = \{q\}$. For all positive integers i, if $i \leq n$ then let $\mathcal{A}_i = (Q_i, E_i, I_i, F_i)$ be the ϵ-free Boolean automaton defined as follows: $Q_i = \{q_{1i}, q_{2i}\}$, E_i is the function from $Q_i \times Q_i$ into the set of all finite subsets of $\mathcal{B}(P) \times (\Sigma_a \cup (\{?, !\} \times \Sigma_s))$ such that for all $q, r \in Q_i$, for all $\phi \in \mathcal{B}(P)$,

- for all $\alpha \in \Sigma_a$, $(\phi, \alpha) \notin E_i(q, r)$,
- for all $x \in \{?, !\}$, for all $\sigma \in \Sigma_s$, $(\phi, x\sigma) \in E_i(q, r)$ iff $\phi = \top$, $x = ?$, $\sigma = \sigma_i$, $q = q_{1i}$ and $r = q_{2i}$ or $\phi = \top$, $x = !$, $\sigma = \sigma_i$, $q = q_{2i}$ and $r = q_{1i}$,

$I_i = \{q_{1i}\}$ and $F_i = \{q_{2i}\}$. Let $\mathcal{A}_0 = (Q_0, E_0, I_0, F_0)$ be the ϵ-free Boolean automaton defined by induction on exp as follows:

Basis: Case $exp = \alpha$**.** In this case, $\mathcal{A}_0 = (Q_0, E_0, I_0, F_0)$ is defined as follows: $Q_0 = \{q_I, q_F\}$, E_0 is the function from $Q_0 \times Q_0$ into the set of all finite subsets of $\mathcal{B}(P) \times (\Sigma_a \cup (\{?, !\} \times \Sigma_s))$ such that for all $q, r \in Q_0$, for all $\phi \in \mathcal{B}(P)$,
- for all $\beta \in \Sigma_a$, $(\phi, \beta) \in E_0(q, r)$ iff $q = q_I$, $r = q_F$, $\phi = \top$ and $\beta = \alpha$,
- for all $x \in \{?, !\}$, for all $\sigma \in \Sigma_s$, $(\phi, x\sigma) \notin E_0(q, r)$,

$I_0 = \{q_I\}$ and $F_0 = \{q_F\}$. The Boolean automaton $\mathcal{A}_0$ is represented in Fig. 8. The reader may easily verify that for all valuations V, $lang(exp) = tr(\mathcal{A}_0^V)$. Remark that $0 = n$. Note also that I_0 and F_0 are singletons.

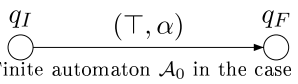


Figure 8. Finite automaton $\mathcal{A}_0$ in the case $exp = \alpha$.

Hypothesis: exp' and exp'' are regular expressions with squaring such that there is an ϵ-free Boolean automaton $\mathcal{A}'_0 = (Q'_0, E'_0, I'_0, F'_0)$ such that for all valuations V, $lang(exp') = tr((\mathcal{A}'_0 \otimes \mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_{n'})^V)$ where $n' = 2 \times op(exp')$ and there is an ϵ-free Boolean automaton $\mathcal{A}''_0 = (Q''_0, E''_0, I''_0, F''_0)$ such that for all valuations V, $lang(exp'') = tr((\mathcal{A}''_0 \otimes \mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_{n''})^V)$ where $n'' = q \times op(exp'')$. We also assume that I'_0, F'_0, I''_0 and F''_0 are singletons.

Step: The cases $exp = \epsilon$, $exp = exp' \circ exp''$, $exp = exp' \cup exp''$ and $exp = exp'^+$ are similar.

Case $exp = exp'^2$**.** In this case, $\mathcal{A}_0 = (Q_0, E_0, I_0, F_0)$ is defined as follows: $Q_0 = Q'_0 \cup \{q_I, q, r, q_F\}$, E_0 is the function from $Q_0 \times Q_0$ into the set of all finite subsets of $\mathcal{B}(P) \times (\Sigma_a \cup (\{?, !\} \times \Sigma_s))$ such that for all $s, t \in Q_0$, for all $\phi \in \mathcal{B}(P)$,

- for all $\beta \in \Sigma_a$, $(\phi, \beta) \in E_0(s,t)$ iff $s, t \in Q'_0$ and $(\phi, \beta) \in E'_0(s,t)$,
- for all $x \in \{?, !\}$, for all $\sigma \in \Sigma_s$, $(\phi, x\sigma) \in E_0(s,t)$ iff $s, t \in Q'_0$ and $(\phi, x\sigma) \in E'_0(s,t)$ or $s = q_I$, $t = q'_I$, $\phi = \top$, $x = !$ and $\sigma = \sigma_{n'+1}$ or $s = q'_F$, $t = q$, $\phi = \top$, $x = !$ and $\sigma = \sigma_{n'+2}$ or $s = q$, $t = q_I$, $\phi = \top$, $x = ?$ and $\sigma = \sigma_{n'+1}$ or $s = q'_F$, $t = r$, $\phi = \top$, $x = ?$ and $\sigma = \sigma_{n'+2}$ or $s = r$, $t = q_F$, $\phi = \top$, $x = ?$ and $\sigma = \sigma_{n'+1}$,

$I_0 = \{q_I\}$ and $F_0 = \{q_F\}$. The Boolean automaton $\mathcal{A}_0$ is represented in Fig. 9. The reader may easily verify that for all valuations V, $lang(exp) = tr((\mathcal{A}_0 \otimes \mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_{n'} \otimes \mathcal{A}_{n'+1} \otimes \mathcal{A}_{n'+2})^V)$. Remark that $n' + 2 = n$. Note also that I_0 and F_0 are singletons.

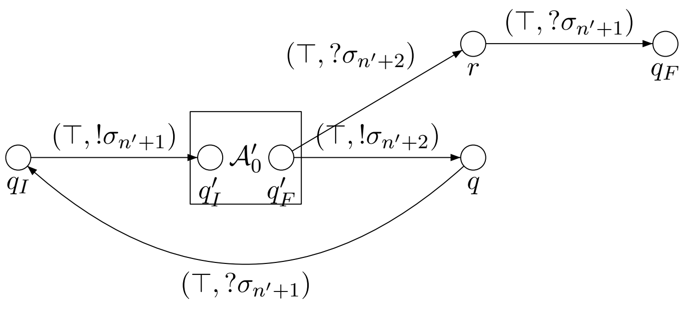


Figure 9. Finite automaton $\mathcal{A}_0$ in the case $exp = exp'^2$.

The reader may easily verify that $lang(exp) = \Sigma_a^*$ iff there is a valuation V such that $\mathcal{A} \sqsubseteq (\mathcal{A}_0 \otimes \mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$. Similarly, the reader may easily verify that $lang(exp) = \Sigma_a^*$ iff there is a valuation V such that $\mathcal{A} \equiv (\mathcal{A}_0 \otimes \mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$. As a result,

Proposition 4.4 *Let $R \in \{\sqsubseteq, \equiv\}$. The VD problem for R is EXPSPACE-hard.*

Similarly,

Proposition 4.5 *Let $R \in \{\sqsubseteq, \equiv\}$. The BFD problem for R is EXPSPACE-hard.*

According to Meyer and Stockmeyer [13], for all deterministic Turing machines M solving the universality problem for regular expressions with squaring, there exists a constant $c > 1$ such that M needs at least space c^n on some input of length n for infinitely many n. Suppose that there is a deterministic algorithm f solving the BFS problem for R and such that for all constants $c > 1$, f needs at least space c^n on some input of length n for finitely many n only. Let M_f be the deterministic Turing machine that behaves as follows given a regular expression *exp* with squaring:

(i) M_f computes $n = 2 \times op(exp)$;

(ii) M_f computes the strongly asynchronous finite automaton $\mathcal{A} = (Q, E, I, F)$ defined as above;

(iii) for all positive integers i, if $i \leq n$ then M_f computes the ϵ-free Boolean automaton $\mathcal{A}_i = (Q_i, E_i, I_i, F_i)$ defined as above;

(iv) M_f computes the ϵ-free Boolean automaton $\mathcal{A}_0 = (Q_0, E_0, I_0, F_0)$ defined by induction on *exp* as above;

(v) M_f simulates f on input $\mathcal{A}$ and $\mathcal{A}_0$ and $\mathcal{A}_1$, ..., $\mathcal{A}_n$ until it is about to return a value ϕ_f;

(vi) if ϕ_f is a Boolean tautology then return the value "accept" else return the value "reject";

Obviously, M_f solves the universality problem for regular expressions with squaring and for all constants $c > 1$ M_f needs at least space c^n on some input of length n for finitely many n only: a contradiction. As a result,

Proposition 4.6 *Let $R \in \{\sqsubseteq, \equiv\}$. For all deterministic algorithms f solving the BFS problem for R, there exist a constant $c > 1$, such that f needs at least space c^n on some input of length n for infinitely many n.*

5 Simulation and Bisimulation

Let us recall that given two finite automata $\mathcal{A}_1$ and $\mathcal{A}_2$, testing whether $\mathcal{A}_1$ simulates $\mathcal{A}_2$ can be done in polynomial time.

Upper bound

Let $\mathcal{A} = (Q, E, I, F)$ be a strongly asynchronous finite automaton and $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1)$, ..., $\mathcal{A}_n = (Q_n, E_n, I_n, F_n)$ be ϵ-free Boolean automata. We now define a deterministic algorithm which returns the value "accept" iff there is a valuation V such that $\mathcal{A} \longleftarrow (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$:

(i) for all valuations V
    (a) compute $(\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$;
    (b) check if $\mathcal{A} \longleftarrow (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$;

(ii) if one of these calls returns the value "accept" then return the value "accept" else return the value "reject";

Obviously, the deterministic algorithm above is exponential-time-bounded. Similarly, an exponential-time-bounded deterministic algorithm which returns the value "accept" iff there is a valuation V such that $\mathcal{A} \longleftrightarrow (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ can be defined. As a result,

Proposition 5.1 *Let $R \in \{\longleftarrow, \longleftrightarrow\}$. The VD problem for R is in EXPTIME.*

Let $\mathcal{A} = (Q, E, I, F)$ be a strongly asynchronous finite automaton, $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1)$, ..., $\mathcal{A}_n = (Q_n, E_n, I_n, F_n)$ be ϵ-free Boolean automata and ϕ be a Boolean formula. We now define a deterministic algorithm which returns the value "accept" iff for all valuations V, $\mathcal{A} \longleftarrow (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ iff $\widehat{V}(\phi) = 1$:

(i) for all valuations V
  (a) compute $(\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$;
  (b) compute $\widehat{V}(\phi)$;
  (c) check if $\mathcal{A} \longleftarrow (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ iff $\widehat{V}(\phi) = 1$;

(ii) if all these calls returns the value "accept" then return the value "accept" else return the value "reject".

Obviously, the deterministic algorithm above is exponential-time-bounded. Similarly, an exponential-time-bounded deterministic algorithm which returns the value "accept" iff for all valuations V, $\mathcal{A} \longleftrightarrow (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ iff $\widehat{V}(\phi) = 1$ can be defined.

Proposition 5.2 *Let $R \in \{\longleftarrow, \longleftrightarrow\}$. The BFD problem for R is in EXPTIME.*

Let $\mathcal{A} = (Q, E, I, F)$ be a strongly asynchronous finite automaton and $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1)$, ..., $\mathcal{A}_n = (Q_n, E_n, I_n, F_n)$ be ϵ-free Boolean automata. We now define a deterministic algorithm which returns a Boolean formula ϕ such that for all valuations V, $\mathcal{A} \longleftarrow (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ iff $\widehat{V}(\phi) = 1$:

(i) $\phi := \bot$;

(ii) for all maximal consistent conjunctions ψ of P-literals
  (a) compute the unique valuation V such that $\widehat{V}(\psi) = 1$;
  (b) compute $(\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$;
  (c) if $\mathcal{A} \longleftarrow (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ then $\phi := \phi \vee \psi$;

Obviously, the deterministic algorithm above is exponential-time-bounded. Similarly, an exponential-time-bounded deterministic algorithm which returns a Boolean formula ϕ such that for all valuations V, $\mathcal{A} \longleftrightarrow (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$ iff $\widehat{V}(\phi) = 1$ can be defined. As a result,

Proposition 5.3 *Let $R \in \{\longleftarrow, \longleftrightarrow\}$. The BFS problem for R is solvable by means of an exponential-time-bounded deterministic algorithm.*

Lower bound

By giving a polynomial time reduction of the simulation problem of a strongly asynchronous finite automaton by means of a product of strongly asynchronous finite automata, which is known to be *EXPTIME*-hard [15], to the VD problem for $\longleftarrow$, we prove that the VD problem for $\longleftarrow$ is *EXPTIME*-hard. Given a strongly asynchronous finite automaton $\mathcal{A} = (Q, E, I, F)$ and strongly asynchronous finite automata $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1), \ldots, \mathcal{A}_n = (Q_n, E_n, I_n, F_n)$, let $\mathcal{A}'_1 = (Q'_1, E'_1, I'_1, F'_1)$, ..., $\mathcal{A}'_n = (Q'_n, E'_n, I'_n, F'_n)$ be the ϵ-free Boolean automata defined as follows: $Q'_i = Q_i$, E'_i is the function from $Q'_i \times Q'_i$ into the set of all finite subsets of $\mathcal{B}(P) \times (\Sigma_a \cup (\{?, !\} \times \Sigma_s))$ such that for all $q, r \in Q'_i$, for all $\phi \in \mathcal{B}(P)$,

- for all $\alpha \in \Sigma_a$, $(\phi, \alpha) \in E'_i(q, r)$ iff $\phi = \top$ and $\alpha \in E_i(q, r)$,

- for all $x \in \{?, !\}$, for all $\sigma \in \Sigma_s$, $(\phi, x\sigma) \notin E'_i(q, r)$,

$I'_i = I_i$ and $F'_i = F_i$. The reader may easily verify that $\mathcal{A} \longleftarrow \mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n$ iff there is a valuation V such that $\mathcal{A} \longleftarrow (\mathcal{A}'_1 \otimes \ldots \otimes \mathcal{A}'_n)^V$. As a result,

Proposition 5.4 *The VD problem for $\longleftarrow$ is EXPTIME-hard.*

Similarly,

Proposition 5.5 *The BFD problem for $\longleftarrow$ is EXPTIME-hard.*

Proof We prove that the Boolean formula decision problem for $\longleftarrow$ is *EXPTIME*-hard by giving a polynomial time reduction of the simulation problem of a strongly asynchronous finite automaton by means of a product of strongly asynchronous finite automata, which is known to be *EXPTIME*-hard [15], to the Boolean formula decision problem for $\longleftarrow$. Given a strongly asynchronous finite automaton $\mathcal{A} = (Q, E, I, F)$ and strongly asynchronous finite automata $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1)$, ..., $\mathcal{A}_n = (Q_n, E_n, I_n, F_n)$, let $\mathcal{A}'_1 = (Q'_1, E'_1, I'_1, F'_1)$, ..., $\mathcal{A}'_n = (Q'_n, E'_n, I'_n, F'_n)$ be the ϵ-free Boolean automaton defined as above and $\phi = \top$. Suppose that $\mathcal{A} \longleftarrow \mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n$. Hence, there is a simulation Z of $\mathcal{A}$ by $\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n$. Let V be a valuation. Obviously, Z is a simulation of $\mathcal{A}$ by $(\mathcal{A}'_1 \otimes \ldots \otimes \mathcal{A}'_n)^V$. Therefore, for all valuations V, $\mathcal{A} \longleftarrow (\mathcal{A}'_1 \otimes \ldots \otimes \mathcal{A}'_n)^V$ iff $\widehat{V}(\phi) = 1$. Reciprocally, suppose that for all valuations V, $\mathcal{A} \longleftarrow (\mathcal{A}'_1 \otimes \ldots \otimes \mathcal{A}'_n)^V$ iff $\widehat{V}(\phi) = 1$. Let V be a valuation. Thus, there is a simulation Z of $\mathcal{A}$ by $(\mathcal{A}'_1 \otimes \ldots \otimes \mathcal{A}'_n)^V$. Obviously, Z is a simulation of $\mathcal{A}$ by $\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n$. Consequently, $\mathcal{A} \longleftarrow \mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n$.$\square$

We prove that the VD problem for $\longleftrightarrow$ is *PSPACE*-hard by giving a polynomial time reduction of the acceptance problem of a linear-space-bounded deterministic Turing machine, which is known to be *PSPACE*-hard, to the VD problem for $\longleftrightarrow$. Let *acc*, α_1, α_2, ... be an enumeration of Σ_a. Given a linear-space-bounded deterministic Turing machine $M = (Q_M, q^0_M, q^1_M, \Sigma_M, \delta_M)$ and a word $\boldsymbol{w} \in \Sigma^*_M$, let n be the length of $\boldsymbol{w}$. Let $\Sigma_f = \{acc, \alpha_1, \ldots, \alpha_n\}$. Let $\mathcal{A} = (Q, E, I, F)$ be the strongly asynchronous finite automaton defined as follows: $Q = (Q_M \times \{1, \ldots, n\}) \cup \{\bot\}$, E is the function from $Q \times Q$ into the set of all finite subsets of $\mathcal{B}(P) \times \Sigma_f$ such that for all $(q, i), (r, j) \in Q$, for all $\phi \in \mathcal{B}(P)$,

- for all $\alpha \in \Sigma_f$, $(\phi, \alpha) \in E((q, i), (r, j))$ iff $\phi = \top$, $\alpha = \alpha_i$ and there are $u, v \in \Sigma_M$, there is $d \in \{-1, +1\}$ such that $\delta_M(q, u, r, v, d)$ is defined and $j = i + d$ or $\phi = \top$, $\alpha = acc$, $q = q^1_M$, $r = q^1_M$ and $j = i$,
- for all $\alpha \in \Sigma_f$, $(\phi, \alpha) \in E((q, i), \bot)$ iff $\phi = \top$, $\alpha \neq acc$ and $\alpha \neq \alpha_i$ or for all $r \in Q_M$, for all $u, v \in \Sigma_M$, for all $d \in \{-1, +1\}$, $\delta_M(q, u, r, v, d)$ is not defined,
- for all $\alpha \in \Sigma_f$, $(\phi, \alpha) \notin E(\bot, (r, j))$,
- for all $\alpha \in \Sigma_f$, $(\phi, \alpha) \in E(\bot, \bot)$ iff $\phi = \top$ and $\alpha \neq acc$,

$I = \{(q^0_M, 1)\}$ and $F = \emptyset$. Let $\mathcal{A}_1 = (Q_1, E_1, I_1, F_1)$, ..., $\mathcal{A}_n = (Q_n, E_n, I_n, F_n)$ be the ϵ-free Boolean automata defined as follows: $Q_i = (Q_M \times \Sigma_M) \cup \{\bot_i\}$, E_i is the function from $Q_i \times Q_i$ into the set of all finite subsets of $\mathcal{B}(P) \times (\Sigma_f \cup (\{?, !\} \times \Sigma_s))$ such that for all $(q, u), (r, v) \in Q_i$, for all $\phi \in \mathcal{B}(P)$,

- for all $\alpha \in \Sigma_f$, $(\phi, \alpha) \in E_i((q, u), (r, v))$ iff $\phi = \top$, $\alpha = \alpha_i$ and there is $d \in \{-1, +1\}$ such that $\delta_M(q, u, r, v, d)$ is defined,

- for all $\alpha \in \Sigma_f$, $(\phi, \alpha) \in E_i((q,u), \perp_i)$ iff $\phi = \top$, $\alpha \neq acc$ and $\alpha \neq \alpha_i$ or for all $r \in Q_M$, for all $v \in \Sigma_M$, for all $d \in \{-1, +1\}$, $\delta_M(q,u,r,v,d)$ is not defined,
- for all $\alpha \in \Sigma_f$, $(\phi, \alpha) \notin E_i(\perp_i, (r,v))$,
- for all $\alpha \in \Sigma_f$, $(\phi, \alpha) \in E_i(\perp_i, \perp_i)$ iff $\phi = \top$ and $\alpha \neq acc$,
- for all $x \in \{?, !\}$, for all $\sigma \in \Sigma_s$, $(\phi, x\sigma) \notin E_i((q,u),(r,v))$,
- for all $x \in \{?, !\}$, for all $\sigma \in \Sigma_s$, $(\phi, x\sigma) \notin E_i((q,u), \perp_i)$,
- for all $x \in \{?, !\}$, for all $\sigma \in \Sigma_s$, $(\phi, x\sigma) \notin E_i(\perp_i, (r,v))$,
- for all $x \in \{?, !\}$, for all $\sigma \in \Sigma_s$, $(\phi, x\sigma) \notin E_i(\perp_i, \perp_i)$,

$I_i = \{(q_M^0, w_i)\}$ and $F_i = \emptyset$. Suppose that M does not accept $\boldsymbol{w}$. Let $Z \subseteq Q \times (Q_1 \times \ldots \times Q_n)$ be the binary relation such that

- $(q,i)\ Z\ ((q_1,u_1), \ldots, (q_n,u_n))$ iff $q = q_i$ and $(q_M^0, 1, \boldsymbol{w}) \Longrightarrow_M^* (q, i, u_1 \ldots u_n)$,
- $\cdot\ Z\ (\cdot, \ldots, \cdot, \perp_i, \cdot, \ldots, \cdot)$,
- $\perp\ Z\ (\cdot, \ldots, \cdot)$.

The reader may easily verify that there is a valuation V such that Z is a bisimulation between $\mathcal{A}$ and $(\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$. Hence, there is a valuation V such that $\mathcal{A} \longleftrightarrow (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$. Reciprocally, suppose that there is a valuation V such that $\mathcal{A} \longleftrightarrow (\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$. Therefore, there is a bisimulation Z between $\mathcal{A}$ and $(\mathcal{A}_1 \otimes \ldots \otimes \mathcal{A}_n)^V$. Obviously, M does not accept $\boldsymbol{w}$.

Proposition 5.6 *The VD problem for $\longleftrightarrow$ is PSPACE-hard. The BFD problem for $\longleftrightarrow$ is PSPACE-hard.*

6 Conclusion

This paper presented different new complexity results for the VD problem and the BFD problem within the framework of service composition with constraints. To sum up, the results are given in snapshot of our work below.

R	valuation decision problem	Boolean formula decision problem
$\equiv$	*EXPSPACE*-complete	*EXPSPACE*-complete
$\subseteq$	*EXPSPACE*-complete	*EXPSPACE*-complete
$\leftarrow$	*EXPTIME*-complete	*EXPTIME*-complete
$\leftrightarrow$	*PSPACE*-hard in *EXPTIME*	*PSPACE*-hard in *EXPTIME*

As pointed out by the above table, a still open question is to evaluate the exact complexity of the valuation decision problem for $\longleftrightarrow$ and the Boolean formula decision problem for $\longleftrightarrow$: are they in *PSPACE* or are they *EXPTIME*-hard? Moreover, we focused on the identification of a relevant abstraction to manage conditional actions in the service composition problem. In the future, we intend to explore practical algorithmic approaches to handle the Boolean formula synthesis problem.

References

[1] Berardi, D., D. Calvanese, G. De Giacomo, R. Hull, M. Mecella: *Automatic composition of transition-based semantic Web services with messaging.* In: Very Large Data Bases. ACM (2005) 613–624.

[2] Berardi, D., D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella: *Automatic services composition based on behavioral descriptions.* Int. Journal of Cooperative Information Systems **14** (2005) 333–376.

[3] Berardi, D., F. Cheikh, G. De Giacomo, F. Patrizi: *Automatic service composition via simulation.* Int. Journal of Foundations of Computer Science **19** (2008) 429–451.

[4] Berardi, D., M. Pistore, P. Traverso: *Automatic Web service composition by on-the-fly belief space search.* In: Proceedings of ICAPS'06, (2006) 358–361.

[5] Foster, H.: *A Rigorous Approach to Engineering Web Service Compositions.* Thesis of the University of London (2006).

[6] Foster, H., S. Uchitel, J. Magee, J. Kramer: *WS-Engineer: a model-based approach to engineering Web service compositions and choreography.* In: Test and Analysis of Web Services. Springer (2007) 87–119.

[7] Fu, X., T. Bultan, J. Su: *Analysis of interacting BPEL Web services.* In: Int. World Wide Web Conference. ACM (2004) 621–630.

[8] Héam, P.-C., O. Kouchnarenko, J. Voinot: *How to handle QoS aspects in Web services substitutivity verification.* In: Enabling Technologies: Infrastructure for Collaborative Enterprises. IEEE (2007) 333–338.

[9] Hoffman, J., P. Bertoli, M. Pistore: *Web service composition as planning, revisited.* In: Proceeedings of AAAI'07 AAAI (2007) 1013–1018.

[10] Laroussinie, F., Ph. Schnoebelen: *The state explosion problem from trace to bisimulation equivalence.* In: Foundations of Software Science and Computational Structures. Springer (2007) 192–207.

[11] Marconi, A., M. Pistore, P. Poccianti, P. Traverso: *Automated Web service composition at work: the amazon/mps case study* In: Proceedings of ICWS'07, IEEE, (2007) 767–774.

[12] Marconi, A., M. Pistore, P. Traverso: *Automated composition of Web services: the astro approach* In: IEEE Data Engineering Bulletin **31** (2008) 23–26.

[13] Meyer, A., L. Stockmeyer: *The equivalence problem for regular expressions with squaring requires exponential space.* In: Switching and Automata Theory. IEEE (1972) 125–129.

[14] Mitra, S., R.J. Kumar, S. Basu: *Automates CHoregrapher Synthesis for Web service composition using I/O Automata.* In: Proceedings of ICWS'07 IEEE (2007) 364–371.

[15] Muscholl, A., I. Walukiewicz: *A lower bound on Web services composition.* In: Foundations of Software Science and Computational Structures. Springer (2007) 274–286.

[16] Pathak, J., S. Basu, R. Lutz, V. Honavar: *Parallel web service composition in moscoe: A choregraphy based approach.* In: Proceedings of ECOWS'96, IEEE (2006) 3–12.

[17] Pistore, M., A. Marconi, P. Bertoli, P. Traverso: *Automated composition of Web services by planning at the knowledge level.* In: Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (2005) 1252–1259.

[18] Pistore, M., P. Traverso, P. Bertoli: *Automated composition of Web services by planning in asynchronous domains.* In: Proceedings of ICAPS'05, AAAI, (2005) 2–12.

[19] Pistore, M., P. Traverso, P. Bertoli, A. Marconi: *Automated synthesis of composite BPEL4WS web services.*. In: Proceedings of ICWS'05, IEEE, (2005) 293–301.

[20] Rabinovich, A.: *Complexity of equivalence problems for concurrent systems of finite agents.* In: Information and Computation (1997). volume 139, 111–129.

[21] Sardina, S., F. Patrizi, G. De Giacomo: *Behaviour composition in the presence of failure.* In: Proceedings of KR'08, AAAI, (2008). 640–650.

[22] Sawa, D.: *Equivalence Checking of Non-flat Systems Is EXPTIME-Hard.* In: Int. Conf. on Concurrency Theory. Springer (2003), 237–250.

[23] Singh, M., M. Huhns: *Service-Oriented Computing. Semantics, Process, Agents.* Wiley (2005).

[24] Trainotti, M., M. Pistore, G. Calabrese, G. Zacco, G. Lucchese, F. Barbon, P. Bertoli, P. Traverso: *Sipporting composition and execution of Web services.* In: Proceedings of ICSOC'05, Springer, (2005). 495–501.

Using Assertions to Enhance the Correctness of Kmelia Components and their Assemblies

Pascal André, Gilles Ardourel, Christian Attiogbé, Arnaud Lanoix

LINA CNRS UMR 6241 - University of Nantes
2, rue de la Houssinière
F-44322 Nantes Cedex, France
Email: {FirstName.LastName}@univ-nantes.fr

Abstract

The Kmelia component model is an abstract formal component model based on services. It is dedicated to the specification and development of correct components. This work enriches the Kmelia language to allow the description of data, expressions and assertions when specifying components and services. The objective is to enable the use of assertions in Kmelia in order to support expressive service descriptions, to support client/supplier contracts with pre/post-conditions, and to enhance formal analysis of component-based system. Assertions are used to perfom analysis of services, component assemblies and service compositions. We illustrate the work with the verification of consistency properties involving data at component and assembly levels.

Keywords: Component, Assembly, Datatype, Assertions, Property Verification

1 Introduction

The Kmelia component model [7] is an abstract formal component model dedicated to the specification and development of correct components. A formal component model is mandatory to check various kind of properties for component-based software systems: correctness, liveness, safety; to find components and services in libraries according to their formal requirements; to refine models or to generate code. The key concepts of Kmelia are component, service, assembly and composition. One important feature is the use of services as first class entities. A service has a state, a dynamic behaviour which may include communication actions, an interface made of required and provided subservices. The composition of components is based on the interaction between their linked services. Linking components by their services in component assemblies establishes a bridge to service oriented abstract models.

In [7] we introduced the syntax and semantics for the core model and language. It has been incrementally enriched later. We mainly focused on the dynamic aspects of composition: interaction compatibility in [7] and component protocols with service composition in [6]. Following this incremental approach, we consider in this article

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

an enrichment of the data and expressions in the kmelia model and its impact on the language syntax, its semantics and the verification of properties. Our guiding objective is twofold: 1) enable the definition of assertions (with invariant, pre/post conditions, and properties of services, components, and compositions), 2) increase the expressiveness of the action statements so as to deal with real size case studies.

Assertions are useful (i) to define *contracts* [1] on services; contracts increase the confidence in assembly correctness and they are a pertinent information when looking for candidates for a required service, (ii) to ensure the consistency of components respecting the invariant. The actions implement a functional part of the services which should then be proved to be consistent with the contracts. Therefore the correctness verification aspects of the Kmelia model is enhanced.

Motivations. Modelling real life systems requires the use of data types to handle states, actions and property descriptions. The state of the art shows that most of the abstract components models [4,13,24,12]. They enable various verifications of the interaction correctness but they lack expressiveness on the data types and do not provide assertions mechanisms and the related verification rules. As an example, in Wright the dynamic part based on CSP is largely detailed (specification and verification) while the data part is minor [4]. In the proposal of [22] the data types are defined using algebraic specifications, which are convenient to marry with the symbolic model checking of state transition systems but this proposal does not deal with contracts and assertions.

Contribution. In this work, we enrich the Kmelia model with data and assertions at the service and composition levels in order to deal with safe services, component consistency and assembly contracts. *First*, the Kmelia language is enriched with data and assertions so as to cover in an homogeneous way structural, dynamic and functional correctness with respect to assertions. *Second*, we deal with state space visibility and access through different levels of nested components; in addition to service promotion we define variable promotions and the related *access rules* from component state in *component compositions*. *Last*, feasibility of proving component correctness using the assertions is presented. We show how structural correctness is verified and how the associated properties are expressed with the new data language.

To design it , we have established a trade off [2] between the desired expressiveness of our language and the verification concerns. To avoid the separation of analysis tools and to work on the same abstract model, we advocate for an approach where both data and dynamic part are integrated in a unique Kmelia language.

The article is structured as follows. Section 2 gives an overview of the Kmelia abstract model and introduces its new features. In Section 3 a working example is introduced to illustrate the use of data and assertions. The formal analysis issue is treated in Section 4; we present various analysis to be performed and we focus on component consistency and on checking assembly links. The formal analysis are based on the formal descriptions of Section 2 also many details are omitted in this paper. Section 5 concludes the article and draws some discussions and perspectives.

[1] Our contract definitions are related to classical results of works such as *design-by-contracts* [20].

[2] We thought to encapsulate statements from other formal data languages such as Z, B, OCL or CASL, with the idea to reuse existing tool supports for checking syntax and properties. That approach was not convincing due to a lack of expressiveness, or a weak tool support or integration problems.

2 The Kmelia Model and its new Features

We enriched the Kmelia language of [7] to allow the description of datatypes, expressions and first order logic predicates. This section revisits the Kmelia model, focusing on its new features.

2.1 Data types and expressions

We enrich the Kmelia language by designing a small but expressive data language. This enables us to deal homogeneously with the expression of the properties related to the component level and to the composition level.

Basic types such as Integer, Boolean, Char, String with their usual operators and semantics are permitted. Abstract data types like record, enumeration, range, arrays and sets are allowed in Kmelia. User-defined record types are built over the above *basic* types. Specific types and functions may be defined and imported from libraries. A Kmelia *expression* is built with constants, variables and elementary expressions built with standard arithmetic and logical operators. An assignment is made of a variable at the left hand side and an expression at the right hand side.

Assertions (pre-/post-conditions and invariants) are first order logic *predicates*. In a post-condition of a service, the keyword old is used to distinguish the before and after variable states. This is close to OCL's **pre** or Eiffel's **old** keywords. Guards in the service behaviour are also predicates. All the assertions must conform to an observability policy described in Section 2.3.

2.2 Components

A **component** is one element of a component type. A component is referenced with a variable typed using the component type; for example c:C where c is a variable and C a component type. The access to a state variable v of c is denoted c.v.

A **component type** C is a 8-tuple $\langle \mathcal{W}, \mathcal{A}, \mathcal{N}, \mathcal{M}, \mathcal{I}, \mathcal{D}, \nu, \mathcal{CS} \rangle$ with:

- $\mathcal{W} = \langle T, V, type, Inv, Init \rangle$ the state space where T is a set of types, V a set of variables, $type : V \rightarrow T$ the function that map variables to types, Inv an invariant defined on V and $Init$ the initialisation of the variables of V.
- $\mathcal{A}$ a finite set of elementary actions (based on the expressions).
- $\mathcal{N}$ a finite set of service names with $\mathcal{N}^P$ (provided services) and $\mathcal{N}^R$ (required services) two disjoint finite sets of names[3]: $\mathcal{N} = \mathcal{N}^P \uplus \mathcal{N}^R$.
- $\mathcal{M}$ a finite set of message names.
- $\mathcal{I} = \mathcal{I}^P \uplus \mathcal{I}^R$ the component interface which is the union of two disjoint finite sets of names $\mathcal{I}^P$ and $\mathcal{I}^R$ such that $\mathcal{I}^P \subseteq \mathcal{N}^P \wedge \mathcal{I}^R \subseteq \mathcal{N}^R$.
- $\mathcal{D}$ is the set of service descriptions with the disjoint provided services ($\mathcal{D}^P$) and required services ($\mathcal{D}^R$) sets: $\mathcal{D} = \mathcal{D}^P \uplus \mathcal{D}^R$.
- $\nu : \mathcal{N} \rightarrow \mathcal{D}$ is the function mapping service names to service descriptions. Moreover there is a projection of the $\mathcal{N}$ partition on its image by ν:
  $s \in \mathcal{N}^P \Rightarrow \nu(s) \in \mathcal{D}^P \wedge s \in \mathcal{N}^R \Rightarrow \nu(s) \in \mathcal{D}^R$

[3] $\uplus$ denotes the disjoint union of sets

- $\mathcal{C}S$ is a set of constraints related to the services of the interface of C in order to control the usage of the services.

Observability of the component state. To preserve the abstraction and encapsulation of components, the state of a component is accessed only through its provided services. Nevertheless to understand the specification of a service (i.e. its contract) we need to *observe* its context (an exposed part of its component state space). Similarly a composite component requires observable informations from its components. Therefore we define V^O as the subset of the **observable** variables of V. Consequently the state invariant Inv is composed of an observable (Inv^O defined on V^O) and a non-observable part. The notion of observability is also applied to service pre/post conditions with the specific rules described in section 2.3. Observability is a kind of visibility related to contracts.

2.3 Services

The behaviour of a component relies on the behaviours of its services which are a kind of concurrent processes. A service models a functionality *activated* by a call. An activated service runs its behaviour and shares the component state with other activated services of the same component. During its evolution a service may activate other services by calling them or communicate with them by messages. Due to dependencies and interactions between services, the actions of several activated services may interleave or synchronise. Only one action of an activated service may be observed at time. Formally a *service* s of a component with type C[4] is defined by a 3-tuple $\langle \mathcal{I}S, l\mathcal{W}, \mathcal{B}\rangle$ with:

- The service interface $\mathcal{I}S$ is defined by a 6-tuple $\langle \sigma, \mu, v\mathcal{W}, Pre, Post, \mathcal{D}I\rangle$ where
  - σ is the service signature $\langle name, param, ptype, Tres\rangle$ with $name \in \mathcal{N}$, $param$ a set of parameters, $ptype : param \rightarrow T$ the function mapping parameters to types and $Tres \in T$ the service result type;
  - $v\mathcal{W} = \langle vT, vV, vtype, vInv, vInit\rangle$ is a *virtual state space* with vT a set of types, vV a set of variables, $vtype : vV \rightarrow vT$ the function mapping context variables to types and $vInv$ an invariant defined on vV and $vInit$ the optional initialisation of the variables of vV;
  - μ is a set of message signatures $\langle mname, mparam, mptype\rangle$ where $mname \in \mathcal{M}$, $mparam$ and $mptype$ are similar to those of the service signature;
  - Pre is a pre-condition defined on the union of the variables in V, vV, and $param$: $V \cup vV \cup param$;
  - $Post$ is a post-condition defined on $V \cup vV \cup param \cup \{$ result $\}$, where the predefined result variable of type $Tres$ denotes the service result;
  - $\mathcal{D}I$ is the *service dependency*; it is composed by services on which the current service depends. $\mathcal{D}I$ is a 4-tuple $\langle sub, cal, req, int\rangle$ of disjoint sets where $sub \subseteq \mathcal{N}^P$ (resp. $cal \subseteq \mathcal{N}^R$, $req \subseteq \mathcal{N}^R$, $int \subseteq \mathcal{N}^P$) contains the provided services names (resp. the ones required from the caller, from any component or from the component itself) in the scope of s.

[4] and by extension a service of a component $c : C$

- $l\mathcal{W} = \langle lT, lV, ltype, lInv, lInit\rangle$ is the local state space where lT is a set of types, lV a set of local variables, $ltype : lV \rightarrow lT$ the function mapping local variables to types, $lInv$ a local state invariant defined on lV (mostly $lInv = true$) and $lInit$ the initialisation of the variables of lV.
- The behaviour $\mathcal{B}$ of a service s is an *extended labelled transition system* (eLTS) with state and transitions. The necessary details are given on the example of Section 3. The full background is provided online in references [7,6].

The state space $l\mathcal{W}$ local to a service is used only in the service behaviour $\mathcal{B}$ but not used in the assertions.

Virtual state spaces. A required service is an abstraction of a service provided by another component. Since that component is unknown when specifying the required service, it may be necessary to describe this “imaginary” component. We introduce the notion of a *virtual state space* $v\mathcal{W}$ in order to abstract the service context. For a *provided* service this virtual context is always empty.

Observability vs. service state space. Let s be a service of a component type C. The distinction between observable and non-observable variables of the component state space is revisited[5] according to the following table:

Service	Variables		Invariant	
state space	Observable part	Non-observable part	Observable part	Non-observable part
`Provided` s	V^O	V	Inv^O	Inv
`Required` s	vV	V	$vInv$	Inv

The pre-/post-conditions of s must respect the well-formedness rules related to the observable, non-observable and virtual contexts according to the following table:

Service	pre-condition		post-condition	
Assertions scope	Observable Pre^O	Non-observable Pre^{NO}	Observable $Post^O$	Non-observable $Post^{NO}$
`Provided` s	$V^O \cup param$	none	$V^O \cup param \cup \{$ result $\}$	$V \cup param \cup \{$ result $\}$
`Required` s	$vV \cup param$	$V \cup param$	$vV \cup param \cup \{$ result $\}$	none

Figure 1 summarises (in the context of a composition as described in Section 2.4) the relations between state spaces, observability and contracts. The boxes denote components (a, b) and compositions (c). The grey (resp. white) "funnel" denote provided (resp. required) services.

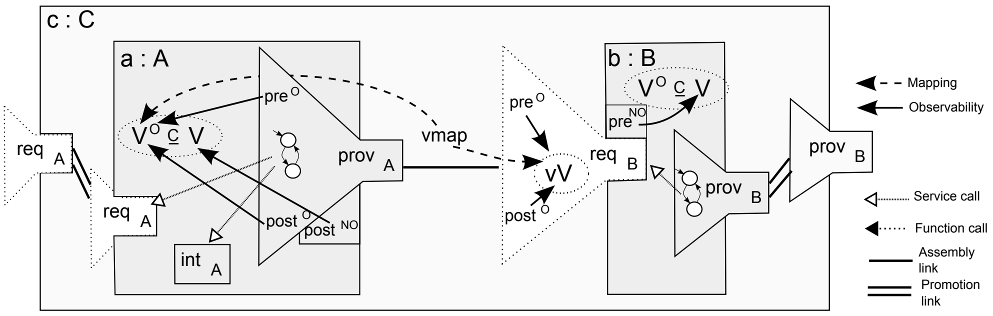


Fig. 1. State variables scope and assertion scope

[5] it is not a partition here because of the supplementary variables in *param* and *result*

The observable pre-/post-conditions of service $prov_A$ (resp. req_B) refer to the observable state V^O of a (resp. the virtual state vV of req_B). These conditions will be used to check the contracts implicitly supported by the assembly links and the composition links. In particular the virtual state vV of req_B should map with a subset of V of a. Non-observable pre-conditions (resp. post-conditions) are meaningless for a provided service (resp. required service) because they prevent safe assembly and promotion contracts. The non-observable pre-condition of service req_B gives call conditions on the (caller) component state variables$V$ of b. The non-observable post-condition of service $prov_A$ refer locally to the whole state V of a and should establish the non-observable part of the invariant of a.

2.4 Assembly and Composition

An *assembly* is a set of components that are linked (*horizontal composition*) through their services. An assembly is one element of an *assembly type*. An *assembly link* associates a required service to a provided one. Considering the rich interface of a Kmelia service (see 2.3), we need an explicit matching mechanism, to link properly the 6-tuples defining given service interfaces; therefore, additionally to signatures and dependency (via sublinks) mapping we now define *context* and *message mappings*. When needed, message or service parameters re-ordering must be handled through adaptation mechanisms [5].

Assembly context and message mapping. Consider a required service sr of a component cr of type CR linked to a provided service sp of another component cp of type CP. The virtual state space variables (vV_{sr}) of sr must be "instantiated" using the *observable* variables of sp (V^O_{CP}) by a mapping (total) function $vmap : vV_{sr} \to exp(V^O_{CP})$ where $exp(X)$ denotes an expression over the variables of X.
Each message name of sr is mapped to a message name of sp by a mapping (total) function $mmap : mname_{sr} \to mname_{sp}$.

A *composition* is the encapsulation of an assembly into a component (the composite) where some features (variables and services) of the nested components can be promoted at the composite level. *Promotion links* are used to promote services. The mappings and rules are similar to the ones of assembly, they are not detailed here.

State variables promotion. An observable variable $vo \in V^O_C$ from a component $c : C$ can be promoted as a variable $vp \in V_{CP}$ of a composite component $cp : CP$. Formally, there are a function $prom : V_{CP} \to V^O_C$ which establishes the *variable promotion*, i.e. a bridge between the variable names. In the Kmelia syntax, $(vp, vo) \in prom$, is written vp FROM c.vo. The promoted variables retain their types ($type(vp) = type(vo)$) and are accessed (*read-only* at the composite level) in their effective contexts using a service of the sub-component that defines the variables. This guarantees the encapsulation principle.

Now Kmelia services are equipped with expressive means (pre-/post-conditions, observability, virtual context) to describe contracts. Section 3 illustrates them on a working example. They are used to check services and assemblies correctness as described in Section 4.

3 A Working Example

The example is a simplified *Stock Management* application including a *vending* main service. This process manages product references (catalog) and product storage (stock). Administrators have specific rights, they can add or remove references under some consistency business rules such as: *a new reference must not be in the catalog* or *a removable reference must have an empty stock level.*

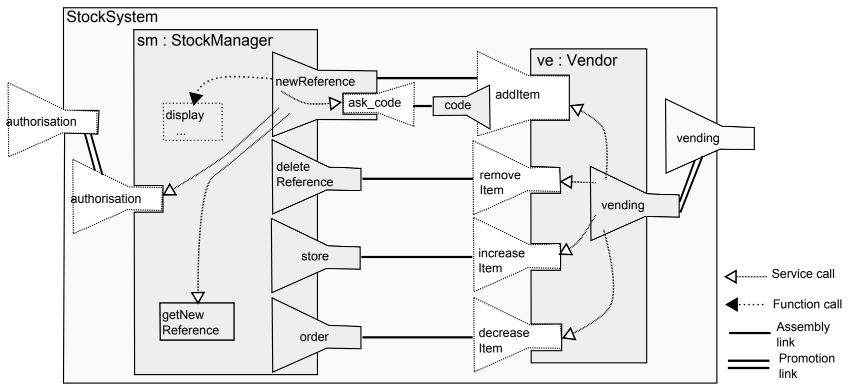


Fig. 2. Simplified Assembly of the Stock Case Study

The system is designed as a reusable component type StockSystem. It encapsulates an assembly of two components: sm:StockManager and ve:Vendor. The former is the core business component to manage references and storage. The latter is the system access interface. The main vending service is promoted at the StockSystem level. In this paper we focus only on the addItem and newReference services. According to the vending service, a user may add a new item in the stock management using the required service addItem of the Vendor component.

```
required addItem () : Integer
Interface
  subprovides : {code}
  Virtual Variables
     catalogFull : Boolean;
     catalogEmpty : Boolean     //possibly catalogSize
  Virtual Invariant not(catalogEmpty and catalogFull)
  Pre not catalogFull
  //No LTS
  Post (Result <> noReference) implies (not catalogEmpty)
End
```

The required service addItem is fulfilled with the provided service newReference which gets a new reference and performs the update of the system if there is an available new reference (see the listing 2). The links and sublinks are explicitly defined in the composition part of a composite component, as detailed in the listing 3.

The nested services represent the *service dependency* $\mathcal{DI}$. For example, the required service addItem provides a special code subservice[6]. Similarly the provided service newReference requires a ask_code service from its caller (see the ***calrequires*** declaration in the interface of newReference in the listing 1).

Inside the components, the different arrows represent various kind of calls: function call (with no side effects), service call (according to the service dependency).

[6] In Kmelia, a subservice of a service s, is a service that belongs to the interface (*subprovides*) of s.

The newReference service calls the display function (declared in the predefined Kmelia library), a service getNewReference required internally (from the same component) and the ask_code service required to its caller.

Data types in Kmelia. The data types are explicitly defined in a **TYPES** clause or in the shared libraries (predefined or user-defined). As an example, the following library (named Stocklib) declares some specific types, functions and constants.

```
TYPES
    ProductItem :: struct {id: Integer; desc: String; quantity: Integer} ;
CONSTANTS
    maxRef : Integer := 100;
    emptyString : String := "" ;
    noReference : Integer := −1;
    noQuantity : Integer := −1
```

These data types in this part are quite concrete; more abstract data types are in the process to be included in the predefined library.

A Kmelia component and observable state. The listing 1 is an extract from the Kmelia specification of the StockManager component. The state of StockManager declares among the other variables, the *observable* variable catalog which can be used for context mapping in the assembly links but also in promoted variables for composite components. Two arrays (plabels and pstock) are used to stock the labels of current references and their available quantity. The invariant states that: the catalog has an upper bound; all references in the catalog have a label and a quantity; the unknown references have no entries in the two arrays pstock and plabels. The assertions in Kmelia are possibly named predicates; the labels in front of the invariant lines are names used in this specification.

Listing 1: Kmelia specification StockManager State

```
COMPONENT StockManager
INTERFACE
  provides : {newReference, removeReference, storeItem, orderItem}
  requires : {authorisation}
USES {STOCKLIB}
TYPES
  Reference :: range 1..maxRef
VARIABLES
  vendorCodes : setOf Integer; //authorised administrators
  obs catalog : setOf Reference; // product id = index of the arrays
  plabels : array [Reference] of String; //product description
  pstock : array [Reference] of Integer //product quantity
INVARIANT
  obs @borned: size(catalog) <= maxRef,
  @referenced: forall ref : Reference | includes(catalog,ref) implies
     (plabels[ref] <> emptyString and pstock[ref] <> noQuantity),
  @notreferenced: forall ref : Reference | excludes(catalog,ref) implies
     (plabels[ref] = emptyString and pstock[ref] = noQuantity)
INITIALIZATION
  catalog := emptySet;
  vendorCodes := emptySet;     //filled by a required service
  plabels:= arrayInit(plabels,emptyString); //consistent with ..
  pstock := arrayInit(pstock,noQuantity);  //..empty catalog
```

A Kmelia service with its assertions. The listing 2 gives the specification of the provided service newReference. It provides a new reference if its running goes well. The pre-condition is that the catalog does not reach its maximal size. The post-condition is decomposed into several observable/non-observable named parts.

It states that we may have a result ranging in 1..maxRef or no reference at all, in the latter case the catalog remains unchanged.

Listing 2: Kmelia specification Provided Service with assertions

```
provided newReference () : Integer  //Result = ProductId or noReference
Interface
    calrequires : {ask_code} #required from the caller
    intrequires : {getNewReference}
Pre
    obs size(catalog) < maxRef #the catalog is not full
Variables # local to the service
  c : Integer;      # c : input code given by the user
  res:Reference;
  d : String;              # product description
Initialization
  res := noQuantity;
Behavior
Init i # the initial state
Final f # a final state
{  i -- c := __CALLER!!ask_code() --> e1,
        # gets the password on the ask_code (service) channel
  e1 -- [not(c in vendorCodes)]
        display("adding a reference is not allowed") --> end,
  e1 -- [c in vendorCodes] __CALLER?msg(d) --> e2,
        # gets the product description
  e2 -- [d = emptyString]
        display("adding an EmptySet description is not allowed") --> end,
  e2 -- [d <> emptyString] res := __SELF!!getNewReference() --> e4,
  e4 -- {catalog := including(catalog,res); //add new reference
         pstock[res] := 0; //default stock is null
         plabels[res] := d //product description is the one provided
        }--> end,
  end -- __CALLER!!newReference(res) --> f
        # the caller is informed from the Result and the service ends.
}
Post
 obs @resultRange:((Result >=1 and Result <=maxRef) or (Result = noReference)),
 obs @resultValue:(Result <> noReference) implies (notIn(old(catalog),Result)
                   and catalog = add(old(catalog),Result)),
 obs @noresultValue:(Result = noReference) implies Unchanged{catalog},
 @refAndQuantity: (Result <> noReference) implies
          (pstock[Result] = 0 and plabels[Result] <> emptyString and
          (forall i : Reference | (i <> Result) implies
             (pstock[i] = old(pstock)[i] and plabels[i] = old(plabels)[i] ))),
 @NorefAndQuantity: (Result = noReference) implies Unchanged{pstock,plabels}
```

The behaviour of a service defines a list of transitions e1 −−label−−> e2 where e1 and e2 are state names. A transition label is a guarded combination of actions [guard] action∗. An action is either an *elementary action* from $\mathcal{A}$ (expression) or a *communication action* (service interaction). The syntax of a communication action is channel(! | ? | ⁈ | ‼) message(param∗) where the channel denotes a reference in the service dependency $\mathcal{DI}$, the single char operators are message interactions (send/receive) and the double char operators are service interactions (call, result). The channel _***CALLER*** stands for the caller service, _***SELF*** stands for a service of the same component (internal call), _rs stands for a required service. In this article we will not consider further the behaviour.

Context and message mappings. The context and message mappings (see 2.4) are specified in assembly links. In the listing 3, variables of the virtual context of addItem are associated with an expression on the variables of the context of newReference i.e. the observable state variables of the component sm. In this example, there are no message mapping because only the standard msg message (declared in the predefined Kmelia library) is used.

Listing 3: Kmelia specification StockSystem

```
COMPONENT StockSystem
INTERFACE
  provides : {vending}
  requires : {authorisation}
COMPOSITION
  Assembly
    Components
      sm : StockManager;
      ve : Vendor
    Links ////////////assembly links//////////
      lref: p-r sm.newReference, ve.addItem
         context mapping
           ve.catalogEmpty == empty(sm.catalog),
           ve.catalogFull == size(sm.catalog) = MaxInt
         sublinks : {lcode}
      lcode: r-p sm.ask_code, ve.code
      ...
  End // assembly
  Promotion
    Links ////////////promotion links//////////
      lvend: p-p ve.vending, SELF.vending
      laut: r-r sm.authorisation, SELF.authorisation
END_COMPOSITION
```

In the next section, we show how this Kmelia specification is analysed using our COSTO[7] tool and a specific verification approach using the B method and tools.

4 Formal Analysis and Experimentations

Components, assemblies and compositions should be analysed according to various facets. Tables 1 and 2 give an overview of the verification requirements that we consider to validate a Kmelia specification. Some of them were achieved before, in particular the behavioural compatibility of services and components, treated in [7]: it was achieved using model-checking techniques provided by existing tools (Lotos/CADP[8] and MEC[9]); the involved parts of the Kmelia specifications were automatically translated into the input languages of these tools and checked.

In this section, we address aspects related to data type checking and assertion checking; the main goal is to analyse parts of a Kmelia specification using its new features such as the assertions. Formal verification tools are necessary to check assertions consistency. Our approach consists in reusing existing tools such as the B tools and especially the Rodin[10] framework. We design a systematic verification method that enables us to reuse the proof obligations generated by the B tools for our specific purpose.

Event-B and Rodin framework. Rodin is a framework made of several tools dedicated to the specification and proof of Event-B models. Event-B [1] extends the classical B method [2] with specific constructions and usage; it is intended to the modelling of general purpose systems and for reasoning on them. Proof obligations (POs) are generated to ensure the consistency of the considered model, i.e. the preservation of the **INVARIANT** by the **EVENTS**. Other POs ensure that a *refined* model is consistent, i.e. the abstract **INVARIANT** is preserved and the refined events

[7] COmponent Study TOolkit dedicated to the Kmelia language
[8] http://www.inrialpes.fr/vasy/cadp/
[9] http://altarica.labri.fr/wiki/tools:mec_4
[10] http://rodin-b-sharp.sourceforge.net

Analysis	**Status**
Static rules: Scope + name resolution + type-checking	done
Observability rules	in progress (see 2.3)
Component interface consistency	done
Services dependency consistency: $\mathcal{DI}$ well-formed vs. $\mathcal{I}$ and $\mathcal{D}$ (component) $\mathcal{DI}$ vs. $\mathcal{B}$ (eLTS)	done
Simple constraint checking (parameters, query, protocol, . . .)	in progress
Local eLTS checking (deadlocks, guard, subprovides, . . .)	in progress
Invariant consistency vs. pre/post conditions: provided services : $Inv^O \wedge Pre^O \Rightarrow Post^O \wedge Inv^O$ $Inv \wedge Pre \Rightarrow Post^{NO} \wedge Inv$ required services : $vInv \wedge Pre^O \Rightarrow Post^O \wedge vInv$	 experimental (a) experimental (b) experimental (c)
Consistency between service assertions and eLTS: eLTS vs. *Post* the post condition should be established required service R calls vs. Pre_R the context must ensure the precondition (local+virtual) eLTS vs. subprovided service SP annotations Pre_{SP} the context must ensure the precondition (local)	not yet

Table 1
Formal analysis of a simple Kmelia component

Analysis	State
Static rules: Scope + name resolution + type-checking	done
Observability rules: promoted variables	done
Link/sublink consistency: assembly and composition signature matching service dependency matching (subprovides, callrequires)	done [7]
context mapping (*cm* function) and observability rules message mapping	
Assembly Link Contract correctness: $cm(Pre^O_R) \Rightarrow Pre^O_P$ $Post^O_P \Rightarrow cm(Post^O_R)$	 experimental (d) experimental (e)
Provided Promotion Link Contract correctness: PP is at the composite level $cm(Pre^O_P P) \Rightarrow Pre^O_P$ $Post^O_P \Rightarrow cm(Post^O_{PP})$	 experimental (f) experimental (g)
Required Promotion Link Contract correctness: RR is at the composite level $cm(Pre^O_R) \Rightarrow Pre^O_{RR}$ $Post^O_{RR} \Rightarrow cm(Post^O_R)$	 experimental (h) experimental (i)
eLTS (behaviour) compatibility	done [7]

Table 2
Formal analysis of a Kmelia assembly and compositions

do not contradict their abstract counterparts.

POs can be discharged automatically or interactively, using the Rodin provers.

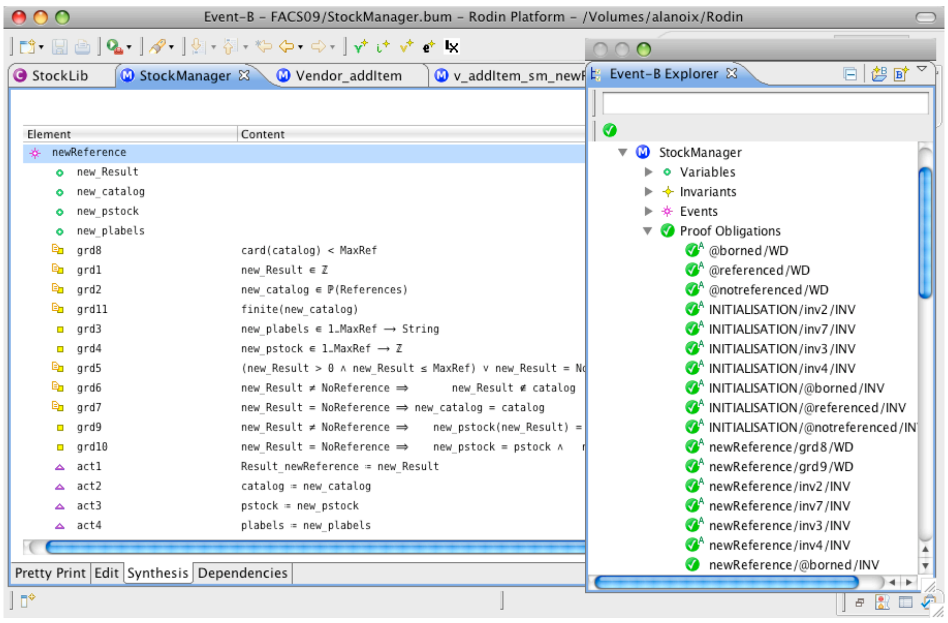
Verifying Kmelia specifications using Event-B. The main idea is, first to consider a part of the Kmelia specification involved in the property to be verified (a service, a component, a link of an assembly, an assembly, etc), then to build from this part of the specification, a set of (Event-)B models in such a way that the POs generated for them correspond to the specific obligations we needed to check the Kmelia specification assertions. Using B to validate components assembly contracts has been investigated in [15,18].

We systematically [11] build some Event-B models, with an appropriate structure as explained below, to check some of the proof obligations presented in Tables 1 and 2. The Event-B models are currently built by hand.

(i) For each component and its provided services, we generate an Event-B model. The proof of the consistency of this model ensures the proof of the rules (a) and (b) for the invariant consistency at the Kmelia level.

(ii) For each required service (and its "virtual context") we have to generate an Event-B model. Its B consistency establishes the rule (c).

(iii) For each assembly link between a required service req and a provided one prov, we give an Event-B model of the observable part of prov, which *refines* the Event-B model of the required service req previously checked.
- the consistency proof of the Event-B model ensures the rule (a) for the invariant consistency at the Kmelia level;
- the refinement proof establishes both the rules (d) and (e) for the Kmelia assembly correctness.



```
Element        Content
newReference
  new_Result
  new_catalog
  new_pstock
  new_plabels
  grd8         card(catalog) < MaxRef
  grd1         new_Result ∈ ℤ
  grd2         new_catalog ∈ ℙ(References)
  grd11        finite(new_catalog)
  grd3         new_plabels ∈ 1‥MaxRef → String
  grd4         new_pstock ∈ 1‥MaxRef → ℤ
  grd5         (new_Result > 0 ∧ new_Result ≤ MaxRef) ∨ new_Result = No
  grd6         new_Result ≠ NoReference ⇒       new_Result ∉ catalog
  grd7         new_Result = NoReference ⇒ new_catalog = catalog
  grd9         new_Result ≠ NoReference ⇒    new_pstock(new_Result) =
  grd10        new_Result = NoReference ⇒    new_pstock = pstock ∧
  act1         Result_newReference ≔ new_Result
  act2         catalog ≔ new_catalog
  act3         pstock ≔ new_pstock
  act4         plabels ≔ new_plabels
```

```
StockManager
  Variables
  Invariants
  Events
  Proof Obligations
    @borned/WD
    @referenced/WD
    @notreferenced/WD
    INITIALISATION/inv2/INV
    INITIALISATION/inv7/INV
    INITIALISATION/inv3/INV
    INITIALISATION/inv4/INV
    INITIALISATION/@borned/INV
    INITIALISATION/@referenced/INV
    INITIALISATION/@notreferenced/IN
    newReference/grd8/WD
    newReference/grd9/WD
    newReference/inv2/INV
    newReference/inv7/INV
    newReference/inv3/INV
    newReference/inv4/INV
    newReference/@borned/INV
```

Fig. 3. Rodin

We are not going to deal in this article with the details of the translation procedure [12]. Kmelia invariant and pre-condition translations are quite systematic, whereas the post-condition concept does not exist into the B language. Therefore we abstract the post-condition by using an **ANY** substitution that satisfies the post-condition (once translated) as proposed in the context of UML/OCL to B

[11] applying defined rules which are not yet fully automatised

[12] The specifications and results are available in http://www.lina.sciences.univ-nantes.fr/coloss/projects/kmelia/index_en.php

translations [19]. Figure 3 depicts the Event-B translation into Rodin of the service newReference of StockManager.

Experimental results. Applying our method on the case study presented in Section 3, we obtain the Event-B models structured as depicted in Fig 4. These models are studied within Rodin; it tooks a few hours to write and check the models. We can verify the Kmelia components StockManager and Vendor before checking the assembly StockSystem. The Event-B model StockManager is used to prove the preservation of the invariant assertions by the provided services. The refinement v_addItem_sm_newReference is used to check the assembly link between the services newReference and addItem. The Table 3 gives an idea about the number of POs that are to be discharged to ensure the correctness of the Kmelia specification.

Studying the example within Rodin revealed some errors in our initial Kmelia specification. For example, the post-condition of newReference was wrong; one of the associated POs could not be discharged. After the feedback in our Kmelia specifications, the error was corrected.

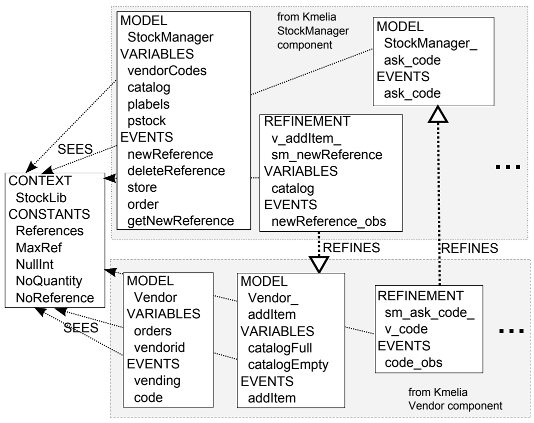


Fig. 4. Event-B Models

	Auto.	**Manual**	**Total**
StockManager	16	3	19
Vendor_addItem	2	1	3
v_addItem_sm _newReference	22	1	23

Table 3
Rodin Proof obligations

In a general manner, the assertions associated to Kmelia services help us to ensure the correctness of the assembly link by considering the required-provided relationship as a refinement from the required service to the provided one. When the assertions are wrong, the proofs fail, which means the assembly link is wrong.

5 Discussion and Conclusion

In this article we have presented enrichments to the Kmelia abstract component model: a data language for Kmelia expressions and predicates; visibility features for component state in the context of composite components; contracts in the composition of services. The formal specification and analysis of the model are revisited accordingly. The syntactic analysis of Kmelia is effective in the COSTO tool that supports the Kmelia model. We have proposed a method to perform the necessary assertions verification using B tools: the contracts are checked through preliminary experimentations using the Rodin framework. We have illustrated the contribution with an example which is specified in Kmelia, translated manually and verified using Rodin.

Discussion. Our work is more related to abstract and formal component models like SOFA or Wright, rather than to the concrete models like CORBA, EJB or .NET. The Java/A [9] or ArchJava [3] models do not allow the use of contracts. We have already emphasized (see Section 1) the fact that most of the abstract models deal mainly with the dynamic part of the components. Some of them [16,23] take datatypes and contracts into account but not the dynamic aspects. Some other ones [11,13] delay the data part to the implementation level.

In [14] *may/must* constraints are associated to the interactions defined in the component interfaces to define behavioural contracts between client and suppliers. In Kmelia, the distinction between a supplier constraint and the client is done from a methodological point of view rather than a syntactic rule. The use of B to check component contracts has been studied in [15,18] in the context of UML components.

Fractal [17] proposes different approaches based on the separation of concerns: the common structural features are defined in Fractal ADL [21] ; dynamic behaviours are implemented by Vercors [8] or Fractal/SOFA [12] and the use of assertions are studied in ConFract [16]. In ConFract contracts are independent entities which are associated to several participants, not to services and links as in our case; their contracts support a rely/guarantee mechanism with respect to the (vertical) composition of components.

In [10] a component (a component type in Kmelia) is a model in the sense of the algebraic specifications. Dynamic behaviours are associated to components but not to services, which are simple operations. The component provided and required interfaces are type specifications and composing component is based on interface (or type) refinement. In Kmelia components are assembled on their services; therefore the main issue is not to refine types as in [10] but rather to check contracts as in [25]. More specifically our case is more related to the *plugin matching* of [25].

Perspectives. Several aspects remain to be dealt with regarding assertions and the related properties, composition and correctness of component assemblies. First, we need to implement the full chain of assertion verification especially the translation *KmlToB* which is necessary to automatically derive the necessary Event-B models to check the assertions and the assemblies. Second, we will integrate high level concepts and relations for data types. In particular, we plan to integrate some kind of objects and inheritance in the type system but also component types. Assertions in this context are more difficult to specify and to verify.

Another challenging point is the support for interoperability with other component models. We assume that in real component applications, a component assembly is built on components written in various specification languages. When connecting services (or operations) we can at least check the matching of signatures. If the specification language of the corresponding services or components accepts contracts (resp. service composition, service behaviour) we can provide corresponding verification means.

References

[1] J.-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[2] Jean-Raymond Abrial. *The B-Book Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[3] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[4] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[5] Pascal André, Gilles Ardourel, and Christian Attiogbé. Adaptation for hierarchical components and services. *Electron. Notes Theor. Comput. Sci.*, 189:5–20, 2007.

[6] Pascal André, Gilles Ardourel, and Christian Attiogbé. Defining Component Protocols with Service Composition: Illustration with the Kmelia Model. In *6th International Symposium on Software Composition, SC'07*, volume 4829 of *LNCS*. Springer, 2007. http://www.lina.sciences.univ-nantes.fr/coloss/projects/kmelia/index_en.php.

[7] Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking Component Composability. In *5th International Symposium on Software Composition, SC'06*, volume 4089 of *LNCS*. Springer, 2006. http://www.lina.sciences.univ-nantes.fr/coloss/projects/kmelia/index_en.php.

[8] Tomás Barros, Antonio Cansado, Eric Madelaine, and Marcela Rivera. Model-checking distributed components: The vercors platform. *Electron. Notes Theor. Comput. Sci.*, 182:3–16, 2007.

[9] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A component model for architectural programming. *Electr. Notes Theor. Comput. Sci.*, 160:75–96, 2006.

[10] Michel Bidoit and Rolf Hennicker. An algebraic semantics for contract-based software components. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST'08)*, volume 5140 of *LNCS*, pages 216–231. Springer, July 2008.

[11] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and Its Support in Java. *Software Practice and Experience*, 36(11-12), 2006.

[12] Tomáš Bureš, Martin Děcký, Petr Hnětynka, Jan Kofroň, Pavel Parízek, František Plášil, Tomáš Poch, Ondřej Šerý, and Petr Tůma. *CoCoME in SOFA*, volume 5153, pages 388–417. Springer-Verlag, 2008.

[13] Carlos Canal, Lidia Fuentes, Ernesto Pimentel, JosÃľ M. Troya, and Antonio Vallecillo. Adding Roles to CORBA Objects. *IEEE Trans. Softw. Eng.*, 29(3):242–260, 2003.

[14] Cyril Carrez, Alessandro Fantechi, and Elie Najm. Behavioural contracts for a sound composition of components. In Hartmut König, Monika Heiner, and Adam Wolisz, editors, *FORTE 2003, IFIP TC6/WG 6.1)*, volume 2767 of *LNCS*, pages 111–126. Springer-Verlag, September 2003.

[15] S. Chouali, M. Heisel, and J. Souquières. Proving component interoperability with B refinement. *Electronic Notes in Theoretical Computer Science*, 160:157–172, 2006.

[16] Philippe Collet, Jacques Malenfant, Alain Ozanne, and Nicolas Rivierre. Composite contract enforcement in hierarchical component systems. In *Software Composition, 6th International Symposium (SC 2007)*, volume 4829, pages 18–33, 2007.

[17] Thierry Coupaye and Jean-Bernard Stefani. Fractal component-based software engineering. In *Object-Oriented Technology, ECOOP 2006*, pages 117–129. Springer, 2006.

[18] A. Lanoix and J. Souquières. A trustworthy assembly of components using the B refinement. *e-Informatica Software Engineering Journal (ISEJ)*, 2(1):9–28, 2008.

[19] Hung Ledang and Jeanine Souquières. Integration of uml and b specification techniques: Systematic transformation from ocl expressions into b. In *9th Asia-Pacific Software Engineering Conference (APSEC 2002)*. IEEE Computer Society, 2002.

[20] B. Meyer. *Object-Oriented Software Construction*. Professional Technical Reference. Prentice Hall, 2nd edition, 1997. http://www.eiffel.com/doc/oosc/.

[21] ObjectWeb Consortium. Fractal adl. [Online]. Available: http://fractal.ow2.org/fractaladl/index.html. [Accessed: Jun. 17, 2009], 2009.

[22] Sebastian Pavel, Jacques Noye, Pascal Poizat, and Jean-Claude Royer. Java Implementation of a Component Model with Explicit Symbolic Protocols. In *4th International Symposium on Software Composition, SC'05*, volume 3628 of *LNCS*. Springer, 2005.

[23] H. Schmidt. Trustworthy components-compositionality and prediction. *J. Syst. Softw.*, 65(3):215–225, 2003.

[24] D.M. Yellin and R.E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

[25] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transaction on Software Engeniering Methodolology*, 6(4):333–369, 1997.

A Formal Framework for Structural Reconfiguration of Components under Behavioural Adaptation

Antonio Cansado, Carlos Canal, Gwen Salaün, and Javier Cubo

Department of Computer Science, University of Málaga
Campus de Teatinos, 29071, Málaga, Spain
Emails: {acansado,canal,salaun,cubo}@lcc.uma.es

Abstract

A major asset of modern systems is to dynamically reconfigure systems to cope with failures or component updates. Nevertheless, designing such systems with off-the-shelf components is hardly feasible: components are black-boxes that can only interact with others on compatible interfaces. Part of the problem is solved through Software Adaptation techniques, which compensates mismatches between interfaces. Our approach aims at using results of Software Adaptation in order to also provide reconfiguration capabilities to black-box components.

This paper provides two contributions: (i) a formal framework that unifies behavioural adaptation and structural reconfiguration of components; this is used for statically reasoning whether it is possible to reconfigure a system. And (ii), two cases of reconfiguration in a client/server system in which the server is substituted by another one with a different behavioural interface, and the system keeps on working transparently from the client's point of view.

Keywords: Components, reconfiguration, behavioural adaptation, formal methods

1 Introduction

The success of Component-Based Software Development comes from creating complex systems by assembling smaller, simpler components. Nevertheless, building systems based on off-the-shelf components is difficult because components must communicate on compatible interfaces. It is even more difficult when the system must be able to reconfigure because components must provide reconfiguration capabilities. Here, we understand by reconfiguration the capacity of changing the component behaviour and/or component implementation at runtime [15]. For example, we are interested in upgrading a component, substituting a component by another, adding new components to a running system, and so on.

In general, components are black-box modules of software that come with behavioural specifications of their interfaces. Therefore, there is no access to the source code of components, but it is possible to use tool-assisted techniques to analyse the

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

behaviour of a component assembly [5,8]. Some applications of this analysis are used in Software Adaptation [24] to work out behavioural mismatches in the components' interfaces. In [19], an adaptation contract defines rules on how mismatches can be worked out and a tool generates an adaptor that orchestrates the system execution and compensates incompatibilities existing in interfaces.

On the other hand, there is little support to analyse whether a reconfiguration is safe. Enabling system reconfiguration requires designers to define (i) when a component can be reconfigured, (ii) which kind of reconfiguration is supported by the component, and (iii) which kind of properties the reconfiguration holds. For example, ensuring that some parts of the system can be reconfigured without system disruption. Our approach aims at providing a formal framework that helps answering these questions.

There are several related approaches in the literature. SOFA 2.0 [8] proposes *reconfiguration patterns* in order to avoid uncontrolled reconfigurations which lead to errors at runtime. This allows adding and removing components at runtime, passing references to components, *etc.*, under predefined structural patterns. There are other more general approaches that deal with distributed systems and software architectures [12,13], graph transformation [1,23] or metamodelling [11].

Our goal is to reconfigure components that have not been designed with reconfiguration capabilities in mind. Moreover, we target reconfiguration of components that may be involved in an ongoing transaction without stopping the system. This fits in a context where reconfiguration may be triggered at any moment and a component must be substituted at runtime.

Since substituting a component usually requires finding a perfect match, reconfiguration is usually limited to instances (or subtypes) of the same component. Our approach is to exploit behavioural adaptation to further allow reconfiguration. That is, we target reconfiguration scenarios in which both the former and the new component need some adaptation in order to allow substitution. We build on the basis that components are provided with behavioural interfaces and the composition with adaptation contracts. Then, we show that in some cases the information found in the adaptation contract can be used to endow black-box components with reconfiguration capabilities.

This paper's contributions are: (i) we present a formal model that includes behavioural adaptation and structural reconfiguration of components. With this framework it is possible to verify properties of the complete system, including those involving reconfiguration of a component by another one with a different behavioural interface. And (ii), we present two examples of substitutability in which a server is substituted by another one with a different behavioural interfaces. We also show how can we build a fault-tolerant system by constraining the system's behaviour.

This paper is structured as follows: Firstly, Section 2 provides some background on formalisms that will be used throughout the paper. Then Section 3 introduces a client/server system that is used as running example. Section 4 provides the formal model that allows structural reconfiguration and behavioural adaptation. Section 5 provides applications of our approach for designing reconfigurable systems based on (non-reconfigurable) black-box components. Then, Section 6 presents related works on reconfiguration and behavioural adaptation and Section 7 concludes this paper.

2 Background

This work builds on previous work on hierarchical behavioural models [5] and Software Adaptation [9]. We recall in this section the main definitions that are used in this paper.

2.1 Networks of Synchronised Automata

We assume that component interfaces are equipped both with a signature (set of required and provided operations), and a protocol. We model the behaviour of a component as a Labelled Transition System (LTS). The LTS transitions encode the actions that a component can perform in a given state. In the following definitions, we frequently use indexed vectors: we note $\tilde{x}_I$ the vector $\langle ..., x_i, ...\rangle$ with $i \in I$, where I is a countable set.

Definition 2.1 [LTS]. A *Labelled Transition System (LTS)* is a tuple $\langle S, s_0, L, \rightarrow\rangle$ where S is the set of states, $s_0 \in S$ is the initial state, L is the set of labels, $\rightarrow$ is the set of transitions : $\rightarrow \subseteq S \times L \times S$. We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \rightarrow$.

Communication between components are represented using *actions* relative to the emission and reception of messages corresponding to operation calls, or internal actions performed by a component. Therefore, in our model, a *label* is either the internal action τ or a tuple (M, D) where M is the message name and D stands for the communication direction (*!* for emission, and *?* for reception).

Definition 2.2 [Network of LTSs]. Let *Act* be an action set. A *Net* is a tuple $\langle A_G, J, \tilde{O}_J, T\rangle$ where $A_G \subseteq Act$ is a set of global actions, J is a countable set of argument indexes, each index $j \in J$ is called a *hole*. O_j is the sort of the hole j with $O_j \subset Act$. The transducer T is an LTS $(S_T, s_{0T}, L_T, \rightarrow_T)$, and $L_T = \{\overrightarrow{v} = \langle a_g.\tilde{\alpha}_I\rangle,\ a_g \in A_G,\ I \subseteq J \wedge \forall\, i \in I, \alpha_i \in O_i\}$

Synchronisation between components is specified using *Nets* [5]. *Nets* are inspired by *synchronisation vectors* [2], that we use to synchronise a number of processes and can describe dynamic configurations of processes, in which the possible synchronisations change with the state of the Net; they are *transducers* [18].

A transducer in the Net is encoded as an LTS whose labels are synchronisation vectors ($\overrightarrow{v}$), each one describing a particular synchronisation between the actions ($\tilde{\alpha}_I$) of different argument processes; the result of this synchronisation is seen as a global action a_g. Each state of the transducer T corresponds to a given configuration of the network in which a given set of synchronisations is possible. Some of those synchronisations can trigger a change of state in the transducer leading to a new configuration of the network; that is, it encodes a dynamic change on the configuration of the system.

Definition 2.3 [Sort]. The *Sort* is the set of actions that can be observed from outside the automaton. It is determined by its top-level node, L for an LTS, and A_G for a Net:

$$Sort(\langle S, s_0, L, \rightarrow\rangle) = L \qquad\qquad Sort(\langle A_G, J, \tilde{O}_J, T\rangle) = A_G$$

A Net is a generalised parallel operator. Complex systems are built by combining LTSs in a hierarchical manner using Nets at each level. There is a natural typing compatibility constraint for this construction, in term of the sorts of formal and actual parameters. The standard compatibility relation is Sort inclusion: a system *Sys* can be used as an actual argument of a Net at position j only if it agrees with the sort of the hole O_j, *i.e.* $Sort(Sys) \subseteq O_j$.

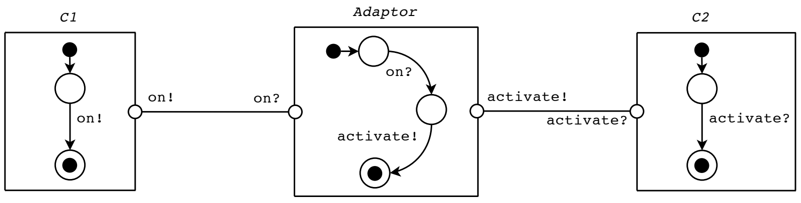
2.2 Specification of Adaptation Contracts

While building a new system by reusing existing components, behavioural interfaces do not always fit one another, and these interoperability issues have to be faced and worked out. Mismatches may be caused by different message names, a message without counterpart (or with several ones) in the partner, etc. The presence of mismatch results in a deadlocking execution of several components [3,10].

Adaptors can be automatically generated based on an abstract description of how mismatch situations can be solved. This is given by an *adaptation contract*.

In this paper, the adaptation contract $\mathcal{AC}$ between components is specified using *vectors* [9]. Each action appearing in one vector is executed by one component and the overall result corresponds to a synchronisation between all the involved components. A vector may involve any number of components and does not require interactions to occur on the same names of actions. Moreover, a vector may synchronise actions performed by sub-processes in a hierarchical fashion.

For example, a vector $v = \langle C1 : on!, C2 : activate?\rangle$ denotes that the action $on!$ performed by component $C1$ corresponds to action $activate?$ performed by component $C2$. This does not mean that both actions have to take place simultaneously, nor one action just after the other; the adaptor will take into account their respective behaviour as specified in their LTS, accommodating the reception and sending of actions to the points in which the components are able to perform them.



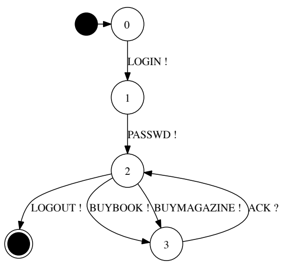
3 Running Example

This section presents the running example used in the following sections. It consists of a client/server system in which the server may be substituted by an alternative server component. This can be needed in case of server failure, or simply for a change in the client's context or network connection that made unreachable the original server. We suppose none of the components have been designed with reconfiguration capabilities.

The client wants to buy books and magazines as shown in its behavioural interface in Fig. 1(a). The two servers A and B have behavioural interfaces depicted in

Figs. 1(c) and 2(b) respectively. Server A can sell only one book per transaction [1]; on the other hand, server B can sell a bounded number of books and magazines.

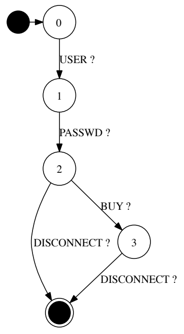
Initially, the client is connected to server A; we shall call this configuration c_A. The client and the server agree on an adaptation contract $\mathcal{AC}_{C,A}$ (see Fig. 1(b)). Naturally, under configuration c_A the client can only buy at most one book in each transaction but it is not allowed to buy magazines because this is not supported by server A. The latter is implicitly defined in the adaptation contract (Fig. 1(b)) because there is no vector allowing the client to perform the action $buyMagazine!$. Finally, server A does not send the acknowledgement $ack?$ (see v_4 in Fig. 1(b)) expected by the client; this must also be worked out by the adaptor.



(a) LTS of Client C

$$
\begin{aligned}
v_1 &= \langle C : login!, A : user?\rangle\\
v_2 &= \langle C : passwd!, A : passwd?\rangle\\
v_3 &= \langle C : buyBook!, A : buy?\rangle\\
v_4 &= \langle C : ack?, A : \epsilon\rangle\\
v_5 &= \langle C : logout!, A : disconnect?\rangle
\end{aligned}
$$

(b) Adaptation Contract $\mathcal{AC}_{C,A}$



(c) LTS of Server A

Fig. 1. Configuration c_A.

In an alternative configuration c_B the client is connected to server B whose protocol is depicted in Fig. 2(b). Similarly, the client and the server agree on an adaptation contract $\mathcal{AC}_{C,B}$ (see Fig. 2(a)). Under configuration c_B, the client can buy a bounded number of books and magazines. In Fig. 2(a), we see that vector v_5 allows the client to buy magazines. Moreover, server B sends a different acknowledgement for each product (see v_4 and v_6 in Fig. 2(a)).

We shall study reconfiguration from c_A to c_B which substitutes A by B. It is worth noting that A and B do not have the same behavioural interfaces. Not only B provides additional functionality *w.r.t.* A, but also B does not have the same names for the actions (and potentially the ordering of actions may be different as well). For instance, v_1 of $\mathcal{AC}_{C,A}$ (see Fig. 1(b)) says that the $login!$ action of the client relates to $user?$ of server A. On the other hand, this $login!$ action must be related to $connect?$ of server B (see v_1 of $\mathcal{AC}_{C,B}$ in Fig. 2(a)).

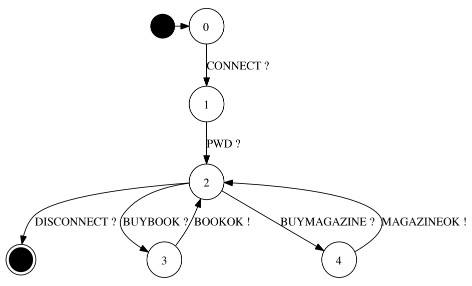
4 Formal Model

This section provides the formal model that enables reconfiguration and behavioural adaptation. We define a *reconfiguration contract* to determine how the system may

[1] A transaction starts in the LTS's initial state and ends in the final one. The definition of LTS in Section 2.1 does not include final states, though one can understand as final state a state with no outgoing transitions.

$v_1 = \langle C : login!, B : connect?\rangle$
$v_2 = \langle C : passwd!, B : pwd?\rangle$
$v_3 = \langle C : buyBook!, B : buyBook?\rangle$
$v_4 = \langle C : ack?, B : bookOk!\rangle$
$v_5 = \langle C : buyMagazine!, B : buyMagazine?\rangle$
$v_6 = \langle C : ack?, B : magazineOk!\rangle$
$v_7 = \langle C : logout!, B : disconnect?\rangle$

(a) Adaptation Contract $\mathcal{AC}_{C,B}$



(b) LTS of Server B

Fig. 2. Configuration c_B.

evolve in terms of structural changes. Then, we provide a formal model based on *Nets* for this kind of systems.

4.1 Reconfiguration Contract

A system architecture consists of a finite number of components. Each *configuration* is a subset of these components connected together by means of adaptation contracts.

Definition 4.1 [Configuration]. A configuration $c = \langle P, \mathcal{AC}, S^\star\rangle$ is a static structural configuration of a system. P is an indexed set of components. $\mathcal{AC}$ is an adaptation contract of components in P. $S^\star$ is a set of *reconfiguration states* defined upon P; these are states in which reconfiguration is allowed.

Changing a configuration by another is what we call a reconfiguration. This is specified in a *reconfiguration contract* which separates reconfiguration concerns from the business logic. Each configuration can be thought of a static system and the "dynamic" part is specified by a reconfiguration operation. The reconfiguration states are left out-of-scope in this paper. We assume here they are given by the designer, though we show how some of them can be obtained in our examples.

Definition 4.2 [Reconfiguration Contract]. A reconfiguration contract $\mathcal{R} = \langle C, c_0, \rightarrow_\mathcal{R}\rangle$ is defined as:

C is a set of static configurations, where $c_i \in C$, $i \in \{0..n\}$, is a static configuration. $c_0 \in C$ is the initial configuration. $\rightarrow_\mathcal{R} \subseteq C \times R_{op} \times C$ is a set of reconfiguration operation, with reconfiguration operation $R_{op} \subseteq S_i^\star \times S_j^\star$, $S_i^\star \in c_i$, $S_j^\star \in c_j$.

From the definition above, reconfiguration can take place in the middle of a transaction and the new configuration can have a new adaptation contract. Therefore, this allows reconfiguring a component by another one that implements a different behavioural interface. Nevertheless, for guaranteeing consistency this can only happen at predefined states. A state of the source configuration ($s_i^\star$) defines when a configuration can be reconfigured. On the other hand, a state of the target configuration ($s_j^\star$) says what is the starting state in the target configuration to resume the execution.

Example

In the running example, there are two configurations:
$c_A = \langle\{C, A\}, \mathcal{AC}_{C,A}, S^\star_A\rangle$ and $c_B = \langle\{C, B\}, \mathcal{AC}_{C,B}, S^\star_B\rangle$.

The reconfiguration contract $\mathcal{R} = \langle C, c_A, \rightarrow_\mathcal{R}\rangle$ is given by:
$C = \{c_A, c_B\}$, and $\rightarrow_\mathcal{R} = \{c_A \xrightarrow{r} c_B\}$, with $r = (s^\star_A, s^\star_B)$.

r must specify the pairs of *reconfiguration states* on which reconfiguration can be performed. Since both servers have different behavioural interfaces, it is not straight-forward to determine how reconfiguration can take place after a transaction between the client and the server has started. Therefore, in the simplest scenario reconfiguration from c_A to c_B is only allowed at the initial states of the client and the server. This is specified as a unique reconfiguration state $s^\star_i \in S^\star_i$, $i \in \{A, B\}$ for each configuration; more precisely, $s^\star_A = \{C : s_0, A : s_0\}$ and $s^\star_B = \{C : s_0, B : s_0\}$. In Section 5 we will study how other pairs of reconfiguration states can be obtained.

4.2 Building Verifyable Systems

This section shows how to build Nets for the reconfiguration contract above. We have previously defined the system and now we generate a behavioural model of the complete system that can be fed into model-checking tools. There are two benefits in this approach.

Firstly, is it easier to verify properties related to reconfiguration if the complete system is modelled. In our running example, we can prove that the client may only buy magazines if the system is reconfigured towards a configuration c_B.

Secondly, a *Net* is close enough to the structure of a program that it should be possible to implement the *Net* in a component model framework such as *Fractal* [7]. This would provide us, at runtime, a component system with predictable behaviour under reconfiguration.

As the adaptor represents the interactions between the adapted components, we will use states of the adaptor to identify states in which reconfiguration will be applied. Let A_P be the adapter LTS generated by $\mathcal{AC}$ [10], we add a reconfiguration action $r_{s^\star_i}?$ for each $s^\star_i \in S^\star$. This action leaves the adaptor in same state, defined as the state in which all components are in the state given by $s^\star_i$.

Definition 4.3 [Network of Configuration]. A configuration $c = \langle P, \mathcal{AC}, S^\star\rangle$, defines a Net $c_{Net} = \langle A_G, J, \tilde{O}_J, T\rangle$, as:

Let A_P be the adapter LTS generated by $\mathcal{AC}$, with a reconfiguration action $r_{s^\star_i}?$ for each $s^\star_i \in S^\star$. Each $a! \in Sort(p_i)$ defines an action $v = \{p_i : a!, A_P : a?\} \in A_G$. Each $a? \in Sort(p_i)$ defines an action $v = \{p_i : a?, A_P : a!\} \in A_G$. Each reconfiguration action $r_{s^\star_i}?$ defines an action $v = \{A_P : r_{s^\star_i}?\} \in A_G$. Each process p_i and A_P is an argument of J, with $Sort(p_i) \subseteq O_{p_i}$ and $Sort(A_P) \subseteq O_{A_P}$.

The Net transducer T is defined as: a unique state $s_T = s_{0_T} \in S_T$; and a transition $s_T \xrightarrow{v} s_T$ for each $v \in A_G$.

In the definition above, a *Net* closely represents a configuration; the root of the *Net*'s tree is the synchronisation operator given by the Net's transducer, and the *Net*'s leaves are the components p_i and A_P. The transducer actions are synchronisation vectors that relate actions between p_i and the adaptor; note that due to the adaptation process, these actions are exact dual operations. The transducer has additional transitions taking care of reconfiguration capabilities, though they are not used within a configuration.

We construct a system that allows reconfiguration between *configuration*s based on the *reconfiguration contract* and the construction of a network for a *configuration*.

Definition 4.4 [Network of Reconfiguration Contract]. A $\mathcal{R} = \langle C, c_A, \rightarrow_{\mathcal{R}} \rangle$ defines a Net $\mathcal{R}_{Net} = \langle A_G, J, \tilde{O}_J, T \rangle$, with:

Each configuration $c_i \in C$ defines a Net $c_{i_{Net}} \in J$. Each reconfiguration operation r defines a process P_r used as argument of J. This process will be in charge of initialising the target configuration. The global actions A_G are defined upon the union of the global actions of each configuration, and the reconfiguration transactions. Formally, $A_G = [\bigcup_{\forall c_i \in C} \mathit{Sort}(c_{i_{Net}})] \cup \{r : c_i \xrightarrow{r} c_j \in \rightarrow_{\mathcal{R}}, i \neq j\}$.

The sorts of each configuration $c_{i_{Net}}$ define $\tilde{O}_J$, *i.e.* $\mathit{Sort}(c_{i_{Net}}) = O_i$. The Net transducer $T{=}(S_T, s_{0_T}, L_T, \rightarrow_T)$ orchestrates the system execution. Each configuration *Net* $c_{i_{Net}}$ is represented by a state $s_i \in S_T$.

Actions within a configuration are transitions over the same configuration state, *i.e.*, there is a transition $s_i \xrightarrow{a_g} s_i$ for each $a_g \in \mathit{Sort}(c_{i_{Net}}) \setminus \bigcup r$.

For each reconfiguration operation $c_i \xrightarrow{r} c_j$ we add a reconfiguration state s_r in the transducer (see Fig. 3). In this state, only actions within P_r can be performed, *i.e.*, $s_r \xrightarrow{r_\alpha} s_r$ where r_α are actions in P_r. The reconfiguration starts with an action $r_{start} = r_{s_i^\star}?$ that changes the transducer state from s_i to s_r. Reconfiguration actions are performed (r_α), and finally the system changes to a configuration c_j in a state defined by r by performing the action $r_{end} = r_{s_j^\star}?$.

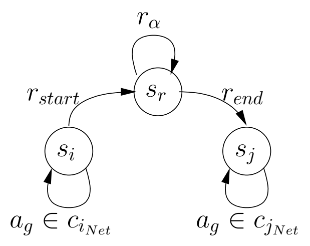


Fig. 3. Transducer representing a reconfiguration from configuration c_i to configuration c_j.

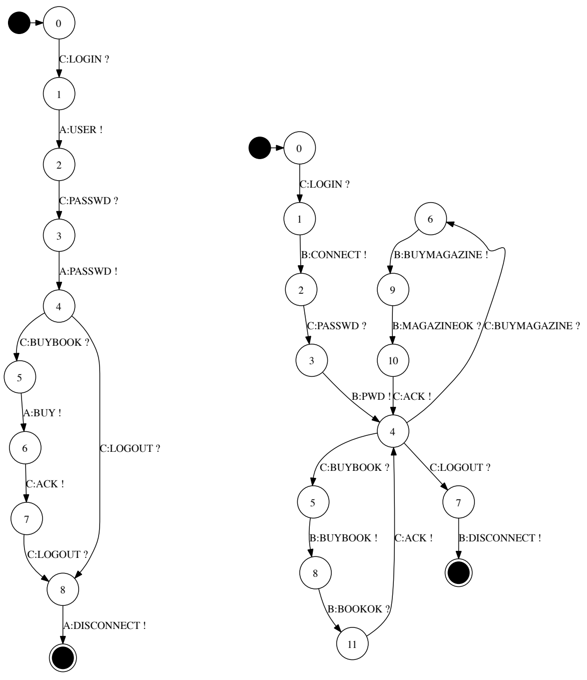
The reconfiguration contract is mapped into a Net $\mathcal{R}_{Net}$. The transducer of $\mathcal{R}_{Net}$ is an LTS that has each of the configurations as states, and transitions representing either a reconfiguration r or the allowance of a (non-reconfiguring) action a_g within a configuration.

Each of the configurations c_i is mapped into a Net $c_{i_{Net}}$ that is used as an argument of the $\mathcal{R}_{Net}$'s holes. Hence, we build a tree of processes in which the root is $\mathcal{R}_{Net}$, the internal nodes are configurations, and leaves are LTSs of the components' behaviour.

Reconfigurations operations eventually move the system from a configuration to another (see Fig. 3). The role of the state s_r is to halt system execution and initialise the new configuration so that it starts at the target reconfiguration state.

Example

Back to the running example, the first step in creating the *Net* is to generate the adaptors $A_{C,A}$ (Fig. 4(a)) and $A_{C,B}$ (Fig. 4(b)) in the form of LTSs. This is done by the `Compositor` tool [14]. Based on the adaptation contracts, `Compositor` automatically generates an adaptor for each configuration that is guaranteed to orchestrate deadlock-free interactions between the client and a server.



(a) Adaptor $A_{C,A}$ (b) Adaptor $A_{C,B}$

Fig. 4. Adaptors

The *Net* for c_A is created as follows [2]:

(i) We add a reconfiguration action $s_A^\star \xrightarrow{r_{s_A^\star}?} s_A^\star$ in the adaptor.

(ii) A_G has two actions for each vector in $\mathcal{AC}_{C,A}$, *i.e.*, $v_1 = \langle C : login!, A : user?\rangle$ generates the actions $v_1! = \langle C : login!, A_{C,A} : login?\rangle$ and $v_1? = \langle A_{C,A} : user!, A : user?\rangle$. A_G has the action $r_{s_A^\star}? = \langle A_{C,A} : r_{s_A^\star}?\rangle$.

(iii) LTSs of the components and the adaptor are the *Net*'s holes: $J = \{C, A, A_{C,A}\}$; $\tilde{O}_J = \{Sort(C), Sort(A), Sort(A_{C,A})\}$.

(iv) T has a unique state s_T, with $s_T \xrightarrow{a_g} s_T$, $a_g \in A_G$.

No initialisation of server B is needed after the reconfiguration because the system can only be reconfigured in its initial state. Therefore, P_r is a dummy automaton with a unique state and no transition.

[2] For the sake of size, the *Net* of c_B is created in a similar way.

Finally, we create $\mathcal{R}_{Net}$: the transducer has states s_A, s_B, $s_{r_{A,B}}$. Non-reconfiguration transitions are: $s_A \xrightarrow{a_g} s_A, a_g \in c_{A_{Net}}$ and $s_B \xrightarrow{a_g} s_B, a_g \in c_{B_{Net}}$. Reconfiguration transitions are: $s_A \xrightarrow{r_{s^\star_A}?} s_{r_{A,B}}$ and $s_{r_{A,B}} \xrightarrow{r_{s^\star_B}?} s_B$.

In this example, reconfiguration is only allowed in the initial state. Nevertheless, our goal is to allow reconfiguration in arbitrary states in which the client and the server have already started a transaction. This is the goal of Section 5.

5 Contract-Aware Reconfiguration

We have shown in the previous section that the running example can be reconfigured at the initial stage of the transaction. Nevertheless, more interesting are scenarios in which reconfiguration can take place at arbitrary stages of the transaction. With that purpose, Section 5.1 defines a *test* that determines whether it is possible to reconfigure the component system at a certain stage of the transaction. Afterwards, Section 5.2 studies how to design a fault-tolerant system in which reconfiguration can take place between two different configurations back and forth.

5.1 History-Aware Substitutability

Now, suppose the previous example needs to be reconfigured during an ongoing transaction between C and A. This would be the case where a client connects to the server A, logs in, and before disconnecting, A is substituted by B. Unfortunately, A and B do not provide such reconfiguration capabilities and it is not possible to substitute A by B without behavioural adaptation because they have different behavioural interfaces.

The client must not abort the ongoing transaction t. Therefore, if the execution trace of C (σ_C from now on) is valid in the new configuration c_B, then C could have been interacting with B from the very beginning. We proceed by initialising B under this assumption, *i.e.*, finding an execution trace σ_B such that the actions performed by C in t are feasible in c_B when B has performed σ_B.

As the adaptor orchestrates the execution between the two parties, we can use the state of the adaptor as the global system state. We say that $t = \langle \sigma_{p_1}, \sigma_{p_2} \rangle \rightsquigarrow_s A_{p_1,p_2}$ when actions performed by P_1 in σ_{p_1} and P_2 in σ_{p_2} lead the adaptor A_{p_1,p_2} to state s.

Definition 5.1 [Trace Compliant]. Let $t \rightsquigarrow_s A_{C,S}$, with $t = \langle \sigma_C, \sigma_S \rangle$ where σ_C and σ_S are traces of C and S respectively.

S' is *trace compliant* to S given t if there exists $t' = \langle \sigma_C, \sigma_{S'} \rangle \rightsquigarrow_{s'} A_{C,S'}$.

Not all components are *trace compliant* given t. However, if t' exists, the new configuration c_B can reach a (deadlock-free) state $s^\star_1$ that simulates the execution of c_A when the latter reaches the state $s^\star_0$. Therefore, it is possible to build a reconfigurable system that starts in a configuration c_A and may reconfigure towards a configuration c_B when the client has performed the actions in t. This reconfiguration does not affect the client in the sense it does not need to abort t nor to rollback. In fact, a new transaction t' is created such that the client continues working on transparently, though it is warned that the adaptation contract has changed; even so, its previous actions are valid and considered in the running transaction (t').

Trace compliance assures that the client's history is unchanged, though it says nothing about future actions. Therefore, it is possible to provide new functionality in the new configuration while maintaining consistency.

Example

In the running example, the login phase is similar in both configurations from the client's point of view. Therefore, it is easy to notice that both configurations are *trace compliant* given $\sigma_C = \{login!;\ passwd!\}$. This allows us to define a reconfiguration from the state $s_0^\star = \{C : s_2, A : s_2\}$ to the state $s_1^\star = \{C : s_2, B : s_2\}$. The initialisation required by B is the result of finding t', *i.e.*, $\sigma_B = \{connect?;\ pwd?\}$ (this specifies P_r). Therefore, meanwhile the client is logged in, the server can be substituted at runtime by another component with a different behavioural interface. Then, the client may log in server A, and after a reconfiguration to configuration c_B the client can buy several books and magazines.

On the other hand, c_B is not *trace compliant* to c_A given an arbitrary trace σ_C. For example, if the client has bought a magazine (or several books) in σ_C, then this trace cannot be simulated in c_A because A only supports transactions in which at one book is sold. Still, c_B is *trace compliant* to c_A for traces that do not include buying several books or at least one magazine; we shall explore this scenario in Section 5.2.

5.2 Fault-Tolerant System

The previous example allowed reconfiguring the system from c_A to c_B, though reconfiguring the system from c_B to c_A is only possible on some of the traces performed by the client. We investigate here how to design a fault-tolerant system by constraining the behaviour of the previous system. We will guarantee that server A can always be substituted by B (and B by A likewise) transparently from the client's point of view.

One way of ensuring two components are mutually substitutable is by means of a bisimulation equivalence [22]. A more relaxed equivalence takes into account only the behaviour of the servers given the constraints imposed by the environment. In our case, these constraints are summarised within the adaptors because they express the allowed behaviour performed by both parties.

An even more relaxed condition checks whether the adaptors $A_{C,A}$ and $A_{C,B}$ are bisimilar[3] from the client's point of view. There are two options: either to ignore the actions performed by the servers, or to map the actions of one server into another (through yet another adaptation contract). Both solutions allow servers to implement different behavioural interfaces; we have chosen to explore the former.

Even under this relaxed condition it is difficult to find two servers that provide clients with the same functionalities. More likely is that some of the actions allowed to the client are common in both configurations. Hence, if the system were to work on this reduced behaviour it would be possible to provide reconfiguration capabilities

[3] We leave open which kind of bisimulation is used; for example, strong, weak, or branching bisimulation.

as shown in Section 5.1. Our solution is to create new adaptors that are equivalent from the client's point of view. That is:

$$A^R_{C,A} \equiv_C A^R_{C,B} \text{ iff } \begin{cases} \forall\, t = \langle \sigma_C, \sigma_A \rangle \rightsquigarrow_{s_0^\star} A^R_{C,A},\ \ \exists\, t' = \langle \sigma_C, \sigma_B \rangle \rightsquigarrow_{s_1^\star} A^R_{C,B} \\ \forall\, t' = \langle \sigma_C, \sigma_B \rangle \rightsquigarrow_{s_1^\star} A^R_{C,B},\ \ \exists\, t = \langle \sigma_C, \sigma_A \rangle \rightsquigarrow_{s_0^\star} A^R_{C,A} \end{cases}$$

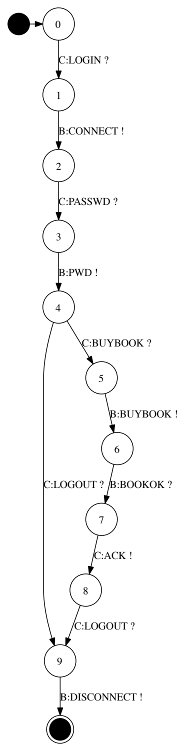
These are adaptors that constrain the behaviours of the client and the servers such that we can perform reconfiguration at any moment (from the client's point of view). They are found as follows:

(i) Compute $C^R = \mathit{Product}((\mathit{Hide}(L_A)\ \mathit{in}\ A_{C,A}), (\mathit{Hide}(L_B)\ \mathit{in}\ A_{C,A}))$. If C^R is deadlock-free, proceed.

(ii) Compute $A^R_{C,A} = \mathit{Product}(C^R, A_{C,A})$ and $A^R_{C,B} = \mathit{Product}(C^R, A_{C,B})$.

Product synchronises matching labels of both parties into a new synchronised action. *Hide* makes a set of labels L be internal actions (τ).

C^R finds the behaviour that can be performed by the client in both adaptors. This is done by hiding the actions performed by the servers (L_A and L_B) and building the synchronised product. If there are no traces that can be performed by the client in both adaptors, there is a deadlock in C^R and it is not possible to build a fault-tolerant system. Otherwise, using C^R , we constrain each adaptor to this client behaviour which yields, by construction, trace compliant $A^R_{C,A}$ and $A^R_{C,B}$ given any transaction.



Example

In the running example, the constrained system allows the client to buy at most one book in each transaction. In fact, we find that $A^R_{C,A} \equiv A_{C,A}$ but $A^R_{C,B}$ allows only the traces in which the client buys a book or nothing at all (see Fig. on the right). In this scenario, either server A or B suits the client's actions, and thus reconfiguration is possible.

Any σ_C feasible in one of these new adaptors is feasible in the other. Therefore, we can apply the procedure from Section 5.1 for finding pairs of states on which reconfiguration can be applied. We build this way a system that can be reconfigured from one configuration to the other back and forth.

6 Related Work

Dynamic reconfiguration [15] is not a new topic and many solutions have already been proposed in the context of distributed systems and software architectures [12,13], graph transformation [1,23], software adaptation [20,19], metamodelling [11], or reconfiguration patterns [8]. On the other hand, Software Adaptation is a recent solution to build component-based systems accessed and reused through their public interfaces. Adaptation is known as the only way to compose black-box components with mismatching interfaces. However, only few works have focused so far on the reconfiguration of systems whose correct execution is ensured using adaptor components. In the rest of this section, we focus on approaches that tackled reconfiguration aspects for systems developed using adaptation techniques.

First of all, in [20], the authors present some issues raised while dynamically reconfiguring behavioural adaptors. In particular, they present an example in which a pair of reconfigurations is successively applied to an adaptor due to the upgrade of a component in which some actions have been first removed and next added. No solution is proposed in this work to automate or support the adaptor reconfiguration when some changes occur in the system.

Most of the current adaptation proposals may be considered as global, since they proceed by computing global adaptors for closed systems made up of a predefined and fixed set of components. However, this is not satisfactory when the system may evolve, with components entering or leaving it at any time, *e.g.*, for pervasive computing. To enable adaptation on such systems, an incremental approach should be considered, by which the adaptation is dynamically reconfigured depending on the components present in the system. One of the first attempts in this direction is [4], whose proposal for incremental software construction by means of refinement allows for simple signature adaptation. However, to our knowledge the only proposal addressing incremental adaptation at the behavioural level is [21,19]. In these papers, the authors present a solution to build step by step a system consisting of several components which need some adaptations. To do so, they propose some techniques to (i) generate an adaptor for each new component added to the system, and (ii) reconfigure the system (components and adaptors) when a component is removed.

Compared to [20,21,19], our goal is slightly different since we do not want to directly reconfigure adaptor behaviours, but we want to substitute both a component and its adaptor by another couple component-adaptor while preserving some properties of the system such as trace compliance.

Some recent approaches found in the literature [6,17,16] focus on existing programming languages and platforms, such as BPEL or SCA components, and suggest manual or at most semi-automated techniques for solving behavioural mismatch. In particular, the work presented in [16] deal with the monitoring and adaptation of BPEL services at run-time according to Quality of Services attributes. Their approach also proposes replacement of partner services based on various strategies either syntactic or semantic. Although replaceability ideas presented in this paper are close to our reconfiguration problem, they mainly deal with QoS characteristics whereas our focus is on behavioural issues.

7 Conclusions

This paper has presented a novel framework that supports the design of reconfigurable systems. The formal model defines reconfiguration as a transition from a (static) configuration to another. Each configuration specifies a component assembly with its own set of components and connections, and a *reconfiguration contract* defines *when* the configuration can be reconfigured and *which* is the starting state in the new configuration in order to resume the execution.

We have integrated Software Adaptation in the framework in order to further enable reconfiguration. We have shown that, based on adaptation contracts, it is possible to substitute a component by another one that implements a different behavioural interface; this potentially includes mismatches in actions, ordering of actions, and functionality. Briefly, we build on the basis that for some cases it is possible to find execution traces in which configurations are similar in some sense, and thus it is possible to simulate the execution of a system in another one with a different behavioural interface.

Perspectives. We have presented an initial step on reconfiguration of components with mismatching behavioural interfaces. The framework is expressive and suitable for our needs and we have presented a case-study of runtime configuration. There are many open questions, though, that we wish to continue working on:

We have shown that in a simple setting it is possible to find system states in which it is safe to perform reconfiguration. Nevertheless, for more general scenarios finding correspondences between states based on trace compliance is not enough. We believe that it is necessary to refine what are the necessary properties the new configuration must hold. For example, by endowing a reconfiguration contract with invariants under the form of temporal properties.

An alternative is to augment component LTSs with sub-transactions. In our client/server example, the server would specify that a sub-transaction starts when buying a product and ends when the acknowledgement has been sent to the client. Then, it would be possible to buy magazines from server B (which is not supported by server A), and then substitute B by A.

We also plan to address other scenarios in which a component is substituted by a component assembly. For example, where the server is substituted by a component implemented by a front-end component and a back-end component. Our formal model can formalise the behaviour of such systems, though there are new challenges on how to simulate actions of the former configuration in the new one.

Finally, we plan to integrate this framework within a component model such as *Fractal*. We believe that the model built on *Nets* can be implemented in the runtime platform of Fractal in order to provide components with *safe* reconfiguration capabilities.

Acknowledgements. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Science and Innovation, and project P06-TIC-02250 funded by the Andalusian local Government.

References

[1] N. Aguirre and T. Maibaum. A Logical Basis for the Specification of Reconfigurable Component-Based Systems. In *Proc. of FASE'03*, volume 2621 of *LNCS*, pages 37–51. Springer, 2003.

[2] A. Arnold. *Finite transition systems. Semantics of communicating systems.* Prentice-Hall, 1994.

[3] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. SYNTHESIS: A Tool for Automatically Assembling

Correct and Distributed Component-based Systems. In *Proc. of ICSE'07*, pages 784–787. IEEE Computer Society, 2007.

[4] R. J. Back. Incremental Software Construction with Refinement Diagrams. Technical Report 660, Turku Center for Computer Science, 2005.

[5] T. Barros, R. Ameur-Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed fractal components. *Annals of Telecommunications*, 64(1):25–43, 2009.

[6] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2006.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[8] T. Bureš, P. Hnetynka, and F. Plášil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[9] J. Cámara, J. A. Martin, G. Salaün, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel. Itaca: An integrated toolbox for the automatic composition and adaptation of web services. In *ICSE*, pages 627–630. IEEE, 2009.

[10] C. Canal, P. Poizat, and G. Salaün. Model-Based Adaptation of Behavioural Mismatching Components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.

[11] A. Ketfi and N. Belkhatir. A Metamodel-Based Approach for the Dynamic Reconfiguration of Component-Based Software. In *Proc. of ICSR'04*, volume 3107 of *LNCS*, pages 264–273. Springer, 2004.

[12] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[13] J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.

[14] R. Mateescu, P. Poizat, and G. Salaün. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 84–99, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] N. Medvidovic. ADLs and Dynamic Architecture Changes. In *SIGSOFT 96 Workshop*, pages 24–27. ACM, 1996.

[16] O. Moser, F. Rosenberg, and S. Dustdar. Non-Intrusive Monitoring and Adaptation for WS-BPEL. In *Proc. of WWW'08*, pages 815–824, 2008.

[17] H. R. Motahari-Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-Automated Adaptation of Service Interactions. In *Proc. of WWW'07*, pages 993–1002, 2007.

[18] E. Najm, A. Lakas, A. Serouchni, E. Madelaine, and R. de Simone. ALTO: an interactive transformation tool for LOTOS and LOTOMATON. In *Proc. of Lotosphere Workshop and Seminar*, 1992.

[19] P. Poizat and G. Salaün. Adaptation of Open Component-Based Systems. In *Proc. of FMOODS'07*, volume 4468, pages 141–156. Springer, 2007.

[20] P. Poizat, G. Salaün, and M. Tivoli. On Dynamic Reconfiguration of Behavioural Adaptation. In *Proc. of WCAT'06*, pages 61–69, 2006.

[21] P. Poizat, G. Salaün, and M. Tivoli. An Adaptation-based Approach to Incrementally Build Component Systems. In *Proc. of FACS'06*, volume 182, pages 39–55, 2007.

[22] I. Černá, P. Vařeková, and B. Zimmerova. Component substitutability via equivalencies of component-interaction automata. *Electronic Notes in Theoretical Computer Science (ENTCS) series*, 182:39–55, 2007.

[23] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC / SIGSOFT FSE 2001*, pages 21–32. ACM, 2001.

[24] D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

PobSAM: Policy-based Managing of Actors in Self-Adaptive Systems

Narges Khakpour[1] , Saeed Jalili[2]

Tarbiat Modares University, Tehran, Iran

Carolyn Talcott[3]

SRI International, Menlo Park, California

Marjan Sirjani[4]

Reykjavk University, Reykjavk, Iceland
University of Tehran and IPM, Tehran, Iran

MohammadReza Mousavi[5]

Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract

In this paper, we present a formal model, named PobSAM (Policy-based Self-Adaptive Model), for modeling self-adaptive systems. In this model, policies are used as a mechanism to direct and adapt the behavior of self-adaptive systems. A PobSAM model consists of a set of self-managed modules(SMM). An SMM is a collection of autonomous managers and managed actors. Managed actors are dedicated to functional behavior while autonomous managers govern the behavior of managed actors by enforcing suitable policies. To adapt SMM behavior in response to changes, policies governing an SMM are adjusted, i.e., dynamic policies are used to govern and adapt system behavior. We employ the combination of an algebraic formalism and an actor-based model to specify this model formally. Managers are modeled as meta-actors whose policies are described using an algebra. Managed actors are expressed by an actor model. Furthermore, we provide an operational semantics for PobSAM described using labeled transition systems.

Keywords: Adaptive systems, Policy-based Computing, Component-based Design, Algebra, Actor Models

[1] Email: nkhakpour@modares.ac.ir
[2] Email: sjalili@modares.ac.ir
[3] Email: clt@cs.stanford.edu
[4] Email: msirjani@ut.ac.ir
[5] Email: m.r.mousavi@tue.nl

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Motivation 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 [1]: 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 in 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 as intended and avoids undesirable behavior. Hence, one of the main concerns in developing self-adaptive systems is providing mechanisms to trust whether the system is operating correctly, where *formal methods* can play a key role.

Zhang et al. [2] proposed a model-driven approach using Petri Nets for developing adaptive systems. They also presented a model-checking approach for verification of adaptive system [3,4] in which an extension of LTL with "adapt" operator was used to specify the adaptation requirements. In this work, system was modeled using a labeled transition system. Furthermore, authors in [5,6] used labeled transition systems at a low level of abstraction to model and verify embedded adaptive systems. Kulkarni et al. [7] proposed a theorem proving approach to verify the structural adaptation of adaptive systems.

Flexibility is another main concern to achieve adaptation. 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. All the proposed formal models hard-code the adaptation logic which leads to system's inflexibility. Recently, the use of policies has been given attention as a rich mechanism to achieve flexibility in adaptive system. A policy is a rule describing under which condition a specified subject must (can or cannot) do an action on a specific object. In [8,9,10,11,24], policies are used as a structural adaptation mechanism. Additionally, [12,13] proposed architectures for engineering autonomic computing systems that use policies for behavioral adaptations.

This paper In this paper we propose a formal model called PobSAM (Policy-based Self-Adaptive Model) for developing and specifying self-adaptive systems that employs policies as the principal paradigm to govern and adapt system behavior. We model a self-adaptive system as a collection of interacting actors directed to achieve particular goals according to the predefined policies. A PobSAM model consists of a set of Self-Managed Modules(SMMs). An SMM is composed of a collection of autonomous managers and managed actors. Autonomous managers are meta-actors responsible for monitoring and handling events by enforcing suitable policies. Each manager adapts its policies dynamically in response to the changing circumstances according to adaptation policies. The behavior of managed actors is governed by managers, and cannot be directly controlled from outside.

PobSAM has a formal foundation that employs an integration of algebraic formalisms and Actor-based models. The computational (functional) model of PobSAM is based on actor-based models while an algebraic approach is proposed to specify policies of managers. Operational semantics of PobSAM is described with labeled transition systems. The proposed model is suitable for cases where the set of governing policies of each context is known in advance.

In our previous work [14], we proposed a formal model for policy-based self-adaptive systems using an actor-based language Rebeca [15]. In [14], we added policies as rules to the Rebeca code and we focused on policy conflict detection. Combining adaptation concerns with system functionality in [14] increases the complexity of the model as well as the formal verification process. In order to address these drawbacks, we need an approach in which adaptation concerns are separated from system functionality. Here, we extracted the policy rules, specified by an algebraic formalism, from Rebeca code. Moreover, the policies are presented in two classes, separating the policies governing the actor behavior from the policies which determine the adaptation strategy (when and how the system passes the adaptation phase safely). Additionally, here we added modules as an encapsulation mechanism in which each module can manage itself autonomously.

Contribution Formal methods are proposed for the analysis of adaptive systems, mainly at the low levels of abstraction, and flexible policy-based approaches are proposed for designing adaptive systems without formal foundation. Here, we propose a flexible policy-based approach with formal foundation to support modeling and verification of self-adaptive systems. Policies allow us to separate the rules that govern the behavioral choices of a system from the system functionality giving us a higher level of abstraction; so, we can change system behavior without changing the code or functionality of the system. We are also concerned about the adaptation strategy, to pass the adaptation phase safely and at the right moment. As an example, we are able to change and reason about the scheduling of jobs using policies independent of the system code. Although our approach can support both structural and behavioral adaptation, in this paper, we focus on the behavioral adaptation. The formal foundation, the modular model, and separation of adaptation rules will help us in developing rigorous analysis techniques.

Structure of the paper This paper is organized as follows. In Section 2 we introduce an example to illustrate our approach. Section 3 introduces the PobSAM model in brief. Sections 4 and 5 introduce the syntax and semantics of PobSAM respectively. Section 6 presents related work and compares our approach with the existing approaches. In Section 7, we present our conclusions and plans for the future work.

2 Smart Home

In a home automation system, sensors are devices that provide smart home with the physical properties of the environment by sensing the environment. In addition, actuators are physical devices that can change the state of the world in response to the sensed data by sensors. The system processes the data gathered by the sensors, then it activates the actuators to alter the user environment according to the predefined set of policies. Smart homes can have different features. Here, we take into account three features including: (1) The lighting control which allows lights to switch on/off automatically depending on several factors. In addition, the intensity of the lights placed in a room can be adjusted according to the predefined policies. (2) Doors/Windows management that enable inhabitants to manage windows and doors automatically. In addition, if windows have blinds, these should be rolled up and down automatically too. (3) Heating control which allows inhabitants to adjust the heating of the house to their preferred value. The heating control will adjust itself automatically in order to save energy.

The smart home system is required to adapt its behavior according to the changes of the environment. To this aim, we suppose that the system runs in normal, vacation and fire modes and in each context it enforces various sets of policies to adapt to the current conditions. For the reason of space, here we only identify policies defined for lighting control module while the system runs in normal and fire modes as follows:

Defined policies in the normal mode

P1 Turn on the lights automatically when night begins.

P2 Whenever someone enters an empty room, the light setting must be set to default.

P3 When the room is reoccupied within T1 minutes after the last person has left the room, the last chosen light setting has to be reestablished.

P4 The system must turn the lights off, when the room is unoccupied.

Defined policies in the fire mode

P1 Turn on the emergency light.

P2 Disconnect power outlets.

P3 When the fire is distinguished, turn off the emergency light.

3 Modeling Concepts of PobSAM

Self-Managed Module (SMM) is the policy-based building block of PobSAM. A PobSAM is composed of a set of SMMs. An SMM, in turn, may contain a number of SMMs structured hierarchically. An SMM is a set of actors which can manage their behavior autonomously according to predefined policies. PobSAM 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. In the smart home case study, we consider three SMMs including *LightClModule*, *TempClModule* and *DWClModule* to manage lighting, temperature and doors/windows respectively.

An SMM structure can be conceptualized as the composition of three layers illustrated in Figure 1.

- **Managed Actors Layer** This layer is dedicated to the functional behavior of SMM and contains computational actors. Actors are governed by autonomous managers using policies to achieve predefined goals. Henceforth, we use the terms managed actors and actors interchangeably.
- **Autonomous Managers Layer** Autonomous managers are meta-actors that can operate in different configurations. Each configuration consists of two classes of policies: governing policies, and adaptation policies. Using governing policies,

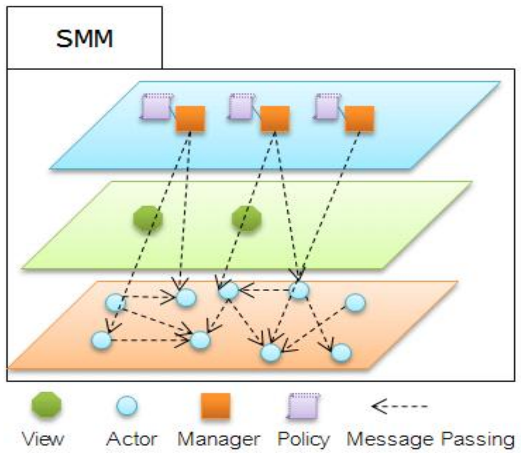


Fig. 1. The PobSAM Model

the manager directs the behavior of actors by sending messages to them. Adaptation policies are used to switch between different configurations to adapt the module behavior properly. Moreover, a manager may has its local variables too.

- **View Layer** In PobSAM, each actor provides its required state information to the relevant managers. Not all aspects of the operational environment have direct influence on the behavior of managers, the views provide only the required information for managers. The view layer is composed of views that provides a view or an abstraction of an actor's state that is adequate for the managers' needs. The distinction between the underlying computational environment and the required state information of actors makes analyzing managers much simpler.

Example 3.1 In our example, *LightClModule* module comprises an autonomous manager named *LightMngr*, two *light* actors (*light1* and *light2*), an outlet actor and a number of views indicating the overall light intensity of room and the status of the lights.

4 PobSAM Syntax

Figure 2 shows a typical SMM containing manager meta-actors, views and actors, which we elaborate in the sequel.

4.1 Actors

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 Rebeca is used. Rebeca [15] is an actor-based language for modeling concurrent asynchronous systems which allows us to model the system as a set of reactive objects called rebecs interacting by message passing. Each rebec provides methods called message servers (msgsrv) which can be invoked by others. Each rebec has an unbounded buffer for coming messages, called queue.

```
SMM SMM1
 Managers
   Manager ManagerName1(InitialConfiguration11)
     // definition of local variables
     Datatype var1;
     // manager's view
     <ViewName11,..., ViewName1n>;
     // definition of manager's configurations in terms of
     // their governing and adaptaion policies
     Configurations
         ConfigurationName1={gp11,...,gp1m}<ap11|...|ap1n>;
         .
         .
         .
     EndC
     Policies
         //Definition of governing policies(gps)
         GoverningPolicyName1: on eventi if condi do actionsi;
         .
         .
         .
         //Definition of adaptation policies(aps)
         AdaptationPolicyName1:
              on eventj if condj switchto Configuration1 when condk priority Oj;
         .
         .
         .
      EndP
    EndM
   // definition of other managers
 EndMS
 Views
   //definition of views
   Datatype1 ViewName1 as expr1;
   Datatype2 ViewName2 as expr2;
     .
     .
     .
 EndV
 Actors
   //definition of actors
   reactiveclass Classname1() {
      Knownrebecs{}
      Statevars{  Public datatype v1;
                  Private datatype v2;}
      msgsrv initial() {}
      msgsrv msgsrv1(){}
   }
   main {
      ...
      Classname1 rebec1(...):(...);
      ...
   }
 EndA
 EndSMM
```

Fig. 2. The Typical Syntax of a PobSAM Model

Furthermore, the rebecs' state variables (statevars) are responsible of capturing the rebec state. The known rebecs of a rebec (Knownrebecs) denotes the rebecs to which it can send messages. In our extension, an actor can expose a number of its state variables to the managers (Figure 2). The exposed state variables are used in the definition of views.

Example 4.1 In the *LightClModule* SMM, the managed layer comprises a set of *light* rebecs controlled by *LightMngr*. We consider a reactive class named *Light* to model the lights which contains *setIntensity*, *switchOn* and *switchOff* message servers as well as *intensity* and *status* state variables.

4.2 Views

In PobSAM, the views are defined in terms of the public state variables of actors. A view variable could be an actual state variable, or a function or a predicate applied to state variables. Views enable managers not to be concerned about internal behavior of actors and they provide an abstraction of actor's state to managers.

Example 4.2 In the *LightClModule* SMM, the *LightMngr* does not require the exact values of the lights intensities and providing overall intensity as low, medium or high values is sufficient to decide. The overall intensity is defined based on the *intensity* statevar of the *light* rebecs as a view.

4.3 Managers

In our model, policies direct the system behavior, and adaptation is achieved by changing policies. A manager can be in various configurations enforcing different policy sets. As shown in Figure 2, a manager is defined in terms of its possible configurations, its view of the actor layer and its local variables.

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

$$P, Q \overset{\text{def}}{=} P \cup Q | P - Q | P \cap Q | \{p\} | \emptyset$$

$P \cap Q$ means that intersection of policy sets P and Q is used to direct actors. P-Q reduces policy set P by eliminating all the policies in the second set Q. $P \cup Q$ represents the union of P and Q governing the actors simultaneously. {p} denotes a policy set with the simple policy p as its member. $P \cap Q$ and $P \cup Q$ are commutative and associative.

A simple action policy p=$[o,\varepsilon,\psi,\alpha]$ consists of priority o, event ε , optional condition ψ and action α. In PobSAM, events are defined as the execution of a message server, sending a message to a rebec, creating new actor or holding a specific condition in the system. Actions can be composite or simple. A simple action is of the form $r.\ell(\upsilon)$ which denotes message $\ell(\upsilon)$ is sent to actor r. Composite actions are created by composing simple actions as follows:

$$\alpha, \beta \overset{\text{def}}{=} \alpha;\beta | \alpha \parallel \beta | \alpha + \beta | [\omega ? \alpha : \beta] | r.\ell(\upsilon)$$

Thus a composite action can be the sequential $(\alpha;\beta)$ or parallel execution $(\alpha \parallel \beta)$ of actions α and β. Also, an action can be chosen to execute non-deterministically(+). Term $([\omega?\alpha : \beta])$ represents that action α is chosen to be executed if ω holds, else β will be chosen. + and $\parallel$ are commutative and associative.

Example 4.3 The governing policies of the *lightMngr* in the normal configuration are as follows, where *night*, *occpd*, *rccpd* and *unoccpd* denote events and *c1*, *c2*, *d1* and *d2* are arguments indicating the light intensity.

$$\begin{aligned}
p_{n1} &\overset{\text{def}}{=} [1, night, true, light1.switchon() \parallel light2.switchon()]\\
p_{n2} &\overset{\text{def}}{=} [2, occpd, true, light1.setIntensity(d1) \parallel light2.setIntensity(d2)]\\
p_{n3} &\overset{\text{def}}{=} [3, rccpd, true, light1.setIntensity(c1) \parallel light2.setIntensity(c2)]\\
p_{n4} &\overset{\text{def}}{=} [4, unoccpd, true, light1.switchoff() \parallel light2.switchoff()]
\end{aligned}$$

Adaptation Policies
One of the main characteristics of a formal model to specify a self-adaptive system is considering adaptation semantics. To this end, we should deal with a number of issues such as "the time at which an adaptation is performed in the system" , "the time at which the manager's policies are modified" , "the time at which the enforcement of new policies begin after modifying policies" or "the ability to restricting the system behavior during adaptation".

Whenever an event requiring adaptation occurs, relevant managers in different SMMs are informed. However, the adaptation cannot be done immediately and when the system reaches a safe state, the managers switch to the new configuration. Therefore, we introduce a new mode of operation named adaptation mode in which a manager runs before switching to the next configuration. While the manager is in the adaptation mode, it is likely that events occur which need to be handled by managers. To handle these cases, we introduce two kinds of adaptations named loose adaptation and strict adaptation. Under loose adaptation a manager enforces old policies, while under strict adaptation all events will be ignored until the manager exits the adaptation mode and the system reaches a safe state. For example in our smart home example, when *LightMngr* is in the fire configuration and there is a request for the vacation mode, while fire has not been put out, it keeps enforcing policies of the fire configuration by switching to the loose adaptation mode. Also, when *LightMngr* is in the normal configuration, once fire is detected, it stops enforcing its current policies by switching to the strict adaptation mode.

A simple configuration C is defined as $C \overset{\text{def}}{=} \langle P, A\rangle$ where P and A indicate the governing policy set and the adaptation policy of C respectively. Adaptation policies are defined using an algebraic language as follows:

$$A \overset{\text{def}}{=} \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta} | A \oplus A$$

in which D, δ , γ , λ and ϑ, respectively denote an arbitrary configuration, the conditions of triggering adaptation, the conditions of applying adaptation, adaptation type (loose or strict) and the priority of adaptation policy. The simple adaptation policy, $\lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta}$, 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 becomes true, it will perform adaptation. Adaptation policies of a manager is defined as the composition, $\oplus$, of the simple adaptation policies. Here, composition of two policies means that those policies are potentially to be triggered.

$\oplus$ is associative and commutative. D is defined as follows where ω is an arbitrary condition:

$$D, D' \stackrel{\text{def}}{=} [\omega?D : D']|D\Box D'|C$$

Terms $[\omega?D : D']$ and $D\Box D'$ represent conditional and non-deterministic choices respectively. In conditional choice, configuration D is chosen if ω holds, else D' will be chosen. Non-deterministic choice means that the choice between configuration D or D' is made non-deterministically. This operator is associative and commutative.

Example 4.4 In our smart home example, there are three configurations including C_n, C_v and C_f. Formal specification of *lightMngr*'s configurations are as follows in which p_{mj} denotes policy j defined in mode C_m. *vacReq*, *fire*, *firePutout* and *comebackHome* are events while *isPutout* , *isCnfrmd* and *onVac* are *lightMngr*'s local variables. As an example, when the *lightMngr* is in the *fire* mode and there is a request for going on vacation, while fire has net been put out, it can not switch to the C_v configuration.

$C_n \stackrel{\text{def}}{=} \langle\{p_{n1}, p_{n2}, p_{n3}, p_{n4}\}, \lfloor C_f\rfloor_{fire,true,S,1} \oplus \lfloor C_v\rfloor_{vacReq,isCnfrmd,L,2}\rangle$

$C_f \stackrel{\text{def}}{=} \langle\{p_{f1}, p_{f2}, p_{f3}\}, \lfloor onVac?C_v : C_n\rfloor_{firePutout,true,L,1} \oplus \lfloor C_v\rfloor_{vacRequest,isPutout,L,2}\rangle$

$C_v \stackrel{\text{def}}{=} \langle\{p_{v1}, p_{v2}, p_{v3}, p_{v4}, p_5\}, \lfloor C_f\rfloor_{fire,true,S,1} \oplus \lfloor C_n\rfloor_{comebackHome,true,L,2}\rangle$

5 Operational Semantics of PobSAM

5.1 Operational Semantics of Actors and Views

The operational semantics of our extension of Rebeca does not differ from that of Rebeca[15]. However, any changes in state of the rebecs used in the definition of views must be reflected to the views. Let $I_1, I_2, \ldots, I_n$ denote the defined views of SMM S and η denote the set of defined events that S is concerned with. The state of a view is determined by its current value that is modified by the related events occurring at the actor level. After execution of a message server, the changes of public state variables must be reflected in the views state, too. We specify the operational semantics of the view layer as a labeled transition system.

Let S_B, A_B and $T_B \subseteq S_B\times A_B\times S_B$ be the set of states, the set of actions and the state transition relation of the transition system of the actor layer respectively. The state transition relation of the view layer $T_I \subseteq S_I \times A_I \times S_I$ is defined based on T_B, where S_I and A_I are the set of states and the set of actions of the view layer transition system respectively and $S_I = \langle I_1, I_2, ..., I_n\rangle$. Suppose $I_j(x_1, x_2, .., x_m)$ denotes an arbitrary view defined on public state variables $x_1, x_2, .., x_m$ and $I_j|_{\sigma_s}$ denotes the state of I_j , where its state variables are substituted with their corresponding values in state $\sigma_s in S_B$. For each triple $\langle\sigma_s, a, \sigma_t\rangle \in T_B$, we consider an associated transition $\langle\sigma'_s, a, \sigma'_t\rangle \in T_I$ where ${\sigma'}_s = \langle I_1|_{\sigma_s}, I_2|\sigma_s, ..., I_n|\sigma_s\rangle$ and $\sigma'_t = \langle I_1|_{\sigma_t}, I_2|_{\sigma_t}, ..., I_n|_{\sigma_t}\rangle$ if and only if $a\in \eta$ or $\exists I_k|1 \leq k \leq n \wedge I_k|_{\sigma_s} \neq I_k|_{\sigma_t}$, i.e.

$$\frac{\sigma_s \xrightarrow{a} \sigma_t \in T_B, (a \in \eta) \vee (\exists I_k|1 \leq k \leq n \wedge I_k|_{\sigma_s} \neq I_k|_{\sigma_t})}{\sigma'_s \xrightarrow{a} \sigma'_t}$$

$$(\text{NPE1})\frac{\mathcal{R} = \{\text{p} | (p.\varepsilon \wedge p.\psi) = true\} \wedge \mathcal{R} \neq \emptyset}{\mathcal{M}_{\emptyset,\emptyset}C[0] \xrightarrow{\varepsilon,\tau} [\mathcal{M}]_{\mathcal{R},\emptyset}C[0]} \quad (\text{NPE2})\frac{p \in \mathcal{R} \wedge (\nexists q \in \mathcal{R} | q.o > p.o)}{[\mathcal{M}]_{\mathcal{R},\emptyset}C[0] \xrightarrow{p.\varepsilon \wedge p.\psi, enf(p)} [\mathcal{M}]_{\mathcal{R},p}C[p.\alpha]}$$

$$(\text{NPE3})\frac{\alpha \longrightarrow \alpha'}{[\mathcal{M}]_{\mathcal{R},p}C[\alpha + \beta] \xrightarrow{true,\tau} [\mathcal{M}]_{\mathcal{R},p}C[\alpha']}$$

$$(\text{NPE4})\frac{}{[\mathcal{M}]_{\mathcal{R},p}C[[\omega?\alpha : \beta]] \xrightarrow{\omega,\tau} [\mathcal{M}]_{\mathcal{R},p}C[\alpha]} \quad (\text{NPE5})\frac{}{[\mathcal{M}]_{\mathcal{R},p}C[[\omega?\alpha : \beta]] \xrightarrow{\neg\omega,\tau} [\mathcal{M}]_{\mathcal{R},p}C[\beta]}$$

$$(\text{NPE6})\frac{\alpha \rightarrow \alpha'}{[\mathcal{M}]_{\mathcal{R},p}C[\alpha \parallel \beta] \xrightarrow{true,\tau} [\mathcal{M}]_{\mathcal{R},p}C[\alpha' \parallel \beta]}$$

$$(\text{NPE7})\frac{}{[\mathcal{M}]_{\mathcal{R},p}C[r.\ell(v); r'.\ell'(v')] \xrightarrow{\ell,\ell' \notin \{sendAck,waitAck\},\tau} [\mathcal{M}]_{\mathcal{R},p}C[r.\ell(v); r.sendAck(r'); r'.waitAck(sendAck(r')); r'.\ell'(v')]}$$

$$(\text{NPE8})\frac{}{[\mathcal{M}]_{\mathcal{R},p}C[r.\ell(v); \beta] \xrightarrow{true,send(r,\ell(v))} [\mathcal{M}]_{\mathcal{R},p}C[\beta]}$$

$$(\text{NPE9})\frac{}{[\mathcal{M}]_{\mathcal{R},p}C[0] \xrightarrow{true,\tau} [\mathcal{M}]_{\mathcal{R}-p,\emptyset}C[0]} \quad (\text{NPE10})\frac{}{[\mathcal{M}]_{\emptyset,\emptyset}C[0] \xrightarrow{true,\tau} \mathcal{M}_{\emptyset,\emptyset}C[0]}$$

Fig. 3. Rules of policy enforcement

5.2 Operational Semantics of Managers

We use a labeled transition system to define the operational semantics of managers in which labels have two components. The first component indicates the activation condition of the transition while the second component denotes the action of the transition. Assume that M is a logical expression defined on state variables, the transitions are of the form $P \xrightarrow{M,a} Q$ meaning "if M holds then P has an action a leading to Q". Henceforth, we denote a transition by $P \xrightarrow{\mu} Q$ where $\mu = (M, a)$.

The behavior of a manager depends on the mode in which it is running. A manager can run in different modes such as normal execution, adaptation and policy enforcement. To distinguish managers in different modes, we use different notations. Let $[\mathcal{M}]^s_{\mathcal{R},p}C[\alpha]$ indicate manager $\mathcal{M}$ in the enforcement mode in which,

- C is the configuration in which $\mathcal{M}$ is running and $C \overset{\text{def}}{=} \langle P, A\rangle$.
- $\mathcal{R}$ is the set of triggered policies to be enforced.
- p is the current policy being enforced by $\mathcal{M}$.
- α is the action of a recent policy being executed by $\mathcal{M}$.
- s denotes $\mathcal{M}$'s view of current context in addition to its local variables.

$\mathcal{M}^s_{\emptyset,\emptyset}C[0]$, $|\mathcal{M}|^s_{\mathcal{R},p}C[\alpha]$ and $\|\mathcal{M}\|^s_{\emptyset,\emptyset}C[0]$ indicate $\mathcal{M}$ in normal execution, loose adaptation and strict adaptation modes respectively. We ignore s in the definition of operational semantics of managers.

Policy enforcement semantics

Whenever an event is received by a manager, it identifies all the triggered policies with the policy condition evaluated to true. Then it enforces the identified policies based on their priorities. Once a manager enforces all the triggered policies, it evolves to normal mode. Figure 3 gives rules of policy enforcement.

Using NPE1 manager switches to the enforcement mode by identifying the triggered policies to be enforced. NPE2 places the action of a policy with the highest priority in the action part of the manager to be run. NPE3, NPE4 and NPE5 define the semantics of non-deterministic and conditional choices of actions. NPE7 is considered to apply two sequential actions in which r' is an arbitrary actor. To this aim, we use synchronous message passing provided by Extended Rebeca [16]. The corresponding actors of two sequential actions are synchronized after execution of the first action. We considered two message servers for each reactive class named $\mathit{sendAck}(r)$ and $\mathit{waitAck}(r)$ which sends and receives a synchronization message to and from rebec r respectively. NPE8 expresses sending a message to an actor and removing it from the list of actions to be executed. After applying policy p, NPE9 will remove p from the list of activated policies. When there is no policy to be enforced, manager will switch to normal execution mode using NPE10.

As mentioned above, managers in loose adaptation mode are able to enforce policies. Therefore, all the rules introduced for the enforcement mode, except for NPE10, are applicable in loose adaptation mode too.

Policy adaptation semantics

For the sake of readability, we omit p, $[\alpha]$ and $\mathcal{R}$ symbols of managers in normal execution and strict adaptation modes. Figure 4 shows rules for adaptation in strict mode in which B, A′ and B′ denote arbitrary adaptation policies. Furthermore, F and D'' denote a simple adaptation policy and an arbitrary configuration, respectively.

$$(\text{SA1})\frac{\nexists F \in A | (\vartheta < F.\vartheta \wedge F.\delta = true)}{\mathcal{M}\langle P, \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta} \oplus A\rangle \xrightarrow{\delta\wedge\lambda,\tau} \|\mathcal{M}\|\langle P, \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$$

$$(\text{SA2})\frac{}{\|\mathcal{M}\|\langle P, \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\gamma,adapt} \mathcal{M}_{\emptyset,\emptyset}D}$$

$$(\text{SA3})\frac{\mathcal{M}\langle P, B\rangle \xrightarrow{\mu} \mathcal{M}\langle P, B'\rangle}{\mathcal{M}\langle P, A \oplus B\rangle \xrightarrow{\mu} \mathcal{M}\langle P, A \oplus B'\rangle} \qquad (\text{SA4})\frac{\mathcal{M}\langle P, A\rangle \xrightarrow{\mu} \mathcal{M}\langle P, A'\rangle}{\mathcal{M}\langle P, A \oplus B\rangle \xrightarrow{\mu} \mathcal{M}\langle P, A' \oplus B\rangle}$$

$$(\text{SA5})\frac{\|\mathcal{M}\|\langle P, \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\mu} \|\mathcal{M}\|\langle P, \lfloor D' \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}{\|\mathcal{M}\|\langle P, \lfloor D \Box D'' \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\mu} \|\mathcal{M}\|\langle P, \lfloor D' \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$$

$$(\text{SA6})\frac{\|\mathcal{M}\|\langle P, \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\mu} \|\mathcal{M}\|\langle P, \lfloor D' \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}{\|\mathcal{M}\|\langle P, \lfloor D'' \Box D \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\mu} \|\mathcal{M}\|\langle P, \lfloor D' \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$$

$$(\text{SA7})\frac{}{\|\mathcal{M}\|\langle P, \lfloor \omega ? D : D' \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\omega,\tau} \|\mathcal{M}\|\langle P, \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$$

$$(\text{SA8})\frac{}{\|\mathcal{M}\|\langle P, \lfloor \omega ? D : D' \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle \xrightarrow{\neg\omega,\tau} \|\mathcal{M}\|\langle P, \lfloor D' \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$$

Fig. 4. Rules of strict adaptation

SA1 states that in the case of strict adaptation, when adaptation policy conditions hold, manager $\mathcal{M}$ switches to the strict adaptation mode. SA2 asserts that when the condition for applying the adaptation holds, $\mathcal{M}$ will evolve to normal mode and run configuration D. SA5, SA6, SA7 and SA8 define the semantics of non-deterministic and conditional choices of configurations. SA5, SA6, SA7 and SA8 rules have a higher priority than SA2. To this aim, we use the ordered SOS (Structural Operational Semantics) framework [17] and place SA5, SA6, SA7 and SA8 above SA2. Rules of loose adaptation are identical to strict adaptations rules except for SA1 which is as $\dfrac{\nexists F \in A | (\vartheta < F.\vartheta \wedge F.\delta = true)}{\mathcal{M}\langle P, \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta} \oplus A\rangle \xrightarrow{\delta \wedge \neg\lambda, \tau} |\mathcal{M}|\langle P, \lfloor D \rfloor_{\delta,\gamma,\lambda,\vartheta}\rangle}$(LA1).

5.3 Interaction of Managers and Views

The other kind of transitions is related to the interaction of managers and view layer. Figure 5 demonstrates rules for interaction of managers and the view layer of S where $\sigma_s \xrightarrow{a} \sigma_t \in \mathrm{T_I}$ and t indicates the new state of M. s_v and t_v are defined as the projection of σ_s and σ_t on M's view respectively, i.e. $s_v = \sigma_s \uparrow M.v$ and $t_v = \sigma_t \uparrow M.v$. Construct $\sigma_s \uparrow M.v$ denotes only the state variables of σ_s that are in $M.v$ too. IR1, IR2, IR3 and IR4 express changing $\mathcal{M}$'s view of the view layer by state changing at the view layer in the normal, enforcement, loose adaptation and strict adaptation modes respectively, where s(t) is union of $s_v(t_v)$ and lv. lv indicates the current state of M in terms of its local variables.

$$\text{(IR1)}\frac{\sigma_s \xrightarrow{a} \sigma_t, \exists I_j \in s | I_j|_{\sigma_s} \neq I_j|_{\sigma_t}}{\mathcal{M}^s_{\emptyset,\emptyset}\langle P, A\rangle[0] \xrightarrow{true,\tau} \mathcal{M}^t_{\emptyset,\emptyset}\langle P, A\rangle[0]} \qquad \text{(IR2)}\frac{\sigma_s \xrightarrow{a} \sigma_t, \exists I_j \in s | I_j|_{\sigma_s} \neq I_j|_{\sigma_t}}{[\mathcal{M}]^s_{\emptyset,\emptyset}\langle P, A\rangle[0] \xrightarrow{true,\tau} [\mathcal{M}]^t_{\emptyset,\emptyset}\langle P, A\rangle[0]}$$

$$\text{(IR3)}\frac{\sigma_s \xrightarrow{a} \sigma_t, \exists I_j \in s | I_j|_{\sigma_s} \neq I_j|_{\sigma_t}}{|\mathcal{M}|^s_{\mathcal{R},p}\langle P, A\rangle[0] \xrightarrow{true,\tau} |\mathcal{M}|^t_{\mathcal{R},p}\langle P, A\rangle[0]} \qquad \text{(IR4)}\frac{\sigma_s \xrightarrow{a} \sigma_t, \exists I_j \in s | I_j|_{\sigma_s} \neq I_j|_{\sigma_t}}{\|\mathcal{M}\|^s_{\emptyset,\emptyset}\langle P, A\rangle[0] \xrightarrow{true,\tau} \|\mathcal{M}\|^t_{\emptyset,\emptyset}\langle P, A\rangle[0]}$$

Fig. 5. Rules for interaction of managers and the view layer

6 Discussion and Related Work

Flexibility of self-adaptive systems is realized by three different features including separation of concerns, computational reflection and component-based design [18]. We explain how PobSAM can address these requirements in the sequel.

PobSAM decouples the adaptation logic of an SMM from its business logic described at an abstract level using policies. Among the proposed formal approaches to model adaptive systems, [2,20,21] combine the adaptation logic into the business logics. In [3,6,19] the adaptation concerns have been separated; however, all the proposed formal models hard-code the adaptation logic which leads to system's inflexibility. The proposed model permits us to direct/adapt system behavior by enforcing/modifying policies at an abstract level without re-coding actors and managers; thereby it leads to increasing system flexibility and scalability.

Computational reflection is the ability of a system to monitor and change its behavior subsequently. In PobSAM, managers monitor actor's behavior through views and direct/adapt SMMs behavior. Policies provide us a high-level description of what we want without dealing with how to achieve it. Thus, using policies can be a suitable mechanism to determine if the goals were achieved using existing policy refinement techniques.

Furthermore, PobSAM uses SMM as a policy-based building block for a modular model where each component is able to adapt its behavior autonomously. This notion makes PobSAM a suitable model to specify self-organizing and cooperating systems too. Although, in this paper we focused on behavioral adaptation, however, PobSAM can support structural adaptation by joining/leaving an actor or an SMM to/from an SMM dynamically, which is an advantage over the most existing approaches that concentrate on one adaptation type. SMM notion is similar to Self-Managed Cell (SMC) notion proposed in [13] as a paradigm for engineering ubiquitous systems. In this work, an SMC consists of a set of components that constructs an autonomous management domain.

One of the main aspects of modeling a self-adaptive system is specifying adaptation requirements. To this aim, we introduced a two phases adaptation strategy to pass the adaptation phase safely. Upon receiving an adaptation event by a manager, it switches to the adaptation mode. Adaptation mode models transient states during adaptation. When the system reaches a safe state, the adaptation is completed by evolving the manager to the new configuration. We believe that the modular nature of adaptation policies enables us to express adaptation requirements easily and at the high-level of abstraction.

As stated above, PobSAM has decoupled the adaptation layer from the functional layer. Thus, we can verify the adaptation layer independently from the actor layer provided that we have a labeled transition system modeling view behavior. This feature can decrease the complexity of verification procedure.

Dynamic adaptation is a very diverse area of research. While structural adaptation has been given strong attention in the research community(see [22]), fewer approaches tackle behavioral adaptation as we considered. Due to the lack of space, we restrict ourselves to present related work done on formal modeling of self-adaptive systems in addition to applying policy-based approaches in engineering of self-adaptive systems.

Formal verification of adaptive systems is a young research area [23] and only a few research groups already focused on this topic. A model-driven approach was proposed for developing adaptive systems in [2]. In this approach, there are different behavioral variants of a process modeled as Petri Nets. At each time, one Petri Net runs and reconfiguration is carried out by switching between various Petri Nets. In another work [3], they modeled a system as a set of steady-state programs among which the system switches. An extension of LTL with "adapt" operator was used to specify adaptation requirements before, during and after adaptation [4]. Then, they use a model checking approach to verify the system. Kulkarni et al. [7] proposed an approach based on the concept of proof lattice to verify if a system is in a correct state during and after adaptation in terms of satisfying the transitional-invariants. Furthermore, Schneider et al. [5] presented a method to describe adaptation behavior at an abstract level. After deriving transition systems from a system description, they verify the system using model checking. In their later work [6], they proposed a framework for model-based development of adaptive embedded systems using labeled transition systems. In this work, they verify different properties using theorem proving, model checking and specialized verification methods. [19] proposed a coordination protocol for distributed adaptation of the component-based systems and used Colored Petri Nets for formal verification. In our model, adaptation is performed by applying suitable policies in different contexts, which in nature differs from the proposed approaches. We have proposed a formal model of policy-based self-adaptive systems using Rebeca concentrated on policy conflict detection [14]. Combining adaptation concerns with system functionality in this approach causes an increase in the complexity of model as well as formal verification process.

Employing policies as a paradigm to adapt self-adaptive systems has been given considerable attention during recent years. Work in [8,9,10,11,24,26] used policies as the adaptation logic for structural adaptation, while we use policies as a mechanism for governing as well as adapting system behavior. Furthermore, [24] used policies for a simple type of behavioral adaptation named parameterization, too. [25] proposed an adaptive architecture for management of differentiated networks which performs adaptation by enabling/disabling a policy from a set of predefined QoS policies, but this architecture does not have formal foundation. Anthony [26] presents a policy definition language for autonomic computing systems in which the policies themselves can be modified dynamically to match environmental conditions. However, this work does not deal with modeling system and it is limited to proposing an informal policy language.

7 Conclusions and Future Work

We proposed PobSAM as a formal model to develop self-adaptive systems which uses policies as the main mechanism to govern and adapt the system behavior. To this aim, we model a system as the composition of a set of autonomous components named SMMs. Each SMM contains two types of actors: managed actors that are dedicated to the functional layer of system and autonomous managers that coordinate actors to achieve the predefined goals using policies. This model integrates two formal methods including algebra and actor-based model to specify a system. Then, we presented the operational semantics of PobSAM by means of labeled transition systems.

There is much more research to pursue in the area of verification of self-adaptive systems. In this paper, we focused on formal modeling of self-adaptive systems. Verification of different properties of adaptation and functional layers of PobSAM models is an ongoing work. We are going to implement a tool to support our approach too. As our model can support both behavioral and structural adaptations, our future researches will be concentrated on specifying structural adaptations. Extending this model for modeling self-organizing systems in which managers need to coordinate together is considered as a future work, too.

References

[1] C. Hofmeister, “Dynamic Reconfiguration, Ph.D. Thesis” in Computer Science Department, University of Maryland, College Park 1993.

[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] J. Zhang, H. J. Goldsby, and B. H. C. Cheng, “Modular verification of dynamically adaptive systems”, in the 8th ACM international conference on Aspect-oriented software development, Charlottesville, Virginia, 2009, pp. 161-172.

[4] J. Zhang and B. H. C. Cheng, “Using temporal logic to specify adaptive program semantics”, Journal of Systems and Software, Architecting Dependable Systems, vol. 79, pp. 1361-1369, 2006.

[5] 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.

[6] 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.

[7] S. S. Kulkarni and K. N.Biyani, “Correctness of Component-Based Adaptation”, in Component-Based Software Engineering, 2004, pp. 48-58.

[8] 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.

[9] 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.

[10] C. Efstratiou, K. Cheverst, N. Davies, and A. Friday, “Utilising the Event Calculus for Policy Driven Adaptation in Mobile Systems”, in Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks, Monterey, California., 2002, pp. 13-24.

[11] G. Phil and B. Lynne, “A Framework for Policy Driven Auto-adaptive Systems Using Dynamic Framed Aspects”, in Transactions on Aspect-Oriented Software Development II. vol. LNCS 4242, 2006, pp. 30-65.

[12] “Autonomic computing”, IBM Systems Journal, vol. 42, 2003.

[13] M. Sloman and E. Lupu, “Engineering Policy-Based Ubiquitous Systems”, The Computer Journal, to appear, 2009.

[14] N. Khakpour, R. Khosravi, M. Sirjani, and S. Jalili, “A Formal Model for Policy-based Self-Adaptive Systems”, Submitted, 2009.

[15] 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.

[16] M. Sirjani, F. d. Boer, A. Movaghar, and A. Shali, “Extended Rebeca: A Component-Based Actor Language with Synchronous Message Passing”, in Proceedings of the Fifth International Conference on Application of Concurrency to System Design: IEEE Computer Society, 2005.

[17] M. Mousavi, I. Phillips, M. A. Reniers, and I. Ulidowski, ”Semantics and expressiveness of ordered SOS,” Information and Computation, vol. 207, pp. 85-119, 2009.

[18] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, “A Taxonomy of Compositional Adaptation”, Michigan State University Technical Report MSU-CSE-04-17, 2004.

[19] N. H. Kacem, A. H. Kacem, and K. Drira, “A Formal Model of a Multi-step Coordination Protocol for Self-adaptive Software Using Coloured Petri Nets”, International Journal of Computing and Information Sciences , 2009. To appear.

[20] K. N. Biyani and S. S. Kulkarni, “Concurrency and Complexity in Verifying Dynamic Adaptation: A Case Study”, Michigan State University Technical Report MSU-CSE-05-21, August 2005.

[21] M. Gudemann, F. Ortmeier, and W. Reif, “Safety and Dependability Analysis of Self-Adaptive Systems”, in the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, 2007, pp. 177-184.

[22] 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.

[23] 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.

[24] D. Pierre-Charles and L. Thomas, “Towards a Framework for Self-adaptive Component-Based Applications”, in Proceedings of Distributed Applications and Interoperable Systems. LNCS2893, 2003, pp. 1-14.

[25] L. Lymberopoulos, E. Lupu, and M. Sloman, “An Adaptive Policy Based Management Framework for Differentiated Services Networks”, in Proceedings of IEEE Workshop on Policies for Distributed Systems and Networks, Monterey, California, Los Alamitos, California, 2002, pp. 147-158.

[26] R. J. Anthony, “Generic Support for Policy-Based Self-Adaptive Systems”, in Proceedings of the 17th International Conference on Database and Expert Systems Applications, 2006, pp. 108-113.

A systematic approach to construct compositional behaviour models for network-structured safety-critical systems

Johannes Kloos and Robert Eschbach [1]

Testing and Inspections
Fraunhofer Institute for Experimental Software Engineering
Kaiserslautern, Germany

Abstract

This paper considers the problem of model-based testing of a class of safety-critical systems. These systems are built up from components that are connected a network-like structure. The number of possible structures is usually large. In particular, we consider the following issue: For many of these systems, each instance needs its own set of models for testing. On the other hand, the instances that should be tested will have to be chosen so that the reliability statements are generally applicable. Thus, they must be chosen by a domain expert. The approach in this paper addresses both of these points. The structure of the instance of system under test is described using a domain-specific language, so that a domain expert can easily describe a system instance for testing. At the same time, the components and composition operators are formalized. Using a structure description written in the DSL, corresponding test models can be automatically generated, allowing for automated testing by the domain expert. We show some evidence about the feasibility of our approach and about the effort required for modelling an example, supporting our belief that our approach improves both on the efficiency and the expressivity of current compositional test model construction techniques.

1 Introduction

Safety-critical embedded systems such as control applications in industrial automation or train automation are very complex due to their broad range of possible applications. Among these systems, industrial plant controllers, railroad coordination systems, logistic planning systems etc. form a class that is especially adaptable to many different application scenarios. One thing that is common to many of these systems is that they control an underlying network of physical entities that are connected in some way, often with several hierarchy levels. The network formed by these entities induces data flows between neighbouring components and vertically through the different levels of hierarchy. These data flows are observed and manipulated by the control system. As an example, a railway control system for a given segment of the railroad network will use the underlying structure of the tracks, switches and

[1] Email: ⟨firstname⟩.⟨lastname⟩@iese.fraunhofer.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

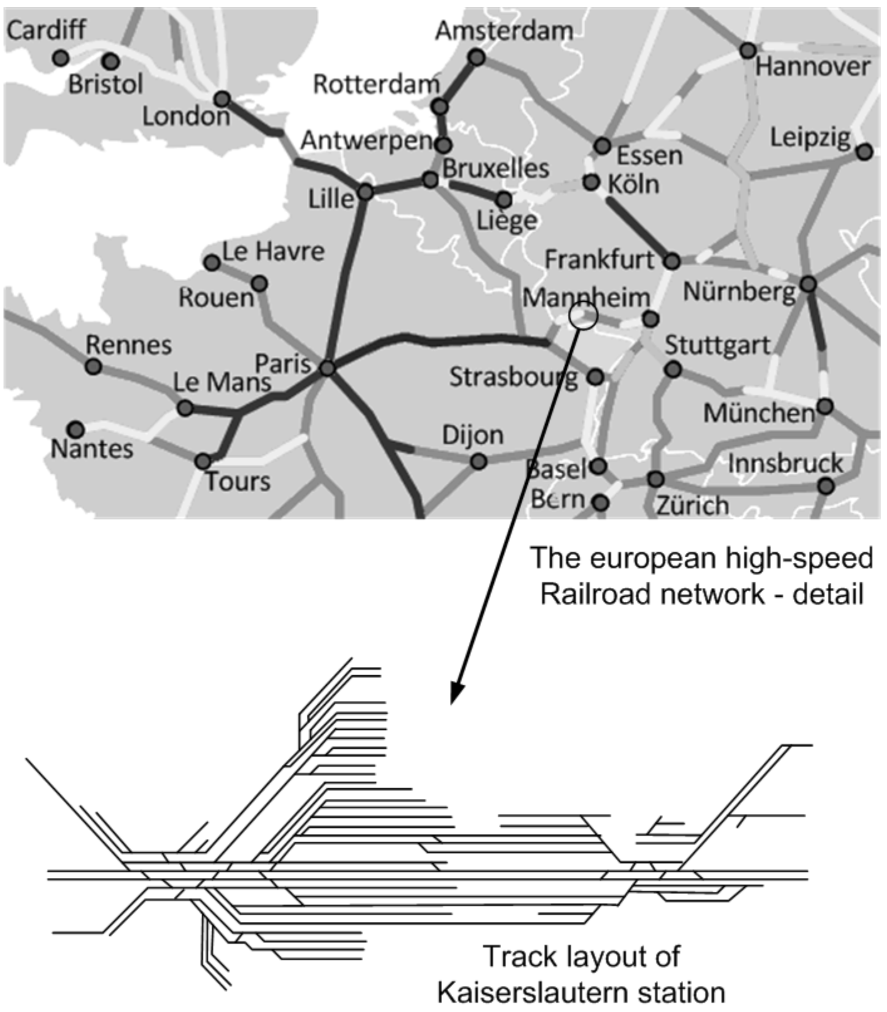


Fig. 1. An example of a hierarchical network structure: The European high-speed railroad network and a detail, Kaiserslautern station

signals on the lowest level of its internal model, aggregating them into larger control units that finally form the network (cf. figure 1). We will call this kind of systems "network-structured systems".

To handle the complexity inherent in these structures, one way to construct such embedded control systems is a modular, component-based approach. For each installation of such a system, e.g., for a given industrial plant or a given railway network, an instance of the embedded system is created by instantiating some basic component types and plugging them together in an appropriate way. For many of these systems, one finds that the number of possible instances, is huge and effectively unbounded.

From the perspective of quality assurance it is not clear how to test such a huge space of instances w.r.t. different quality properties like correctness, safety or reliability. Reliability is one of the most important non-functional quality properties of embedded software systems required by contract or different standards like the

IEC 61508 [10], IEC 61511 [9], CENELEC 50126 [3], ISO CD 26262 [11] or DO178B [17]. Measures like mean time between failures (MTBF) or failure rates are typical reliability indicators.

The problems that a tester faces for these systems are twofold: For one thing, which instances should be tested? If one is interested in generally applicable statements about system reliability, a set of instances that exercises the handling of critical situations is required. In most cases, only a domain expert will be able to decide exactly which instances are relevant here.

On the other hand, even if the instances to test are known, one still has to generate test cases for each of these instances. When one tries to demonstrate the reliability of a system, a large number of test cases per instance is necessary. Multiplied with the number of instances, the amount of required test cases becomes huge.

A common approach to solve the problem of test case construction is the automatic generation of test cases. Among the possible method for generating test cases, model-based test approaches form an important and well-researched subclass. These techniques generate test cases (i.e., test inputs and expected results) from one or more models that describe the behaviour of the system and/or the system's environment. These models are known as test models.

When testing for reliability, one usually chooses techniques that generate test cases based on the operational profile of the system, allowing unbiased estimations of reliability measures. One of these techniques is Model-Based Statistical Testing [20], also known as Statistical Testing. This approach uses a description of the possible system inputs from a user's point of view to generate test cases. This description is formalized as a Markov chain of possible input event sequences. The expected outputs can either be annotated to the Markov chain, or calculated using a second model, the system model, for more complicated systems.

When trying to apply model-based testing techniques to network-structured systems, one finds that each instance requires test models tailored to the underlying network structure. To avoid the overhead of generating test models for each instance, it appears worthwhile to construct test models for each instance from some general information and the description of the concrete instance. As the structure of the instances lends itself to a natural composition-based description, a component-based approach seems promising. Naturally, this approach must be designed in such a way that the instance description given by the domain expert can easily be translated into an instance of the test models.

Given some method that can be used to construct test models for a network-structured system from pre-defined components and a description of the system instance by a domain expert, it becomes possible to carry out testing of this system with little work for each instance, as the most labour-intensive step, namely the construction of the test model components, has to be done only once, and the translation from the domain expert's representation of an instance to test cases can be fully automated (cf. figure 2). Moreover, this approach allows the domain experts to carry out (parts of) testing without the burden of having to learn the details of testing the system under consideration.

In this paper, we present an approach called COMPOSE that allows for the

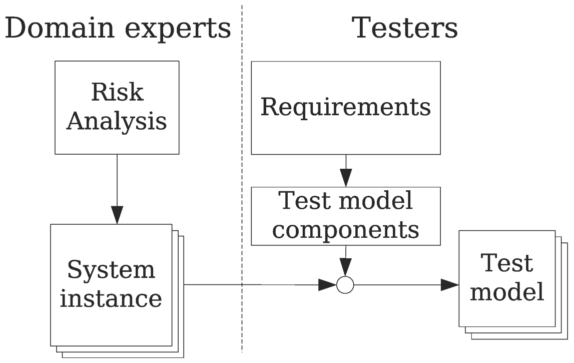


Fig. 2. Elements of the approach

systematic compositional construction of test models using domain-specific information. This approach comprises an integrated technique to describe state-based component behaviour and compose these components according to a specification given in a domain-specific language.

We believe that our approach improves both the efficiency and the expressivity compared to state-of-the-art work in the compositional construction of test models. In particular, we state the following claims:

(i) Constructing one composite model using the modelling techniques of COMPOSE does not require more effort than using other techniques.

(ii) The construction of several composite models for different system instances is more efficient using COMPOSE when compared to other techniques.

(iii) The resulting models produce identical test results.

The rest of this paper is organized as follows: In section 2, prior work is described. The COMPOSE approach is described in section 3. It is illustrated by a running example. Section 4 describes how we plan to evaluate our approach with the claims stated above, and gives some preliminary results. Finally, we summarize our work and point out further improvements and directions.

2 Related Work

Many ideas in this paper are based on common ideas found in component-based software and model-based engineering. The focus of this paper is not on extending these results, but rather on the application of these techniques to software testing, in particular, to model-based testing. It continues the work of a prior case study from the railway domain, which has been described in [12]. As noted there, the use of domain-specific languages for describing compositional models is, in itself, not a new idea; see, e.g., [7].

Often, composition operators are given by describing the action of the operator in natural language, usually together with a mathematical specification (cf. [5,21,13] and various others). We are not aware of any formalizations of composition operators in some machine-readable language, except for some simple examples of operators for process calculi [14].

Most model-based testing methods are based on automata or Markov chains.

Apart from the most commonly described composition operators, namely different forms of parallel composition [18], some more general operators have been defined in the literature.

For Markov chains, only few compositional approaches exist. Parallel compositions are obvious candidates, but are subsumed in other approaches. One general approach that we found is known as Stochastic Automata Networks [4]. It describes a Markov chain as a set of stochastic automata. These automata can interact using synchronized events (i.e., some transitions of the Markov chains are labelled using an event name and may only fire simultaneously) and functional rates (i.e., a transition rate may depend on the current state of some other Markov chain). Another approach is the Test Generation Language [15], which describes how stimulation sequences generated by several different Markov chains can be woven into one stimulation sequence for testing. This is done by giving several operators working on event sequences. In our experience, using a high-level approach like the Test Generation Language is preferable, as it allows clearer descriptions of intent.

For Mealy automata and more general state-event systems, a number of compositional techniques exist. First of all, Statecharts [6] can be seen as a method to describe the composition of state-event systems in a compact notation. Another approach is the use of communicating automata, for example in UPPAAL [2]. Further compositional approaches have cropped up in the area of Model-Driven Development. Some of these approaches (e.g. [19]) describes connectors using process calculi, which is quite close to the approach taken here. As our goal was to have a single technique for the construction of composite Markov chain models and system models, these approaches were not directly applicable to our problem, as these approaches do not provide appropriate handling for Markov chains.

Choosing a different point of view, model-based testing needs to generate sequences of stimulus events. When abstracting away the details of which underlying model is used, we are left with event-generating processes. These processes can be naturally described using process calculi, such as the π calculus. Indeed, the π calculus [14] already has a notion and graphical notation for the composition of different communicating processes. By modelling both component models and composition operators as communicating processes, one can use this kind of process composition to easily describe complex models.

3 The approach

In the following, the approach sketched above will be described in detail. As a running example, the construction of a behavioural model for a Train Control System [12,22] will be shown.

The approach has five steps:

(i) Identify the atomic components.

(ii) Determine appropriate composition operators.

(iii) Describe the behaviour of the atomic components using finite-state automata.

(iv) Define the behaviour of the composition operators, using the stimulus stream model described below.

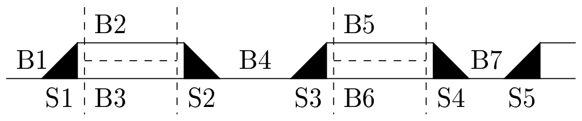


Fig. 3. An example track layout. Notation: The black triangles labeled S1 etc. are switches. B1 up to B7 are blocks, whose borders are the dashes lines.

(v) Define a DSL that uses the composition operators and atomic components to build a system model from the domain description.

Usually, one wishes to have a direct mapping from syntactic components of the DSL to composition operators and atomic components. On the other hand, a true one-to-one mapping will usually not be possible – some information may be implicit in the domain description, or there may be extraneous information.

For the example, an assistance system for a railway operating procedure known as “Zugleitbetrieb” is modelled. This system is called ZLB-PS. Briefly, the operation of this system can be summarized as follows:

(i) ZLB-PS is an assistance system that acts as substitute for the “train sheet”, a paper form that records the position and movement of trains as well as track reservations.

(ii) ZLB-PS uses track-side equipment (e.g. vacancy detectors) to counter-check and enforce the train director’s decisions. All track-side equipment is controlled by ZLB-PS.

(iii) The railroad network is partitioned into blocks, which contain switches and other track-side equipment. Moving a train means moving it from one block to another.

(iv) ZLB-PS can be configured for arbitrary track layouts.

(v) For a train to move from A to B, it must first reserve a track and ensure that this track is set up. During movement, it must notify the train director of arrival and may also indicate departure.

The track layouts used for ZLB-PS are network-like structures, which motivates our use of this example. A simple but typical layout is depicted in figure 3.

3.1 Identification of atomic components

In the first step, the basic components from which the test model is to be built must be identified. As a starting point, one starts from the domain expert’s view of the system, choosing elements of this description as candidates for components. These components are then compared against the (informal) description of the system. One may find that some components are not explicitly described in the system representation chosen above, e.g., with coordination components that have only one instance. These components are then added to the candidate set.

The candidate set can then be considered element by element. One may find that an element does not actually contribute to the behaviour being modelled or does not have a clear-cut behaviour. In this case, the element should be removed or replaced be suitable other components.

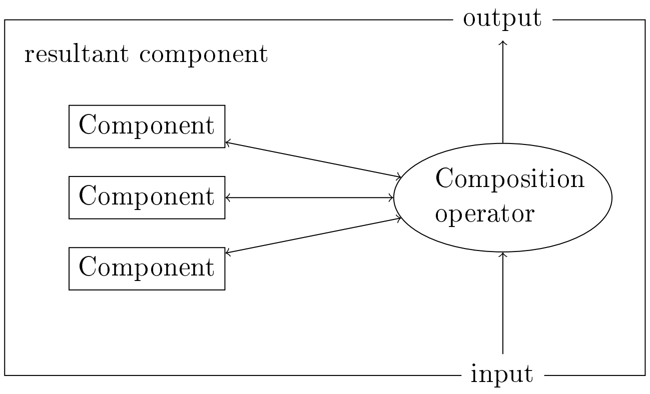


Fig. 4. The mechanics of composition operators; the arrows denote data streams

This step may be iterated when new information from further steps is available, adding, removing or changing components as required. In any case, one should try to keep a simple mapping from the DSL to the components.

As a first approximation, one finds that the structure of a railroad network is made up from lengths of track, switches and block borders. Looking at the system's requirements, it can be seen that the block borders themselves as well as the lengths of track are not directly considered, but make up parts of a block. Hence, one exchanges the track and block border components for a block component.

Also, signals are mentioned. As they can occur only on block borders, it is clear how they can be modelled in the domain expert's description: they correspond to block borders.

Finally, it becomes clear that reservations et cetera are described on a level above blocks, therefore a small mediating component will be needed as well, the central interlocking unit (short CIU). The component types are summarized in Figure 4 as rectangles with thick borders.

3.2 Determination of composition operators

Based on the identified components and the set of possible configurations, one next identifies how these components should be composed. The composition of a set of components yields a new, larger component, allowing a hierarchical description of the resulting system.

For this approach, one takes a black-box view of the components, describing their behaviour solely using streams of events entering and leaving the component. The job of the composition operator is to tie together the components by connecting to their input and output streams and coordinating which data is sent to and received from each component (cf. Figure 4). The possible composition operators can range from very simple operations, such as a simple scatter/gather operator that simply broadcasts inputs to all attached components and passes through their outputs, to complex and application-specific operators.

The railroad example will use two composition operators: At one level, a block contains its signals and switches, who can only operate correctly in cooperation with their environment inside the block. Thus, a "low-level" composition operator is defined, which describes how a block instance is built up from switch models, signal models and the (abstract) model of a general block. This composition operator needs additional information, namely the concrete layout of switches, signals and connections inside the block.

The concrete block instances, together with the central interlocking unit, can then form the description of the complete system. Here, the "high-level" composition operator is used, which describes how the signals from the CIU are distributed to the concrete block instances. Again, this operator needs extra information, namely which blocks are neighbours of a given block.

Again, a summary of the composition operators and their application is shown in Figure 4: The composition operators are shown as ellipses, their configuration information is given by the italic text and the components that are composed using a composition operator are connected to it via arrows. The result of a composition is given by a rectangle (with thin borders) surrounding all necessary information; it forms a new component, which can again be composed with other components.

3.3 Describing the atomic components

The next step is the description of component behaviour. As single components are modelled as Mealy machines, one can use various methods in this step, e.g. by direct construction of the state/transition graph [8], by transformation from Statecharts (cf. [6]) or a wealth of other techniques (e.g., [16]).

For the running example, sequence-based specification was chosen because backwards traceability to the requirements was considered as an important requirement [12]. The application of this method yielded four Mealy machines, which are described in the paper just referenced.

3.4 Describing the composition operators

Now that the interfaces of the components are known, the composition operators can be constructed. To do this, one assumes that each Mealy machine is a (communicating) process with an event input stream and an event output stream. The role of a composition operator is thus to describe how several such Mealy machine processes can be connected to form a larger component with one or more input and output event streams (cf. the section on the determination of composition operators, especially Figure 4.

A good way to describe the behaviour of a composition operator is to use a process calculus, such as CSP or the π-calculus. The approach described in this paper is based on a (slightly extended) version of π-calculus [14]. A companion version of the stochastic π-calculus exists as well, which is used to describe composition of Markov chain-based usage models for software testing. For reference, the following syntax is used for the π calculus:

```
Def ::= name '(' Patterns ')' ':=' Proc
                                   Process definition
Proc ::= '0'                       Null process
    | '(' Proc ')'
    | GuardedProcs
```

| Proc '|' Proc	Parallel composition
| 'replicate' Proc	Proc is replicated, i.e., infinitely many parallel copies are created
| 'new' Bindings '.' Proc	Introduction of new name bindings
| name '(' Terms ')'	Process instantiation
GuardedProcs ::= GuardedProc	
| GuardedProc '+' GuardedProcs	Nondeterministic guarded choice
GuardedProc ::= Guard	When the guard holds, do nothing
| Guard Proc	When the guard holds, perform new bindings and execute Proc
Guard ::= var '(' Patterns ')' '?'	Receive on channel, with data
| var '?'	Receive on channel, no data
| var '(' Terms ')' '!'	Send on channel, with data
| var '!'	Send on channel, no data
Patterns ::= Pattern | Pattern ',' Patterns	
Pattern ::= var	Accept any name/term, bind to var
| '_'	Dummy pattern, accept anything
| name '(' Patterns ')'	Constructor expression
| name '(' ')'	Empty constructor expression
Terms ::= Term | Term ',' Terms	
Term ::= var	The binding of var
| name '(' Terms ')'	Constructor expression
| name '(' ')'	Empty constructor expression
Bindings ::= Binding | Binding ',' Bindings	
Binding ::= var	Normal binding
| var ':' 'imm'	"immediate" binding

Semantics for the π calculus are given in Milner's book [14], while our extensions are described in a technical report [1].

As there is a natural transformation from Mealy machines to π processes, the composition of several Mealy machines using a given composition operator can be described by assuming that the Mealy machines are given by processes M_1 to M_k, and the composition operator by C. Then the resulting model is the π process defined similar to the following:

```
Composite(in, out) ::=
new in1, out1, ..., ink, outk.
    M1(in1, out1) | ... | Mk(ink, outk)
  | O(in, out, in1, out1, ..., ink, outk)
```

In the following example, it will be shown that the π calculus can be used as a kind of "programming language" to describe the operational semantics of the composition operators. This has several advantages:

(i) It is easy to define application-specific operators.

(ii) Tool support for model compositions is possible.

(iii) Operators can be built from other operators using simple language constructs, viz., process instantiation.

For ZLB-PS, two composition operators have to be modelled. These composition operators built as far as possible using standard operators. At first, consider the general structure of the operators.

The high-level composition operator works as follows: An instance of the Central Interlocking Unit is composed with a set of instances of concrete blocks. Whenever a stimulus describing some operation on a path in the network enters the Central Interlocking Unit, all blocks on this path are sent the corresponding stimulus in the reservation protocol. If the stimulus has a synchronous response, a transactional pattern is employed to guarantee that all blocks are in a consistent state.

Therefore, the composition operator works as follows: If the central interlocking unit outputs `request(f,t)`, the path from f to t is computed. Next, each path element is sent `request`in turn. If the request succeeds, the next element in the path is considered (the end of the path is handled in an appropriate way), and if this element is successfully reserved, OK is returned. If the next element fails (be it because the element itself failed or some problem happened later in the path), or if the request does not succeed, NOK is returned. The handling of `tearDown(f,t)` is and `setup`is analogous. On the other hand, the `cancel`stimulus is simply sent to all relevant blocks.

As a π-calculus process, this operator can be expressed as follows:

```
// Entrance to the high-level composition
HighLevel(ciu, listBlocks, adjacencies) :=
  // Initial state: normal operation state
  HighLevelNormal(ciu, adjacencies)
  // Run each block component instance,
  // all of them in parallel.
| Parallel(listBlocks);

// The behaviour of the central interlocking unit
// when not processing a request.
HighLevelNormal(ciu, adjacencies) :=
  // A request is received
  ciu(REQUEST(f, t))? new path, result.
    // Start three processes:
    // A path calculation...
    ( CalculatePath(adjacencies, f, t, path)
    // ... the transaction process that tries
    // to reserve all the blocks on the path (except
    // the first, which is a special case) ...
    | path(first)? Transaction(RESERVE(), path, result)
    // ... and the process that acts on the result,
    // providing an answer and entering the appropriate mode.
    | result(OK())? ciu(OK())!
        HighLevelRequested(ciu, adjacencies, f, t))
```

```
     +result(NOK())? ciu(NOK())! HighLevelNormal(ciu, adjacencies))
  // A teardown is received; basically, the same handling as for
  // a request.
 +ciu(TEARDOWN(f, t))? new path, result.
    ( CalculatePath(adjacencies, f, t, path)
    | path(first)? Transaction(TEARDOWN(), path, result)
    | result(what)? ciu(what)! HighLevelNormal(ciu, adjacencies));

// The behaviour of the central interlocking unit
// when processing a request.
HighLevelRequested(ciu, adjacencies, f, t) :=
  new path, result. (
    CalculatePath(adjacencies, f, t, path)
  | ciu(SETUP())? (Tranasction(SETUP(), path, result)
                   |result(what)? ciu(what)!
     HighLevelNormal(ciu, adjacencies))
   +ciu(CANCEL())? (Distribute(CANCEL(), path)
                   |HighLevelNormal(ciu, adjacencies));
```

The HighLevel operator uses three helper processes, namely CalculatePath, Transaction (which implements the transactional pattern) and Distribute (which distributes a stimulus along a path). These processes can be implemented as follows:

```
Transaction(reqType, path, result) :=
  // end of path: no obstacle to the reservation
  path(END())? result(OK())!
  // still on path: try the request on the current
  // path element\dots
+ path(STEP(s))? s(reqType)! (
    // if it fails, stop transaction
    (s(NOK())? result(NOK())!)
// if it suceeds, try next element\dots
  + s(OK())? new resNext. (Transaction(reqType, path, resNext)
           // hand down result, and call rollback if necessary
         | resNext(OK())? result(OK())!
   +resNext(NOK())? s(ROLLBACK())! result(NOK())!)

// Simple recursive implementation of a
// "for all elements" operation
Distribute(reqType, path) :=
  path(END())?
+ path(STEP(s))? s(reqType)! Distribute(reqType, path);

CalculatePath(adjacencies, from, to, path) :=
  [Code omitted for brevity: A function
   that calculates an arbitrary path through
   the graph given in adjacencies starting at
   from and ending at to. The path is output by sending
a sequence of STEP(s) terms, one for each vertex on the
path, and finally a END() term.]
```

The implementation of the low-level composition operator can be given in a similar way. It re-uses the definitions of `Transaction`, `Distribute` and `CalculatePath`. Again, we omit the implementation itself for brevity.

3.5 Definition of a DSL

The final step of this method is to define a DSL describing instances of the system and mapping it to applications of the composition operators to appropriate component instances.

To describe an instance of a complicated system, domain experts have usually developed their very own notation, e.g., block diagrams of the system or railroad network diagrams. The language is derived from such a notation, and should be designed in a way familiar to domain experts. Additionally, the translation from syntactic elements of the language to instances of the components and composition operators should be easy to implement. Optimally, each syntactic element maps to a fixed set of component instances and composition operator applications.

For the running example, a DSL based on the track layout was chosen (cf. figure Figure 3). It contains, among other information, data on which blocks, switch and signal components exist, and how blocs and switches are interconnected. For convenience, an already-existing XML encoding of the track layout was leveraged, which contained large amounts of extraneous information.

The elements that were relevant for the model construction were the following:

- The definition of a single component. This was used to instance the appropriate component models.
- The introduction of a neighbourhood relationship between two blocks. This data was filled into the block connectivity graph, used by the high-level composition operator.
- The introduction of the follower sets of a switch, where a follower set describes which blocks can be reached when a switch is set in a given position. This information was preprocessed and then used to fill the low-level connective graph for the low-level composition operator.
- The definition of a single block additionally implies the application of the low-level composition operator to this block and all track devices (i.e., switches and signals) belonging to this block.

Finally, the (implied) CIU instance is created once and the high-level composition operator applied to the results of the low-level compositions.

For the track layout given in 3, we get the following component instances and composition operator applications (see also 5:

(i) Five instances of the switch model, one each for S1 to S5.

(ii) Seven instances of the block model.

(iii) Eight instances of the signal model, two each for the blocks B2, B3, B5 and B6.

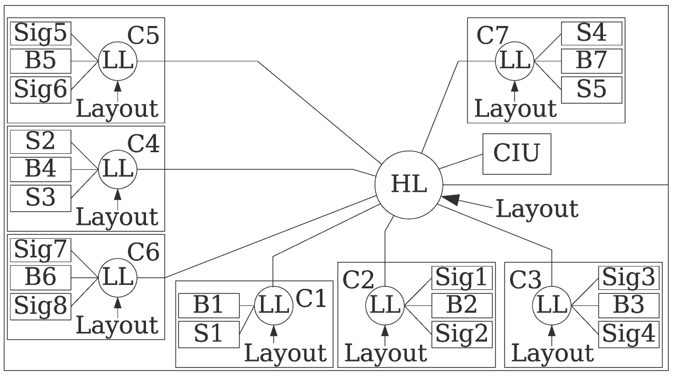


Fig. 5. The result of the composition. Each model (rectangle) is named by some letters symbolizing its type (B means block component, S switch, Sig signal and C concrete block) and, optionally, a number. Composition operators are signified by circles, where HL means high-level composition, and LL low-level composition.

(iv) One instance of the CIU.

(v) Seven applications of the low-level composition operator, namely on for B1 and S1, one for B4 with S2 and S3, one for B7, S4 and S5 and one each for B2, B3, B5 and B6 together with their respective signals. Each of these operators yields a concrete block, called C1 to C7.

(vi) One application of the high-level composition operator, composing the CIU with C1 to C7 and yielding the complete system model.

4 Experiences from the example

As this paper was based on prior work by the authors [12], some limited experience about efficiency is available. In particular, the approach described in [12] works along the same lines as COMPOSE, with all steps but step 4 of the technique virtually identical. Not surprisingly, we find that the implementation of the composition operators as an executable Java program required significantly more effort (a few man-weeks) compared to their specification as π-calculus processes (two or three days).

All in all, we find that the COMPOSE approach seems to perform well: It permits the production of test models for network-structured systems with acceptable effort. This leads us to formulate several hypotheses:

(i) COMPOSE satisfies the requirements described in the introduction: It allows the construction of test models for network-structured systems in a way that is comprehensible for domain experts.

(ii) COMPOSE does not require significantly more effort to construct the artifacts for one composite model than other techniques.

(iii) COMPOSE requires significantly less effort to construct several different composite models, compared to other techniques.

(iv) The models generated using this technique generate the same test cases and/or test verdicts compared to models built by other techniques.

These hypotheses will form the basis of a serious evaluation of the COMPOSE approach. We plan to derive the necessary results by using a controlled experiment and/or an industrial case study. For the comparison to other, existing techniques, Stochastic Automata Networks [4] and Prowell's TGL [15] seem to be the most appropriate candidates.

5 Conclusion and Outlook

In this paper, we have demonstrated an approach called COMPOSE that allows for the systematic compositional construction of behavioral models. This approach comprises an integrated technique to describe state-based component behaviour and compose these components according to a specification given in a domain-specific language. In our approach, component behaviour is described using Mealy machines, while composition operators can be specified using process calculus expressions. This allows a tester or system expert to build models suitable for analysis and model-based testing. On the other hand, the instantiation of a model is described using a domain-specific language, allowing a domain expert to construct instances of the system and environment models for testing without any deeper knowledge about these models. The advantage of this approach is that the domain expert can easily test the system in as many configurations as he deems relevant and important, without having to worry about constructing test cases or computing test results manually.

Our approach is an extension of older work, described in a previous paper [12], where a special case of this approach was demonstrated. The improvement in this paper is that we can now specify the behaviour of the composition operators more easily, so that the approach can be applied with less effort.

Starting from here, we plan to do three things. The first task, which is already close to completion, is the implementation of a tool to support this approach. Next, we plan to apply this approach on different examples, e.g., an example from industrial control. Finally, we will investigate how the concepts behind this approach can be used for quantitative reliability estimation of complex systems.

6 Acknowledgements

The preparation of this work was funded by the Fraunhofer Innovation Cluster "Digital Commercial Vehicle Technology". The Fraunhofer Innovation Cluster "Digital Commercial Vehicle Technology" is funded by the European Union and the state of Rhineland-Palatinate in the context of the Ziel 2-Programm Rheinland-Pfalz. The example used in this paper was created by Siemens Mobility for the ranTest research project. The picture of the German railway network is based on a public domain picture taken from Wikipedia, cf. http://en.wikipedia.org/wiki/File:Bahn-Streckenkarte_Deutschland.png.

References

[1] Bauer, T., R. Eschbach, T. Hussain, J. Kloos and F. Zimmermann, *Komposition von benutzungsmodellen: Operatoren und anwendungsbeispiel*, Technical report, Fraunhofer IESE (2009).

[2] Behrmann, G., A. David and K. G. Larsen, *A Tutorial on Uppaal*, in: M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS (2004), pp. 200–236.

[3] EN 50126, "Railway Applications – The Specification and Demonstration of Reliability, Availability, Maintainability, and Safety (RAMS)." (1999), CENELEC, european Standard.

[4] Farina, A. G., P. Fernandes and F. M. Oliveira, *Representing software usage models with stochastic automata networks*, in: *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering* (2002), pp. 401–407.

[5] Garavel, H. and M. Sighireanu, *A graphical parallel composition operator for process algebras*, in: *Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV 99)* (1998), pp. 185–202.

[6] Harel, D., *Statecharts: A visual formalism for complex systems*, Science of Computer Programming **8** (1987), pp. 231–274.

[7] Haxthausen, A. E. and J. Peleska, *Automatic verification, validation and test for railway control systems based on domain-specific descriptions*, in: S. Tsugawa and M. Aoki, editors, *Proceedings of the 10th IFAC Symposium on Control in Transportation Systems* (2003), p. (pages unknown).

[8] Hopcroft, J. E., R. Motwani and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison-Wesley Longman Publishing Co., Inc., 1990, 1 edition.

[9] IEC 61511, "Functional safety: Safety Instrumented Systems for the process industry sector," (2003), IEC.

[10] IEC 61508, "Functional safety of electrical/electronic/programmable electronic safety related systems," (2005), IEC, (draft).
URL http://www.iec.ch/functionalsafety

[11] ISO/CD 26262, "Road vehicles – Functional safety," ISO.

[12] Kloos, J. and R. Eschbach, *Generating system models for a highly configurable train control system using a domain-specific language: A case study*, in: *5th Workshop on Advances in Model Based Testing (A-MOST'2009)*, 2009, p. n/a.

[13] Milne, G., *The formal description and verification of hardware timing*, IEEE Transactions on Computers **40** (1991), pp. 811–826.

[14] Milner, R., "Communicating and Mobile Systems: The pi-Calculus," Cambridge University Press, 1999.

[15] Prowell and S.J., *Using Markov Chain Usage Models to Test Complex Systems*, System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on (2005), pp. 318c–318c.

[16] Prowell, S. J. and J. H. Poore, *Foundations of Sequence-based Software Specification*, IEEE Trans. Softw. Eng. **29** (2003), pp. 417–429.

[17] DO178B, "Software Considerations in Airborne Systems and Equipment Certification," RTCA.

[18] Schneider, K., "Verification of Reactive Systems - Formal Methods and Algorithms," Texts in Theoretical Computer Science (EATCS Series), Springer, 2003.

[19] Spitznagel, B. and D. Garlan, *A compositional formalization of connector wrappers*, in: *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (2003), pp. 374–384.

[20] Whittaker, J. A. and J. H. Poore, *Statistical Testing for Cleanroom Software Engineering*, in: *Proc. Twenty-Fifth Hawaii International Conference on System Sciences*, 1992, pp. 428–436.

[21] Yevtushenko, N., T. Villa, R. Brayton and A. Petrenko, *Solution of Parallel Language Equations for Logic Synthesis*, in: *Proceedings of the International Conference on Computer Aided Design*, 2001, pp. 103–110.

[22] Zechner, A., "Entwurf einer generischen Systemarchitektur für den Zugleitbetrieb," Diplomarbeit, TU Braunschweig (2006).

A lightweight approach to customizable composition operators for Java-like classes

Giovanni Lagorio[1], Marco Servetto[2] and Elena Zucca[3]

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Genova, Italy

Abstract

We propose a formal framework for extending a class-based language, equipped with a given class composition mechanism, to allow programmers to define their own derived composition operators. These definitions can exploit the full expressive power of the underlying computational language.
The extension is obtained by adding *meta-expressions*, that is, (expressions denoting) class expressions, to conventional expressions. Such meta-expressions can appear as class definitions in the class table.
Extended class tables are reduced to conventional ones by a process that we call *compile-time execution*, which evaluates these meta-expressions.
This mechanism poses the non-trivial problem of guaranteeing soundness, that is, ensuring that the conventional class table, obtained by compile-time execution, is well-typed in the conventional sense.
This problem can be tackled in many ways. In this paper, we illustrate a lightweight solution which enriches compile-time execution by partial typechecking steps. Conventional typechecking of class expressions only takes place when they appear as class definitions in the class table. With this approach, it suffices to introduce a unique common type **`code`** for meta-expressions, at the price of a later error detection.

Keywords: Modular composition, Java-like languages, Meta-programming, Type systems

Introduction

Support for code reuse is a key feature which should be offered by programming languages, in order to automate and standardize a process that programmers should, otherwise, do by hand: duplicating code for adapting it to best solve a particular instance of some generic problem. Two different strategies which can be adopted to achieve code reuse are *composition languages* and *meta-programming*.

In the former approach programmers can write fragments of code (classes in the case of Java-like languages) which are not self-contained, but depend on other fragments. Such dependencies can be later resolved by combining fragments via composition operators, to

[1] Email: `lagorio@disi.unige.it`
[2] Email: `servetto@disi.unige.it`
[3] Email: `zucca@disi.unige.it`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

obtain different behaviours. These operators form a *composition language*. Inheritance (single and multiple), mixins and traits are all approaches allowing one to combine classes, hence they define a composition language in the sense above.

The limitation of this approach is that the users, provided with a fixed set of composition mechanisms, cannot define their own operators, as it happens, e.g., with function/method definitions.

In meta-programming, programmers write (meta-)code that can be used to generate code for solving particular instances of a generic problem. In the context of Java-like languages, *template meta-programming* is the most widely used meta-programming facility [4], as, e.g., in C++, where templates, which are parametric functions or classes [5], can be defined and later instantiated to obtain highly-optimized specialized versions. The instantiation mechanism requires the compiler to generate a temporary (specialized) source code, which is compiled along with the rest of the program. Moreover, template specialization allows to encode recursive computations, that can be thought of as compile-time executions. This technique is very powerful, yet can be very difficult to understand, since its syntax and idioms are esoteric compared to conventional programming. For the same reasons, maintaining and evolving code which exploits template meta-programming is rather complex, see, e.g., [3]. Moreover, well-formedness of generated source code can only be checked "a posteriori", making the whole process hard to debug.

Here, our aim is to distill the best of these two approaches, that is, to couple disciplined meta-programming features with a composition language, in the context of Java-like classes. More precisely, we propose a formal framework for extending a class-based language, equipped with a given class composition mechanism [6] to allow programmers to define their own derived composition operators. These definitions can exploit the full expressive power of the underlying computational language.

The extension is obtained as follows: *meta-expressions*, that is, (expressions denoting) class expressions, are added to conventional expressions. Then, such meta-expressions can appear as class definitions in the class table. Extended class tables are reduced to conventional ones by evaluating these meta-expressions. This meta-circular approach implies compile-time execution as in template meta-programming, with the advantage of a familiar meta-language, since it just coincides with the conventional language the programmers are used to.

This mechanism, which is trivial in itself, poses the non-trivial problem of guaranteeing soundness, that is, ensuring that the conventional class tables, obtained by compile-time execution, are well-typed (in the conventional sense).

Ideally, typing errors in generated source code should be detected statically, that is, without requiring reduction at the meta-level at all, as it happens, e.g., in MetaML [16]. However, this would require to introduce sophisticated types for meta-expressions. In this paper, we illustrate instead a lightweight solution which enriches compile-time execution by typechecking steps. Conventional typechecking of class expressions only takes place when they appear as the right-hand side of class definitions in the class table. With this approach, it suffices to introduce a unique common type `code` for meta-expressions, at the

[4] A very limited form of meta-programming is offered by Java/C# generics, which are classes parametric in some type parameters, that can be compiled once. Java generics are a source level feature, which is compiled away, while generics are fully supported by C#.

[5] Note that C++ supports, along type-parameters, other kinds of template parameters.

[6] In this paper for sake of simplicity we illustrate the approach on only one composition operator, that is, **`extends`**.

price of a later error detection.

The paper is organized as follows: in Section 1 we informally introduce our approach by means of some examples, in Section 2 we provide a formalization on a very simple class composition language, and in Section 3 we summarize the contribution of the paper and draw related and further work. The Appendix contains the proof of soundness. This paper is the full version of [8].

1 Examples

In order to show how to use meta-programming as a tool for better composing software, we introduce a language allowing one to compose classes by means of some operators. In such a language, a class declaration associates a *class expression* with the name of the declared class. The simplest form of class expression is the *base class*, that is, a set of field and method declarations. For instance, the literal `{ int answer() { return 42; } }` denotes a base class declaring a single method named `answer`. In our example language, we can give the name `C` to that class body by writing:

```
class C = {  int answer() { return 42; }  }
```

This is the exact Java syntax, with the exception of the extraneous symbol `=`. Of course, here the symbol `=` can be followed by any arbitrarily involved class composition expression, in place of a simple base class literal.

Since our aim here is to explain how our approach works, rather than proposing a specific composition language, for simplicity we consider a very simple language offering just a single binary operator, **`extends`**, allowing one to combine two classes in a way that should feel natural to Java programmers: the left operand *extends*, that is, overrides, the right operand.

For instance, writing

```
{ int a() { return 1; } } extends
{ int a() { return 0; } int b() { return 0; } }
```

is equivalent to write:

```
{ int a() { return 1; } int b() { return 0; } }
```

To add a meta-programming facility to this simple language, we allow class (composition) expressions to be used as expressions of a newly introduced type: **`code`**.

For instance, the following program

```
class C = {
  code m() {
   return { int one() { return 1; }  };
  }
}
class D = new C().m()
```

declares two classes, `C` and `D`. The former, `C`, declares a single method named `m`, which returns a value of type **`code`**. This value, in turn, is a base class declaring the (non-meta) method[7] `one`. The latter class, `D`, is declared using an expression that has to be evaluated in order to obtain the corresponding class body. In this example, the body of `D` is the value returned by the method `m` of `C`, so this program could be equivalently written as:

[7] We call *meta-methods* the methods involving code manipulation.

```
class C = /* ...as before... */
class D = {  int one() { return 1; }  }
```

One very basic use of this mechanism allows to obtain conditional compilation. For instance, in the previous example we could have written:

```
class C = {
  code m() {
      if (DEBUG) return /* ...debug version... */;
      return /* ...as before... */;
  }
}
```

The following (meta-)method:

```
code mixin(code parent) {
  return { /* ... */ } extends parent;
}
```

behaves like a mixin, extending in some way a parent class passed as argument.

Note that the code in the extension can select arbitrary fields or methods of the parent class. This is allowed because we do not typecheck a class expression until it is associated to a class name in the class table. This choice allows for an incredible leeway in writing reusable code, at the price of a late error detection. The situation is very similar to what happens with C++ templates [15,13].

The class to be used as parent could be constructed, having a generic list type, `List<>`, by chaining an arbitrary number of classes:

```
code chain(List<code> parents){
  if (parents.isEmpty()) return Object;
  return parents.head() extends this.chain(parents.tail());
}
```

This is indeed similar to mixin composition, with the advantage that the operands of this arbitrarily long composition do not have to be statically known.

Finally, the following example is a graphic library that adapts itself with respect to its execution environment, without requiring any extra-linguistic mechanisms: [8]

```
class GraphicalLibrary {
  code produceLibrary() {
    code result = BaseGraphicalLibrary;
    String producer = System.getProperty("sys.vcard.brand");
    if (producer.equals("NVIDIA"))
        result = NVIDIASupport extends result;
    else if (producer.equals("ATI"))
        result = ATISupport    extends result;
    else
        throws new CompilationError(
                      "No compatible hardware found");
    if (System.getProperty("os.name").contains("Windows"))
      result = CygwinAdapter extends result;
    return result;
  }
}
```

The method `produceLibrary` builds a platform-specific library by combining the generic library `BaseGraphicalLibrary` with the brand-specific drivers (represented by the two classes `NVIDIASupport` and `ATISupport`) and wrapping the result, if required on the specific platform, with the class `CygwinAdapter`, which emulates a Linux-like environment on Windows operating systems.

[8] To keep the example compact, we do not detail all the classes named in the example, and we simply assume that they are declared elsewhere.

cp	$::= \overline{\texttt{class}\ C\ = ce}$	(conventional) program
ce	$::= C \mid B \mid ce\ \texttt{extends}\ ce'$	class expression
B	$::= \{fds\ mds\}$	base class
fds	$::= \overline{fd}$	field declarations
fd	$::= T\ f\texttt{;}$	field declaration
mds	$::= \overline{md}$	method declarations
md	$::= T\ m(\overline{T\ x})\ \{\texttt{return}\ e;\}$	method declaration
e	$::= x \mid e.f \mid e.m(\overline{e}) \mid \texttt{new}\ C\ (\overline{e}) \mid (C)e$	(runtime) expression
v	$::= \texttt{new}\ C\ (\overline{v})$	value
T	$::= C$	type
CT	$::= \langle\overline{C}, fds, mhs\rangle$	class type
mhs	$::= \overline{mh}$	method headers
mh	$::= T\ m(\overline{T})$	method header
Δ	$::= \overline{C{:}CT}$	class type environment
Γ	$::= \overline{x{:}T}$	parameter type environment

Figure 1. Syntax and types of the conventional language

In this way the compilation of the same source produces customized versions of the library depending on the execution platform. In other words, this approach can be used to write *active libraries* [2], that is, libraries that interact dynamically with the compiler, providing better services, as meaningful error messages, library-specific optimizations and so on.

2 Formalization

Figure 1 shows syntax, values and types of our conventional language, using the overbar notation to denote a (possibly empty) sequence. [9]

The top section of the figure defines the syntax, where we assume infinite sets of class names C, field names f and method names m. As already mentioned, to keep the presentation minimal we consider a class composition language with only one operator, `extends`. This conventional language is very similar to Featherweight Java [6], FJ for short, but the operator `extends` composes two class expressions, rather than the name of an existing class with a class body (base class).

Reduction rules are as in FJ and are omitted. The only difference is that look-up, formally expressed by the function $mbody$, needs to be generalized, as shown in Figure 2, to take into account that `extends` composes two class expressions. We omit the analogous trivial generalization of the function $fields$. Values v of the conventional language are as in FJ.

[9] This notation for metavariables is the analogous of the Kleene-star in BNF style.

$$(\text{<-DIRECT})\ \frac{}{\Delta \vdash C{<}C_i}\ \begin{array}{l}\Delta(C) = \langle C_1 \ldots C_n, \_, \_\rangle\\ i \in 1..n\end{array} \qquad (\text{<-TRANS})\ \frac{\begin{array}{c}\Delta \vdash C{<}C'\\ \Delta \vdash C'{<}C''\end{array}}{\Delta \vdash C{<}C''}$$

$$(\le\text{-REFL})\ \frac{}{\Delta \vdash C{\le}C} \qquad (\le\text{-STRICT})\ \frac{\Delta \vdash C{<}C'}{\Delta \vdash C{\le}C'}$$

$$(\text{METHOD-T})\ \frac{\begin{array}{l}\Delta; x_1{:}T_1, \ldots x_n{:}T_n, \texttt{this}{:}C \vdash e{:}T'\\ \Delta \vdash T'{\le}T\end{array}}{\Delta; C \vdash T\ m(T_1\ x_1 \ldots T_n\ x_n)\{\texttt{return}\ e;\}{:}T\ m(T_1 \ldots T_n)}$$

$$(\text{NAME-T})\ \frac{}{\Delta; C' \vdash C{:}\langle C, \mathit{fds}, \mathit{mhs}\rangle}\ \Delta(C) = \langle \_, \mathit{fds}, \mathit{mhs}\rangle$$

$$(\text{BASIC-T})\ \frac{\begin{array}{l}\Delta; C \vdash md_1{:}mh_1\\ \ldots\\ \Delta; C \vdash md_n{:}mh_n\end{array}}{\Delta; C \vdash \{\mathit{fds}\ md_1 \ldots md_n\}{:}CT}\ CT = \langle \emptyset, \mathit{fds}, mh_1 \ldots mh_n\rangle$$

$$(\text{EXTENDS-T})\ \frac{\begin{array}{c}\Delta; C \vdash ce_1{:}\langle \overline{C}_1, \mathit{fds}_1, \mathit{mhs}_1\rangle\\ \Delta; C \vdash ce_2{:}\langle \overline{C}_2, \mathit{fds}_2, \mathit{mhs}_2\rangle\end{array}}{\Delta; C \vdash ce_1\ \texttt{extends}\ ce_2{:}\langle \overline{C}_1{\cup}\overline{C}_2, \mathit{fds}_1{\cup}\mathit{fds}_2, \mathit{mhs}_1{\cup}\mathit{mhs}_2\rangle}\ \mathit{dom}(\mathit{fds}_1){\cap}\mathit{dom}(\mathit{fds}_2) = \emptyset$$

$$(\text{PROGRAM-T})\ \frac{\begin{array}{l}\Delta, \Delta'; C_1 \vdash ce_1{:}CT_1\\ \ldots\\ \Delta, \Delta'; C_n \vdash ce_n{:}CT_n\end{array}}{\Delta \vdash C_1 = ce_1 \ldots C_n = ce_n{:}\Delta'}\ \begin{array}{l}\Delta' = C_1{:}CT_1 \ldots C_n{:}CT_n\\ \Delta, \Delta' \vdash \_{<}\_\ \text{acyclic}\end{array}$$

$\mathit{mbody}_{cp}(C, m) = \mathit{mbody}_{cp}(ce, m)$ if $cp(C) = ce$
$\mathit{mbody}_{cp}(\{\ldots C\ m(\ldots)\{\texttt{return}\ e;\}\ldots\}, m) = e$
$\mathit{mbody}_{cp}(ce_1\ \texttt{extends}\ ce_2, m) = \mathit{mbody}_{cp}(ce_1, m)$ if $\mathit{mbody}_{cp}(ce_1, m)$ defined,
  $\mathit{mbody}_{cp}(ce_2, m)$ otherwise

Figure 2. Typing rules and look-up of the conventional language

Typing rules are shown in Figure 2.

The first four rules define the subtyping relation. Note that, since a class definition can contain many class names as subterms [10], in our generalization a class can be a direct subtype of many others. However, method look-up function *mbody* gives precedence to the

[10]For instance, **class** C = D **extends** E.

$$
\begin{array}{lll}
p & ::= \overline{\texttt{class}\ C\ = e} & \text{(generalized) program} \\
e & ::= \ldots \mid C \mid B \mid e\ \texttt{extends}\ e' & \\
v & ::= \texttt{new}\ C\,(\overline{v}) \mid ce & \\
T & ::= C \mid \texttt{code} &
\end{array}
$$

$$
\text{(META-RED)}\ \frac{e \xrightarrow{cp} e'}{cp\ (C = e)\ p \longrightarrow cp\ (C = e')\ p}
$$

$$
\text{(EXTENDS-1)}\ \frac{e_1 \xrightarrow{cp} e_1'}{e_1\ \texttt{extends}\ e_2 \xrightarrow{cp} e_1'\ \texttt{extends}\ e_2}
\qquad
\text{(EXTENDS-2)}\ \frac{e \xrightarrow{cp} e'}{v\ \texttt{extends}\ e \xrightarrow{cp} v\ \texttt{extends}\ e'}
$$

$$
\text{(}\leq\text{-REFL-CODE)}\ \frac{}{\Delta \vdash \texttt{code} \leq \texttt{code}}
\qquad
\text{(T-NAME)}\ \frac{}{\Delta; \Gamma \vdash C{:}\texttt{code}}
\qquad
\text{(T-BASIC)}\ \frac{}{\Delta; \Gamma \vdash B{:}\texttt{code}}
$$

$$
\text{(T-EXTENDS)}\ \frac{\Delta; \Gamma \vdash e_1{:}\texttt{code} \quad \Delta; \Gamma \vdash e_2{:}\texttt{code}}{\Delta; \Gamma \vdash e_1\ \texttt{extends}\ e_2{:}\texttt{code}}
$$

Figure 3. Meta-expressions and compile-time execution

left operand as in standard FJ.

Rule (METHOD-T) is as in FJ, typing rules for expressions are also as in FJ and are omitted.

The typing judgment $\Delta;\, C \vdash ce{:}\langle\overline{C}, \mathit{fds}, \mathit{mhs}\rangle$ assigns a class type to a class expression ce appearing as (subterm of) the definition of class C, needed to type method bodies in base classes in ce. This class type models the type information which can be extracted from ce, and consists of three components: a set $\overline{C}$ of class names (those appearing as subterms in ce, which are, hence, the direct supertypes of C), a set of field declarations and a set of method headers extracted from method declarations. As usual, we assume that these sets are well-formed only if a field (method) name appears only once, and write dom to denote the set of declared names. In rule (EXTENDS-T), this assumption implicitly ensures that a method can be overriden only with the same type, whereas the additional side condition prevents hiding of fields, and both are standard FJ requirements.

In rule (PROGRAM-T), standard FJ typing rule for programs is generalized to open programs, that is, programs which can refer to already compiled classes, modeled by the left-side class type environment Δ. We denote by Δ, Δ' concatenation of two class type environments with disjoint domain.

Figure 3 shows how the conventional language is extended to allow customizable composition operators.

As already mentioned, this is achieved as follows: *meta-expressions*, that is, (expressions denoting) class expressions, are added to conventional expressions, as shown in the second production. These meta-expressions have a special primitive type `code` which is added to types (fourth production, and typing rules in the last section of the figure). An example of meta-expression is

```
new C().m({ int m(){ return 2;} })
```

In particular, a class expression is seen as a value of type `code` (third production).

Moreover, such meta-expressions can appear as class definitions in the program (first production).

Then, *compile-time execution* consists in reducing this (generalized) program to a conventional program, where all right-hand sides of class declarations are values, that is, class expressions. This is modeled by the relation $p \longrightarrow p'$, whose steps are *meta-reduction* steps, that is, steps of reduction of a meta-expression. More precisely, as formalized by rule (META-RED), in a (generalized) program it is possible to reduce the right-hand-side of a class declaration in the context of a conventional fragment cp of the program. We have assumed, without any loss of generality, that in a generalized program the conventional part comes first. The relation $e \xrightarrow{cp} e$ is the standard FJ reduction of an expression in the context of a (conventional) program, enriched by the rules (EXTENDS-1) and (EXTENDS-2).

We consider now the issue of soundness. Compile-time execution can: (1) not terminate; (2) get stuck; (3) reduce to a program where the right-hand-side of some class declaration is a value different from a class expression; (4) reduce to a program where some class declaration is ill-typed; (5) reduce to a well-typed program.

Indeed, there is no way to have *both* terminating metaprograms and a fully metacircular approach (over a Turing complete base-language) because, to guarantee the termination, one has to restrict either the resources used by metaprograms or the metalanguage itself.

To prevent (2)-(3)-(4), hence to guarantee that compile-time execution always produces a well-typed program when terminates, we can take different approaches. In this paper, we propose a simple technique which integrates meta-reduction with typechecking, as shown in Figure 4.

In this approach, reduction of a program involves some typechecking steps, which can either succeed or fail. In the latter case the program reduces to `error`.

More in detail, during compile-time execution each class declaration `class` $C = e$ in the program can be annotated with the following meaning:

- empty annotation: initial state, no check has been performed yet;
- annotation `code`: e is a well-typed meta-expression;
- annotation CT, for some class type CT: e is (a well-typed meta-expression which denotes) a well-typed class expression of type CT.

We will use $\tilde{p}$ as metavariable for annotated programs. More precisely, checked compile-time execution is defined on annotated programs of the following form:

$$\tilde{p} ::= cp{:}\Delta\ cp'{:}\texttt{code}\ [\texttt{class}\ C = e{:}\texttt{code}]\ p \mid \texttt{error}$$

where square brackets denote optionality, and e is not of the form ce. Moreover, for any cp conventional program, $cp{:}\texttt{code}$ is the program obtained by annotating each class declaration by `code`, and, for any Δ s.t. $dom(cp) = dom(\Delta)$, $cp{:}\Delta$ is the program obtained by annotating each class declaration with the type associated in Δ to the corresponding class name.

We have assumed, without any loss of generality, that in an annotated program the $cp{:}\Delta$

$$\text{(META-RED)}\ \frac{e \xrightarrow{cp} e'}{cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e{:}\texttt{code})\ p \longrightarrow cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e'{:}\texttt{code})\ p}$$

$$\text{(META-CHECK)}\ \frac{}{\begin{array}{l}cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e)\ p \longrightarrow\\ \quad cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e{:}\texttt{code})\ p\end{array}}\ \Delta;\emptyset \vdash e{:}\texttt{code}$$

$$\text{(META-CHECK-ERROR)}\ \frac{}{cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e)\ p \longrightarrow \texttt{error}}\ \begin{array}{l}\nexists cp'' \subseteq cp',\, cp'' \neq \emptyset \text{ s.t. } closed(\Delta, cp'')\\ \Delta, \emptyset \nvdash e{:}\texttt{code}\end{array}$$

$$\text{(CHECK)}\ \frac{}{cp{:}\Delta\ cp'{:}\texttt{code}\ \tilde{p} \longrightarrow cp{:}\Delta\ cp'{:}\Delta'\ \tilde{p}}\ \begin{array}{l}cp' \neq \emptyset\\ \Delta \vdash cp'{:}\Delta'\end{array}$$

$$\text{(CHECK-ERROR)}\ \frac{}{cp{:}\Delta\ cp'{:}\texttt{code}\ \tilde{p} \longrightarrow \texttt{error}}\ \begin{array}{l}cp' \neq \emptyset\\ closed(\Delta, cp') \text{ or } \tilde{p} = \emptyset\\ \nexists\Delta' \text{ s.t. } \Delta \vdash cp'{:}\Delta'\end{array}$$

Figure 4. Checked compile-time execution

part comes first, then the cp:`code` part, then the others. In particular, in the initial program conventional class declarations appear first and are annotated `code`. Moreover, reduction rules ensure that at each intermediate step there is at most one class declaration which has been annotated `code` but is not reduced yet. This is formalized later by the subject reduction property, that is, Theorem 2.2.

Rule (META-RED) models a (safe) meta-reduction step. Indeed, meta-reduction is only performed w.r.t. a conventional program cp which has been previously succesfully typechecked. Note that, here as in the following two rules, there can be another portion of the program cp' which has already been reduced, but for which it is still impossible to perform a conventional typechecking step. This happens when cp' refers to some class names whose definition is still unavailable, see the first example in the following.

Rule (META-CHECK) and (META-CHECK-ERROR) model a typechecking step at the meta-level. That is, the first class declaration in the program which is not annotated yet is examined, to check that its right-hand side e is a well-typed meta-expression. The expression is typechecked w.r.t. to the portion of the conventional program cp which has been already successfully typechecked. If the typechecking step succeeds, then the class declaration is annotated `code`. Otherwise, an error is raised only if it is not possible to perform a further conventional typechecking step on cp', since any non-empty subset of cp' refers to some class names whose definition is still unavailable. This is expressed by the side-condition: $closed(\Delta, p)$ holds when p only refers to class names that are either in $dom(\Delta)$ or in $dom(p)$ itself (the trivial formal definition is omitted).

Rule (CHECK) and (CHECK-ERROR) model a conventional typechecking step. A successful typechecking step takes place if there is a portion of the conventional program cp' which can be typechecked w.r.t. the current class type environment Δ. An error is raised, instead, if no successful typechecking step is possible and, moreover, there is no hope it will be possible in the future, since either cp' only refers to class names which are already available, or there are no other class definitions to reduce.

We show now some examples illustrating how checked compile-time execution works.

First we give an example of successful compile-time execution. We abbreviate by B the base class `{ int one(){ return 1; } }`. The program

```
class C = { code m() { return B; } } : code
class D = {  int m() { return new E().one();} } : code
class E = new C().m()
```

is reduced by (CHECK) to

```
class C = { code m() { return B; } } : ⟨∅,∅, code m() ⟩
class D = {  int m() { return new E().one();} } : code
class E = new C().m()
```

is reduced by (META-CHECK) to

```
class C = { code m() { return B; } } : ⟨∅,∅, code m() ⟩
class D = {  int m() { return new E().one();} } : code
class E = new C().m() : code
```

is reduced by (META-RED) to

```
class C = { code m() { return B; } } : ⟨∅,∅, code m() ⟩
class D = {  int m() { return new E().one();} } : code
class E = { int one() { return 1; }  } : code
```

is reduced by (CHECK) to

```
class C = { code m() { return B; } } : ⟨∅,∅, code m() ⟩
class D = {  int m() { return new E().one();} } : ⟨∅,∅, int m() ⟩
class E = { int one() { return 1; }  } : ⟨∅,∅, int one() ⟩
```

Compile-time execution checks that class `C` is well-typed. Note that it is not possible to check class `D` since it refers to class `E` that has no associated class expression yet. Hence, expression `new C().m()` is checked to be of type `code`. At this point, reduction of this expression can take place, and finally the resulting class `D` is checked to be well-typed. Finally, also the class `D` is verified to be well-typed.

The second example shows a case when compile-time execution terminates with an `error`.

```
class C = { code m() { return B; } }: code
class D = new C().k()
```

is reduced by (CHECK) to

```
 class C = { code m() { return B; } } : ⟨∅,∅, code m() ⟩
 class D = new C().k()
```

is reduced by (META-CHECK-ERROR) to `error`.

Compile-time execution checks that class `C` is well-typed,and then checks whether the expression `new C().k()` is of type `code`. This is not the case, since class `C` has no methods named `k`. Moreover, because no standard typechecking steps are possible, since there are no other classes, an `error` is raised.

In the last example we abbreviate by B the base class

`{ int one(){ return new C().k(); } }`.

```
class C = { code m() { return B; } } : code
class D = new C().m()
```

is reduced by (CHECK) to

```
class C = { code m() { return B; } } : ⟨∅,∅, code m() ⟩
class D = new C().m()
```

is reduced by (META-CHECK) to

```
class C = { code m() { return B; } } :⟨∅,∅, code m() ⟩
class D = new C().m() : code
```

is reduced by (META-RED) to

```
class C = { code m() { return B; } } :⟨∅,∅, code m() ⟩
class D = { int one() { return new C().k(); } } : code
```

is reduced by (CHECK-ERROR) to `error`.

Compile-time execution checks that class `C` is well-typed, then checks that the expression `new C().m()` is of type `code`, then reduces this expression. Finally, the check that the resulting class `D` is well-typed fails since class `C` has no methods named `k`.

This example also illustrates that standard typechecking of class expressions only takes place when they are associated to a class name in the class table. For instance, the fact that base class B is ill-typed is known from the beginning, but is only detected when B is associated to `D`. This choice allows for more expressive power, at the cost of a later error detection. In further work we will investigate smarter strategies allowing one to discover some inconsistencies earlier, for instance using type constraints as in [1].

In order to state our soundness result, we define a judgment $\vdash \tilde{p}\ \text{OK}$ which states that annotations in $\tilde{p}$ are correct.

$$\text{(OKERROR)}\ \frac{}{\vdash \texttt{error}\ \text{OK}} \qquad \text{(OK1)}\ \frac{\emptyset \vdash cp{:}\Delta}{\vdash cp{:}\Delta\ cp'{:}\texttt{code}\ p\ \ \text{OK}}$$

$$\text{(OK2)}\ \frac{\emptyset \vdash cp{:}\Delta \qquad \Delta;\emptyset \vdash e{:}\texttt{code}}{\vdash cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e{:}\texttt{code})\ p\ \ \text{OK}}\ e \neq ce$$

Soundness is formally expressed by the usual progress and subject reduction properties. Proofs are in the Appendix.

Theorem 2.1 (Progress) *If* $\vdash \tilde{p}\ \text{OK}$, *then either* $\tilde{p} \longrightarrow \tilde{p}'$ *or* $\tilde{p} = \texttt{error}$ *or* $\tilde{p}$ *is of the form* $cp{:}\Delta$.

Theorem 2.2 (Subject reduction) *If* $\vdash \tilde{p}\ \text{OK}$ *and* $\tilde{p} \longrightarrow \tilde{p}'$ *then* $\vdash \tilde{p}'\ \text{OK}$.

3 Conclusion

We have presented a framework for extending a Java-like language (that is, a class-based statically typed language with nominal types) with class composition operators blended into conventional expressions, thus using meta-programming as a flexible tool for composing software. Compile-time execution reduces extended class tables to conventional ones, by evaluating meta-expressions. Safety is ensured by a lightweight approach, where conventional typechecking of class expressions only takes place when they appear as class definitions in the class table.

An important advantage of this lightweight approach is that it is *modular*, in the sense that it can be applied on top of an existing Java-like language. In particular, checked compile-time execution is defined on top of typechecking and reduction relations of the underlying Java-like language, as formally shown in Figure 4. Correspondingly, an implementation could basically consist in [11] an algorithmic version of the rules in Figure 4 where (META-RED) steps and (META-CHECK)-(CHECK) steps invoke the JVM and the Java compiler, respectively.

Metaprogramming approaches can be classified by two properties: whether the meta-language coincides with the conventional language (the so-called *meta-circular* approach), and whether the code generation happens during compilation. MetaML [16], Prolog [14] and OpenJava [17] are meta-circular languages, while C++ [7], D [4], Meta-trait-Java [12] and MorphJ [5] use a specialized meta-language. [12] Almost any dynamically typed language allows some sort of meta-circular facility, typically by offering an *eval* function. Such a function allows to run arbitrary code, represented by an input string. Regarding code generation, MetaML and Prolog performs the computation at run time, while C++, D, Meta-trait-Java, MorphJ and OpenJava use compile-time execution. Again, dynamically typed languages providing an *eval* function allow runtime meta-programming. [13]

The work presented in this paper lies in the area of meta-circular compile-time execution.

Among the above mentioned approaches, [17] is the one showing more similarities with ours. In OpenJava, programmers can add new language constructs on top of Java, and define the semantics of these new constructs by writing *meta-classes*, that is, particular Java classes which instruct the OpenJava compiler on how to perform a type-driven translation into standard Java. These meta-classes use the reflection-based *Meta Object Protocol (MOP)* to manipulate the source code. In the same way it is even possible to change the semantics of standard Java language constructs. A similar capability of specifying within the code instructions for contextual compilation has been recently provided in Java 6 by annotations.

However, this approach, besides being lower-level, has a very different goal w.r.t. ours, that is, to make easy for programmers to extend and possibly change the behaviour of an existing language, in rather arbitrary ways. In our case, instead, syntax and semantics of both the underlying language and the language for composing classes are fixed. The programmer is only allowed to define its own derived composition operators by using the whole expressive power of the underlying language. Note also that both approaches produce standard Java code; however, in our case this code is obtained by an algorithm which interleaves standard Java compilation and execution steps, rather than by a unique preprocessing step.

In this paper, we have illustrated our approach on a minimal class composition language, to be able to analyze in isolation the safety issue. Further work will be carried out in two complementary directions. On the one hand, we will design a richer composition language suitable for our aims, likely a subset/variant of Featherweight Jigsaw [11,9,10]. On the other hand, we will study alternative approaches to guarantee safety, ranging from a fully static analysis based on sophisticated types, as in MetaML [16], to intermediate solutions still including dynamic checks, but allowing earlier error detection. Moreover, to test

[11] Besides a parser for the extended language.

[12] The latest version of D seems to include a limited form of metacircular compilation.

[13] Some dynamically typed languages like Groovy allows meta-circular compile-time excution while they are translated into bytecode (or some other abstract representation).

the applicability of our proposal, we will develop a prototype which exploits our ideas by extending a real language such as Java.

Aknowledgments

We warmly thank the anonymous referees for their valuable help in improving the paper.

References

[1] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2005*. ACM Press, January 2005.

[2] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde, and Todd L. Veldhuizen. Generative programming and active libraries. In Mehdi Jazayeri, Rüdiger Loos, and David R. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer, 2000.

[3] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*. Springer, 2004.

[4] Digital Mars. D programming language, 2007. http://www.digitalmars.com/.

[5] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In *ECOOP'07 - Object-Oriented Programming*, pages 399–424. Springer, August 2007.

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.

[7] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, 2003.

[8] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Customizable composition operators for Java-like classes (extended abstract). In *ICTCS'09 - Italian Conf. on Theoretical Computer Science*, 2009.

[9] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw - a minimal core calculus for modular composition of classes. In Sophia Drossopoulou, editor, *ECOOP'09 - Object-Oriented Programming*, number 5653 in Lecture Notes in Computer Science. Springer, 2009.

[10] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw - a minimal core calculus for modular composition of classes. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, August 2009. Submitted for journal publication.

[11] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Flattening versus direct semantics for Featherweight Jigsaw. In *FOOL'09 - Intl. Workshop on Foundations of Object Oriented Languages*, 2009.

[12] John Reppy and Aaron Turon. Metaprogramming with traits. In Erik Ernst, editor, *ECOOP'07 - Object-Oriented Programming*, number 4609 in Lecture Notes in Computer Science, pages 373–398. Springer, 2007.

[13] Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Bern, February 2005.

[14] Leon Shapiro and Ehud Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, April 1994.

[15] Bjarne Stroustrup. *The C++ Programming Language*. Reading. Addison-Wesley, special edition, 2000.

[16] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[17] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Kilijian, and Kozo Itano. OpenJava: A class-based macro system for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science, pages 117–133. Springer, 2000.

A Proofs

Theorem A.1 (Progress) *If* $\vdash \tilde{p}$ OK, *then either* $\tilde{p} \longrightarrow \tilde{p}'$ *or* $\tilde{p} = \texttt{error}$ *or* $\tilde{p}$ *is of the form* $cp{:}\Delta$.

Proof By case analysis on the definition of $\vdash \tilde{p}$ OK.

(OKERROR) Trivial.

(OK1) We have $\vdash cp{:}\Delta\ cp'{:}\texttt{code}\ p$ OK and $\emptyset \vdash cp{:}\Delta$. We distinguish two subcases: either there exists a non-empty cp'' such that $cp'' \subseteq cp'$ and $\mathit{closed}(\Delta, cp'')$ or not.

cp'' **exists**

In this case,

- if $\Delta \vdash cp''{:}\Delta'$ for some Δ', then we can apply rule (CHECK),
- otherwise we can apply rule (CHECK-ERROR).

cp'' **not exists**

In this case,

- if p is empty and $cp'{:}\texttt{code}$ is not empty, then we can apply rule (CHECK-ERROR),
- if p is empty and $cp'{:}\texttt{code}$ is empty, then the program is of the form $cp{:}\Delta$,
- if p is not empty, then it is of the form $(C = e)\ p'$. In this case, if $\Delta; \emptyset \vdash e{:}\texttt{code}$, then we can apply rule (META-CHECK), otherwise rule (META-CHECK-ERROR).

(OK2) We have $\vdash cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e{:}\texttt{code})\ p$ OK, with e not of the form ce, and $\emptyset \vdash cp{:}\Delta$, $\Delta; \emptyset \vdash e{:}\texttt{code}$. From these last two judgments and the fact that e is not a value, by the progress property of the conventional language we know that $e \xrightarrow{cp} e'$, hence we can apply rule (META-RED).

□

Lemma A.2 (Weakening) $\Delta; \Gamma \vdash ce{:}CT$ *implies* $\Delta, \Delta'; \Gamma \vdash ce{:}CT$.

Lemma A.3 *If* $\emptyset \vdash cp{:}\Delta$ *and* $\Delta \vdash cp'{:}\Delta'$ *then* $\emptyset \vdash cp, cp'{:}\Delta, \Delta'$.

Proof Since $\Delta \vdash cp'{:}\Delta'$ has been deduced by rule (PROGRAM-T), we have

(i) for each $C = ce$ in cp', $\Delta, \Delta' \vdash ce{:}\Delta'(C)$,

(ii) $\Delta, \Delta' \vdash \_ < \_$ is acyclic by side condition.

Analogously, since $\emptyset \vdash cp{:}\Delta$ holds, we have that, for each $C = ce$ in cp, $\Delta \vdash ce{:}\Delta(C)$. Hence, by Lemma A.2, $\Delta, \Delta' \vdash ce{:}\Delta(C)$, and we can apply (PROGRAM-T) getting the thesis. □

Theorem A.4 (Subject reduction) *If* $\vdash \tilde{p}$ OK *and* $\tilde{p} \longrightarrow \tilde{p}'$ *then* $\vdash \tilde{p}'$ OK.

Proof By case analysis on the definition of $\tilde{p} \longrightarrow \tilde{p}'$.

(META-RED) We have

(i) $\tilde{p} \equiv cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e{:}\texttt{code})\ p \longrightarrow \tilde{p}' \equiv cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e'{:}\texttt{code})\ p$,

(ii) $e \xrightarrow{cp} e'$,

(iii) $\emptyset \vdash cp{:}\Delta$ and $\Delta; \emptyset \vdash e{:}\texttt{code}$, since $\vdash \tilde{p}$ OK holds.

From (ii) and (iii), by the subject reduction property of the conventional language, we get that $\Delta; \emptyset \vdash e'{:}\texttt{code}$. Hence, we can apply (OK2) with this premise and get $\vdash \tilde{p}'$ OK.

(META-CHECK) We have

(i) $\tilde{p} \equiv cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e)\ p \longrightarrow \tilde{p}' \equiv cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e{:}\texttt{code})\ p$.

(ii) $\Delta;\emptyset \vdash e{:}\texttt{code}$ by side condition.
(iii) $\emptyset \vdash cp{:}\Delta$ since $\vdash \tilde{p}\ \ \textsc{ok}$ holds.
Hence, we can apply (OK2) with premises (ii) and (iii) and get $\vdash \tilde{p}'\ \ \textsc{ok}$.

(META-CHECK-ERROR) We have $cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e)\ p \longrightarrow \texttt{error}$, hence we get the thesis by rule (OKERROR).

(CHECK) We have
(i) $cp{:}\Delta\ cp'{:}\texttt{code}\ \tilde{p} \longrightarrow cp{:}\Delta\ cp'{:}\Delta'\ \tilde{p}$,
(ii) $\Delta \vdash cp'{:}\Delta'$ by side condition,
(iii) $\emptyset \vdash cp{:}\Delta$, since $\vdash cp{:}\Delta\ cp'{:}\texttt{code}\ \tilde{p}\ \ \textsc{ok}$ holds,
From (ii) and (ii) we get $\emptyset \vdash cp, cp'{:}\Delta, \Delta'$ by Lemma A.3. Hence, we an apply (OK1) with this premise and get $\vdash cp{:}\Delta\ cp'{:}\Delta'\ \tilde{p}\ \ \textsc{ok}$.

(CHECK-ERROR) We have $cp{:}\Delta\ cp'{:}\texttt{code}\ \tilde{p} \longrightarrow \texttt{error}$, hence we get the thesis by rule (OKERROR).

□

Integrating extra-functional properties in component deployment dependencies

Meriem Belguidoum[1] and Fabien Dagnat[2]

[1,2] *Institut TELECOM, TELECOM Bretagne*
European University of Brittany
[1] *University of Mentouri, Constantine, Algeria*

Abstract

Component-Based Software Engineering (CBSE) is a widely used approach for the software design, particularly when addressing large scale software. The common practice is to build software by composing large collections of components. Such software requires a complex management of their dependencies to be deployed successfully and safely. Therefore, all component dependencies, functional and extra-functional one must be precisely and formally specified. In a previous work, we have proposed a formal language to specify functional dependencies and a formal deployment framework to manage them. Based on this work, we propose an extension with extra-functional dependencies specification and management for component deployment. With this extension, it is possible to specify that a component provides or requires a service with specific extra-functional properties (such as security level, version information, resource consumption level, etc.). We present here how specifying extra-functional component dependencies and how managing them to be able to ensure success and safety of component installation and deinstallation.

Keywords: Component-based software, Safe deployment, Extra-functional properties

1 Introduction

Component-Based Software Engineering (CBSE) [12] stresses the idea that a software system is a composition of pre-existing and newly developed components. One of the main contributions that CBSE has to this idea is the reuse of software components to save out development effort. The component reuse leads to a complex management of deployment dependencies [13]. Indeed, removing a shared component may have an impact on components which depend on it. Therefore, to deploy components we have to know explicitly all their functional and extra-functional dependencies to be able to manage them safely. Dependencies represent the relations between provided services and required one. Functional dependencies describe the intended behaviours of components or system and extra-functional dependencies include constraints on properties of components or system.

[1] Email: m.belguidoum@gmail.com

[2] Email: fabien.dagnat@telecom-bretagne.eu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Functional issues of component-based software engineering have been well investigated [5], [12], but very little research has been performed concerning non-functional properties. Moreover, a formal verification of extra-functional dependencies for deployment of component based system has not been really investigated.

In a previous work [2], we have proposed a component deployment framework which verifies the dependency description using a set of inference rules to guarantee the success and the safety of installation, deinstallation and upgrading [4] operations.

Our aim in this work is to extend the proposed framework to be able to verify the success and the safety of software deployment with respect to extra-functional properties. Therefore, in this paper, we propose an extension of our dependency language to enable the specification of extra-functional properties in addition to the usual functional one. With this extension, it is possible to specify that a component provides or requires a service with specific extra-functional properties (such as security level, version information, resource consumption level, ...). The properties are used in the description of components, services and target system. This extension requires:

(i) To be able to distinguish component instances. A component may be present several times in the system with different extra-functional properties. For example, depending on requirements, deploying an instance of a component with good QoS or another with less resource consumption.

(ii) To be able to distinguish service instances in a component description. A component may provide or require several times the same service with different extra-functional properties. For instance, a component may provide the same service with different properties : version, security level, language, etc.

(iii) To be able to determine if properties of an available component or an available service satisfy the corresponding constraints of the required component or the required service. For example, the available component having *version 3* satisfies the required component with the constraint: *version greater than 2*.

After this introduction, we present in section 2 the related work. In section 3, we describe more precisely what properties are and how they are described in the next section. Section 4 presents how the verification of extra-functional properties is integrated using deployment rules (proposed in [2]) to ensure the safety of installation and deinstallation operations. Finally, Section 5 concludes and discusses future work.

2 Related work

Several research using UML have been investigated to model extra-functional properties of software. Most important among these approaches is [7] in which a new UML profile [9] is proposed in order to model extra-functional properties in models based on the service component architecture specification. It shows how services and extra-functional behaviors can be modeled and described in a loosely coupled implementation environment by extending UML with profiles. Another interesting approach proposed by Skene et al. [11]. They present SLAng a language for precisely specifying service-level agreements (SLAs) using the precise UML (pUML) definition of the semantics of UML [6]. There, meta-models are used to specify both the syntax and the semantics of a modelling language. These approaches are interesting, however, the properties formalisation remains very low.

Another language for modelling non-functional properties of component-based systems is CQML (*Component Quality Modelling Language*) proposed in [1] it is based on UML and OCL (*Object Constraint Language*) [10]. It is a rich lexical language for QoS specification but it lacks formality and precise semantics. In [14] an interesting formal specification of timeliness properties of a component-based system is presented. It uses extended temporal logic of actions TLA+ [8] to describe the system. This work is somewhat similar to our work in that it attempts to provide a formal description of extra-functional properties. However, our aim is to verify and prove the safety of component deployment using predicate logic which is more simple and feasible comparing to TLA+ formalism.

Finally, we can quote OCL [10] the *Object Constraint Language* which is a formal language used to describe expressions on UML models. It provides the power of first-order predicate which is similar to our formalism. However, we prefer integrating extra-functional properties in our dependency language which is already proposed with its inference engine in our deployment framework [2] to be able to verify and prove deployment success and safety.

3 Dependencies with properties

In this paper, we present a basic notion of property: it has a *name* and a *type*. The *type* constrains the possible values that the property may be associated with. It also constrains the available operations on the corresponding property. For example, properties of number type offer comparison operations such as $>$, $<$, $\geq$, $\leq$ that are not available on a string type, etc. Properties are introduced in our deployment framework in the generic way by introducing them in our metamodel [3]. A deployed entity may be a complete system (defined as a set of components), a component or a service. In our dependency language we add in the provided side a list of *property* generally denoted by $\mathcal{E}$ with their values $[p_1 = v_1, \ldots, p_n = v_n]$. For example, a service s with a property named $version$ of value 1 is denoted by $s[version = 1]$. It is clear that when specifying a service (or a component) a property cannot be bound twice to different values. A *property constraint* specifies the constraint that the matching components or services must ensure. In the context of this paper, constraints on number property type may use any usual comparison operator ($>$, $\geq$, $<$, $\leq$, $=$, $\neq$) and constraints on string property type may use only equality or inequality.

Finally, a system is composed of components. A component provides (a composition of) services. Each provided services have a requirement which represents a conditional expression involving other entities. The dependency language grammar presented below defines more precisely which kind of composition is possible on provided right side and on required left side (dependency = required $\rightarrow$ provided: if requirements are satisfied then services are provided).

3.1 Basic satisfiability

Our reasoning framework compares the requirement of component dependencies with available services and components in the system. For instance, a required service s constrained by $[version \geq 3]$ can be satisfied by an available service s which has the property $[version = 4]$. For this, we define the notion of basic satisfiability of a property $\mathcal{E}$ in a constraint φ and denotes it $\varphi \leftarrow \mathcal{E}$ (see definition 3.1). Intuitively, this relation is verified if all the names of the constraints φ exist in the set of the names of properties $\mathcal{E}$. To define it more formally, we need to define the notion of *domain* for properties and constraints. The domain is the set of property names that appear in either a property or a constraint:
$dom([]) = \varnothing, dom([p_1 O_1 v_1, \ldots, p_n O_n v_n] = \{p_1\} \cup \cdots \cup \{p_n\}$

where O denote either a comparison operator (for constraints) or a binding operator (for properties).

Definition 3.1 (**Basic satisfiability**) A property $\mathcal{E}$ is satisfied in a constraint $\varphi = [p_1 O_1 v_1, \ldots, p_n O_n v_n]$ if and only if

(i) all the names of constraints are included in the set of the property names $\mathcal{E}$: $dom(\varphi) \subseteq dom(\mathcal{E})$

(ii) the constraint φ is true when substituting all properties by their values in $\mathcal{E}$: $(\mathcal{E}(p_1) O_1 v_1) \wedge \cdots \wedge (\mathcal{E}(p_n) O_n v_n)$

The satisfiability $(\varphi \leftarrow \mathcal{E})$ of a property of an entity x in the constraint of another entity x' is calculated as follows:
$(x', \varphi) \leftarrow (x, \mathcal{E}) = (x' = x) \wedge (\varphi \leftarrow \mathcal{E})$, where x and x' are components or services.

3.2 Dependency specification

Dependency specification is represented as the composition of provided services that require a composition of required services and components. Dependencies are composed using three operators $\#$, $\bullet$ and ?. Intuitively, these operators correspond respectively to disjunction (one of the sub-requirements must be met), conjunction (the two sub-requirement must be met) and optional dependency (if the sub-requirement is met, some services may be provided). Dependencies are described in more details in [2], the formal description follows a grammar based on predicate logic. In this paper, we extend this grammar by adding properties at the level of the provided services and the constraints on the properties at the level of required and forbidden components and services (see definition 3.2).

Definition 3.2 (**Dependencies**) The dependencies or intra-dependencies represent the relations between each provided service (which may have extra-functional properties) of a component and its requirements (which may have constraints on extra-functional properties). Dependencies are described using the predicate logic and are specified by the following grammar:

$$D ::= P \Rightarrow s\,\mathcal{E} \;\mid\; D \bullet D \;\mid\; D \,\#\, D \;\mid\; ?\,D \qquad P ::= \mathit{true} \;\mid\; P \wedge P \;\mid\; Q$$

$$Q ::= Q \vee Q \;\mid\; \varphi \;\mid\; \neg c\,\varphi \;\mid\; \neg s\,\varphi \;\mid\; c\,\varphi\,.s\,\varphi \;\mid\; s\,\varphi \qquad \mathcal{E} \;::=\;\mid\; [v = \mathit{value}\;\mathcal{E}']$$

$$\mathcal{E}' ::= \mid\, , v = \mathit{value}\;\mathcal{E}' \qquad \varphi \;::=\;\mid\; [v\;O\;\mathit{value}\;\varphi'] \qquad \varphi' ::= \mid, v\;O\;\mathit{value}\;\varphi'$$

$$O ::= > \mid \geq \mid < \mid \leq \mid = \mid \neq$$

The basic dependency $P \Rightarrow s\,\mathcal{E}$ correspond to a provided service s with properties $\mathcal{E}$ which has a requirement P. This requirement is expressed by a term called a predicate (in conjunctive normal form). Predicates are composed by usual logical operators (conjunction and disjunction). It represents: (1) constraints on the system φ, (2) a forbidden component with its property constraints $\neg c\ \varphi$, (3) a forbidden service with its property constraints $\neg s\ \varphi$, (4) a service provided by a specific component with their property constraints $c\ \varphi\ .s\ \varphi$ and (5) a service provided by any component with its property constraints $s\ \varphi$. As it is explained earlier, properties $\mathcal{E}$ represent the set of the corresponding values of properties (variable v, value), constraints φ represent the set of the triplet (variable v, operator O, value). In this paper, we consider variables as strings and values as string, number and boolean types.

For example, we assume that the dependency description of the mail server `postfix` is as follows:

$$([\mathit{FreeDisk} \geq 1380] \wedge \neg c_{sendmail} \wedge s_{lib}[\mathit{version} > 1.2] \Rightarrow s_{MTA}[\mathit{SecLevel} = 2])$$
$$\bullet\ ?(s_{amavis} \Rightarrow s_{AV}[\mathit{SecLevel} = 2])$$

It means that the component `postfix` needs 1380kb of free disk space and a service s_{lib} in a version greater than 1.2 and it conflicts with the component `sendmail`. When these conditions are satisfied, it can be installed and it will provide the service s_{MTA} (for Mail Transfer Agent) with a security level equal to 2. It may also provide an Anti virus service s_{AV} having security level value equal to 2 if the service s_{amavis} is available.

3.3 Context description

The description of a system is called a *context*. A context is composed of (1) its environment denoted by $\mathcal{E}$ or $ctx.\mathcal{E}$ (contains the value of the system properties), (2) the set of its components denoted $\mathcal{C}$, (3) the dependency graph between these components denoted $ctx.\mathcal{G}$. The structure of (2) and (3) are complex and are described in more detail below.

As a system may contain several instances of the same component, we need to identify all these instances. For a component c, the instances are named (c, num) where num is the instance number which is already installed in the system (see definition 3.3).

Definition 3.3 (CalcNum) The function *CalcNum* calculates the new component instance as follows:

$$CalcNum(c, Ctx) = max(\{0\} \cup \{n|(c, n, \ldots) \in Ctx.\mathcal{C}\}) + 1$$

The set of installed components contain component information. It is represented by $\mathcal{C}$ a set of six-tuples $(c, num, \mathcal{E}_c, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c)$ storing for each installed instance of a component c its number num, its properties $\mathcal{E}_c$, its provided services $\mathcal{P}_s$, its forbidden services $\mathcal{F}_s$ and its forbidden components $\mathcal{F}_c$. Each provided service is stored together with its environment $(s, \mathcal{E}_s)$. Forbidden services and forbidden components are stored with their associated constraints $\mathcal{F}_s$ (resp. $\mathcal{F}_c$), they are denoted by (s, φ_s) (resp. (c, φ_c)).

Finally, a dependency graph stores all dependencies between components of a system. It can be viewed as a model of the interconnection between component of the system.

Indeed, as components can provide several times the same service with different properties, it is necessary to identify the services to distinguish them. In our model, the service identifier, denoted by id_s.

A node of the graph $\mathcal{G}$ represents the identifier of an available service ($id_s = (id_c, s, \mathcal{E}_s)$), where $id_c = (c, num)$ represents the component identifier (the provider of the service s) and $\mathcal{E}_s$ represents its properties. An arc is a couple of nodes $id_{s1} \xmapsto{x,\varphi_c,\varphi_s} id_{s2}$, where x is the kind of dependency: M for mandatory or O for optional, id_{s2} requires id_{s1} with the constraints φ_c and φ_s, φ_c are constraints on the component which provides the service id_{s1} and φ_s are constraints on the required service id_{s1}.

Constraints must be stored in the graph to be able to ensure the safety of deployment operations that need to change the graph. For example, if a service s_1 has the security level $(ls = 3)$ and it is used by another service s_2 with a constraint on the security level $(ls > 2)$, let suppose that s_1 must be substituted by another service $s_3[ls = 2]$, s_3 cannot replace s_1 because the constraint of the service s_2 $(ls > 2)$ is not satisfied by the property of s_3. As the structure of the context is rather complex, we define utility functions that calculate the set of available services (resp. components) AS (resp. AC) and forbidden services (resp. components) FS (resp. FC) for a given context.

$$\begin{cases} AS(ctx) = \bigcup\{\mathcal{P}_s \mid (\ldots, \mathcal{P}_s, \ldots) \in ctx.\mathcal{C}\} \\ AC(ctx) = \{(c, num) \mid (c, num, \ldots) \in ctx.\mathcal{C}\} \\ FS(ctx) = \bigcup\{\mathcal{F}_s \mid (\ldots, \mathcal{F}_s, \ldots) \in ctx.\mathcal{C}\} \\ FC(ctx) = \bigcup\{\mathcal{F}_c \mid (\ldots, \mathcal{F}_c) \in ctx.\mathcal{C}\} \end{cases}$$

$\mathcal{P}_s$ is the set of service couples with their properties $(s, \mathcal{E}_s)$ and $Ctx.\mathcal{C}$ represents the elements $(c, num, \mathcal{E}_c, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c)$ describing each component c with its identifier num, all its properties $\mathcal{E}_c$, all its services $\mathcal{P}_s$, all its forbidden services $\mathcal{F}_s$ and all its forbidden components $\mathcal{F}_c$.

Adding properties in components and services and adding constraints on these properties in requirements makes difficult the calculation of available and forbidden components and services. The *Forbidden* function determines whether a provided component or a service are forbidden in the context (see the definition 3.4). The function *Available* determines whether a required component or a service required are available in the context (see definition 3.5).

Definition 3.4 (Forbidden) The $Forbidden$ function checks whether the *provided* component $(c, \mathcal{E}_c)$ respectively a *provided* service $(s, \mathcal{E}_s)$ are not forbidden in the context by verifying their *Satisfiability* (see definition 3.1) in the sets FC respectively FS with the properties of the corresponding component respectively a service. It is defined as follows:

$$\begin{cases} Forbidden(ctx, c, \mathcal{E}_c) = \exists (c', \varphi_c) \in FC(ctx) \mid (c', \varphi_c) \leftarrow (c, \mathcal{E}_c) \\ Forbidden(ctx, s, \mathcal{E}_s) = \exists (s', \varphi_c) \in FS(ctx) \mid (s', \varphi_s) \leftarrow (s, \mathcal{E}_s) \end{cases}$$

Definition 3.5 (Available) The function *Available* checks whether a *required* component, a *required* service or a *required* service provided by a specific component are available in the context. It verifies the *satisfiability* of available component in the constraint of the *required* component. It verifies the *satisfiability* of available service in the constraint of the *required* service and it verifies the *satisfiability* of available service from a specific component in the constraint of the *required* service from a specific component. It is defined as follows:

$$\begin{cases} Available(ctx, c, \varphi_c) = \exists (c', \mathcal{E}_c) \in AC(ctx) \mid (c, \varphi_c) \leftarrow (c', \mathcal{E}_c) \\ Available(ctx, s, \varphi_s) = \exists (s', \mathcal{E}_s) \in AS(ctx) \mid (s, \varphi_s) \leftarrow (s', \mathcal{E}_s) \\ Available(ctx, c.s, \varphi_c, \varphi_s) = \\ \exists (c', -, \mathcal{E}_c, \mathcal{P}_s, \ldots) \in ctx.\mathcal{C} \mid (c, \varphi_c) \leftarrow (c', \mathcal{E}_c) \wedge \exists (s', \mathcal{E}_s) \in \mathcal{P}_s \mid (s, \varphi_s) \leftarrow (s', \mathcal{E}_s) \end{cases}$$

Finally, when a service is available in a context, we have to know the set of its identifier. The function *CalcIdS* is used for this purpose. As it is explained in context description, the identifiers represent the graph nodes and we use them to distinguish between the different instances of the same service (provided by the same or different components).

Definition 3.6 (CalcIdS) The function *CalcIdS* calculates the identifiers of the required services (provided by any component or a precise one) which are available in the context. It is based on the satisfiability check, i.e., the verification of the required service constraints with properties of available corresponding service properties. It is defined as follows:

$$\begin{aligned} & CalcIds(ctx, c.s, \varphi_c, \varphi_s) = \\ & \quad \{(c, n, s, \mathcal{E}_s) \mid (c, n, \mathcal{E}_c, \mathcal{P}_s, -, -) \in ctx.\mathcal{C} \wedge (s, \mathcal{E}_s) \in \mathcal{P}_s \wedge (\varphi_c \leftarrow \mathcal{E}_c) \wedge (\varphi_s \leftarrow \mathcal{E}_s)\} \\ & CalcIds(ctx, s, \varphi_s) = \\ & \quad \{(c, n, s, \mathcal{E}_s) \mid (c, n, \mathcal{E}_c, \mathcal{P}_s, -, -) \in ctx.\mathcal{C} \wedge (s, \mathcal{E}_s) \in \mathcal{P}_s \wedge (\varphi_s \leftarrow \mathcal{E}_s)\} \end{aligned}$$

4 Deployment description

Once the basic notions of property, dependency and context are defined we can present the rules that ensure that deployment operations are safe. In this section, the discussion is centered on the installation and deinstallation operations.

4.1 Installation

The installation process is divided in two phases: a first step ensure component *installability* (its requirements are satisfied and its provides services do not conflict

Predicates:

$$\text{PTrue: } ctx \vdash_P true \qquad \text{PAnd: } \frac{C \vdash_P P_1 \qquad ctx \vdash_P P_2}{ctx \vdash_P P_1 \wedge P_2} \qquad \text{POrL: } \frac{ctx \vdash_P Q_1}{ctx \vdash_P Q_1 \vee Q_2}$$

$$\text{POrR: } \frac{ctx \vdash_P Q_2}{ctx \vdash_P Q_1 \vee Q_2} \qquad \text{PVar: } \frac{\varphi \leftarrow ctx.\mathcal{E}}{ctx \vdash_P \varphi} \qquad \text{PNotS: } \frac{\neg Available(ctx, s, \varphi_s)}{ctx \vdash_P \neg\, s\ \varphi_s}$$

$$\text{PNotC: } \frac{\neg Available(ctx, c, \varphi_c)}{ctx \vdash_P \neg\, c\ \varphi_c} \qquad \text{PServ: } \frac{Available(ctx, s, \varphi_s)}{ctx \vdash_P s\ \varphi_s}$$

$$\text{PComp: } \frac{Available(ctx, c.s, \varphi_c, \varphi_s)}{ctx \vdash_P c\ \varphi_c.s\ \varphi_s}$$

Dependencies:

$$\text{CTriv: } \frac{ctx \vdash_P P \qquad \neg Forbidden(ctx, s, \mathcal{E}_s)}{ctx \vdash_C P \Rightarrow s\,\mathcal{E}_s} \qquad \text{CAnd: } \frac{ctx \vdash_C D_1 \qquad ctx \vdash_C D_2}{ctx \vdash_C D_1 \bullet D_2}$$

$$\text{COpt: } ctx \vdash_C\ ?\,D \qquad \text{COrL: } \frac{ctx \vdash_C D_1}{ctx \vdash_C D_1 \# D_2} \qquad \text{COrR: } \frac{ctx \vdash_C D_2}{ctx \vdash_C D_1 \# D_2}$$

Fig. 1. Installability rules

with the context) and a second step calculate the *effect* resulting from the installation.

4.1.1 Installability

Installability involves (1) ensuring that any required service is available in the context (see definition 3.5), (2) ensuring that none of the components that will be installed is forbidden and also that none of its provided services is forbidden (see definition 3.4). It is defined as follows:

Definition 4.1 (**Installability**) A component c with a dependency D and a set of properties $\mathcal{E}_c$ is *installable* within a context ctx ($ctx \vdash_C c : D, \mathcal{E}_c$) iff the component with its corresponding properties is not forbidden and its dependency D is verified by the checking rules of Fig. 1. The general installability rule is defined below:

$$\text{CComp: } \frac{ctx \vdash_C D \qquad \neg Forbidden(ctx, c, \mathcal{E}_c)}{ctx \vdash c : D, \mathcal{E}_c}$$

The installability rules presented in Fig. 1 are similar to those proposed in [2]. The difference is in the verification of the availability of required components and services using *satisfiability* relation.

Simple dependencies $P \Rightarrow s\,\mathcal{E}_s$ are verified if the predicate P is true and $s\,\mathcal{E}_s$ is not forbidden (CTriv). The evaluation of a predicate P in the context ctx follows propositional logic and is presented in the first part of the figure (rules denoted by

$$\text{GTrue: } ctx, id \vdash_G true \Rightarrow \varnothing \qquad \text{GAnd: } \frac{ctx, id \vdash_G P_1 \Rightarrow \mathcal{G}_1 \qquad ctx, id \vdash_G P_2 \Rightarrow \mathcal{G}_2}{ctx, id \vdash_G P_1 \wedge P_2 \Rightarrow \mathcal{G}_1 \cup \mathcal{G}_2}$$

$$\text{GOr: } \frac{ctx, id \vdash_G Q_1 \Rightarrow \mathcal{G}_1 \qquad ctx, id \vdash_G Q_2 \Rightarrow \mathcal{G}_2}{ctx, id \vdash_G Q_1 \vee Q_2 \Rightarrow \mathcal{G}_1 \cup \mathcal{G}_2} \qquad \text{GVar: } ctx, id \vdash_G \varphi \Rightarrow \varnothing$$

$$\text{GNotS: } ctx, id \vdash_G \neg s'\varphi \Rightarrow \varnothing \qquad \text{GNotC: } ctx, id \vdash_G \neg c'\varphi \Rightarrow \varnothing$$

$$\text{GServC: } \frac{Available(ctx, c'.s', \varphi_c, \varphi_s)}{ctx, id \vdash_G c'\varphi_c.s'\varphi_s \Rightarrow \{id' \xmapsto{\mathsf{M},\varphi_c,\varphi_s} id \mid id' \in CalcIds(ctx, c'.s', \varphi_c, \varphi_s)\}}$$

$$\text{GServ: } \frac{Available(ctx, s', \varphi_s)}{ctx, id \vdash_G s'\varphi_s \Rightarrow \{id' \xmapsto{\mathsf{M},[],\varphi_s} id \mid id' \in CalcIds(ctx, s', \varphi_s)\}}$$

Fig. 2. Graph calculation rules

$\vdash_P$). During installability, optional dependencies are ignored (COpt) because such dependencies may be unavailable without preventing component installation. The conjunction of dependencies is verified when the two dependencies are valid (CAnd), the disjunction ($D_1 \# D_2$) means `if` D_1 `then` COrL `else if` D_2 `then` COrR.

4.1.2 Installation

Once the component installability is checked using installability rules, the effect of its installation in the target system must be calculated. This effect gathers the new available services (with their properties), the new forbidden services and components (with their constraints) and the new dependencies (as a dependency graph). Before presenting the installation rules, we describe first some functions which calculate the installation effect.

Definition 4.2 (Graph calculation) The dependency graph $\mathcal{G}$ is built as components are deployed. For each component we compute the subgraph that corresponds to it to build the global graph. The subgraph of a component c which provides services having identifiers id_s is calculated from each predicat P of its dependency ($ctx, id_s \vdash_G P \Rightarrow \mathcal{G}$) by the rules presented in Fig. 2.

The difference between these rules and those presented in [2] is the use of service identifier for as nodes and constraints of components and services as labels. The rules GTrue, GAnd, GOr, GVar, GNotS, GNotC are exactly identical and consists in the gathering process of new arcs. The two last rules are the only rules that add new arcs. The rule GServC bind all potential providers of each required service to the current provided service. The set of providers is calculated using the previously defined *CalcIds* of the definition 3.6. Each new binding is labeled by $(\mathsf{M}, \varphi_c, \varphi_s)$, meaning that the dependency is mandatory and the constraints imposed on the required component and its service are φ_c and φ_s. The rule GServ is similar, but links the service id with any component providing it.

The second function necessary to calculate installation effect is *CalcF* that gathers of all forbidden entities (services or components).

Definition 4.3 (CalcF) The function *CalcF* calculates the forbidden components and the forbidden services using the condition of the dependency predicate. It computes them as follows:

$$\begin{cases} CalcF(true) = CalcF(s\ \varphi) = CalcF(c\ \varphi.s\ \varphi) = CalcF(\varphi) = \varnothing, \varnothing \\ CalcF(P_1 \wedge P_2) = \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2 \quad \text{where } CalcF(P_i) = \mathcal{F}_s^i, \mathcal{F}_c^i \\ CalcF(Q_1 \vee Q_2) = \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2 \quad \text{where } CalcF(Q_i) = \mathcal{F}_s^i, \mathcal{F}_c^i \\ CalcF(\neg\, s\varphi) = \begin{cases} \{(s,\varphi)\}, \varnothing \ \texttt{if} \ \neg Available(ctx, s, \varphi) \\ \varnothing, \varnothing \ \texttt{else} \end{cases} \\ CalcF(\neg\, c\varphi) = \begin{cases} \varnothing, \{(c,\varphi)\} \ \texttt{if} \ \neg Available(ctx, c, \varphi) \\ \varnothing, \varnothing \ \texttt{else} \end{cases} \end{cases}$$

Definition 4.4 (Installation) The installation of a component c with a dependency D in a context ctx adds: the new component instance (its number num) with its provided services $\mathcal{P}_s$, forbidden services $\mathcal{F}_s$ and forbidden components $\mathcal{F}_c$ and the subgraph ($\mathcal{G}$) (see rules in Fig. 3) into the context.

$$\text{IComp:}\ \frac{ctx, c, num \vdash_I D \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G} \qquad num = CalcNum(c, ctx)}{ctx \vdash_I c : D, \mathcal{E}_c \Rightarrow (c, num, \mathcal{E}_c, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c), \mathcal{G}}$$

The main rule that calculates the effect of the installation is the rule ITriv. The provided services represent the couples of services and their properties $(s, \mathcal{E}_s)$. The sets of forbidden services and forbidden components are calculated with the function *CalcF*. The dependency graph is calculated using the rules of Fig. 2 which are denoted by $(\vdash_G)$ in the rule ITriv. The function *Forbidden* checks whether the provided services with all their properties are not in conflict with the constraints of the context.

4.1.3 An installation example

We assume that the component `POSTFIX` (denoted by C_{PX}) provides the service S_{MTA} with the property version equals to $(v = 3)$ if a set of requirements are fulfilled. These requirements are: Free Disk Space $(FDS \geq 1380)$, the component `SENDMAIL` (C_{SM}) having a version greater or equal to 2 $(v \geq 2)$ is forbidden, the library S_{lib} with a version greater or equal to 3 is required $(v \geq 3)$. The description of `POSTFIX` dependency is as follows: $D_1 \bullet D_2$:

$$\begin{cases} D_1 = [FDS \geq 1380] \wedge C_{SM}\ [v \geq 2] \wedge S_{lib}\ [v \geq 3] \Rightarrow S_{MTA}\ [v = 3] \\ D_2 = ?(C_A\ [v \geq 3, NS \geq 2].S_{amavis}\ [NS \geq 3] \Rightarrow S_{AV}\ [NS = 4]) \end{cases}$$

Let's have the following initial context with environment properties $\mathcal{E}$ and a set of components $\mathcal{C}$

$$\begin{cases} \boldsymbol{\mathcal{E}} = \{FDS = 500000, OS = LINUX, RAM = 128\}; \\ \boldsymbol{\mathcal{C}} = \{(C_1, 1, [], (S_{lib}\ [v = 3]), \varnothing, \varnothing), (C_A, 1, [v = 4, NS = 3], (S_{amavis}\ [NS = 3]), \varnothing, \varnothing)\}; \end{cases}$$

The installability proof

The installability proof is demonstrated using the installability rules of the Fig. 1. It is as follows:

$$\text{ITriv: } \frac{ctx, id_s \vdash_G P \Rightarrow \mathcal{G} \qquad \neg Forbidden(s, \mathcal{E}_s) \qquad CalcF(P) = \mathcal{F}_s, \mathcal{F}_c}{ctx, id \vdash_I (P \Rightarrow s, \mathcal{E}_s) \Rightarrow \{s, \mathcal{E}_s\}, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}$$

$$\text{INot}_1\text{: } \frac{ctx \vdash_P \neg P}{ctx, id \vdash_I (P \Rightarrow s, \mathcal{E}_s) \Rightarrow \bot} \qquad \text{INot}_2\text{: } \frac{Forbidden(s, \mathcal{E}_s)}{ctx, id \vdash_I (P \Rightarrow s, \mathcal{E}_s) \Rightarrow \bot}$$

$$\text{IOpt}_1\text{: } \frac{ctx, id \vdash_I D \Rightarrow \bot}{ctx, id \vdash_I \,?\,D \Rightarrow \varnothing, \varnothing, \varnothing, \varnothing}$$

$$\text{IOpt}_2\text{: } \frac{ctx, id \vdash_I D \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}{ctx, id \vdash_I \,?\,D \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \{s \xmapsto{\mathsf{O}, \varphi_c, \varphi_s} s' \mid s \xmapsto{-, \varphi_c, \varphi_s} s' \in \mathcal{G}\}}$$

$$\text{IAnd}_1\text{: } \frac{ctx, id \vdash_I D_1 \Rightarrow \bot}{ctx, id \vdash_I D_1 \bullet D_2 \Rightarrow \bot} \qquad \text{IAnd}_2\text{: } \frac{ctx, id \vdash_I D_2 \Rightarrow \bot}{ctx, id \vdash_I D_1 \bullet D_2 \Rightarrow \bot}$$

$$\text{IAnd}_3\text{: } \frac{ctx, id \vdash_I D_1 \Rightarrow \mathcal{P}_s^1, \mathcal{F}_s^1, \mathcal{F}_c^1, \mathcal{G}_1 \qquad ctx, id \vdash_I D_2 \Rightarrow \mathcal{P}_s^2, \mathcal{F}_s^2, \mathcal{F}_c^2, \mathcal{G}_2}{ctx, id \vdash_I D_1 \bullet D_2 \Rightarrow \mathcal{P}_s^1 \cup \mathcal{P}_s^2, \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2, \mathcal{G}_1 \cup \mathcal{G}_2}$$

$$\text{IOrL: } \frac{ctx, id \vdash_I D_1 \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}{ctx, id \vdash_I D_1 \# D_2 \Rightarrow \mathcal{P}_s, \mathcal{F}_c, \mathcal{F}_s, \mathcal{G}}$$

$$\text{IOrR: } \frac{ctx, id \vdash_I D_1 \Rightarrow \bot \qquad ctx, id \vdash_I D_2 \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}{ctx, id \vdash_I D_1 \# D_2 \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}$$

Fig. 3. Installation rules

$$\textbf{CComp}\ \frac{\textbf{CAnd}\ \dfrac{\dfrac{\mathbf{A}}{Ctx \vdash_C D_1} \qquad ?(C_A\ [v \geq 3, NS \geq 2].S_{amavis}\ [NS \geq 3] \Rightarrow S_{AV}\ [NS = 4])}{Ctx \vdash_C D_1 \ \bullet\ D_2} \qquad \neg Forbidden(Ctx, C_{PX}, [])}{Ctx \vdash C_{PX} : D_1 \ \bullet\ D_2}$$

$$\mathbf{A} = \textbf{CTriv}\ \frac{\dfrac{\mathbf{B}}{Ctx \vdash_P [FDS \geq 1380] \wedge \neg C_{SM}\ [v \geq 2] \wedge S_{lib}\ [v \geq 3]} \neg Forbidden(Ctx, S_{MTA}, [v = 3])}{Ctx \vdash_C [FDS \geq 1380] \wedge \neg C_{SM}\ [v \geq 2] \wedge S_{lib}\ [v \geq 3] \Rightarrow S_{MTA}\ [v = 3]}$$

$$\mathbf{B} = \textbf{PTriv}\ \frac{\textbf{PVar}\ \dfrac{500000 \geq 1380}{Ctx \vdash_P FDS \geq 1380} \quad \textbf{PNotC}\ \dfrac{\neg Available(Ctx, C_{SM}, [v \geq 2])}{Ctx \vdash_P \neg C_{SM}\ [v \geq 2]} \quad \textbf{PServ}\ \dfrac{Available(Ctx, S_{lib}, [v \geq 3])}{Ctx \vdash_P S_{lib}\ [v \geq 3]}}{Ctx \vdash_P [FDS \geq 1380] \wedge \neg C_{SM}\ [v \geq 2] \wedge S_{lib}\ [v \geq 3]}$$

The installation proof
Once `POSTFIX` is installable, we compute the installation effect using installation rules of Fig. 3. The context will be updated with new provided services, new forbidden components, new forbidden services and the new dependency subgraph. D_1 respectively D_2 are demonstrated by the set of inference rules denoted respectively

by **A'** and **B**.

$$\textbf{IComp}\ \dfrac{\textbf{IAnd3}\ \dfrac{\dfrac{\textbf{A'}}{Ctx, C_{PX}, 1 \vdash_I D_1 \Rightarrow \mathcal{P}_s^1, \mathcal{F}_s^1, \mathcal{F}_c^1, \mathcal{G}_1} \qquad \dfrac{\textbf{B}}{Ctx, C_{PX}, 1 \vdash_I D_2 \Rightarrow \mathcal{P}_s^2, \mathcal{F}_s^2, \mathcal{F}_c^2, \mathcal{G}_2}}{Ctx, C_{PX}, 1 \vdash_I D_1 \bullet D_2 \Rightarrow \mathcal{P}_s^1 \cup \mathcal{P}_s^2, \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2, \mathcal{G}_1 \cup \mathcal{G}_2} \qquad 1 = CalcNum(C_{PX}, Ctx)}{Ctx \vdash_C C_{PX} : D_1 \bullet D_2 \Rightarrow (C_{PX}, 1, \mathcal{P}_s^1 \cup \mathcal{P}_s^2, \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2), \mathcal{G}_1 \cup \mathcal{G}_2}$$

$$\textbf{A'=ITriv}\ \dfrac{\textbf{GAnd}\ \dfrac{\textbf{GServ}\ \dfrac{Available(Ctx, S_{lib}, [v \geq 3])}{Ctx, C_{PX}, 1, S_{MTA}, [v = 3] \vdash_G S_{lib}\ [v \geq 3] \Rightarrow \mathcal{G}_1} \qquad Ctx, C_{PX}, 1, S_{MTA}, [v = 3] \vdash_G C_{SM}\ [v \geq 2] \Rightarrow \varnothing \\ Ctx, C_{PX}, 1, S_{MTA}, [v = 3] \vdash_G [FDS \geq 1380] \Rightarrow \varnothing}{Ctx, C_{PX}, 1, S_{MTA}, [v = 3] \vdash_G P \Rightarrow \mathcal{G}_1} \\ Forbidden(Ctx, S_{MTA}, [v = 3]) \qquad CalcF(P) = \varnothing, \{(C_{SM}, [v \geq 2])\}}{Ctx, C_{PX}, 1 \vdash_I (P \Rightarrow S_{MTA}, [v = 3]) \Rightarrow \{(S_{MTA}, [v = 3])\}, \varnothing, \{(C_{SM}, [v \geq 2])\}, \mathcal{G}_1}$$

$$\textbf{B=IOpt2}\ \dfrac{\textbf{ITriv}\ \dfrac{\textbf{GServ}\ \dfrac{Available(Ctx, S_{amavis}, [NS \geq 3])}{Ctx, C_{PX}, 1, S_{AV}, [NS = 4] \vdash_G S_{amavis}\ [NS \geq 3] \Rightarrow \mathcal{G}_2} \\ \neg Forbidden(Ctx, S_{AV}, [NS = 4]) \qquad CalcF(C_A\ [v \geq 3, NS \geq 2].S_{amavis}\ [NS \geq 3]) = \varnothing, \varnothing}{Ctx, C_{PX} \vdash_I (C_A\ [v \geq 3, NS \geq 2].S_{amavis}\ [NS \geq 3] \Rightarrow S_{AV}.[NS = 4]) \Rightarrow \{(S_{AV}, [NS = 4])\}, \varnothing, \varnothing, \mathcal{G}_2}}{Ctx, C_{PX} \vdash_I ?(C_A\ [v \geq 3, NS \geq 2].S_{amavis}\ [NS \geq 3] \Rightarrow S_{AV}.[NS = 4]) \Rightarrow \{(S_{AV}, [NS = 4])\}, \varnothing, \varnothing, \mathcal{G}_2}$$

$$\begin{cases} \mathcal{P}_s^1, \mathcal{F}_s^1, \mathcal{F}_c^1 = \{(S_{MTA}, [v = 3])\}, \{\}, \{(C_{SM}, [v \geq 2])\} \\ \mathcal{G}_1 = \{(C_1, 1, lib\ [v = 3]) \xmapsto{M, [], [v \geq 3]} (PX, 1, S_{MTA}\ [v = 3]\} \\ \mathcal{P}_s^2, \mathcal{F}_s^2, \mathcal{F}_c^2 = \{S_{AV}\ [NS = 4]\}, \{\}, \{\} \\ \mathcal{G}_2 = \{C_A\ [v = 4, NS = 3], 1, S_{amavis}\ [NS = 3] \xmapsto{O, [v \geq 3, NS \geq 2], [NS \geq 3]} PX, 1, S_{AV}\ [NS = 4]\} \end{cases}$$

After the installation of `POSTFIX`, the context is updated. The resulting dependency graph represents the union of $\mathcal{G}_1$ and $\mathcal{G}_2$ presented above. The component `POSTFIX` (C_{PX},1,[],$\mathcal{P}_s$,$\mathcal{F}_c$,$\mathcal{F}_s$) is added to the set of components:

$$\begin{aligned} \boldsymbol{\mathcal{C}} = \{&(C_1, 1, [], (S_{lib}\ [v = 3]), \varnothing, \varnothing), (C_A, 1, [v = 4, NS = 3], (S_{amavis}\ [NS = 3]), \varnothing, \varnothing), \\ &\boldsymbol{(C_{PX}, 1, [], \{(S_{MTA}\ [v = 3]), (S_{AV}, [NS = 4])\}, (S_{SM}\ [v \geq 2]), \varnothing)}\} \end{aligned}$$

Notice that any component `SENDMAIL` (S_{SM}) having a version greater or equal to 2 ($v \geq 2$) is forbidden otherwise (if $v < 2$) the component S_{SM} is not forbidden.

4.2 Deinstallation

The deinstallation follows the same approach as installation. The first phase checks the feasibility of deinstallation, and the second phase calculates the effect of deinstallation on the context.

4.2.1 Deinstallability

The principle of verification is exactly the same as presented in [2]. The difference lies in the dependency syntax and management of components and services identifiers (*instances*). Indeed, we can remove an instance of a component (c, num) if none of its instances of provided services id_s is used necessarily by other components directly on indirectly (see definition 4.5).

Definition 4.5 (Mandatory dependencies (MD)) The set of mandatory dependencies (MD) of a service id_s in a dependency graph $\mathcal{G}$ is the set of nodes that have a mandatory dependencies and belong to all graph paths from id_s to the leaves:

$$MD(\mathcal{G}, id_s) = \bigcup \{\{id_{s'}\} \cup MD(\mathcal{G}, id_{s'}) \mid id_s \xmapsto{(\mathsf{M}, -, -)} id_{s'} \in \mathcal{G}\}$$

Definition 4.6 (Optional dependencies (OD)) The set of optional dependencies (OD) of a service id_s in a dependency graph $\mathcal{G}$ is the set of nodes that have optional dependencies and belong to all graph paths from id_s to the leaves:

$$OD(\mathcal{G}, id_s) = \bigcup \{\{id_{s'}\} \cup OD(\mathcal{G}, id_{s'}) \mid id_s \xmapsto{(\mathsf{O},-,-)} id_{s'} \in \mathcal{G}\}$$

Definition 4.7 (Deinstallability) A component (c, num) can be deinstalled from the context ctx iff all the instances of its provided services are not used by other components in a mandatory way, i.e. there is no mandatory dependency in all paths from all these provided serviced to the graph leaves. It is checked using the following rule:

$$\text{Check-DI: } \frac{(c, num, \mathcal{E}_c, \mathcal{P}_s, -, -) \in ctx.\mathcal{C} \qquad \bigcup\{MD(\mathcal{G}, (c, num, s, \mathcal{E}_s)) \mid (s, \mathcal{E}_s) \in \mathcal{P}_s\} = \varnothing}{ctx \vdash_D (c, num)}$$

4.2.2 Deinstallation

The structure of the context containing all instances separately makes it possible to deinstall an instance only by removing the corresponding tuple from $\mathcal{C}$. Therefore, the deinstallation effect is a set of nodes that have to be removed from the dependency graph and is calculated in the same way as in [2]. The difference lies in the management of component and service identifiers. The nodes that have to be removed are those corresponding to the nodes of provided services of corresponding component and those that are used directly or indirectly in an optional way (see definition 4.6) and do not use any other service.

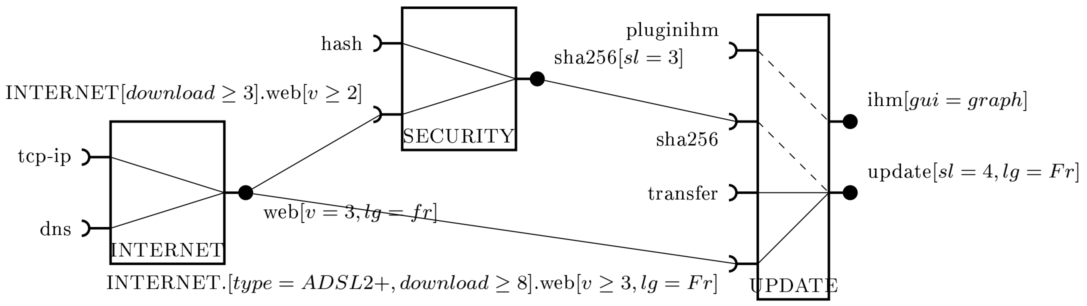
Definition 4.8 (Deinstallation) The deinstallation rule calculates the set of nodes that have to be removed from the graph with their corresponding arcs. The removed nodes are those provided by the component (c, num) and those which have only optional arcs in each path from them to graph leaves and are not tails of any arc. The calculation of removing nodes follows this rule:

$$\text{Effet-DI: } \frac{(c, num, \mathcal{E}_c, \mathcal{P}_s, -, -) \in ctx.\mathcal{C}}{ctx \vdash_E (c, num) \Rightarrow \bigcup_{(s, \mathcal{E}_s) \in \mathcal{P}_s} (\{(s, \mathcal{E}_s)\} \cup \{id_{s'} \in OD(ctx.\mathcal{G}, (c, num, s, \mathcal{E}_s)) \mid \nexists n, n \xmapsto{-} id_{s'} \in ctx.\mathcal{G}\})}$$

After deinstallation, the component tuple is removed from $ctx.\mathcal{C}$ and the set N is removed from the dependency graph:
$\mathcal{G} \setminus N = \{n_1 \overset{x}{\longmapsto} n_2 \mid n_1 \overset{x}{\longmapsto} n_2 \in \mathcal{G} \wedge n_1 \notin N \wedge n_2 \notin N\}$ where N is the set of nodes calculated by the rule Effet-DI ($ctx \vdash_E c \Rightarrow N$).

4.2.3 A deinstallation example

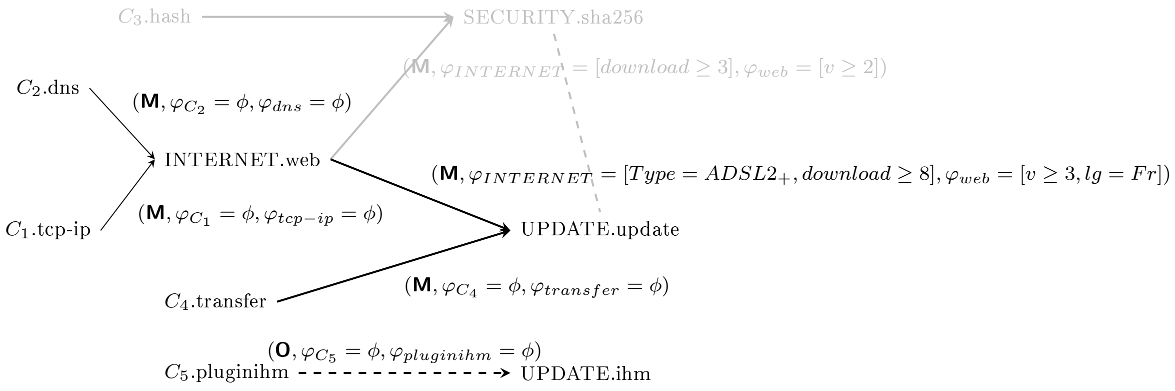
An example of dependencies of a component assembly is illustrated in Fig. 4. This assembly is composed of the components `INTERNET`, `SECURITY` and `UPDATE` which are represented in boxes. The intra-dependencies are inside the boxes, the required services are represented in the left side of each box and the provided one are in the right side. Dependencies can be necessary or optional (represented by dotted line). To update applications the component `UPDATE` needs (1) a transfer function (`transfer`), (2) a service `web` with the version and language constraints: $[v \geq 3, lg = Fr]$ which is provided by the component `INTERNET` and must satisfy the constraints $[type = ADSL2+, download \geq 8Mb/s]$ and (3) may uses a hash function (`sha256`) provided by SECURITY. When all these requirements are fulfilled the component UPDATE provides the service `update` with the security level and language properties $[sl = 4, lg = Fr]$. Its provides also the optional service `ihm` with a graphical user interface $[gui = graph]$ which depend on the service `pluginihm`. Fig. 5 shows the dependency graph of this component assembly including the gray part.



hash
pluginihm
sha256[$sl = 3$]
INTERNET[$download \geq 3$].web[$v \geq 2$]
SECURITY
ihm[$gui = graph$]
sha256
tcp-ip
update[$sl = 4, lg = Fr$]
transfer
web[$v = 3, lg = fr$]
dns
INTERNET
INTERNET.[$type = ADSL2+, download \geq 8$].web[$v \geq 3, lg = Fr$]
UPDATE

Fig. 4. Example of a component for the UPDATE

Let's verify the deinstallation of the component SECURITY with its services (see Fig. 4 and Fig. 5). The service `SECURITY.sha256` provided by the component SECURITY is optional for the component UPDATE. The set of mandatory dependencies (MD) of the service provided by the component SECURITY is empty. Therefore, this component is deinstallable. The nodes which are removed from the graph are calculated by the rule Effect-DI. Indeed, the service `UPDATE.update` which is part of the optional dependency of the service `SECURITY.sha256` has two requirements `INTERNET`$[\varphi]$`.web`$[\varphi]$ and `C4.transfet`, it represents the tail of two arcs. Therefore, the component UPDATE can not be deinstalled. The removed service is `SECURITY.sha256`. The resultant graph is represented in Fig. 5.



C_3.hash
SECURITY.sha256
$(\mathbf{M}, \varphi_{INTERNET} = [download \geq 3], \varphi_{web} = [v \geq 2])$
C_2.dns
$(\mathbf{M}, \varphi_{C_2} = \phi, \varphi_{dns} = \phi)$
INTERNET.web
$(\mathbf{M}, \varphi_{INTERNET} = [Type = ADSL2_+, download \geq 8], \varphi_{web} = [v \geq 3, lg = Fr])$
$(\mathbf{M}, \varphi_{C_1} = \phi, \varphi_{tcp-ip} = \phi)$
C_1.tcp-ip
UPDATE.update
$(\mathbf{M}, \varphi_{C_4} = \phi, \varphi_{transfer} = \phi)$
C_4.transfer
$(\mathbf{O}, \varphi_{C_5} = \phi, \varphi_{pluginihm} = \phi)$
C_5.pluginihm
UPDATE.ihm

Fig. 5. The dependency graph after the deinstallation of the component SECURITY

5 Conclusion and Future work

In this paper, we have presented an extension of the deployment formalisation which is described in [2]. This extension concerns the integration of extra-functional properties in the deployment reasoning engine. Therefore, the description of dependencies, the description of the context and the description of deployment rules are extended by properties and constraints on them. Thus, in the requirement side of dependencies we express the components and the services constraints and in the provided side of the dependencies we add properties of both services and components.

This extension is not limited to the component and service names, its advantage is to be able to manage different instances of the same component and the same service according to their properties and depending on requirements and needs. Therefore, we will identify services and components by their instances and not by their names. Consequently, the verification of installation deinstallation is based on instance *satisfiability* and not only name comparison. For that, deployment rules presented in [2] are extended to take into account the verification of extra-functional properties. The integration of extra-functional properties in substitution phase presented in [4] will be treated in another paper.

Finally, we plan to deal with deployment policies which are often associated with extra-functional properties. These policies are used to control and personalise deployment. For this reason, we have to formalise policies with their associated rules for each deployment phase to improve deployment quality while maintaining safety.

References

[1] Aagedal, J., "Quality of Service Support in Development of Distributed Systems," Phd thesis, University of Oslo (2001).

[2] Belguidoum, M. and F. Dagnat, *Dependency management in software component deployment*, in: *FACS'06-International Workshop on Formal Aspects of Component Software* (2006).

[3] Belguidoum, M. and F. Dagnat, *Dependability in software component deployment*, in: *DepCoS-RELCOMEX* (2007), pp. 223–230.

[4] Belguidoum, M. and F. Dagnat, *Formalization of component substitutability*, in: *FACS'07-International Workshop on Formal Aspects of Component Software* (2007).

[5] Cheesman, J. and J. Daniels, "UML Components: A Simple Process for Specifying Component-Based Software," Addison Wesley Professional, 2001.

[6] Clark, A., A. Evans and S. Kent, *Engineering Modelling Languages: A Precise Metamodelling Approach*, in: *Fundamental Approaches to Software Engineering*, Grenoble, France, 2002, pp. 159–173.

[7] G.Ortiz and J. Hernandez, *Toward UML Profiles for Web Services and their Extra-Functional Properties*, in: *ICWS '06: Proceedings of the IEEE International Conference on Web Services* (2006), pp. 889–892.

[8] Lamport, L., "Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[9] Object Management Group, *UML profile for schedulability, performance, and time specification*, Technical report, OMG (2002).
URL http://www.omg.org/cgi-bin/doc?ptc/02-03-02

[10] Object Management Group, *UML 2.0 OCL Specification.*, Technical report, OMG (2003).
URL http://www.omg.org/cgi-bin/doc?formal/02-06-39

[11] Skene, J., D. D. Lamanna and W. Emmerich, *Precise service level agreements*, in: *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (2004), pp. 179–188.

[12] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," Addison-Wesley, 1998.

[13] Vieira, M. and D. Richardson, *The role of dependencies in component-based systems evolution*, in: *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution* (2002), pp. 62–65.

[14] Zschaler, S., *Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof*, Software and Systems Modelling (SoSyM) (2009), to appear.