4th International Workshop on

Formal Aspects of Component Software

FACS’07

Sophia-Antipolis, France

September 19-21, 2007

Preliminary Proceedings

Guest Editors: Markus Lumpe and Eric Madelaine
Electronic Notes in Theoretical Computer Science

4th International Workskop on
FORMAL ASPECTS OF COMPONENT SOFTWARE
FACS’07

Sophia-Antipolis, France

September 19-21, 2007

Preliminary Proceedings

Guest Editors:

MARKUS LUMPE and ERIC MADELAINE
Contents

Preface v
Program vi
Committees viii

SLIM KALLEL, ANIS CHARFI, AND MOHAMED JMAIEL
Aspect-based Approach for Specifying and Enforcing Architecture Invariants 1

BERHARD SCHÄTZ
Modular Functional Specification of Reactive Components 16

JAVIER CUBO, GWEN SALAÜN, CARLOS CANAL, ERNESTO PIMENTEL, AND PASCAL POIZAT
A Model-Based Approach to the Verification and Adaptation of WF/.NET Components 30

MILA MAJSTER-CEDERBAUM, MORITZ MARTENS, AND CHRISTOPH MINNAMEIER
Liveness in Interaction Systems 47

VASU ALAGAR AND MUBARAL MOHAMMAD
A Component Model for Trustworthy Real-Time Reactive Systems Development 62

DAVID PAULO PEREIRA AND ANA C. V. DE MELO
A CSP Architectural Model for Fault-tolerant Systems 79

MERIEM BELGUIDOUm AND FABIEN DagnAT
Formalization of Component Substitutability 94

YOUCEF HAMMAL
Substitutability Relations for Active Components 109

JEAN-BAPTISTE RACLET
Residual for Component Specifications 125

JAVIER CÁMARA, GWEN SALAÜN, AND CARLOS CANAL
Multiple Concern Adaptation for Run-time Composition in Context-Aware Systems 140

FABRICIO FERNANDES AND JEAN-CLAUDE ROYER
The STSLib Project: Towards a Formal Component Model Based on STS 156

JASMIN CHRISTIAN BLANCHETTE AND OLAF OWEX
An Open System Operational Semantics for an Object-Oriented and Component-Based Language 175

JIN-HYUN KIM, JAE-HWAN SIM, AND JIN-YOUNG CHOI
Resource-Oriented Design Framework for Embedded System Components 190
MUCK VAN WEERDENBURG
Process Algebra with Local Communication .......................... 205

TOBIAS BLECHMANN AND CHRISTEL BAIER
Checking Equivalence for Reo Networks .............................. 220
Preface

On behalf of the Organizing Committee we are pleased to present the preliminary proceedings of the 4th International Workshop on Formal Aspects of Component Software (FACS’07).

The objective of the FACS workshop series 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 emerged as a promising paradigm to deal with the ever increasing need for mastering system complexity, for enabling evolution and reuse, and for driving software engineering into sound production and engineering standards. Soon, however, it became a popular technology long before well-understood and widely adopted formal foundations have emerged. Issues like mathematical models for components, their interaction and composition, or rigorous approaches to verification, deployment, testing and certification remain open research questions and challenging opportunities for formal methods. Moreover, new challenges are raised by applications of this paradigm to safety-critical, mobile, or reconfigurable systems.

FACS’07 is the fourth 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 held in Pisa, Italy, in September 2003, co-located with FM’03. Next, FACS’05 was organized as a standalone event in October 2004 at UNU-IIST. Then FACS’06 was hosted in Prague by Charles University.

Considering the persisting interest of the participants in the topics, FACS’07 was scheduled again as a separate event, this time to be hosted by INRIA in Sophia-Antipolis, France, in September 2007. The scientific program includes 15 regular papers and 2 invited talks, delivered by Corina Pasareanu from the NASA Ames Research Center - Moffett Field, USA and by Eugenio Zimeo from the University of Sannio - Benevento, Italy. 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.

The FACS’07 proceedings will be published in the series Electronic Notes in Computer Science. We would like to thank Michael Mislove, Managing Editor of the series, for his continuing support.

Without the support of INRIA Sophia-Antipolis 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 Antonio Causado, Patricia Maleyran, Monique Simonetti at INRIA Sophia-Antipolis, and to Kitty Chan Iok Sam at UNU-IIST Macau, for their help in managing all practical aspects of this event. A special thanks to Vladimir Mencl at university of Canterbury (New-Zealand) for his support during the whole process.

September 2007

Markus Lumpe & Eric Madelaine
Program Co-chairs
FACS’07
Program

Wednesday, september 19th

12:30-14:00 Lunch
14:00-14:15 Welcome
14:15-15:45 Session 1: Modular Reasoning
   Chair: TBA
   Slim Kallel, Anis Charfi, and Mohamed Jmaiel:
   Aspect-based Approach for Specifying and Enforcing Architecture Invariants
   Berhard Schätz: Modular Functional Specification of Reactive Components

15:45-16:15 Coffee break
16:15-17:45 Session 2: Verification Techniques & Behavioral Analysis
   Chair: TBA
   Javier Cubo, Gwen Salaïn, Carlos Canal, Ernesto Pimentel, and Pascal Poizat:
   A Model-Based Approach to the Verification and Adaptation of WF/.NET
   Mila Majster-Cederbaum, Moritz Martens, and Christoph Minnameier:
   Liveness in Interaction Systems

18:00 SC meeting

Thursday, september 20th

9:00-10:30 Session 3: Component Models
   Chair: TBA
   Vasu Alagar and Mubaral Mohammad:
   A Component Model for Trustworthy Real-Time Reactive Systems
   David Paulo Pereia and Ana C. V. de Melo:
   A CSP Architectural Model for Fault-tolerant Systems

10:30-11:00 Coffee break
11:00-12:30 Session 4: Substitutability
   Chair: TBA
   Meriem Belguidoum and Fabien Dagnat:
   Formalization of Component Substitutability
   Youcef Hammal: Substitutability Relations for Active Components

12:30-14:00 Lunch
14:00-15:00 Invited talk 1: Corina Pasareanu:
   Learning Based Assume Guarantee Reasoning

15:00-15:30 Coffee break
15:30-17:00 **Session 5: Component Composition & System Adaptation**  
Chair: TBA  
Jean-Baptiste Raclet: *Residual for Component Specifications*  
Javier Cámara, Gwen Salaün, and Carlos Canal:  
*Multiple Concern Adaptation for Run-time Composition in Context-Aware*

17:00-18:00 **Demonstrations**  
20:00 **Workshop dinner in Juan les Pins**

**Friday, September 21st**

9:00-10:00 **Invited talk 2: Eugenio Zimeo**  
*From Component-Based to Service-Oriented Computing: Towards Self-Evolution*

10:00-10:30 **Coffee break**

10:30-12:45 **Session 6: Tools & Frameworks**  
Chair: TBA  
Fabricio Fernandes and Jean-Claude Royer:  
*The STSLib Project: Towards a Formal Component Model Based on STS*  
Jasmin Christian Blanchette and Olaf Owe:  
*An Open System Operational Semantics for an Object-Oriented and*  
Jin-Hyun Kim, Jae-Hwan Sim, and Jin-Young Choi:  
*Resource-Oriented Design Framework for Embedded System Components*

12:45-14:00 **Lunch**

14:00-15:30 **Session 7: Component Interaction**  
Chair: TBA  
Muck van Weerdenburg: *Process Algebra with Local Communication*  
Tobias Blechmann and Christel Baier: *Checking Equivalence for Reo Networks*

15:30-16:30 **Discussion session with coffee/refreshments**

16:30 **Workshop ends**  
17:00 **Bus to the airport**
Program Committee

- Luis Barbosa, Universidade do Minho, Portugal
- Frank S. de Boer, CWI, The Netherlands
- Christiano Braga, Universidad Complutense de Madrid, Spain
- Carlos Canal, Universidad de Malaga, Spain
- Paolo Ciancarini, Universita di Bologna, Italy
- Jose Fiadeiro, University of Leicester, United Kingdom
- Rolf Hennicker, Ludwig-Maximilians-Universität München, Germany
- Atsushi Igarashi, Kyoto University, Japan
- Einar Broch Johnsen, Universitetet i Oslo, Norway
- Bengt Jonsson, Uppsala University, Sweden
- Mathai Joseph, Tata Consultancy Services Limited, India
- Markus Lumpe, Swinburne University of Technology, Australia
- Eric Madelaine, INRIA, Centre Sophia Antipolis, France
- Vladimir Mencl, University of Canterbury, New Zealand
- Corina Pasareanu, NASA Ames, USA
- Frantisek Plasil, Charles University, Czech Republic
- Ralf Reussner, Universität Karlsruhe, Germany
- Bernhard Schätz, Technical University of Munich, Germany
- Joseph Sifakis, VERIMAG, France
- Carolyn Talcott, SRI International, USA
- Dang Van Hung, IIST UNU, Macau

Organizing Committee

- Eric Madelaine, INRIA, Centre Sophia Antipolis, France
- Antonio Cansado, INRIA, Centre Sophia Antipolis, France
- Monique Simonetti, INRIA, Centre Sophia Antipolis, France
- Patricia Maleyran, INRIA, Centre Sophia Antipolis, France
- Vladimir Mencl, University of Canterbury, New Zealand
- Kitty Chan Iok Sam, UNU-IIST, Macau
Invited Talks

Corina Pasareanu, Robust Software Engineering group, NASA Ames, Research Center - Moffett Field, USA

Learning Based Assume Guarantee Reasoning

Abstract:
Assume-guarantee reasoning is a “divide and conquer” approach to the verification of large systems that makes use of “assumptions” about the environment of a system component. Coming up with appropriate assumptions used to be a difficult manual process. We will present techniques for performing assume-guarantee verification of software in a fully automated fashion. In the heart of these techniques lies a framework that uses an off-the-shelf learning algorithm for regular languages, namely L*, to compute the assumptions. We will also discuss the application of the techniques to a number of NASA case studies. This is joint work with Dimitra Giannakopoulou, Howard Barringer, Jamie Cobleigh, and Mihaela Gheorghiu.

Eugenio Zimeo, Research Center on Software Technology, University of Sannio - Benevento, Italy

From Component-Based to Service-Oriented Computing: Towards Self-Evolution

Abstract:
Service oriented architectures have gained increasing popularity as they significantly extend the potential of software components to implement global information systems where loosely coupled pieces of functionality are published, consumed, and combined with other functions over the Internet. This approach enables a two-level developing model, where functions are implemented by individual services and processes are built as compositions of services. The talk focuses on the design and enactment of processes that orchestrate services over the Web, reviewing basic technologies and presenting emerging approaches that enable self-evolution. This will be analyzed for different aspects of a web process, including control-flow and service binding, and using different architectural models, from centralized orchestration to fully distributed composition and enactment.
Using Aspects for Enforcing Formal Architectural Invariants

Slim Kallel$^1$
Software Technology Group
Darmstadt University of Technology
Darmstadt, Germany

Anis Charfi
SAP Research, CEC Darmstadt
Darmstadt, Germany

Mohamed Jmaiel
ReDCAD Laboratory
National Engineering School of Sfax
Sfax, Tunisia

Abstract

Formal methods such as Z and Petri nets can be used to specify invariants that should hold during the execution of component-based applications such as those regarding changes in the architecture of the application and valid sequences of architecture reconfigurations. Integrating logic for checking and enforcing these invariants into the application’s implementation is generally done by adding appropriate code to the functional application code. In this paper, we discuss several limitations of this approach that may ensue in a disconnection between the application implementation and its formal specification.

We propose an approach for specifying and enforcing architectural constraints, which combines formal methods and Aspect-Oriented Programming. We use the Z notation for describing the architectural invariants of the application and Petri nets for modeling coordination protocols. At the implementation level, aspects intercept architecture reconfiguration events and check according to the formal specification and the coordination protocol whether a reconfiguration action can be performed.

Keywords: Software architecture, Z notation, Petri nets, Aspect-Oriented Programming

1 Introduction

The software applications of today’s organizations consist generally of several distributed software components. Such applications are characterized by a dynamic architecture, which evolves over the time in order to respond to the user requirements. For instance, new components may be added and existing connections between the components may be modified during execution. When such reconfigurations are
done it is necessary to ensure that no faults are caused and that the software application works correctly.

To guarantee the reliability and consistency of the architectural evolution of distributed component-based applications, we propose using formal specifications. In this way, one could define the architectural constraints and coordination protocols that must be fulfilled by each reconfiguration of the application. Integrating logic for checking and enforcing architectural constraints and coordination protocols into the application’s implementation is generally done by adding appropriate control code to the functional application code, as shown in [33,26]. These approaches provide the necessary control functionality, but they exhibit several limitations, which may ensue in a disconnection between the application implementation and its formal specification.

First, the control code that implements the constraints is written manually in these approaches. Second, this code is not well-modularized as it is tangled with the functional code of the application and scattered across the implementation of different components, which makes it non reusable. Third, the code that implements the constraints may not be conform to the formal specification. This is accentuated especially by the scattering problem. Fourth, if the formal specification changes, it is necessary to change the code that implements the constraints manually.

To solve these problems, we propose a novel approach for the runtime verification of distributed applications, which combines formal methods and Aspect-Oriented Programming (AOP) [17]. This approach covers the static structural aspect (i.e., specification of components, connections, and constraints), the dynamic aspect (i.e., specification of the reconfiguration operations and their preconditions), and the coordination aspect (i.e., specification of the execution order of reconfiguration operations) of the software architecture. It fosters an organization of distributed component-based software systems in three phases: the formal specification phase, the base code implementation phase, and the aspect code implementation phase. In the first phase, the user specifies the constraints that should be fulfilled when the application evolves: architectural constraints are specified using the Z notation and coordination protocols are specified using Petri nets. In the base code implementation phase, the user writes the functional code of the different components (here we use Java). This code does not contain any control logic. In the aspect code implementation phase, we use AspectJ aspects, which intercept reconfiguration events and check according to the formal specification whether the reconfiguration events can be performed.

This approach yields several benefits. It enables a more reliable control of the architectural evolution of component-based applications as it is based on formal methods. Moreover, using an aspect-oriented module to control architecture reconfiguration operations, mismatches between the implementation of the application and its formal specification are unlikely. In addition, the control code of the component-based distributed application becomes more reusable as it is well-modularized in aspects.

The remainder of this paper is organized as follows. In Section 2, we introduce the Z specification language, Petri nets, and Aspect-Oriented Programming. In Section 3, we present our approach for controlling the architecture evolution of
component-based applications. Section 4 describes our case study collaborative authoring system. Section 5 reports on some related work and section 6 concludes this paper.

2 Background

In this section, we make use of two well-known formal methods, namely Z notation and Petri nets, for specifying respectively architectural constraints and coordination protocols. In addition, we use Aspect-Oriented Programming for modularizing the control and coordination code.

2.1 Z specification language

The Z notation, as presented in [30], is a formal specification language. Z defines a mathematical language, a schema language, and a refinement theory between abstract data types. The mathematical language is based on the set theory and on mathematical logic i.e., first order predicate logic. The schema language allows to describe the state of a system and the manners according to which this state can change. The refinement theory allows to develop a system by building an abstract model from a system design.

A Z specification can be defined as a collection of state schemes and operation schemes. The state schema \texttt{State} describes the system state and the invariant relationships, which are maintained when the system is updated. This schema consists of two parts: a declaration part and a predicate part.

The operation schemes \texttt{Operation} define the possible operations in the system, the relationship between their inputs and outputs, and the state changes resulting from their execution. The operation schema comprises the state \texttt{State} before and the state \texttt{State'} after performing the operation. These two states are represented in the schema language by the \(\Delta \texttt{State}\).

2.2 Petri nets

Petri nets [22] are a graphical and mathematical tool to model and analyze discrete systems. In Petri nets, the different states of a system are modeled by \textit{places} and \textit{tokens}. The events are represented by \textit{transitions} between places. Formally, a Petri net can be defined as a 5-tuple \(< P, T, F, W, M_0 >\), where: \(P = \{p_1, \ldots, p_m\}\) is a finite set of places; \(T = \{t_1, \ldots, t_n\}\) is a finite set of transitions with \((P \cap T) = \emptyset\) and \((P \cup T) \neq \emptyset\); \(F \subseteq (P \times T) \cup (T \times P)\) is a set of arcs; \(W : F \to \mathbb{N}_1\) is a weight function and \(M_0 : P \to \mathbb{N}\) is an initial marking where for each place \(p \in P\) there are \(n \in \mathbb{N}\) tokens.

The system behavior can be described in terms of the system state and its changes. To simulate the dynamic behavior of the system, the state or marking will be changed according to the following rules:
- A transition $t$ is enabled, if each input place $p_i$ is marked with at least $W(p_i, t)$ tokens.

$$\forall p_i \in \bullet t, M(p_i) \geq W(p_i, t).$$ \hspace{1cm} [R1]

- If a transition $t$ is enabled for the marking $M$ then the enabling of $t$ will lead to the new marking $M'$:

$$\forall p_i \in \bullet t, M'(p_i) = M(p_i) - W(p_i, t) + W(t, p_i)$$ \hspace{1cm} [R2]

where $W(P_i, t)$ is the weight of the arc $(P_i, t)$; $W(t, P_i)$ is the weight of the arc $(t, P_i)$ and $\bullet t$ is the set of input places of the transition $t$.

2.3 Aspect-Oriented Programming

Aspect-Oriented Programming [17] is a programming paradigm that allows the modularization of concerns that cut across the implementation of a software application, such as logging, persistence, and security.

According the separation of concerns principle, AOP provides language means to separate the code implementing a crosscutting concern from the functional code of a software application. Using AOP, an application consists of two parts: The base program, which implements the core functionalities, and the aspects, which implement the crosscutting concerns. Aspects are new units of modularity, which aim at modularizing crosscutting concerns in complex systems by using join points, pointcuts, and advices.

Join points are well-defined points in the execution of a program. In AspectJ [16], which is an aspect-oriented extension to Java, join points correspond to method calls, constructor calls, field read/write, etc. The pointcut allows to select a set of join points, where some crosscutting functionality should be executed.

The advice is a piece of code implementing a crosscutting functionality, which can be associated with a pointcut. The advice is executed whenever a join point in the set identified by the pointcut is reached. It may be executed before, after, or instead of the join point at hand; this corresponds respectively to the advice types before, after and around in AspectJ. With an around advice, the aspect can control the execution of the original join point: it can integrate the further execution of the intercepted join point in the middle of some other code (using proceed).

3 Proposed Approach

We propose a centralized approach for controlling the software architecture evolution of component based applications, which combines formal methods and Aspect-Oriented Programming. Our approach consists to specify, validate and enforce architectural constraints in distributed applications at runtime. We distinguish three phases: formal specification, base code implementation, and aspect code implementation (Fig. 1).

The formal specification phase consists in specifying and validating the architectural style (i.e. definition of component and architectural constraints) using the Z notation and model the coordination protocol using Petri nets. The base code implementation phase consists in implementing the functional code of the application

---

2 The term control in our paper has no relation with the control theory.
using any Java-based component model. The aspect code implementation phase is the verification phase which allows the connection between the formal specification and the application.

In our approach, we can define constraints in the architectural style and coordination protocol without modifying any functional code of the application. To do that, we just model a new Z specification and a new Petri net. In this way, we can provide a better control and a strong reuse of code. Using aspects in our approach allows us to separate the control and coordination code from the functional code of the application, which reduces the complexity of distributed applications, and bridges the gap between the application implementation and its formal specification. Moreover, if the formal specification changes, a few well-defined modules need to be changed in a non-invasive way, namely the aspects.

### 3.1 Formal specification phase

Formal specification provides a very effective means for a precise and unambiguous description of a software architecture. This phase consists of three steps: The first is the formal specification of the system in terms of components, relations between them and the architectural constraints. The second step is the specification and validation of the different reconfiguration operations (i.e., adding, deleting, duplicating, connecting and disconnecting components). The third step consists in modeling the coordination protocol which describes the execution order of reconfiguration operations using Petri nets. In the two first steps, we follow the Z-based approach presented in [18], which covers both the static aspect (i.e., system specification) and dynamic aspect (i.e., specification of reconfiguration operations) of software architectures.

**System Specification**: The system specification consists in defining the types of components, the types of relations between components, and the architectural
properties specified in terms of first-order predicates. The system structure is specified using the following Z schema. \( C_i, R_{ij} \) define respectively the component and the relation between them, \( Cstr_i \) represents an architectural constraint. Each component defined in the system schema must be already specified using a Z schema and include the internal behavior \( bhr_i \) in terms of predicate logic.

\[
\begin{align*}
\text{Component}_i, \quad & \text{atts : Types, \ldots} \\
& bhr_1; \ldots; bhr_n \\
\text{System} \quad & \text{C}_i : \text{Component}_i \\
& C_j : F \setminus \text{seq Component}_j \\
& R_{ij} : \text{Component}_i \leftrightarrow \text{Component}_j \\
& Cstr_1; \ldots; Cstr_n
\end{align*}
\]

To verify that our system does not contain any contradictions between the defined constraints, we should verify the system consistency by ensuring that at least one valid state exists [32]. This verification is specified by the following consistency theorem. \( \text{SystemInit} \) corresponds to a Z schema which describes a valid state of the system. In order to validate and reason about the architectural style, we use the tool Z/EVES [20], which supports syntax and type checking as well as theorem proving.

\[
\begin{align*}
\text{SystemInit} \\
\text{System} \quad & C_i = \{\ldots\} \\
& R_{ij} = \{(\ldots, \ldots)\}
\end{align*}
\]

**Specification of reconfiguration operations:** This step consists in specifying the dynamic aspect of architectural styles which is defined through a set of architecture reconfiguration operations. The operations are defined as Z operation schemas and correspond to adding, deleting, duplicating, connecting, and disconnecting components. Each operation schema \( \text{Operation}_i \) defines the pre-conditions and post-conditions of that operation. These conditions are essential to control the architectural evolution of the application. The respective operation will be executed only if its pre-conditions are evaluated to true. We specify and prove also the pre-condition theorem \( \text{PreconditionTheorem} \). This theorem states the pre-conditions that must initially be satisfied to guarantee that the constraints are preserved after the execution of the operation.

\[
\begin{align*}
\text{Operation}_i \\
\Delta \text{System} \quad & C_i : \text{Component}_i \\
\text{pre\_Condition} \quad & \exists \text{System} \land C_i : \text{Component} \\
\text{post\_Condition} \quad & \text{pre\_Conditions} \land \text{pre\_Operation}_i
\end{align*}
\]

\( C_i \) represents the input component and \( \text{pre\_Condition} \) and \( \text{post\_Condition} \) define respectively the pre- and post-conditions (in terms of predicate logic) of the operation \( \text{Operation}_i \).
**Coordination Protocol Modeling:** The coordination protocol describes the dynamic evolution of the architecture by defining the execution order of reconfigurations. We propose to model the coordination protocol using Place/Transition Petri nets, which allow us to prove algebraic properties such as the existence of deadlocks and livelocks. Moreover, determining whether the Petri net is live and bounded, we can prove if the evolution of software architecture finishes correctly.

We model each reconfiguration operation by a transition (i.e. already specified by Z operation schema). The enabling of a transition means that the corresponding action is conform with the coordination constraints. Consequently, the transition can be carried out and a token can be put in the next place. We use the tool P3 [9], which allows the creation, the modeling of Petri nets, and their representation in XML.

### 3.2 Base code implementation phase

This phase consists in implementing the core functionality of the application without including any architecture verification code. The application can be implemented using any Java-based component model (e.g., EJB [31], CCM [23]). The structure of the application must be synchronized with the formal specification, i.e., it must comprise the components and their properties, the relations between them, as well as the implementation of the different reconfiguration operations.

### 3.3 Aspect code implementing phase

In order to verify and control at runtime the software architecture evolution of the component-based applications, we implement an aspect-based module. The aspect module code is separated from the functional application code in two parts: the **control module** contains aspects that check the conformity of each reconfiguration operation against the formal specification of the architectural style in Z, whereas the **coordination module** contains aspects that check and enforce the coordination protocols that are modeled using Petri nets.

**Control module:** This module interprets the architectural style specified in Z and extracts the architectural constraints in order to subsequently verify for each reconfiguration operation whether it can be performed or not. This module is implemented using several aspects. One aspect verifies the static constraints that are specified in the system schema, such as the properties of components. In addition, for each reconfiguration operation, there is an aspect that intercepts the execution of that operation and interprets the pre-conditions that are specified in the system schema and the respective operation schema.

We implemented an evaluator for Z, which evaluates the logic predicate constraints according to the system state in terms of components and their relations. The Z specifications are saved in \texttt{LaTeX} form\(^3\), which facilitates the extraction of the architectural constraints. We implemented an around advice to allow/disallow the reconfiguration operation. The advice works as follow: First, it executes the constraint evaluator with the \texttt{LaTeX} source as input parameter. Second, it interprets

\(^3\) Z/EVES v. 2.3 exports the specification in \texttt{LaTeX} file based on Z-eves.sty
the result using the keyword *proceed*. Then, it extracts the post-condition describing the system request and updates the system state. If the reconfiguration operation does not conform to the Z specification, the aspects prohibit the execution of the operation.

**Coordination module**: This module enforces execution orders of the architecture operations in distributed applications. Similarly to the control module, this module checks the conformity of each reconfiguration operation against the coordination protocol that is modeled as Petri nets using the tool P3, which saves in matrix form the Petri net definition and the current marking in an XML file.

In this module, an aspect checks if a reconfiguration operation can be carried out by verifying whether the corresponding transition is enabled (applying R1 section 2.2.). If that is the case, the aspect executes the reconfiguration operation, and after that updates the marking in the XML file (applying R2 section 2.2.).

4 Case Study

To illustrate our approach, we implemented two applications. The first is a collaborative authoring system based on a client/server style. This application controls and manages shared documents that are located on a server. The authors connect to the server in order to edit and update these documents simultaneously. The second application is a patient monitoring system specified according to the publish/subscribe style. This application allows the nurses in a hospital to control remotely their patients and request patient data by sending a request to an event service, which manages the communication between nurses and bed monitors.

In this section, we explain how the collaborative authoring application is built according to our three-phase approach. The authors can have two roles: the writer role can modify, create, and delete sections of a document, whereas the reviewer role can correct a section and add annotations to it. Problems such as overlaps between sections that are accessed by different actors cannot occur because appropriate constraints that hinder such problems are specified formally and enforced by appropriate control aspects.

4.1 Formal specification phase

**System Specification**: The collaborative authoring system consists of four components. The shared document *SharedDoc* is accessible to all actors, who are authorized either as Writer or as Reviewer. The shared document is defined as a sequence of sections *Section* (specified by the position of the first and last characters of it in the whole document), so that there is no overlap between sections. This constraint [C1] is specified in predicate form as shown below:

\[
\forall i: N \mid 1 \leq i < \#section \quad [C1]
\]

\[
\bullet (section(i + 1)).firstCharacter = (section(i)).lastCharacter + 1
\]
Our system specified in the schema below, consists of writers, reviewers, a shared document and relations between the authors and sections. This relation will be established only between the authors who belong to the system and the sections of the shared document. The conditions \([C2,C3]\) are preserved by verifying the domain \(\text{dom}\) and the range \(\text{ran}\) of each relation. We restrict the number of writers \([C4]\), and we require that a writer can connect only to one section at a given point of time \([C5]\). To ensure that two actors are never connected simultaneously to the same section, we specify the constraint \([C6]\) in the schema below.

\[
\begin{array}{l}
\text{CollaborativeAuthoringSystem} \\
\text{writers} : F \text{Writer} \\
\text{sharedDoc} : \text{SharedDoc} \\
\text{WriterSection} : \text{Writer} \leftrightarrow \text{Section} \\
\text{...}
\end{array}
\]

\[
\begin{array}{l}
\text{dom} \text{ReviewerSection} \subseteq \text{reviewers} \quad [C2] \\
\text{ran} \text{ReviewerSection} \subseteq \{s : \text{Section} \mid s \in \text{ran sharedDoc.section}\} \quad [C3] \\
\#\text{writers} < 5 \quad [C4] \\
\forall w : \text{writers} \land \#(\text{WriterSection}[\{w\}]) \leq 1 \quad [C5] \\
\forall r : \text{reviewers}; w : \text{writers}; s : \text{Section} \mid s \in \text{ran sharedDoc.section} \\
\quad \bullet (r, s) \notin \text{ReviewerSection} \lor (w, s) \notin \text{WriterSection} \quad [C6] \\
\text{...}
\end{array}
\]

In order to verify the consistency of the specified system, we define a valid system state \(\text{InitCASystem}\) and we specify and prove the theorem \(\text{ConsistencyCASystem}\). Our system specification is consistent and does not contain any conflict since the following theorem was proved by Z/EVES.

\[
\text{Theorem} \quad \text{ConsistencyCASystem} \\
\exists \text{CollaborativeAuthoringSystem} \bullet \text{InitCASystem}
\]

**Specification of reconfiguration operations:** In the following, we specify and validate all reconfiguration operations of our system (e.g., insert writer, connect writer, disconnect writer, delete section, ...). The operation schema \(\text{InsertWriter}\) specifies the addition of new writer without connecting the writer to any section of the shared document. The constraint \([C7]\) specifies that the new writer \(w^?\) should not be one of the writers that are already present in the system. We specify also the post-conditions \([C8,C9]\) of the operation in terms of components and their relations.

We proved the theorem \(\text{PreInsertWriter}\), which preserves the system properties while adding a new writer. This operation is conform to the system constraints described in the style schema \(\text{CollaborativeAuthoringSystem}\).
Modeling coordination protocols: In our collaborative authoring system, the writers can create, modify, and delete sections. Then, the reviewers can correct these sections and add annotations. To enforce the activity order specified above, we define a simple coordination protocol, which requires that each section must be created or modified by a writer before it becomes accessible to reviewers for correction. In addition, after a section is corrected, the next reviewer cannot revise it before an author modifies it. These constraints are expressed using the following Petri net.

Each connection and disconnection is modeled by a Petri net transition. At execution time, actions are only executed if their corresponding transitions are possible. In the initial state, there is a token in the places \( p7 \) and \( p8 \) and consequently the transition \( \text{ConnectWriter} \) is enabled. Thus, a writer can connect to the section. After the disconnection of the writer, no other writer can connect because there is no token in \( p8 \). However, a reviewer can connect because there is a token in \( p7 \) and \( p9 \).
4.2 Base code implementation phase

Our collaborative authoring application is implemented as a Client/Server application. The functional level comprises only code providing the core functionalities such as editing, i.e., access control to the shared documents and control of the architecture evolution is out of scope. For instance, in this phase two writers can modify simultaneously the same section of a document.

4.3 Aspect code implementation phase

Control module: The control module manages the evolution of the architecture since new users can connect and/or disconnect to the documents during the execution of the application. In the architectural style of our application, we specified a static constraint expressing that each client can modify one section only at a given point of time (constraint [C5] in the system schema). Therefore, an aspect is necessary to prohibit clients from locking more than one section simultaneously.

Moreover, the architectural style specification disallows overlapping between the sections. This requirement is enforced by an appropriate aspect, which checks if the section requested by an actor does not overlap with sections that are locked by other actors. These control operations are ensured by an aspect that interprets the architectural constraints specified in the style schema CollaborativeAuthoringSystem.

In addition, we have specified a dynamic constraint on the connection of a new writer. Before a new writer can connect, an aspect interprets the formal pre-conditions generated from the operation schema connectWriter and the style schema CollaborativeAuthoringSystem. This aspect defines a pointcut that selects all calls to the method ConnectWriter, which allows an actor to access the shared document as writer. The aspect uses an around advice to allow/disallow a new writer to connect.

```
public aspect SectionAccessControl {
    pointcut permitted(Server S): 
    call( public ∗ ConnectWriter(..)) & target(S)
    void around (Server S): permitted(S) {
        try{
            S.parsingXMLPetriNet();
            if (S.isTransitionEnabled(ConnectWriter)) {
                S.resultat = "Allow";
                ...
                proceed(S);
                Info.updateXMLPetriNet(ConnectWriter);
            } else {
                S.resultat = "Disallow";
                ...
            }
        } catch(Exception e){}
    }
}
```

Listing 1: Coordination Aspect Skeleton

Coordination module: In the collaborative authoring system, the coordination aspects enforce that actions that are executed on the shared document are according to the predefined coordination protocol (Fig. 3). Before each connection
or disconnection to a section, the coordination aspects insure that a writer cannot modify a section before a reviewer corrects it and that two reviewers cannot correct the same section simultaneously.

In the following, we present a skeleton of an aspect which checks whether a section can be executed using the corresponding transition in the Petri net. The aspect *SectionAccessControl* defines a pointcut, which selects calls to the method *ConnectWriter* (lines 2 and 3 in Listing 1). The around advice of this aspect checks according to the XML representation of the Petri net if the transition *ConnectWriter* is enabled (line 7). If that is the case, the actor gets access to the document, i.e., the operation is executed (using *proceed* as shown in line 10) and the advice saves the new marking describing the new system state. If not, the aspect prohibits the execution of the method *ConnectWriter* (lines 13-16).

5 Related Work

Several formal methods have been used for the specification of software architectures. We report in this section on the approaches that cover the static, the dynamic and the coordination aspects of software architecture.

Architecture Description Languages (ADLs) [19] provide means to describe the architecture of a software system. However, most ADLs are limited to the description of predefined dynamism [15]. They support only systems where the possible reconfigurations are at design time. In addition, with the exception of Wright, most ADLs do not have a formal basis. Many works introduce aspect-orientation to the specification level (e.g. ADL, UML [7], etc.). For instance, AO-ADL [27] is an aspect-oriented architecture description language, which models crosscutting concerns using components and provides a mechanism for defining aspect-oriented connector templates. In our current work, we are not interested in defining aspects at such a high level; we just use aspects to enforce formal architectural constraints at the implementation level.

We classify works on formal specification of software architectures according to the used techniques into three classes: based on logic, on graphs, and on process algebras. Some works used logic-based methods e.g., Temporal Logic [1] and Z notation [5]. The first-order logic covers only the static aspect of software architecture and pre- and post-conditions of architecture reconfiguration operations, whereas temporal logic can express at a very high level some coordination properties and temporal constrains. Other approaches use process algebra-based methods e.g., CSP [14] and the π-calculus [24] to model architectural dynamism with mobile processes. Other works are interested in graphs-based approaches to specify the static aspect of software architecture e.g., graph grammars [21] and graph Transformation [11]. These approaches do not cover the three aspects in software architecture (static, dynamic and coordination). In order to solve this problem other works propose multi-formalism approaches combining more than one formal language.

We are interested in works that combine the Z notation or Petri nets with others formal methods. ObjectZ ⁴ were combined with different process algebra Z/CCS [8],

---

⁴ Object-Z(OZ) is an object-oriented extension of the formal specification language Z.
Z/CSP [12], and OZ/CSP [29] and with temporal logic [4]. In these approaches, Z and ObjectZ are used to specify the architectural constraints (i.e. static aspect). However, they do not propose any formal solution to check if the architectural constraints specified in Z are preserved when the architecture evolves. In our approach, we can specify and prove the pre-condition theorem by using the Z/EVES theorem prover. Petri nets and colored Petri nets\(^5\) were combined with different type of temporal logic [10,28] in order to define the architecture constraints and architecture evolution using temporal constraints and specify coordination constraints using Petri nets. However, Z, which is based on predicate logic and set theory allows a low-level description of architectural invariants. To support the temporal properties in our approach, we plan to combine linear temporal logic and Z notation.

In the following, we focus on approaches to the enforcement of architectural constraints. Yan et al. [33] propose an approach to discover the architecture of a system using dynamic analysis. This approach provides a tool called DiscoTect based on online monitors, which are used for system observation in order to describe inconsistencies between implementation and architecture. ArchJava [2] is an extension to Java that seamlessly unifies software architecture with implementation, ensuring that the implementation conforms to the architectural constraints. It extends a practical implementation language to incorporate architectural features and enforce communication integrity. This approach allows to enforce architecture constraints but it focuses only on communication integrity. It also does not support other types of architectural reasoning, such as reasoning about coordination protocols and architectural styles. SonarJ [13] is a commercial Eclipse plug-in\(^6\) allowing the enforcement of architectural constraints in Java programs. The tools mentioned in this paragraph allow an efficient enforcement of architectural constraints but do not have any formal basis. In addition, crosscutting concerns are not supported by these tools.

6 Conclusion

The main contribution of this paper is the control of software architecture evolution in component-based applications in modular way using aspects. Our approach combines Aspect-Oriented Programming and formal methods and enables a reliable and modular verification. The reliability of our approach is ensured by the formal specification and validation of architectural constraints using Z and Petri nets. The use of an aspect-based control module in our approach, improves the modularity and the reusability of control code, as this code is well-modularized using aspects and separated from business logic.

As future work, we will study expressive pointcut languages such as [3,6,25], which allow the expression of temporal relationships in the pointcut to, e.g., express that a certain operation must be called before another. We will also investigate whether and to what extent the usage of such pointcut languages would replace the usage of Petri nets in our approach. We will also target automatic generation of control and coordination aspects.

\(^5\) Colored Petri nets are a high-level extension of petri nets
\(^6\) http://www.eclipse.org
S. Kallel

Acknowledgments

We would like to thank Mira Mezini for several helpful discussions and comments on earlier drafts of this paper.

References


Modular Functional Descriptions

Bernhard Schätz

Technische Universität München, Fakultät für Informatik, Boltzmannstr. 3, D-85748 Garching bei München, Germany
schaetz@in.tum.de

Abstract
The construction of reactive systems often requires the combination of different individual functionalities, thus leading to a complex overall behavior. To achieve an efficient construction of reliable systems, a structured approach to the definition of the behavior is needed. Here, functional modularization supports a separation of the overall functionality into individual functions as well as their combination to construct the intended behavior, by using functional modules as basic paradigm together with conjunctive and disjunctive modular composition.

Keywords: Function, behavior, module, composition, decomposition, refinement, verification

1 Introduction

In many application domains reactive systems are becoming increasingly complex to cope with the technical possibilities and requested functionalities. The behavior provided by the system often is a combination of different functions integrated in an overall functionality; e.g., an embedded controller managing the movement of a car power window combines control of the basic movement, position control to restrict motor overload, as well as power management to avoid battery wear.

Implementing those combinations of individual functions is a complex and error-prone task. Since these functions in general influence each other, a modular development process ensures that the combined functionality respects the restrictions imposed by each individual function. Furthermore, due to the increased demand for possible variants of behavior, in general the development of reactive systems requires the recombination, restriction and extension of functionalities.

Here, the use of functional modules can improve the development process by supporting the modular definition of the basic functions as well as their combination into the overall functionality.

1.1 Contributions

Modular functional development aims at supporting the development process of multi-functional reactive systems by use of modular composition of functions. To

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

- introduce functions as modular units of system construction, which provide a data flow interface describing the values observed and controlled by a function as well as a control interface describing the activation and deactivation of functions.
- use disjunctive and conjunctive composition as a means of combination, which allow to either alternatively or simultaneously combine functional behavior.
- provide automatic proof support to check the refinement between more descriptive and more constructive variants of specifications of functionality.

1.2 Related Approaches

Functions are modules of behavior, used for the construction of complex behavior from basic functionality. They offer interfaces for both data and control flow in a similar fashion to the ports and connectors introduced in [6].

As generally used, e.g., in embedded systems, functions are intended for the description of signal-based reactive systems, using asynchronous communication unlike [5] or [10]. They use a communication paradigm similar to [9], [4], or [3]. Therefore, they provide similar forms of conjunctive and disjunctive compositions as provided for the modules introduced in [4] or the states introduced in [2]. However, while those are targeting the specification of reactive behavior in a constructive fashion, here a more descriptive from is used, using a more generalized form of (conjunctive) composition. In contrast to these constructive approaches, ruling out the introduction of partial behavior either syntactically by restricting compatible alphabets (e.g., [4]) or semantically using interleaving of interactions instead of synchronization (e.g., [2], functions with their less restricted composition allow a more natural modular form of specification.

Due to this form of composition, they are similar to services-oriented descriptions as used in [1]. In contrast to those rather descriptive approaches with a large number of different composition operators, however, functions provide a more constructive form of decomposition. Similar to [8], only conjunction and disjunction are used and a similar semantical models is used; but while there – due to its rather general form of specification without distinction between control and data flow – a more low-level formal form of description is needed, here by means of the explicit introduction ports and locations a more compact form of specification is possible.

Finally, the approach introduced here extends [11] by focusing on refinement rather than refactoring, and providing automated proof support by use of model-checking.

2 Describing Functions

Functions form the building block of the approach presented here. Basically, functions are capsules of behavior, defined by their (external) interface in terms of data and control flow as well as their (internal) implementation. The data flow between the function and its environment is described in form of data signals exchanged between them, allowing the function to observe and control shared signals. The control flow between the function and its environment is described in form of control loca-
Figure 1 shows a function *Power Position Controlled Window* describing the functionality of a power window controller. The capsulated behavior is represented by a box, and identified by a function name (*Power Window*). Interface elements (e.g., *Bat, stp*) are attached to its border; its internal structure is depicted inside the box.

The function observes user interactions via the *But* signal (with *Up, Hd, and Dn* signaling the up, hold, and down position of the switch), the current battery status via the *Bat* signal (with *Hi and Lo* signaling high and low voltage), and the current position of the window via the *Pos* signal (with *On and Of* signaling intermediate or end positions of the window). It controls the motor of the window via the *Mot* signal (with *Hi, Lo, and Zr* signaling upward, downward, or no movement). As shown in Figure 1, the input and output signals accessed by a function are indicated by empty and filled boxes at the border of the function.

To control the activation and deactivation of the function, it can be entered and exited via the control location *stp*. As shown in Figure 1, while interface locations as well as internal control locations are indicated by outlined circles.

In its elementary form, the behavior of the function is described in a state-transition manner. As shown in Figure 1 in case of *Power Position Controlled Window*, its internal control flow is described via locations *dn, stp, and up* (with *stp* being an internal control location as well as an interface location, indicated by the grayed-out line), as well as transitions between these locations. Transitions are influenced by observed signals and influence controlled signals. Furthermore, transitions might be influenced by values of local variables and influence local variables. Thus, if control resides in location *stp*, value *Hi* is received via the *Bat* signal (*Bat?Hi*), and a value *Up* is received via the *But* signal (*But?Up*), then value *Hi* is sent via the *Mot* signal (*Mot!Hi*) and control is transferred to location *up*.

As shown in Section 3, a single transition can be understood as the most basic form of a function. Its interface is defined by the observed and controlled signals (and variables) as well as by its start and end locations.

### 2.1 Decomposing Functionality

The functionality of *Power Position Controlled Window* shown in Figure 1 can be decomposed in simpler functionalities, addressing special aspects of the combined
behavior: *power control*, addressing issues of voltage-dependent window movement; *position control*, addressing issues of end position detection; and *basic window control*, addressing issues of button-controlled window movement.

As shown by the white boxes in Figure 3, to obtain an equivalent functional decomposition *Power Position Window* of function *Power Position Controlled Window*, five basic functions are used:

(i) **Basic Window** movement control, moving the window as requested by the interactions of the user,
(ii) **Position Check**, restricting window movement to positions in between end positions,
(iii) **Position Override**, halting the window if reaching an end position,
(iv) **Power Check**, restricting window movement to situations with sufficient initial voltage,
(v) **Power Override**, disabling window movement if lacking sufficient voltage.

Obviously, all five functions control the motor movement via the Mot signal, interacting to realize the overall behavior. However, their combined behavior does not support a modularization of the behavior of the controller, failing to reflect the separation of concerns into individual functions. Therefore, if restricted to constructive formalisms (like Statecharts [2] or Masaccio [4]), identifying these five functions in a modular fashion is not possible; as a result, ensuring that the overall behavior implements the intended interaction of these functions is requires the use of an additional property-language (e.g., temporal logic).

Figures 2, 4, and 5 show the corresponding basic functionalities.

Function **Basic Window** provides basic window movement functionality, in form of upward movement caused by a Hi-value for the Mot-signal initiated by a Up-value for the But-signal in location stp; holding (But?Up) or relasing (But?Hd) the button continues the upward movement (Mot!Hi), while changing the button (But?Dn) will stop the movement (Mot!Zr). The functionality for the downward movement is supplied in a similar fashion.

The function is activated and deactivated via interface location stp – corresponding to the internal control location representing a stopped motor – or via interface location mv – corresponding to two internal control locations representing either
downward or upward movement of the motor – shown at the interface of Basic Window.

Position control as shown in Figure 4, consists of Position Check, ensuring that the motor is restricted to intermediate positions, and Position Override, ensuring that the motor is stopped if an end-position is reached. Once activated by a motor movement (Mot!{Hi,Lo}), Position Check enforces that further movement requires a non-end position (Pos?Of:Mot!{Hi,Lo}) until deactivation (Pos?Of:Mot!Zr). Interface locations stp and mv correspond to a stopped a moving motor. Position Override provides an override functionality to stop a window movement when an end-position signal is detected.

Similarly, power control consists of Power Check, ensuring that starting the motor movement requires sufficient voltage, and Power Override, ensuring that the motor is not activated in case of insufficient voltage.

2.2 Composing Functionality

To obtain the overall behavior of the controller of the power window, the functions introduced in Section 2.1 are combined. Figure 3 also shows, how Power Position Window is composed to obtain an equivalent functionality like Power Position Controlled Window: Basic Window and Position Check are combined by conjunctive composition – indicated by × – to function Position Controlled, which in turn is combined by disjunctive composition with function Position Override to function Position Window. Using the same pattern of composition, Position Window is combined by conjunctive composition with Power Check to obtain function Power Position Controlled, which in turn is combined by conjunctive composition with Power
Override to function Power Position Window.

Intuitively, disjunctive composition corresponds to the alternative use of composed functions, while conjunctive composition corresponds to the simultaneous use of the composed functions. Obviously, disjunctive composition is not sufficient to obtained the intended functionality, since position control and power control are supposed to restrict basic window movement. Similarly, simple conjunctive composition does not lead to a reasonable behavior, since basic movement, position control, and power control as defined above are in conflict to each other. Therefore, a more sophisticated form of combination is needed, describing the priorities between these (sub-)functions.

Figures 6 and 7 describe these prioritized compositions. As shown in the left-hand side of Figure 6, at the top level, the Power Position Window is realized by disjunctive composition – indicated by a light background used inside the box representing a function – of the Power Override function together with the Controlled Position Window, ensuring that a lack of voltage does result in a blocked window movement. Activation and deactivation of the disjunctive composition Power Position Window via the stp interface location corresponds to the activation and deactivation of either sub-function via stp. Furthermore, as Power Override and Power Position Controlled share the interface location stp, activation may pass from one to other and back. As the signal interfaces (Bat, But, Pos, and Mot) are linked to the corresponding interfaces of the sub-functions, signals are passed between the environment of Power Position Window and the currently activated sub-function.

As Controlled Position Window is obtained by conjunctive composition – indicated by a dark background used inside the box representing a function – of Power Check and Position Window, any window movement is only initialized in case of sufficient voltage. Activation and deactivation of the conjunctive composition Power Position Controlled via interface location stp corresponds to the simultaneous activation and
deactivation of both sub-functions via $stp$. Furthermore, the signal interfaces (Bat, But, Pos, and Mot) of Power Position Controlled Power Check and Position Window are simultaneously observed and controlled by the sub-functions via their corresponding linked signal interfaces.

A similar construction is applied to ensure position control. As Position Window is realized by disjunctive composition of Position Override and Controlled Window, detection of an end position stops the movement of the window. By conjunctive composition of Position Check and Basic Window to form Controlled Window, the basic window movement is restricted to intermediate positions of the window.

3 Modeling Functions

In this section, functions are introduced as building blocks for the construction of reactive behavior. Since functions are a generalization of components, the difference between functions and components from a methodical perspective is discussed, before giving a formal and compositional definition of functions based on [4].

3.1 Components and Functions

A component communicates with its environment via its interface. A component has a completely specified behavior: for each behavior of the environment (in form of a history of input messages received by the component) its reaction (in terms of histories of output messages) is defined. In approaches like [9], [4], or [12] this is defined as input enabledness, input permissiveness, or input completeness. As introduced in [12], in contrast to a component, a function behavior needs not be totally defined. For a partial specification, it is possible to have a behavior of the
environment where no behavior of the function is defined by the specification.

This distinction plays an important role when combining components or functions. Generally, syntactic restrictions (e.g., disjointness of output interfaces and data states), ensure that the composition of components results in a component (with input total behavior); e.g., [4] uses such a restriction. Due to their more general nature, such a restriction is not required for functions [12]. However, as a result, the combinations of functions (e.g., manual window control, position control) may lead to conflicts (e.g., upward movement of window by manual control vs. stop of movement by position control) resulting in undefined behavior.

To define a formal framework for the construction of functions, in the following subsection we introduce a basic model, and then supply some operators for the construction of complex functions from basic ones.

### 3.2 Semantics: State-Based Functions

Since functions are intended for the modular specification of components with input complete behavior, as semantical basis in the following we use a formalization similar to [4] to introduce a set \( \text{Fun} \) of functional descriptions as well as its interpretation; however in contrast to the former, we generalize it to support the description of functions with their partially defined behavior, especially allowing the introduction of new partially by simultaneous combination as defined in Subsection 3.2.4. In the following, \( \text{Fun} \) corresponds to the set of function terms, starting from basic functions and using operators to form more complex descriptions.

#### 3.2.1 Basics

The structural aspects of a function are defined by its input ports \( \text{In} \), its output ports \( \text{Out} \) – with \( \text{In} \cap \text{Out} = \emptyset \) –, its variables \( \text{Var} \) – with \( \text{In} \cup \text{Out} \subseteq \text{Var} \) as special monitored and controlled variables – as well as its control locations \( \text{Loc} \). To describe the behavior of a function, we use the concepts

**State:** A state \( s \in \overrightarrow{\text{Var}} = \text{Var} \to \text{Val} \) maps variables to their current values.

**Observation:** An observation is either a triple \((a, t, b)\) consisting of a finite sequence \(t\) of states corresponding to an execution starting at location \(a\) and ending at location \(b\), changing variables according to \(t\); or it is a pair \((a, t)\) consisting of a finite sequence \(t\) of states, corresponding to a partial execution, starting at location \(a\). Since in the following only continuous functions are introduced, a restriction to finite observations is sufficient.

**Behavior:** The behavior of a function is the set \( \text{Obs} \) of all its observations. Consequently, \( \text{Obs} \) is prefix-closed, i.e., \((a, t, b) \in \text{Obs} \) implies \((a, t) \in \text{Obs} \), and \((a, t) \in \text{Obs} \) implies \((a, s) \in \text{Obs} \) for any prefix \(s\) of \(t\).

For a state \( s : \text{Var} \to \text{Val} \) with \( \text{Var}' \subseteq \text{Var} \) we use notation \( s \upharpoonright \text{Var}' \) for restrictions \((s \upharpoonright \text{Var}')(v) = s(v)\) for all \(v \in \text{Var}'\). This restriction is extended to sequences of states through point-wise application. For sequences \(r\) and \(t\) we use the notation \(r \circ t\) to describe the concatenation of \(r\) and \(t\); furthermore, \(\langle \rangle\) describes the empty sequence.
3.2.2 Basic Functions

The most basic function performs only one step of computation. When entered through its entry location, it reads the currently available input; it produces some output, depending on its current variable state and the available input, and changes its variable state; it then terminates by exiting via its exit location. To describe a basic function, we use the notation described in [6]. Figure 4 shows such a basic function Position Override with input port Pos and output port Mot, entry location mv, and exit location stp. Its behavior is described by a labeled transition from mv to stp with a structured label Pos?On : Mot!Zr. The first part of the label, its pre-part, states that whenever signal On is received via port Pos, then the transition is enabled. The second part of the label, its post-part, states that, whenever the transition is triggered, in the next state signal Zr is sent via output port Mot. These parts correspond to terms But = Stp and Mot = Zr using variables ‘v with v ∈ Var for values of v prior to execution of the transition, and variables v’ with v ∈ Var for values of v after its execution. The interface of Position Override is defined by In = {But}, Out = {Mot}, its variables by Var = In ∪ Out, and its locations by Loc = {mv, stp}. Abstracting from a concrete graphical representation, a basic function is described as the structure (a, pre, post, b) with entry location a, exit location b, pre-condition pre over Var, and post-condition post over Var × Var, pre and post are obtained from the corresponding terms by interpretation over Var and Var’, resp. Its behavior is the set containing all elements

• (a, before ◦ after, b)
• (a, before ◦ after)
• (a, before)
• (a, ⟨⟩)

with pre(before) ∧ post(before, after). Consequently, the behavior of Position Override is the set consisting of all observations (mv, before ◦ after, stp), (mv, before ◦ after), (mv, before), and (mv, ⟨⟩), such that before(But) = Stp as well as after(Mot) = Zr.

3.2.3 Alternative Combination

Similar, e.g., to Or-combination used in Statecharts [2], we use alternative combination to describe sequential behavior. The behavior of an alternative combination of two functions corresponds to the behavior of either function. Function Position Window in the left-hand side of Figure 7 shows the alternative combination of functions Position Override and Position Controled. It shares all the structural aspects of either function, and thus uses input ports But and Pos, as well as output port Mot. Furthermore, by means of the common interface location stp, either Position Override or Position Controled can be activated and deactivated. Furthermore, by means of the shared internal control location mv, activation may be passed from Position Controled to Position Override. Formally, the alternative combination of two functions A and B results in a function described by A + B that

• uses the input and output ports as well as variables of either function: In(A+B) = In(A) ∪ In(B), Out(A+B) = Out(A) ∪ Out(B), Var(A+B) = Var(A) ∪ Var(B)
• accesses their control locations: Loc(A+B) = Loc(A) ∪ Loc(B)
exhibits the behavior of either function: \((a, t, b) \in \text{Obs}(A+B)\) if \((a, t \uparrow \text{Var}(A), b) \in \text{Obs}(A)\) or \((a, t \uparrow \text{Var}(B), b) \in \text{Obs}(B)\); \((a, t) \in \text{Obs}(A+B)\) if \((a, t \uparrow \text{Var}(A)) \in \text{Obs}(A)\) or \((a, t \uparrow \text{Var}(B)) \in \text{Obs}(B)\)

Intuitively, the combined function offers observations that can be entered and exit via one of its sub-functions. If the sub-functions share a common entry location, observations of either function starting at that entry location are possible; similarly, if they share a common exit location, observations ending at that common exit location are possible. To ensure a well-defined function, we require that for two functions \(A\) and \(B\) conditions \(\text{In}(A) \cap \text{Out}(B) = \emptyset\) and \(\text{In}(B) \cap \text{Out}(A) = \emptyset\) must hold to be alternatively composable. Obviously, functions \(A + B\) and \(B + A\) and \(A + A\) are equivalent in the sense of having the same interface and behavior.

### 3.2.4 Simultaneous Composition

Besides alternative combination, functions can be combined using simultaneous combination. The behavior of a simultaneous combination of two functions corresponds to the joint behavior of both functions.\(^1\) Function \textit{Position Controlled} in the right-hand side of Figure 7 shows the simultaneous combination of functions \textit{Position Check} and \textit{Basic Window}. Its interface consists of input ports \(\text{In} = \{\text{But}\}\) of \textit{Position Check} and \(\text{In} = \{\text{Pos}\}\) of \textit{Basic Window} as well as output port \(\text{Out} = \{\text{Mot}\}\) of both sub-functions; its locations \(\text{Loc} = \{\text{stp, mv}\}\) are the shared locations of these functions. Formally, the simultaneous combination of two functions \(A\) and \(B\) results in a function described by \(A \times B\) that

- use the input and output ports as well as variables of each function: \(\text{In}(A \times B) = \text{In}(A) \cup \text{In}(B) \setminus \text{Out}(A \times B), \text{Out}(A \times B) = \text{Out}(A) \cup \text{Out}(B), \text{Var}(A \times B) = \text{Var}(A) \cup \text{Var}(B)\)
- accesses their \textit{shared} control locations: \(\text{Loc}(A \times B) = \text{Loc}(A) = \text{Loc}(B)\)
- exhibits the combined behavior of each function: \((a, t, b) \in \text{Obs}(A \times B)\) if \((a, t \uparrow \text{Var}(A), b) \in \text{Obs}(A)\) and \((a, t \uparrow \text{Var}(B), b) \in \text{Obs}(B)\); \((a, t) \in \text{Obs}(A \times B)\) if \((a, t \uparrow \text{Var}(A)) \in \text{Obs}(A)\) and \((a, t \uparrow \text{Var}(B)) \in \text{Obs}(B)\)

Intuitively, the combined functions offers observations that can be offered by both functions. To ensure a well-defined function, we require condition \(\text{Loc}(A) = \text{Loc}(B)\) for functions \(A\) and \(B\) to be simultaneously composable. Note that unless we require the standard interface constraint \((\text{Var}(A) \setminus \text{In}(A)) \cap (\text{Var}(B) \setminus \text{In}(B)) = \emptyset\) imposed for the composition of components, simultaneous combination of functions may result in output or variable conflicts, leading to the introduction of (additional) partiality in the behavior of the combined functions. Obviously, \(A \times B\) and \(B \times A\) are equivalent in the sense of exhibiting the same interface and behavior.

### 3.2.5 Hiding Locations

Hiding a location of a function renders the location inaccessible from the outside. At the same time, when reaching a hidden location the function does immediately continue its execution along an enabled transition linked to the hidden location. In Function \textit{Position Window} in the left-hand side of Figure 7, control location \textit{mv} is

---

\(^1\) Note that this differs essentially from \textit{And}-composition in Statecharts describing interleaved composition.
hidden to enable immediate position override. Formally, by hiding a location \( l \) from a function \( A \) we obtain a function described by \( A \setminus l \) that

- uses the input and output ports and variables of \( A \): \( \text{In}(A \setminus l) = \text{In}(A), \text{Out}(A \setminus l) = \text{Out}(A), \text{Var}(A \setminus l) = \text{Var}(A) \)

- accesses the control locations of \( A \) excluding \( l \): \( \text{Loc}(A \setminus l) = \text{Loc}(A) \setminus \{l\} \)

- exhibits the behavior of \( A \) if entered/exited through locations excluding \( l \) and continuing execution at \( l \): \( (a, t_1 \circ s_1 \ldots t_{n-1} \circ s_{n-1}, b) \in \text{Obs}(A \setminus l) \) if \( (a, t_1 \circ s_1, l), (l, s_{n-1} \circ t_{n-1}, b) \in \text{Obs}(A) \) as well as \( (l, s_{i-1} \circ t_i \circ s_i, l) \in \text{Obs}(A) \) for \( i = 2, \ldots, n-1 \), \( t_j \in S^* \), and \( s_j \in S; (a, t_1 \circ t_2 \circ \ldots) \in \text{Obs}(A \setminus l) \) if \( (a, t_1, l) \in \text{Obs}(A) \) and \( (l, t_i, l) \in \text{Obs}(A) \) for \( i > 1 \).

Obviously, \((S \setminus a) \setminus b\) and \((S \setminus b) \setminus a\) are equivalent in the sense of exhibiting the same interface and behavior. We write \( A \setminus \{a, b\} \) for \((A \setminus a) \setminus b\).

### 3.2.6 Renaming Locations

Renaming a location of a function changes the interface of the function, possibly unifying control locations. As, e.g., shown in the left-hand side of Figure 2, the distinct control locations corresponding to the upward and downward movement of the window are renamed to the unique control location \( \text{mv} \).

Formally, by renaming a location \( l \) in a function \( A \) to location \( m \) we obtain a function described by \( A[l/m] \) that

- uses the input and output ports and variables of \( A \): \( \text{In}(A[l/m]) = \text{In}(A), \text{Out}(A[l/m]) = \text{Out}(A), \text{Var}(A[l/m]) = \text{Var}(A) \)

- accesses the control locations of \( A \) excluding \( l \) and including \( m \): \( \text{Loc}(A[l/m]) = \text{Loc}(A) \setminus \{l\} \cup \{m\} \)

- exhibits the behavior of \( A \) after renaming: for \( a, b \neq l \), \( (a, t, b) \in \text{Obs}(A[l/m]) \) if \( (a, t, b) \in \text{Obs}(A) \) as well as \( (a, t) \in \text{Obs}(A[l/m]) \) if \( (a, t) \in \text{Obs}(A) \). Furthermore for \( a, b \neq l \), \( (a, t, m) \in \text{Obs}(A[l/m]) \) if \( (a, t, l) \in \text{Obs}(A) \) and \( (m, t, b) \in \text{Obs}(A[l/m]) \) if \( (l, t, b) \in \text{Obs}(A), (m, t, m) \in \text{Obs}(A[l/m]) \) if \( (l, t, t) \in \text{Obs}(A) \), and finally \((m, t) \in \text{Obs}(A[l/m]) \) if \((l, t) \in \text{Obs}(A) \).

### 3.2.7 Hiding Variables

Hiding a variable of a function renders the variable unaccessible from the outside. Formally, by hiding a variable \( v \) from a function \( A \) we obtain a function described by \( A \setminus v \) that

- uses the input and output ports and variables of \( A \) excluding \( v \): \( \text{In}(A \setminus v) = \text{In}(A) \setminus \{v\}, \text{Out}(A \setminus v) = \text{Out}(A) \setminus \{v\}, \text{Var}(A \setminus v) = \text{Var}(A) \setminus \{v\} \)

- accesses the control locations of \( A \): \( \text{Loc}(A \setminus v) = \text{Loc}(A) \)

- exhibits the behavior of \( A \) for arbitrary \( v \): \( (a, t \uparrow \text{Var}(A), b) \in \text{Obs}(A \setminus v) \) if \( (a, t, b) \in \text{Obs}(A) \); \( (a, t \uparrow \text{Var}(A)) \in \text{Obs}(A \setminus l) \) if \( (a, t) \in \text{Obs}(A) \).

Obviously, \((S \setminus v) \setminus w\) and \((S \setminus w) \setminus v\) are equivalent in the sense of exhibiting the same interface and behavior. We write \( A \setminus \{v, w\} \) for \( (A \setminus v) \setminus w\).
4 Applying Functions

As introduced in the previous sections, functions are intended to support the modular construction of complex functionalities in the development process by combining individual pieces of reactive behavior. However, while the descriptive form of general functional descriptions eases the combination and reuse of functions and the reasoning about the overall functionality, for the final implementation of the intended behavior in general more constructive forms of descriptions are used, as provided, e.g., by corresponding tools like or [2], [3], or [6]. As stated in Section 1, these descriptions correspond to a restricted form of functions, avoiding the introduction of partiality and ensuring input enabledness. On this constructive level, input enabledness is either established implicitly by completion (as, e.g., in [6]) or explicitly by analysis (as, e.g., in [3]).

However, to integrate these different applications of functional descriptions are integrated in a function-based development process, the more descriptive and more constructive forms must linked by an implementation relation, introduced in the following.

4.1 Implementation

While using the general descriptive form supports a structured development of the overall functionality, a more compact and constructive variant is generally more preferable for its effective implementation.

As shown in the examples, the definition of Power Position Window from the basic functions (Basic Window, Position Check, Position Override, Power Check, Power Override) leads to a more structured description. In contrast, the definition of Power Position Controlled Window is more suited for implementation using state-of-the-art tools. Thus, in a function-based development process, the former should be used in the early stages of defining the function under development, while the latter should be used in the latter stages. However, for a sound and integrated development process, it is furthermore necessary to establish an implementation relation between those functions.

Formally, a function $F_1$ is said to implement a function $F_2$ iff

- they provide the same closed signal interface: $\text{In}(F_1) = \text{In} = \text{In}(F_2)$ and $\text{Out}(F_1) = \text{Out} = \text{Out}(F_2)$ and $\text{Var}(F_1) = \text{In} \cup \text{Out} = \text{Var}(F_2)$
- they provide the same control interface: $\text{Loc}(F_1) = \text{Loc}(F_2)$
- every possible observation of $F_1$ is also a possible observation of $F_2$: $\text{Obs}(F_1) \subseteq \text{Obs}(F_2)$

Basically, functional refinement corresponds to standard trace inclusion. Since here continuous reactive systems are considered with simultaneous input/output actions using a signal-based communication with input enabledness, partial execution traces provide a suitable semantical basis. Obviously, this notion of implementation is transitive and reflexive. Furthermore, the operators introduced in Section 3 are monotonic with respect to this implementation relation.

Using this notion of implementation, Power Position Controlled Window is an
implementation of Power Position Window and vice versa.

4.2 Proof Support

To effectively use the implementation relation in a sound development process, (automatic) support for the verification of the implementation relation between two functions is necessary. Since the behavior of functions is defined by (possibly infinite) sets of finite traces, and the implementation relation is defined by the inclusion relation over those sets, a trace-based formalism is best-suited.

Therefore, here WS1S (weak second order monadic structure with one successor function) is used to implement automatic proof support. This formalism is, e.g., supported by the modelchecker Mona [7]. Using WS1S, functions are specified by predicates over sets of traces. The operators introduced in Section 3 can be directly implemented, allowing a compositional construction of the corresponding trace sets. Similarly, the implementation relation can be defined as a relation on trace sets. Besides proving the refinement relation between two functions, Mona can be used to generate a counter-example for functions violating the refinement relation.

5 Conclusion

The increasing complexity of reactive behavior integrating different interacting functionalities requires a construction process supporting the modular description of individual functions as well as their composition into the overall behavior.

Therefore, we suggest functional modular development using functions as construction units, with transitions as the most basic form, as well as disjunctive and conjunctive composition to combine modules. Offering separation of concern by modular composition of functions, reasoning about the overall behavior is simplified by conjunctive and disjunctive construction of functionalities. Additionally, reuse of modular functionalities is simplified when constructing variants of reactive behavior. Finally, using automatic proof support, the implementation of the integrated modular behavior through a more-constructive form of functional description can be established.

References


A Model-Based Approach to the Verification and Adaptation of WF/.NET Components

Javier Cubo, Gwen Salaün, Carlos Canal, Ernesto Pimentel
Department of Computer Science, University of Málaga
Campus de Teatinos, 29071, Málaga, Spain
Email: {cubo,salaun,canal,ernesto}@lcc.uma.es

Pascal Poizat
INRIA/ARLES Project-Team, France, and
IBISC FRE 2873 CNRS - Université d'Évry, France
Email: pascal.poizat@inria.fr

Abstract
This paper presents an approach which supports verification and model-based adaptation of software components and services implemented using Windows Workflow Foundation (WF). First, we propose an abstract description of WF workflows, and we formalise the extraction of Labelled Transition Systems from these workflows. Next, verification and adaptation are applied using respectively model-checking techniques and existing model-based adaptation approaches. Last, we explain how a WF workflow can be generated from an adaptor protocol.

Key words: Software Components, Services, Composition,
Model-based Adaptation, WF Workflows, Model-checking

1 Introduction
Software Adaptation [4,7] is a promising research area which aims at supporting the building of component systems by reusing software entities. These can be adapted in order to fit specific needs within different systems. In such a way, application development is mainly concerned with the selection, adaptation and composition of different pieces of software rather than with the programming of applications from scratch. Many approaches dedicated to model-based adaptation [5,8,16,19,23,25] focus on the behavioural interoperability level, and aim at generating new components called adaptors which are used to solve mismatch in a non-intrusive way. This process is completely automated being given an adaptation mapping which is an abstract description of how mismatch can be solved with respect to the behavioural interfaces of
components. However, most of these approaches are independent of the implementation framework, and few of them relate with existing programming languages and platforms. To the best of our knowledge, the only attempts in this direction have been carried out using COM/DCOM [16], BPEL [6], and SCA components [19].

In this paper, we focus on Windows Workflow Foundation (WF) [24] which belongs to the .NET Framework 3.0 developed by Microsoft. We have chosen WF because this platform supports the behavioural descriptions of components/services using workflows. In addition, the .NET Framework is widely used in private companies, and makes the implementation of services easier thanks to its workflow-based graphical support and the automation of the code generation. More than dealing only with adaptation of WF components, our approach also allows the verification of such components by extracting abstract descriptions from them and by using model-checking tools. This work extends and formalises the ideas sketched in [9].

Our approach is summarised in Figure 1. To make the verification and adaptation possible, in a first stage, abstract behavioural descriptions (Labelled Transition Systems, LTSs) have to be extracted from WF workflows. Next, being given a set of LTSs, mismatch detection is computed to check whether the involved components need adaptation or not. If a mismatch exists, we apply adaptation techniques that aim at generating an adaptor protocol/LTS from a mapping. Assessment techniques are then helpful to check that the adaptor is as expected. If not, another mapping may be proposed. We emphasise that formal verification of WF components takes place twice: when detecting mismatch, and when assessing the resulting system (components+adaptor). Last, once the designer is satisfied by the abstract adaptor, the corresponding WF workflow is generated.

![Fig. 1. Overview of our approach for the adaptation of WF components](image)

The remainder of the paper is organised as follows. We give an overview of WF, and we define an abstract notation for WF workflows in Section 2. We present in Section 3 a simple on-line computer sale example, and the WF components it relies on. Section 4 formalises the extraction of LTSs from
WF workflows. In Section 5, we focus on the verification and adaptation of WF components based respectively on model-checking techniques and model-based adaptation. Section 6 presents the encoding of an adaptor LTS into a WF workflow. In Section 7, the contributions of our approach are compared to related work. Finally, in Section 8, we conclude the paper and present future work.

2 WF Workflow Notation

In this paper, we present a representative kernel of the WF activities, namely Code, Terminate, InvokeWebService, WebServiceInput, WebServiceOutput, Sequence, IfElse, Listen with EventDriven activities, and While. The reader interested in more details may refer to [24]. We also introduce a textual and abstract notation for the aforementioned WF activities.

2.1 WF Overview

WF belongs to the .NET Framework 3.0, and is supported by Visual Studio 2005. The available programming languages to implement workflows in Visual Studio 2005 are Visual Basic and C#. In this work, C# has been chosen as the implementation language.

The Code activity is meant to execute user code provided for execution within the workflow. The Terminate activity is used to finalise the execution of a workflow. A WF InvokeWebService activity calls a Web service and receives the requested service result back. If such an invoke has to be accessed by another component C, it has to be preceded by a WebServiceInput activity, and followed by a WebServiceOutput activity. Hence, C will interact with this new service using these two input/output activities that enable and disable the data reception and sending, respectively, with respect to the invoked Web service. WF-based XML Web services require at least one WebServiceInput and one or more WebServiceOutput activities. The input and output activities are related, thus each output activity must be associated with an input activity. It is not possible to have an instance of WebServiceInput without associated outputs, as well as having outputs without at least one WebServiceInput.

The Sequence construct executes a group of activities in a precise order. The WF IfElse activity corresponds to an if-then-else conditional expression. Depending on the condition evaluation, the IfElse activity launches the execution of one of its branches. If none of the conditions is true, the else branch is executed.

The Listen activity defines a set of EventDriven activities that wait for a specific event. One of the EventDriven activities is fired when the expected message is received. Last, the While construct defines an activity that is fired as many times as the While condition is true.
2.2 Abstract Notation for WF Workflows

Here, we define a textual and abstract notation for WF workflows. This notation makes abstract several implementation details. Our proposal considers as input textual workflows instead of their graphical description. Table 1 formalises the grammar for the textual notation of WF activities $A$, where $C$, $C_i$ are boolean conditions, and $I$, $I_i$ (inputs), $O$, $O_i$ (outputs) are parameters of activities.

$$A ::= \text{Code} \quad \text{executes a chunk of code}$$
$$\text{ | Terminate} \quad \text{ends a workflow’s execution}$$
$$\text{ | InvokeWebService}(O_1, \ldots, O_n, I) \quad \text{calls a Web service (WS)}$$
$$\text{ | WebServiceInput}(I_1, \ldots, I_n) \quad \text{receives data from a WS}$$
$$\text{ | WebServiceOutput}(O) \quad \text{sends data to a WS}$$
$$\text{ | Sequence}(A_1, A_2) \quad \text{executes first } A_1 \text{ and then } A_2$$
$$\text{ | IfElse}((C_1, A_1), \ldots, (C_n, A_n), A_{n+1}) \quad \text{executes } A_i \text{ if } C_i \text{ is true, or } A_{n+1} \text{ otherwise}$$
$$\text{ | Listen}(E_1, \ldots, E_n) \quad \text{fires one of the } E_i \text{ branches}$$
$$\text{ | While}(C, A) \quad \text{executes } A \text{ while } C \text{ is true}$$

$$E ::= \text{EventDriven} (\text{WebServiceInput}(I), A) \quad \text{executes } A \text{ when } I \text{ is received}$$

Table 1
Grammar for the abstract notation of WF workflows

3 Running Example: On-line Computer Sale

This section introduces an on-line computer sale example. It consists of a system whose purpose is to sell computer material such as PCs, laptops, or PDAs to clients. As a starting point we reuse two components: a Buyer and a Supplier. These components have been implemented using WF./NET, and their workflows are presented in Figure 2.

First, the Supplier receives a request under the form of two messages that indicate the type of the requested material, and the max price to pay (type and price, (1) and (2) respectively in Fig. 2). Then, it sends a response indicating if the request can be replied positively (reply, (3)). Next, the Supplier can terminate the session, receive and reply other requests ((4), (5) and (6)), or receive an order of purchase (buy, (7)). In the latter case, a confirmation is sent (ack, (8)) emphasising if the purchase has been realised.
Fig. 2. WF workflows for the Supplier (left) and Buyer (right) components correctly or not.

The Buyer can submit a request (request, (9) in Fig. 2), in which it indicates the type of material he/she wants to purchase and the price to pay. Next, once he/she has received a response (reply, (10)), the Buyer may realise another request ((11) and (12)), buy the requested product (purchase and ack, (13) and (14)), or end the session (stop, (15)).

In both Supplier and Buyer we have split the workflows of Figure 2, presenting them into two parts. On the left-hand side, we show the initial execution belonging to the first request, and on the right-hand side we present the loop offering the possibility of executing other requests, performing a purchase or finalising. We identify the names of certain activities, whose functionality is the same, with an index (such as type_1 and type_2, or invokeType_1 and invokeType_2 in Supplier), because WF does not accept activities identified using the same name. In the Buyer component, the messages with the code suffix, such as request_1.code, correspond to the execution of C# code. Last, some WebServiceInput and WebServiceOutput activities may be meaningless with respect to the component functionality, and appear in the WF workflow only because WF obliges their presence before and after InvokeWebService activities. In Figure 2, these activities are identified with tau identifiers.

To illustrate the textual notation defined in Section 2.2, we apply it on the Supplier WF workflow. We focus on the While construct, and present a
part of the Listen activity it contains. The condition of the While construct is true because the component loops on requests until it receives an order of purchase, or until the system stops.

```plaintext
Sequence
  (...,
  While
    (true,
     Listen
      (EventDriven
        (WebServiceInput(type),
         ...
      ),
      EventDriven
        (WebServiceInput(buy),
         Sequence
          (InvokeWebService(buy, ack),
           Sequence
             (WebServiceOutput(ack),
              Terminate
             )
          )
        )
      )
    )
  )
)
```

Note that we remove in the abstract notation all the suffixes used in the workflows to distinguish activity names. Last, we recall that in the following we consider such an abstract description of WF components, as an input (Section 4) and output (Section 6) to our verification and adaptation proposal.

### 4 Extracting LTSs from WF Workflows

Since we want to reuse existing techniques to make verification and adaptation of WF components, we have first to extract from the abstract WF notation the required model, namely Labelled Transition Systems. A LTS is a tuple \((A, S, I, F, T)\) where: \(A\) is an alphabet (set of events or messages), \(S\) is a set of states, \(I \in S\) is the initial state, \(F \subseteq S\) are final states, and \(T \subseteq S \times A \times S\) is the transition function. The extracted LTSs must preserve the semantics of workflows as encoded in WF/.NET Framework 3.0. A formal proof of semantics preservation between both levels is not achieved yet since WF does not provide a formal semantics. Our encoding has been deduced from our experiments using the WF platform. The main ideas of the LTS obtaining from abstract description of workflow constructs are the following.

- **Code** is internal and hence interpreted as an internal transition, \(\tau\);
• **Terminate** corresponds to a final state;

• **InvokeWebService** corresponds to a sequence of emissions followed by a reception, **WebServiceInput** corresponds to a sequence of receptions, and **WebServiceOutput** corresponds to an emission;

• **Sequence** is translated so as to preserve the order of the involved activities. For this, the final states of the first activity are linked to the initial state of the second activity using ε transitions;

• **IfElse** corresponds to an internal choice. This corresponds to as many τ transitions as there are branches in the IfElse construct (including the else branch). Each of these τ transitions leads to the initial state of the corresponding activity;

• **Listen** corresponds to an external choice. This corresponds to as many outgoing transitions as there are branches in the Listen construct. These transitions are labelled with receptions corresponding to the messages that can be received and target the initial state of the related activity;

• **While** is translated as a looping behaviour, where the choice between termination or loop is encoded using internal non-determinism (τ transitions).

Formally, an LTS \( L = (A, S, I, F, T) \) can be obtained from a abstract workflow represented by activity \( A \) using function \( \text{auf2lts} : WF \rightarrow LTS \). For an LTS \( L = (A, S, I, F, T) \), we define \( X(L) = X \) for every \( X \) in \( \{A, S, I, F, T\} \). This notation is overloaded for activities: for some activity \( A \), we define \( X(A) = X(\text{auf2lts}(A)) \) for every \( X \) in \( \{A, S, I, F, T\} \). Finally, we use new \( s \) to denote the creation of \( s \) as a new (fresh) state in the LTSs we are building. \( \text{auf2lts} \) can be defined inductively on the structure of WF activities as follows:

**Code** \( \mapsto (\{\tau\}, \{\text{new } s_1, \text{new } s_2\}, s_1, \{s_2\}, \{s_1 \xrightarrow{\tau} s_2\}) \)

**Terminate** \( \mapsto (\emptyset, \{\text{new } f\}, f, \{f\}, \emptyset) \)

**InvokeWebService** \((O_1, \ldots, O_n, I) \mapsto (\{O_1!\}, \ldots, O_n!, I?), \bigcup_{i \in \{0, \ldots, n+1\}} \{\text{new } s_i\}, s_0, \{s_{n+1}\}, (\bigcup_{i \in \{0, \ldots, n\}} s_{i-1} \xrightarrow{O_i} s_i) \cup \{s_n \xrightarrow{I} s_{n+1}\}) \)

**WebServiceInput** \((I_1, \ldots, I_n) \mapsto (I_1?, \ldots, I_n?), \bigcup_{i \in \{0, \ldots, n\}} \{\text{new } s_i\}, s_0, \{s_n\}, \bigcup_{i \in \{1, \ldots, n\}} s_{i-1} \xrightarrow{I_i} s_i \)

**WebServiceOutput** \((O) \mapsto (O!), \bigcup_{i \in \{0, 1\}} \{\text{new } s_i\}, s_0, \{s_1\}, \{s_0 \xrightarrow{O!} s_1\}) \)

**Sequence** \((A_1, A_2) \mapsto (A(A_1) \cup A(A_2) \cup \{\epsilon\}, S(A_1) \cup S(A_2), I(A_1), F(A_2), T(A_1) \cup T(A_2) \cup \{f \xrightarrow{\tau} I(A_2) \mid f \in F(A_1)\}) \)

**IfElse** \((C_1, A_1), \ldots, (C_n, A_n), A_{n+1} \mapsto (\bigcup_{i \in \{1, \ldots, n+1\}} A(A_i)) \cup \{\tau\} \)
Listen(EventDriven(WebServiceInput(I_1), A_1),..., EventDriven(WebServiceInput(I_n), A_n)) \mapsto (\bigcup_{i \in [1,...,n+1]} S(A_i) \cup \{new s\}, s, \bigcup_{i \in [1,...,n+1]} F(A_i), \\
\bigcup_{i \in [1,...,n+1]} (T(A_i) \cup \{s \xrightarrow{\tau} I(A_i)\}))

While(C, A) \mapsto (A(\mathcal{A}) \cup \{\epsilon\} \cup \{\tau\}, S(\mathcal{A}) \cup \{new s, new f\}, s, F(\mathcal{A}) \cup \{f\}, \\
T(\mathcal{A}) \cup \{s \xrightarrow{\tau} I(\mathcal{A})\} \cup \{s \xrightarrow{\tau} f\} \cup \{f' \xrightarrow{l} I(\mathcal{A}) \mid f' \in F(\mathcal{A})\})

Once the LTS is constructed, \(\epsilon\) transitions are removed [15]. LTS does not support the description of data expressions, consequently conditions appearing in While and IfElse constructs are abstracted away while extracting LTS. Likewise, WebServiceInput and WebServiceOutput activities identified with tau identifiers (see Fig. 2) are translated as \(\tau\) transitions in the corresponding LTS.

Initial and final states in the LTS come respectively from the explicit initial and final states that appear in the workflow. There is a single initial state that corresponds to the beginning of the workflow. Final states correspond either to a Terminate activity or to the end of the whole workflow. Accordingly, several final states may appear in the LTS because several branches in the workflow may lead to a final state. Initial and final states are respectively depicted in LTSs using bullet arrows and hollow states.

Let us illustrate the extraction of LTSS from abstract WF workflows on our running example (Figure 3). The messages that appear in the Buyer LTS come from the output and input parameters that appear in its invoke activities. As far as the Supplier component is concerned, the invoke activities are made abstract because they correspond to interactions with external components (in charge of the material database), and are not of interest for the composition at hand. Therefore, the observable messages in this case are coming from the input and output messages surrounding the invoke activities. All the \(\tau\) transitions in LTSs are removed using a behavioural equivalence (\(\tau\ast.a\) reduction [12]) before the adaptation process to favour efficiency and readability. To identify unambiguously component messages in the adaptation process, their names are prefixed by the component identifier, respectively \(b\) for Buyer, and \(s\) for Supplier.

5 Verification and Model-Based Adaptation in WF

This section presents our approach to verify and compose/adapt WF components. Verification techniques are useful in two cases: first, they may help to identify mismatch situations, and, in a second step, they allow to check if the adaptor works correctly, since the designer writes the mapping by hand.
therefore it may contain some errors that will be reflected in the adaptor protocol.

5.1 Detection of Mismatch Cases

First of all, let us introduce verification techniques that can be used to check component LTSs, and their composition with the adaptor LTS (see Section 5.4). All existing model-checking tools that accept automata-based format as input are good candidates to these checks, namely SPIN [14], CADP [12] or mCRL2 [13]. In this paper, we illustrate these ideas with CADP which is a verification toolbox for asynchronous concurrent systems. CADP allows to deal with very large state spaces, and implements various verification techniques such as model checking, compositional verification, equivalence checking, distributed model checking, etc.

The main idea is to generate the full system using parallel composition operators available in CADP (or similar tools), and then to reason on the resulting system using mainly visual checking and model-checking of temporal properties. Model-checking is an automatic technique that efficiently detects subtle architectural flaws. Classical properties such as liveness or safety properties can be easily formalised reusing patterns [17], and then checked against the system model (LTS) using model-checkers, e.g., Evaluator [18] which belongs to CADP.

As regards our running example, we first compute the resulting LTS by composing components Buyer and Supplier and enforcing their interaction on all messages appearing in both components. The resulting LTS consists of a single state with no outgoing transitions. This is quite obvious because both components suffer of mismatch in their first transition (request! in the Buyer versus type? in the Supplier). Indeed, a study of these LTSs points out the three following cases of mismatch:

(i) name mismatch: the Buyer may buy the computer using purchase! whereas the Supplier may interact on buy?;
Cubo et al.

(ii) mismatching number of messages: the Buyer sends one message for each request (request!) while the Supplier expects two messages, one indicating the type (type?), and one indicating the max price (price?);

(iii) independent evolution: the Buyer may terminate with stop! but this message has no counterpart in the Supplier.

5.2 Adaptation Mapping

Now, a mapping should be given to work the aforementioned cases of mismatch out. $\varepsilon$ is used in vectors when some message has no counterpart in a component (see e.g. $V_{\text{price}}$ and $V_{\text{stop}}$ below). We use vectors that define correspondences between messages. More expressive mapping notations exist in the literature, such as regular expressions of vectors [8], but with respect to the example at hand, vectors are enough to automatically retrieve a solution adaptor. A possible mapping for our example is as follows:

- $V_{\text{req}} = (b:\text{request}!, s:\text{type}?)$
- $V_{\text{price}} = (b:\varepsilon, s:\text{price}?)$
- $V_{\text{reply}} = (b:\text{reply}?, s:\text{reply}!)$
- $V_{\text{stop}} = (b:\text{stop}!, s:\varepsilon)$
- $V_{\text{buy}} = (b:\text{purchase}!, s:\text{buy}?)$
- $V_{\text{ack}} = (b:\text{ack}?, s:\text{ack}!)$

The name mismatch can be solved by vector $V_{\text{buy}}$. The correspondence between request! and messages type? and price? can be achieved using two vectors, $V_{\text{req}}$ and $V_{\text{price}}$, where the second contains an independent evolution of component Supplier. The last mismatch is solved using $V_{\text{stop}}$ in which the message stop! is associated to nothing.

5.3 Generation of the Adaptor Protocol

Given a set of component LTSs (Section 4) and a mapping (Section 5.2), we can use existing approaches (here we rely on [8]) to generate the adaptor protocol automatically. This automation is crucial because in some cases, the adaptor protocol may be very hard to derive manually.

Figure 4 presents the Adaptor LTS. Since the adaptor is an additional component through which all the messages transit, all the messages appearing in the adaptor protocol are reversed with respect to the ones in the components. Note first that the adaptor receives the request coming from the Buyer, and splits the message into messages carrying the type and price information. This LTS also shows how the termination is possible along the stop? message, and how the adaptor may interact on different names (purchase? and buy!) to make the interaction possible.
5.4 Assessment Techniques

In this last step, we use model-checking techniques to validate the adaptor LTS generated from the mapping proposed above. First, the LTS corresponding to the composition of both components and the adaptor is computed. Synchronisation is made explicit, and both components interact together through the adaptor. Next, the designer may write some properties to be verified by the final system. In the rest of this section, we show some examples of \( \mu \)-calculus formulas we checked on this system using Evaluator:

(i) a supplier always replies a buyer request

\[
[\text{true}* . \ "b\_request" ]
\mu X. (\langle \text{true} \rangle \text{true} \text{ and } \neg \text{"s\_reply"}) X
\]

(ii) a buyer request is always followed by stop, purchase, or request

\[
[\text{true}* . \ "b\_request" ]
\mu X. (\langle \text{true} \rangle \text{true} \text{ and }
\neg \text{"b\_stop" or "b\_purchase" or "b\_request"}) X
\]

(iii) a buyer request is always followed by stop, or purchase

\[
[\text{true}* . \ "b\_request" ]
\mu X. (\langle \text{true} \rangle \text{true} \text{ and }
\neg \text{"b\_stop" or "b\_purchase"}) X
\]

Properties (i) and (ii) are true, whereas the last one (iii) is false, but this is normal because the system can loop forever on exchanging request/reply messages. If some properties turn out to be false whereas a positive answer was expected, it means that the adaptation mapping contains errors that does not make the system behave as required. In this situation, the mapping must be corrected and assessment applies again.

6 Generating WF Workflows from LTSs

The last step in our proposal is to generate an abstract workflow from an adaptor protocol. Formalising the function \( \text{lts2awf} \) is quite tough, especially because cycles in the adaptor LTS have to be encoded with \text{While} activities which must preserve the LTS behaviour. Therefore, as a first attempt, we give
in this section some guidelines for this encoding.

First, the initial state of the LTS is encoded as the initial state of the workflow. Final states are encoded as Terminate activities. The adaptation process removes all the $\tau$ transitions. Then, all the needed pieces of C# code will be added by hand while refining the abstract workflow into a real WF workflow.

The translation process derives step by step parts of the abstract workflow by focusing on one state of the LTS after the other. We distinguish in the following the translation of transitions corresponding to message activities (InvokeWebService, WebServiceInput, WebServiceOutput), and the generation of structuring activities (Sequence, IfElse, Listen, While). Let us start with messages, and note that the three rules below have to be applied in this order to check if the sequence of messages corresponds to an InvokeWebService before translating it in separate WebServiceOutput or WebServiceInput activities:

- a sequence of transitions with labels holding one or several emissions followed by a reception is encoded as an InvokeWebService activity;
- one or several transitions with receptions as labels are translated into a WebServiceInput activity;
- a transition with one emission corresponds to a WebServiceOutput activity.

Now, we focus on the encoding of the LTS structuring into the abstract workflow:

- a Sequence activity is generated for a sequence of transitions in the LTS corresponding to two successive message activities, and for which no states involve more than one outgoing transition;
- if the state of the LTS to be translated involves two or more outgoing transitions:
  - if all the outgoing transitions hold input messages, a Listen activity is derived,
  - otherwise a conditional choice IfElse activity is generated;
- a cycle in an LTS is translated using a While activity. If several cycles loop on a same state, it corresponds to a single While activity. However if a cycle in the LTS contains another (local) cycle, this latter will also be translated as a While activity nested in the outmost one.

Following these guidelines, an abstract workflow has been derived for our running example that we do not show here for space reasons. Last, this abstract workflow has been refined into a WF workflow (Fig. 5). This refinement step requires the intervention of the designer, to (i) concretise conditions in IfElse and While activities, and (ii) add C# pieces of code to get the adaptor WF component works correctly. Moreover, WF requires addresses of components to be specified in invocations. Therefore, to deploy our adapted system, we have first to update the components workflows to change these addresses.
into the adaptor one. However, this can be done automatically.

![ WF workflow for the Adaptor component ]

Finally, we point out that the simple system presented in this paper has been completely implemented using WF, and the Buyer and Supplier components worked as required thanks to the use of the Adaptor component.

7 Related Work

The first group of related work concerns proposals that aimed at applying adaptor generation approaches to existing implementation platforms. Brogi and Popescu [6] outline a methodology for the automated generation of adaptors capable of solving behavioural mismatches between BPEL processes [1]. In their adaptation methodology they use YAWL workflow as intermediate language. Once the adaptor workflow is generated, they use lock analysis techniques to check if a full adaptor has been generated or only a partial one (some interaction scenarios cannot be resolved).

In [6], the authors chose BPEL. Both BPEL and WF languages allow to design Web services, but WF can also be used to implement any kind of software component. Their respective platforms make the implementation easier thanks to their workflow-based graphical support, and the automated generation of most of the underlying code (XML+Java in BPEL, using Java Application Server included in Netbeans Enterprise, and XML+C# in WF). In this work, we have focused on WF because it is an interesting alternative to BPEL that has not been studied yet. In addition, as a long term purpose, we want our proposal to benefit to the wide number of people that use the .NET Framework in private companies around the world. Compared to [6]
our adaptation approach is able to reorder messages in between components when required.

Inverardi and Tivoli [16] tackle the automatic synthesis of connectors in the COM/DCOM framework, by guaranteeing deadlock-free interactions among components. They may also define properties that the resulting system should verify using liveness and safety properties expressed as specific processes. Compared to this proposal, our approach does not only restrict the adaptor to possible non-deadlocking behaviours [16] but may also address behavioural adaptation. That comes from the notation and adaptation techniques we rely on that allows to deal with possibly complex adaptation scenarios, whereas this approach does not use any mapping language for adaptor specification.

As regards verification of component-based systems, recent approaches have been dedicated to the verification of software components specified using LOTOS, LTSs and synchronisation networks [2,3]. These works present a method and a tool intended to application developers, to build behavioural models of Fractal components on which properties can be verified using CADP. In the Web Service area, different works have been dedicated to verifying Web service description to ensure some properties of systems [10,11,21,22]. Summarising these works, they use model-checking to verify some properties of cooperating Web services described using XML-based languages (DAML-S, WSFL, BPEL, WSCI). Accordingly, abstract representations are extracted from Web service implementations, and some properties may be ensured using ad-hoc or well-known tools (e.g., SPIN, LTSA). Last, Mouakher et al. [20] start with a description of components using UML class and state diagrams that they encode into the B method to use its associated theorem prover, namely Atelier B or B4free, so as to perform compatibility checks. In a second step, they specify adaptors in B, and address their correctness.

Compared to these different proposals, ours focuses on both verification and adaptation of components. We prefer model checking (instead of theorem proving with B for instance) because it makes verification steps easier thanks to a full automation and its adequacy to automata-based models. In addition, adaptation techniques support the automatic generation of adaptors in case verification reveals that components cannot be directly reused (the adaptor is completely specified by hand in [20]).

8 Concluding Remarks

This paper has presented an approach to verify WF components, and in case they cannot be directly composed, we have sketched how an adaptor protocol can be generated, and encoded into a new WF component. We have illustrated the application of our proposal in practice on a simple yet realistic example. This work is promising because it demonstrates that software adaptation can be of real interest for widely used implementation platforms such as the .NET Framework 3.0, and can help the developer in building software applications
by reusing software components or services.

As far as future work is concerned, here is a list of perspectives we will tackle to complement our approach:

• extending the set of WF activities considered in our proposal;
• extending our LTS model with respect to these new activities, and keeping data description at this level;
• formalising both functions \( \text{awf2lts} \) and \( \text{lts2awf} \) to support the automatic extraction and generation of abstract workflows;
• extending our verification and adaptation proposal to deal with this new model;
• implementing our translation functions between LTSs and abstract workflows in a prototype tool;
• implementing in this tool automatic translators between WF workflows (described in XML format) and abstract workflows;
• experimenting the proposal on more complex and realistic examples.

In parallel, we would also like to carry out experiments on the implementation of adaptors using BPEL and the Netbeans Enterprise platform to compare on precise criteria the adequacy of both platforms to apply adaptation in practice.

Acknowledgements. This work has been partially supported by project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and project P06-TIC-02250 funded by Junta de Andalucía.

References


Liveness in Interaction Systems

Mila Majster-Cederbaum

Institute of Computer Science
University of Mannheim
Mannheim, Germany

Moritz Martens

Institute of Computer Science
University of Mannheim
Mannheim, Germany

Christoph Minnameier

Institute of Computer Science
University of Mannheim
Mannheim, Germany

Abstract

Interaction systems were proposed and implemented by Sifakis et al. as a model for the design and study of component based systems. We investigate here the property of liveness in interaction systems where liveness of an action, a component or a set of components means that the action (component, set of components) will repeatedly participate in every run of the global system. We show that deciding liveness is NP-hard. Then we present a characterization of liveness. Finally, by exploiting local information, we establish a polynomial-time criterion that guarantees liveness. We combine the criterion with the characterization to obtain a test for liveness.

Keywords: Component-based Modeling, Interaction Systems, Liveness

1 Introduction

In the last decade a variety of formal approaches to the specification and analysis of component based systems at different levels and with different specific objectives have been proposed [1,2,9,33,34,38,35,7,21,5,23,10,16,15].

We investigate here the approach of interaction systems that was proposed and discussed by Sifakis et al. in [16,17,15,40,41,39] and in more detail in [18]. The model clearly separates the issues of 1. interfaces, 2. behavior of the components

1 Email: mcb@informatik.uni-mannheim.de
2 Email: mmartens@informatik.uni-mannheim.de
3 Email: cmm@informatik.uni-mannheim.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
and 3. interaction between components. It has been implemented successfully in the Prometheus tool [13] and the BIP system [4]. BIP has been extended to a framework for hierarchical components [19] and enriched with contracts as defined in the SPEEDS project [6]. The model was used to specify various applications, e.g. [4,13,14,16,40,27,31,29]. In this approach a component offers a certain set of ports. The communication behavior of a component is given by a labeled transition system where the labels are taken from the port set. That is, the transition system restricts the order of calls to the ports. Components are put together via connectors where a connector is a finite nonempty set of ports such that no two ports stem from the same component. The transitions which the induced global system can perform are regulated by these connectors. One port can be contained in various connectors of various sizes such that the cooperation between components can be regulated freely in a simple way. In approaches using process algebra the basic cooperation scheme is usually fixed and binary and it is cumbersome to realize more flexibility. I/O-automata [23] can be considered as a subclass of interaction systems. They also use some kind of transition systems to model components but have a more restrictive scheme of cooperation. This is also true for the interface-automata defined in [10].

Different aspects of component-based systems have been studied, for example compatibility [37], deadlock-freedom [21,5,1], reliability prediction [22,36,38]. In the framework of interaction systems properties such as local/global deadlock-freedom, progress of a component, availability of ports and robustness against failure of components have been discussed [18,14,24,25]. In general it is difficult to test these properties as they involve the exploration of the global state space. Indeed, it was shown that deciding deadlock-freedom in interaction systems is NP-hard [32]. Recently this result was strengthened by showing that deciding local and global deadlock-freedom is PSPACE-hard [26]. One way to proceed in such situations is to establish criteria that ensure desired properties and can be tested more easily. In [25] for example we presented a condition that ensures deadlock-freedom and can be tested in polynomial time. In [24] we investigated criteria for robustness.

In this work we concentrate on liveness in interaction systems. A component is considered to be live if, no matter how the global system evolves and independently of the point of time we consider, it will repeatedly participate in some step of the system. We show first that deciding liveness of a set of components is NP-hard. Then we give a characterization of liveness of a set of components in an interaction system. Moreover, we establish a criterion that entails liveness and can be tested in polynomial time. We present a hybrid algorithm for testing liveness that combines the characterization and the criterion mentioned above.

The paper is structured as follows. Section 2 summarizes the basic definitions. Section 3 contains the notions of deadlock-freedom and liveness. Liveness is investigated in Sections 4, 5 and 6 which are concerned with NP-hardness respectively present the characterization of, the criterion for liveness, and the hybrid algorithm.

2 Components, Connectors and Interaction Systems

We build on a model for component-based systems, called interaction systems, that was proposed in [16,17,15,40,41]. We start with a set $K$ of components where we
usually refer to a component as $i \in K$. For every component $i \in K$ a set $A_i$ of ports (or actions) is specified which are used for cooperation with other components. The cooperation is determined by so-called connectors. A connector is a finite nonempty set of ports that contains at most one port for every component in $K$. Any nonempty subset of a connector constitutes an interaction of the system. A port may be part of various connectors which may be of different size. Hence the cooperation can be regulated in a very flexible way. An interaction describes a step of the system where the ports contained in that interaction are performed together.

**Definition 2.1** A component system is a pair $CS = (K, \{A_i\}_{i \in K})$ where $K$ is the set of components, $A_i$ is the port set of component $i$, and any two port sets are disjoint. Ports are also referred to as actions.

The union $A = \bigcup_{i \in K} A_i$ of all port sets is the port set of $K$. A finite nonempty subset $c$ of $A$ is called a connector for $CS$, if it contains at most one port of each component $i \in K$, that is $|c \cap A_i| \leq 1$ for all $i \in K$. A connector set is a set $C$ of connectors for $CS$ that covers all ports and contains only maximal elements:

1. $\bigcup_{c \in C} c = A$
2. $c \subseteq c' \Rightarrow c = c'$ for all $c, c' \in C$.

If $c$ is a connector, $I(c)$ denotes the set of all nonempty subsets of $c$ and is called the set of interactions of $c$. For a set $C$ of connectors $I(C) = \bigcup_{c \in C} I(c)$ is the set of interactions of $C$. We also call connectors $c \in C$ the maximal interactions. For component $i$ and interaction $\alpha$, we put $i(\alpha) = A_i \cap \alpha$. We say that component $i$ participates in $\alpha$, if $i(\alpha) \neq \emptyset$.

We give a simple example to illustrate these concepts. We will extend this example throughout the text whenever we encounter new notions.

**Example 2.2** We consider a component system $CS_5 = (K_5, \{A_i\}_{i \in K_5})$ consisting of five components, where $K_5 := \{1, 2, 3, 4, 5\}$ and the port sets of the components are given by $A_1 := \{a_1\}, A_2 := \{b_1, b_2\}, A_3 := \{d_1, d_2\}, A_4 := \{e_1, e_2\}$, and $A_5 := \{f_1, f_2\}$. Additionally we fix the connector set $C_5 := \{\{a_1, b_1\}, \{a_1, e_1\}, \{d_1, f_1\}, \{a_1, e_2, f_2\}, \{b_2, e_2, f_2\}, \{b_2, d_2, e_2\}, \{a_1, d_2, e_2\}\}$. This component system is illustrated in Fig. 1 where the components are shown as boxes and the ports of the components are the black boxes. The connectors are represented by lines connecting the respective ports. Here for example components 1 and 2 may perform their respective first actions together whereas 2, 4, and 5 may perform their respective second actions together.

In the following, we always assume that $K = \{1, \ldots, n\}$ for some $n \in \mathbb{N}$ or that $K$ is countably infinite.

An interaction model for a component system $CS$ is defined by a connector set $C$ together with a set Comp of interactions that are declared to be complete. If an interaction is declared complete it can be performed independently of the environment. Note that it is a design decision which interactions are chosen to be complete. This choice is not restricted in any way and only depends on the system one wishes to model.
**Definition 2.3** Let $C$ be a connector set for the component system $CS$ and let $\text{Comp} \subseteq I(C)$ be a subset of interactions. The pair $IM := (C, \text{Comp})$ is an interaction model for $CS$. The elements of $\text{Comp}$ are called complete interactions.

**Example 2.2 continued** Let $IM_5 := (C_5, \emptyset)$ be an interaction model for $CS_5$, i.e. only interactions in $C_5$ are independent of the environment.

If for some reason the interactions $\{e_2\}$ and $\{f_2, b_2\}$ for example should be independent of other actions, we could set $\text{Comp} := \{\{e_2\}, \{f_2, b_2\}\}$.

The notions presented so far are only concerned with the possible structure of communication between the different components. In a further level of description of the components the order in which a component may perform the actions it provides is restricted. For that purpose for every component $i \in K$ a labeled transition system $T_i$ describing the behavior of $i$ with respect to interaction is introduced. In the simplest case $T_i$ is the “union” of transition systems $T_{ij}$ where $T_{ij}$ is the protocol regulating the cooperation of component $i$ with component $j$.

**Definition 2.4** Let $CS = (K, \{A_i\}_{i \in K})$ be a component system and $IM = (C, \text{Comp})$ an interaction model for $CS$. Let for each component $i \in K$ a transition system $T_i = (Q_i, A_i, \rightarrow_i, Q_i^0)$ be given where $\rightarrow_i \subseteq Q_i \times A_i \times Q_i$ and $Q_i^0 \subseteq Q_i$ is a non-empty set of initial states. We write $q_i \xrightarrow{a_i} q'_i$ instead of $(q_i, a_i, q'_i) \in \rightarrow_i$.

The induced interaction system is given by $Sys := (CS, IM, \{T_i\}_{i \in K})$ where the global behavior $T = (Q, C \cup \text{Comp}, \rightarrow, Q^0)$ is obtained from the local transition systems of the individual components in a straightforward manner:

(i) $Q := \prod_{i \in K} Q_i$, the Cartesian product of the $Q_i$ which we consider to be order independent. We denote states by tuples $q := (q_1, \ldots, q_j, \ldots)$ and call them (global) states.

(ii) $Q^0 := \prod_{i \in K} Q_i^0$, the Cartesian product of the local initial states. We call the elements of $Q^0$ (global) initial states.

(iii) $\rightarrow \subseteq Q \times (C \cup \text{Comp}) \times Q$, the transition relation for $Sys$ defined by...
∀α ∈ (C ∪ Comp) ∀q, q’ ∈ Q : q = (q₁, . . . , q_j, . . .) α q’ = (q’₁, . . . , q’_j, . . .)
⇔ ∀i ∈ K : q_i α_i q’_i if i participates in α and q’_i = q_i otherwise.

The global system can perform either complete or maximal interactions α where α may be performed in a global state q if all partners that are involved in α are offering their corresponding action.

Without loss of generality we always assume that every local state of every component is reachable from some initial state in the local transition system.

**Example 2.2 continued** The behavior of component i is given in Fig. 2 for i ∈ {1, . . . , 5}. For every component i we put Qᵢ⁰ = Qᵢ. The induced global transition system is called T(5). For example in the global state (p₁’₁, p₁’₂, p₂’₁, p₂’₂, p₅’₁) a transition labeled with {d₁, f₁} is enabled. Our example system Sys₅ := (CS₅, IM₅, {Tᵢ}ᵢ∈K₅) is now completely specified.

**Remark 2.5** In what follows, we often mention Sys = (CS, IM, {Tᵢ}ᵢ∈K). It is understood that CS = (K, {Aᵢ}ᵢ∈K), IM = (C, Comp), Tᵢ = (Qᵢ, Aᵢ, −ᵢ, Qᵢ⁰) for i ∈ K, and T are given as above. Usually we will display the local transition systems graphically. If not explicitly stated otherwise the local initial states will be marked by an ingoing arrow which we will omit for component i whenever Qᵢ⁰ = Qᵢ.

### 3 Liveness in Interaction Systems

**Remark 3.1** From now on for all i ∈ K we will assume that Tᵢ has the property that every state offers at least one action.

In order to define liveness we first need a notion of (global) deadlock-freedom. Note that in [25] we also investigated a notion of local deadlock of a subset K’ ⊆ K of components which describes a situation where in the current state every component in K’ needs the cooperation of at least one other component in K’ which in turn does not offer the needed ports.

**Definition 3.2** Let Sys be an interaction system.

(i) Let q ∈ Q. q is reachable in Sys if there is a sequence q⁰ α₀ q¹ α₁ . . . αₙ⁻¹ q such that q⁰ ∈ Q₀ and αᵢ ∈ C ∪ Comp for all 0 ≤ i ≤ n − 1.
(ii) \( Sys \) is called \textit{deadlock-free} if for every reachable state \( q \) there exists \( \alpha \in C \cup Comp \) and \( q' \in Q \) such that \( q \xrightarrow{\alpha} q' \).

This definition is justified by the fact that maximal as well as complete interactions are independent of the environment. They do not have to wait for any other components and can be performed immediately. Started in a global initial state a deadlock-free system may always proceed with some maximal or complete interaction. Thus cyclic waiting involving all components cannot occur, whereas two or more components may be engaged in a local deadlock.

\textbf{Example 2.2 continued} It is easy to see that \( Sys_5 \) is deadlock-free. We have \( Q_i^0 = Q_i \) for all components which means that every global state is reachable. Therefore one has to show that every global state offers at least one maximal interaction which boils down to a simple case distinction.

In deadlock-free systems runs always exist, where a run is simply an infinite thread of execution starting in a reachable state of the system.

\textbf{Definition 3.3} Let \( Sys \) be a deadlock-free interaction system and let \( q \in Q \) be a reachable state. A \textit{run} of \( Sys \) is an infinite sequence \( \sigma : q = q^0 \xrightarrow{\alpha_0} q^1 \xrightarrow{\alpha_1} q^2 \ldots \) with \( q^l \in Q \) and \( \alpha_l \in C \cup Comp \) for all \( l \in \mathbb{N} \).

Let \( i \in K \) be a component and let \( \sigma \) be a run of \( Sys \). If there exists \( l \) such that \( i \) participates in \( \alpha_l \) we say that \( i \) \textit{participates in} \( \sigma \).

Now we can define when a set of components \( K' \subseteq K \) is live. Basically this is the case if for any point of time no matter how the system behaves some component in \( K' \) will eventually participate in some interaction. From now on we identify singleton sets with their element if it is convenient to do so.

\textbf{Definition 3.4} Let \( Sys \) be a deadlock-free interaction system and let \( K' \subseteq K \) be a nonempty set of components. We say that \( K' \) \textit{is live} in \( Sys \) if for every run \( \sigma \) of \( Sys \) there exists some \( i \in K' \) such that \( i \) participates in \( \sigma \).

\textbf{Remark 3.5} It should be noted that this notion of liveness applied to a component \( i \) is the same as requesting that this component should participate infinitely often in a run because runs may start in any reachable state.

This notion of liveness is different from the one introduced for Petri nets [8] that corresponds to our notion of local progress of a component \( i \) in interaction systems [14], which means that at any point in any run we may proceed in such a way that component \( i \) will participate. Clearly liveness of \( i \) implies local progress but not vice versa. A general form of liveness-properties for a component system as a whole is defined in [7]. The questions referring to single components that we are interested in cannot be directly formulated and investigated in [7] because the identity of a component may be lost in the system which means that it is not meaningful to consider liveness of a component.

If \( i \) is live in \( Sys \) any set of components containing \( i \) is also live. The converse does not hold: even if \( K' \) is live in \( Sys \) there does not need to be any \( i \in K' \) that is live. Also note that for \( K' = K \) liveness follows from deadlock-freedom.
4 Deciding Liveness is NP-Hard

We will show that deciding liveness in interaction systems is NP-hard by reducing the question whether a formula $F$ in 3-KNF [11] is not satisfiable to the question of deciding whether a certain component $\kappa$ is live in a certain deadlock-free system. Note that the reduction technique only has to be slightly adapted in the case of I/O-automata [23] yielding an analogous result for this formalism.

The idea of the reduction is as follows. Each clause of $F$ will be represented by one component as will be each literal of every clause. Other than that there is one component $\kappa$ that is live if and only if $F$ is not satisfiable. The clause components only have one state other than the starting state. This state represents an evaluation of the clause to true. The literal-components have a starting state from which two states representing the evaluation of the literal to true respectively false. The choice of the connector set will then make sure that all literals can only be set consistently. That means if one variable is set to a certain value all literals with the same variable must be set appropriately. Then it is clear that a global state where for each clause there is one literal-component that is in its state representing true can be reached if and only if $F$ is satisfiable. This means all clause-components can move to their respective true state if and only if $F$ is satisfiable. Then the choice of the connectors will ensure that it is only possible to start a run not involving $\kappa$ in such a state.

Let $F = k_1 \land \ldots \land k_n$ with $k_i = (l_{i,1} \lor l_{i,2} \lor l_{i,3})$ be a propositional formula in 3-KNF, where $l_{i,1}$, $l_{i,2}$, and $l_{i,3}$ are literals. For $l$ a literal let $\text{var}(l)$ denote the variable occurring in $l$. Let $\text{var}(F) := \{ \text{var}(l_{i,j}) | 1 \leq i \leq n, 1 \leq j \leq 3 \}$ denote the set of variables occurring in $F$.

We construct a deadlock-free interaction system $\text{Sys}(F)$ with component-set $K(F)$ containing a component $\kappa$ such that

$$(F \notin 3\text{-SAT}) \Leftrightarrow (\kappa \text{ is live in } \text{Sys}(F))$$

where 3-SAT is the set of satisfiable formulas in 3-KNF. Besides the component $\kappa$ we represent each clause $k_i$ by a component $(i,0)$ and each literal $l_{i,j}$ by a component $(i,j)$. We define

$$\text{Sys}(F) := \left( CS(F), IM(F), \{ T_{i,j} \}_{j=0,\ldots,3} \cup \{ T_\kappa \} \right)$$

where the components and their port sets are given by:

$$K(F) := \{(i,j) | 1 \leq i \leq n, 0 \leq j \leq 3 \} \cup \{ \kappa \}$$

$$A_{(i,0)} := \{ \text{true}_i, \text{SAT}_i \} \text{ for } 1 \leq i \leq n$$

$$A_{(i,j)} := \{ \text{set}_1(i,j), \text{set}_0(i,j), \text{true}_c(i,j), a_{(i,j)} \} \text{ for } 1 \leq i \leq n, j \neq 0$$

$$A_\kappa := \{ a_\kappa \}$$

We define the following connectors. First we have:

$$\text{sat} := \{ \text{SAT}_1, \ldots, \text{SAT}_n \}$$
Next we define:

\[ \text{set}_1 x := \{ a_\kappa, \text{set}_1 (t_{i_1,j_1}), \ldots, \text{set}_1 (t_{i_m,j_m}) \} \]

and

\[ \text{set}_0 x := \{ a_\kappa, \text{set}_0 (t_{i_1,j_1}), \ldots, \text{set}_0 (t_{i_m,j_m}) \} \]

where \( x = \text{var} \left( t_{i_1,j_1} \right) = \ldots = \text{var} \left( t_{i_m,j_m} \right) \) and there is no other literal \( l \) with \( x = \text{var} \left( l \right) \). Other than that we set:

\[ t_{i,j} := \{ a_\kappa, \text{true} (i,j), \text{true}_i \} \]

\[ c_a := \{ a_\kappa \} \cup \{ a_{(i,j)} | 1 \leq i \leq n, j \neq 0 \} \]

We set

\[ C := \{ \text{sat} \} \cup \{ c_a \} \cup \bigcup_{x \in \text{var} (F)} \{ \text{set}_1 x, \text{set}_0 x \} \cup \bigcup_{1 \leq i \leq n, j \neq 0} \{ t_{i,j} \} \]

and choose \( \text{Comp} \) to be the empty set.

The local transition system for \( \kappa \) is given in Fig. 3 (a). \( T_{(i,0)} \) is given in Fig. 3 (b) and \( T_{(i,j)} \) for \( j \neq 0 \) and \( t_{(i,j)} \) a positive (resp. negative) literal is given in Fig. 3 (c) (resp. (d)).

![Fig. 3. The local transition systems for the components of Sys (F)](image)

**Theorem 4.1** Let \( \text{Sys} (F) \) be defined as above.

(i) Going from \( F \) to \( \text{Sys} (F) \) there is no exponential blow-up in notation.

(ii) \( \text{Sys} (F) \) is deadlock-free.

(iii) \( F \) is not satisfiable if and only if \( \kappa \) is live in \( \text{Sys} (F) \).

The proof can be found in the technical report [28]. The following example illustrates the idea behind the reduction of the above theorem. For a satisfiable formula \( F \) it will be shown how a run in \( \text{Sys} (F) \) can be found in which \( \kappa \) only participates finitely many often. Let \( F = \left( x_1 \lor x_2 \lor x_3 \right) \land \left( \overline{x_1} \lor \overline{x_2} \lor \overline{x_3} \right) \land \left( x_1 \lor \overline{x_2} \lor \overline{x_3} \right) \). \( F \) is satisfiable, namely \( v (F) = 1 \) for \( v (x_1) = 1, v (x_2) = 1, v (x_3) = 0 \).

Consider \( K (F) = \left\{ (1,0), (1,1), (1,2), (1,3), (2,0), \ldots, (3,3), \kappa \right\} \) and \( \text{Sys} (F) := \left( \text{CS} (F), \text{IM} (F), \{ T_{(i,j)} \}_{j=0 \ldots 3} \cup \{ T_{\kappa} \} \right) \) as above. The evaluation \( v \) given above yields the following path starting in \( q^0 \) ending in a state where the connector \( \text{sat} \) can repeatedly be applied resulting in a run as above.

\[ \sigma := q^0 \xrightarrow{\text{set}_1 x} q^1 \xrightarrow{\text{set}_1 x} q^2 \xrightarrow{\text{set}_0 x} q^3 \xrightarrow{t_{(1,1)}} q^4 \xrightarrow{t_{(2,2)}} q^5 \xrightarrow{t_{(3,3)}} q^6 \xrightarrow{\text{sat}} q^6 \xrightarrow{\text{sat}} \ldots \]
In the first step component \((1,1)\) moves to its true-state and \((2,1)\) and \((3,1)\) move to their respective false-state representing the evaluation of \(x_1\) to 1. Analogously the other six literal-components change their state according to \(set1_{x_2}\) and \(set0_{x_2}\). In steps four to six components \((1,0)\), \((2,0)\), and \((3,0)\) move to their true-state together with \((1,1)\), \((2,2)\), respectively \((3,3)\). Note that in the fifth step \(t_{(2,3)}\) could also have been performed because \(q^5_{(2,3)} = q^6_{(2,3)}\) as well. Then \(sat\) is enabled in \(q^6\).

5 Characterizing Liveness of a Set of Components

In this section we consider a (not necessarily finite) deadlock-free interaction system. We present a characterization of all subsets \(K' \subseteq K\) that are live. The benefit of this characterization is that it can be used in combination with the sufficient criterion for liveness that we will present in Sect. 6 for cases where the criterion alone does not imply liveness.

Definition 5.1 Let \(K' \subseteq K\) be a subset of components. Let \(excl (K') := \{\alpha \in C \cup Comp | \exists i \in K' : i(\alpha) = \emptyset\}\) denote the set of maximal or complete interactions in which no component from \(K'\) participates.

Definition 5.2 Let \(Sys\) be a deadlock-free interaction system and let \(K' \subseteq K\) be a non-empty subset of components. We define

\[
\bar{K}' := \{k \in K | \exists \alpha \in excl (K') : k(\alpha) \neq \emptyset\}.
\]

Further we define the following labeled transition system

\[
\bar{T} := (\bar{Q}, excl (K'), \rightarrow)
\]

where \(\bar{Q} := \prod_{k \in K'} Q_k\) and \(\rightarrow \subseteq \bar{Q} \times excl (K') \times \bar{Q}\) is the transition relation which is defined as follows: for any two states \(\bar{p}, \bar{q} \in \bar{Q}\) and any interaction \(\alpha \in excl (K')\) \(\bar{p} \xrightarrow{\alpha} \bar{q}\) \(\iff \forall i \in K' \exists \bar{q}_i \in Q_i \text{ if } i(\alpha) \neq \emptyset\) and \(\bar{p}_i = \bar{q}_i\) otherwise.

\(\bar{K}'\) contains those components that participate in at least one maximal or complete interaction not involving any component from \(K'\). We clearly have \(K' \subseteq K \setminus \bar{K}'\). \(\bar{Q}\) can be understood as the projection of \(Q\) to \(\bar{K}'\) where we only allow those transitions labeled with some \(\alpha \in excl (K')\).

Theorem 5.3 Let \(Sys\) be deadlock-free and let \(K' \subseteq K\). \(K'\) is live in \(Sys\) if and only if \(\bar{T}\) does not contain any infinite path starting in a state \(\bar{q}\) for which there exists \(q' \in \prod_{k \in K \setminus K'} Q_k\) such that \((\bar{q}, q')\) is reachable in \(Sys\).

For finite systems the characterization amounts to cycle detection and involves (partial) global state space analysis in the worst case and its usefulness to detect liveness depends on the size of \(\bar{K}'\). If \(\bar{K}'\) contains very few elements usually it will not be helpful to analyze \(\bar{T}\), because its number of states may still be exponential in the number of components. But even in this case the characterization can be helpful when the set \(excl (K')\) is small and \(\bar{T}\) sparse. In the extreme case every maximal or complete interaction involves some component from \(K'\) and \(excl(K')\) is empty. Then it is clear anyway that \(K'\) is live.
6 Testing Liveness

In this section we present a hybrid algorithm that tests liveness of a subset $K'$ of components. The algorithm is based on a sufficient condition and the characterization given in the previous section applied to a subsystem.

The condition that has to be checked comes down to a reachability analysis in a graph where the components are the nodes. The graph is constructed by checking certain dependencies between pairs of components that can be checked by investigating the local transition systems only. Therefore the graph can be constructed in time polynomial in the number of components and the size of the local transition systems such that the criterion avoids the investigation of the global state space.

In this section we always assume that $Sys$ is a deadlock-free interaction system with a finite set of components $K$ and finite port sets $A_i$.

**Definition 6.1** Let $Sys$ be an interaction system as above and let $j \in K$ be a component.

(i) Let $A'_j \subseteq A_j$ be a subset of actions of $j$. $A'_j$ is inevitable in $T_j$ if only finitely many transitions labeled with $a_j \in A_j \setminus A'_j$ can be performed in $T_j$ before some action from $A'_j$ must be performed.

(ii) Let $\Lambda \subseteq I(C)$ be a nonempty set of interactions and let $j \in K$ be a component. We define $\Lambda \setminus [j] := A_j \cap \bigcup_{\alpha \in \Lambda} \alpha$ the set of ports of $j$ that participate in one of the interactions of $\Lambda$.

A subset of actions of a component is inevitable if on every infinite path in the transition system of that component there are infinitely many transitions that are labeled with some action from that set. The second part of the definition gives a sort of a projection-operator that yields those actions of component $j$ that participate in one of the interactions in $\Lambda$.

In the following we define the graph $G := (K, E)$. The set of edges is given by the union $\bigcup_{m\geq 0} E_m$ where the sets $E_m \subseteq K \times K$ are defined inductively.

**Definition 6.2** Let

$$E_0 := \{(i, j) | A_j \setminus (excl (i) [j]) \text{ is inevitable in } T_j\}$$

and define $E_{n+1}$ inductively as follows:

$$E_{n+1} := \{(i, j) | A_j \setminus (excl (Reach^n (i)) [j]) \text{ is inevitable in } T_j\}$$

Here $Reach^n (i) := \{ j | j \text{ is reachable from } i \text{ in } (K, \bigcup_{m=0}^{n} E_m) \}$.

Let $E := \bigcup_{m=0}^{\infty} E_m$ and define $G := (K, E)$.

Note that $excl (i) [j]$ contains those ports of $j$ that occur in some maximal or complete interaction that does not involve $i$. Thus $A_j \setminus (excl (i) [j])$ is the set of ports of $j$ that only occur in maximal or complete interactions that also involve $i$. Then the intuitive meaning of an edge $(i, j) \in E$ is that $j$ can only participate in finitely many global steps before $i$ also has to participate in such a step.
Theorem 6.3 Let \( K' \subseteq K \) be a set of components. If all components in \( K\setminus K' \) are reachable from \( K' \) in \( G \) then \( K' \) is live in \( Sys \).

Example 2.2 continued We have already explained why \( Sys_5 \) is deadlock-free.

Figure 4 depicts the part of \( G \) only containing the edges from \( E_0 \). The only component that is reachable from 1 is 2, but it can be seen that 3 respectively 5 can only advance finitely many times before component 4 has to participate in some step. Amongst others the edge \((1,4)\) is added to \( E_1 \) in the next iteration step. Hence all components are reachable from component 1 in \( (K,\bigcup_{m=0}^{1} E_m) \) and therefore also in \( G \). Then liveness of component 1 follows from Theorem 6.3 above.

In the following we present the algorithm that tests a given set \( K' \) of components for liveness. The algorithm first applies the sufficient condition of Theorem 6.3 to \( K' \) which causes polynomial cost. If the condition of the criterion is not satisfied then the algorithm applies the characterization of Theorem 5.3 to the set \( Reach \) of components that can be reached in \( G \) from \( K' \). Note that the algorithm requires a system where each global state is a potential starting state. It can easily be adapted to the general case. In this case the algorithm reports “don’t know” if it detects a cycle in \( \overline{Q} \) in the second part of the algorithm as we do not know if the cycle is reachable in the global system. A further refinement could use the techniques of [25] to find out whether the cycle is indeed reachable from a global starting state.

Lemma 6.4 Algorithm 1 terminates and correctly tests a given set \( K' \) of components for liveness. If it yields a positive answer in line 20 the total cost is polynomial in the size of the input.

The else-block starting in line 21 causes cost in the size of \( \overline{Q} \) which is exponential in \( |K'\setminus Reach| \) in the worst case.

Proof. The correctness of the algorithm follows from Theorems 5.3 and 6.3.

Each iteration of the loop in line 5 causes cost polynomial in \( |K|, |C \cup Comp|, \) and \( \sum_{j \in K} |T_j| \), and this iteration will be performed at most \( |K|^2 \) times. Note that the test for inevitability in \( T_j \) from line 9 only causes cost polynomial in \( |T_j| \). It can be performed by checking whether the system \( T'_j \) obtained by deleting all edges labeled with a port from \( A_j \setminus (exc (Reach (i)) [j]) \) does not contain a cycle. The cost of the else-block is dominated by the cost for the computation of \( \overline{Q} \) and the search for a cycle in \( \overline{Q} \).

Example 2.2 continued From the above explanations it is clear that Algorithm 1 launched with \( Sys_5 \) and \( K' = \{1\} \) terminates in line 20 detecting liveness of component 1.
Algorithm 1 LIVENESS $(Sys, K')$

Require: $Sys = (CS, IM, \{T_i\}_{i \in K})$ deadlock-free, $T_i = (Q_i, A_i, \rightarrow_i, Q_i), K' \subseteq K$

Ensure: TRUE if $K'$ is live, FALSE otherwise

1: $V \leftarrow K$, $E \leftarrow \emptyset$, $numberEdges \leftarrow 0$
2: for all $i \in K$ do
3: \hspace{1em} $Reach(i) \leftarrow \{i\}$
4: end for
5: repeat
6: \hspace{1em} $numberEdges \leftarrow |E|$
7: \hspace{2em} for all $i \in K$ do
8: \hspace{3em} for all $j \in K \setminus \{i\}$ do
9: \hspace{4em} if $A_j \setminus (excl(Reach(i))[j])$ is inevitable in $T_j$ then
10: \hspace{5em} $E \leftarrow E \cup \{(i, j)\}$
11: \hspace{4em} end if
12: \hspace{3em} end for
13: \hspace{2em} end for
14: \hspace{1em} for all $i \in K$ do
15: \hspace{2em} $Reach(i) \leftarrow \{j \in K | \exists$ path from $i$ to $j$ in $(V, E)\}$
16: \hspace{1em} end for
17: until $numberEdges = |E|$
18: $Reach \leftarrow \bigcup_{i \in K'} Reach(i)$
19: if $Reach = K$ then
20: \hspace{1em} return TRUE \{K' is live\}
21: else
22: \hspace{1em} compute $excl(Reach)$
23: \hspace{1em} compute $\overline{Reach}$
24: \hspace{1em} $Q \leftarrow \prod_{k \in \overline{Reach}} Q_k$
25: \hspace{1em} if $\not\exists$ cycle in $\overline{Q}$ then
26: \hspace{2em} return TRUE \{K' is live\}
27: else
28: \hspace{2em} return FALSE \{K' is not live\}
29: end if
30: end if

In the following example the else-block will be applied and yields liveness of component 1 if the algorithm is launched with the given system and $K' = \{1\}$.

Example 6.5 Consider a system consisting of the four components 1, 2, 3, and 4 whose behavior is given by Fig. 5 where it is understood that the port sets of the components are given implicitly by the transition systems. For every component $i$ we put $Q_i^0 = Q_i$.

We define $C = \{\{a_1, b_1\}, \{b_2, d_1\}, \{d_2, e_1\}, \{d_1, e_1\}, \{d_2, e_2\}\}$ and set $Comp = \emptyset$.

It is easy to check that the global system is deadlock-free. We apply Algorithm 1 to test liveness of $K' = \{1\}$. The algorithm finds $Reach = \{1, 2\} \neq K$. The else-block will therefore be applied to $\{1, 2\}$. Because $excl(\{1, 2\}) = \{\{d_2, e_1\}, \{d_1, e_1\}, \{d_2, e_2\}\}$ it is clear that $\overline{Reach} = \{3, 4\}$. There is no cycle in $Q$. 
Thus the algorithm affirms liveness of 1.

Had we used Theorem 5.3 directly to test liveness of component 1 we would have got $\bar{K}' = \{2, 3, 4\}$, i.e. a larger transition system would have had to be investigated.

The combination of the criterion from Theorem 6.3 with the characterization of Theorem 5.3 in Algorithm 1 may yield greater benefit if applied to larger examples.

7 Conclusion and Related Work

This work treats various aspects concerning liveness. The contribution is threefold:

(i) We showed that deciding liveness in interaction systems is NP-hard by reducing 3-SAT to resolving the question whether a certain component is live in an interaction system\(^4\).

(ii) We presented a characterization for liveness.

(iii) We established a sufficient criterion for liveness that can be tested in polynomial time. In Algorithm 1 we combined this criterion with the characterization mentioned above.

Liveness has been treated in other settings for component-based systems. For example in the channel-based approach of [7] general liveness-properties have been investigated although no procedures that can be used to test liveness are provided. Moreover Petri-nets have been used for component-based modeling [3]. For Petri-nets a notion of liveness has been investigated in detail [8] and depending on the considered class of Petri-nets the complexity of deciding this property is presented. This notion of liveness corresponds to our notion of local progress [14]. Liveness has been also discussed in [12, 30].

The problem of repeated reachability of a set of accepting states of a Büchi automaton has been dealt with in depth in the context of model checking of LTL formulae, see e.g. [20]. This problem corresponds to our condition in Theorem 5.3. However we propose an alternative idea. We exploit local information about the components and derive a criterion in Theorem 6.3, that guarantees liveness without considering (parts of) the global state space in any form. Algorithm 1 proceeds as follows: only if the criterion fails to establish liveness we apply cycle search in the projection of the global state space to $\text{Reach}$. To implement this part of the algorithm efficiently we could apply the ideas presented in [20].

Another approach to establish properties of systems while avoiding global state space analysis is to exploit compositionality. In [14] we defined a composition oper-

\(^4\) Work in progress suggests the conjecture that the problem is even PSPACE-hard.
ator for interaction systems and presented some first conditions under which properties of subsystems are preserved under composition.

In [14,25,24] we formulated and investigated further properties of interaction systems, in particular global and local deadlock, progress, and robustness. Currently we are enhancing the model by introducing probability. It is then possible to make statements such as “with probability $p$ no deadlock will arise”.

References


A Component Model for Trustworthy Real-Time Reactive Systems Development

Vasu Alagar\textsuperscript{1,2} Mubarak Mohammad\textsuperscript{1,3}

Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada H3G 1M8

Abstract
In this paper a formal description of trustworthy real-time reactive components is given. Component templates are defined and components are defined as instances of a template. A template consists of a structural part and a contract part. All components of a template share the structural and contractual properties while differing in their architectural descriptions and implementations. The behavior of a component is behavior of the architecture associated with the component and it is generated dynamically. Safety and security are identified as the two essential elements of trustworthiness. A rule for composing components is formalized and it is shown that the composition rule preserves trustworthiness property. A brief comparison with SOFA 2.0 model and a discussion of our current research directions are included.

Keywords: Component, RTRS, Safety, Security, Composition

1 Introduction
In this paper we determine safety and security as the two criteria of trustworthiness for RTRS and propose a formal approach to develop a trustworthy system. The development methodology is based on component technology. The goal is to formalize a trustworthy component and define a composition that preserves trustworthiness properties.

Reactive systems belong to the class of computer systems that maintain continuous interaction with their environment through stimulus and response. The class of reactive systems in which the reaction to a stimulus may be strictly regulated by timing constraints is called real-time reactive systems (RTRS). This type of systems has become an essential part of the technological infrastructure of modern societies. It is being used for a long time in safety critical missions, many directly affecting the environment and lives of people. Such systems are required to be trustworthy due to its complexity and the critical contexts in which they operate.

\textsuperscript{1} This research is supported by a Research Grant from Natural Sciences and Engineering Research Council of Canada (NSERC)
\textsuperscript{2} Email: alagar@cse.concordia.ca
\textsuperscript{3} Email: mmoham@cse.concordia.ca

This paper is electronically published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Trustworthiness is the system property that denotes the degree of user confidence that the system will behave as expected [10,28]. In order to trust the system, the trustworthiness credentials of the system should be examined before granting the trust. The challenge in building provably trustworthy RTRS lies in combining safety and security requirements. Research in verifying safety and security properties have progressed in parallel, due to the finding that safety and security can’t be formally specified and verified together in any one formal method [23,32,19]. We suggest the use of component-based development (CBD) [31] as a basis for a unified formal model for the specification and verification of safety and security properties of RTRS.

CBD is the type of software engineering development in which systems are built by constructing units, called components, that perform simple tasks, and assembling them to create composite components that perform complex tasks. Some potential benefits of applying CBD for RTRS include complexity reduction, time and cost savings, predictable behavior, and productivity increase [10].

In the literature, there is an inconsistency in defining components between un-timed and real-time component models. On the one hand, there is a common agreement [10,27,31] that component specification should include both structural and behavioral description. Structural description includes, but not limited to, specifying interfaces, connectors, and composition. These are central concepts in CBD. An interface defines access points to the services provided/requested by components. A connector is a special component that defines the communication between two components. Composition allows building systems by connecting existing components in such a way that preserves their essential properties. On the other hand, in current component-based models for real-time systems, a component is modeled as timed automata [5], duration automata [17], extended finite-state machine [14], and finite-state process [13]. Such modeling techniques focus only on the behavioral aspect and makes no distinction between object-oriented and component-based models. Other real-time component models [9] define "flat" components with restrictive execution model as opposed to hierarchical components found in general un-timed component-based models [4,26]. This shows that there is a gap between un-timed and timed component models. Moreover, all the component models presented focus mainly on safety and liveness properties and don’t provide a formal foundation for the specification and composition of secure components.

We propose a component model that collectively addresses the requirements of RTRS and credentials of trustworthiness. A central challenge in building trustworthy systems [25,28] using CBD method is composing trustworthy components so that the composed component is trustworthy. The main contributions of this paper are (1) a definition of the requirements of a component model for developing trustworthy RTRS, (2) a formal definition for trustworthy hierarchical RTRS components, and (3) a compositional theory for composing components so that safety and security are preserved in the composition. To the best of our knowledge there seems to be no published work that has combined safety and security in a provably correct manner in the development of trustworthy systems.

2 Requirements of a Component Model for Trustworthy RTRS

In this section, we state the requirements of RTRS, elements of trustworthiness, and elements of a component-model. We discuss how CBD can be used to effectively implement
the requirements of RTRS and trustworthiness. The stated requirements form the basis for the formal definitions of our model that will be presented in subsequent sections.

2.1 Requirements of RTRS

Designing RTRS components is more complex than designing non-RTRS components. There are four main requirements that must be satisfied by RTRS under all circumstances [10]:

1. Timeliness: The correct behavior of RTRS depends not only on performing the intended functionality but also depends on the time at which certain functions finish. It is essential that system reactions always satisfy both the functional requirements and the timeliness requirements.

2. Simultaneous processing: In RTRS, many events can occur simultaneously. The behavior of the RTRS is not correct if the system reacts to some stimuli and ignores others.

3. Predictability: For every stimulus there is precisely one kind of reaction. This makes the behavior of RTRS predictable.

4. Dependability: When the environment of the RTRS requests a service from the system, it trusts that the system will react as expected by it. The predicted reaction should satisfy the functional and non-functional requirements expected by the environment. Dependability is defined as the ability to deliver trusted services [3]. In the literature [30], the terms dependability and trustworthiness are used interchangeably.

2.2 Elements of trustworthiness

There is a general agreement [3,28,30] that trustworthiness involves achieving availability, reliability, safety, and security. Below we discuss these elements and point out the consequences of the misbehavior of RTRS in their absence.

1. Availability is the quality of operation in which there is no unforeseen or unannounced disruption of service. A temporary outage of service may not cause big problems for a non-RTRS. The required services can be requested at a later point of time when the system becomes available. However, any service outage for RTRS will violate the requirements of timeliness and may lead to catastrophic consequences.

2. Reliability is the quality of continuing to provide correct services [3]. A RTRS is expected to have a high degree of reliability due to the critical contexts it operates in.

3. Safety is the quality of the operational behavior of the system in which no system action that may lead to catastrophic consequences will happen. Safety includes a set of properties that describe the correct behavior of the system. Safety properties are system specific. Failure to satisfy safety properties could directly affect the availability and reliability of the system due to the incorrect behavior. Hence, ensuring safety properties is very critical for RTRS.

4. Security denotes the acceptable quality of the system before, during, and after every operation. Authorization to request (provide) services, integrity of information provided to clients of components, and confidentiality of stored and communicated information are some of the important aspects in ensuring security of the system. Assuring integrity of data within each component, and ensuring confidentiality of data stores within each
component are issues that we do not address in this paper. Ensuring the integrity of data communicated by a component to its client is part of correctness issue. Enforcing that only authorized clients request and receive services from a component is vital to ensure confidentiality. Authorization in the system is based on user identity [24,28]. A user represents an entity, may be human or system component to whom services are provided and on whose behalf services are requested. Security violations, which in our case is the unauthorized usage of system resources, will directly affect the availability, reliability, and safety of the system.

From the above discussion it can be concluded that safety and security are essential prerequisites for ensuring availability and reliability. Therefore we conclude that the essential credentials for ensuring trustworthiness of RTRS are safety and security. This is the justification for focusing only on these two aspects in this paper.

### 2.3 Elements of a component model

This section is a brief introduction to the elements of a component model. Detailed formal definitions are presented in Section 3. Figure 1 shows a component template composed of a structure part and a contract part. The structure of a template is an abstract external black-box view, called frame, and its internal hierarchical structure, called architecture. The frame consists of the interface types, where each interface type is associated with a set of services. A service may be parameterized with data types. An architecture is a collection of connector types, an abstract view of the tie-ins between interface types. The contract part of the template states the properties required of the system for which the structure is a blueprint.

A component is an instance of a component template. Every component instantiated from a template has one instance of the structure part defined for the template. The frame of the component is a set of interfaces, where each interface belongs to exactly one interface type in the template frame. An architecture instance corresponding to a component frame is an instance of the architecture corresponding to the frame in the template, having as many

<table>
<thead>
<tr>
<th>Data Parameters</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Types</td>
<td>Frame</td>
</tr>
<tr>
<td>Architecture Types</td>
<td>Connector Types</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Constraints</th>
<th>Data Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Security</td>
<td>User Identity</td>
</tr>
<tr>
<td>Reactivity</td>
<td>Time Constraints</td>
</tr>
</tbody>
</table>
instances of connector types as are required for linking the interfaces in the component. A component’s contract constrains the communication pattern at its interfaces and is faithful to the contract part in its template.

A component frame can contain multiple interfaces of an interface type. This enables a RTRS component to interact in a similar way with several other components, receive/send many interactions at the same time, and perform simultaneous processing. Through contracts we can specify the requirements of timeliness, reactivity, safety, and security at different interfaces of the same type. At each interface of a component frame, stimuli and reactions can be regulated, restricted and filtered. Regulating reactions at an interface ensures timeliness, restricting reactions promotes safety, and filtering services at interfaces can protect the component from unauthorized use. By scrutinizing the nature of a stimulus and the response to it at an interface, a request for service can be either authenticated or denied. The data parameters carried by a response at an interface of the component frame can be validated both for integrity and confidentiality. Moreover, obligations associated with responses can be verified at frame interfaces.

Due to space limitation we discuss only the structure and contract formal definitions in this paper. In [2] we discuss the behavior specification of a component, which can be generated automatically from the structure and contract specification.

3 Specifications of Trustworthy Components

In this section we discuss three issues. These are (1) basic definitions that lead to a formal definition, (2) a formal definition of untimed components, and (3) a formal definition of RTRS components that exhibit safe and secure behavior.

3.1 Basic definitions

We use the template notation [26], although conceptually and semantically our definition of frame and architecture are different.

• A component requests/provides a set of services. We assume a finite non-empty set of services Σ, in which every service is either a stimulus or a response. A service cannot be both a stimulus and a response for a component.

• An interface type is an enumerated type whose elements are services from Σ. An interface is an instance of an interface type, it inherits the services listed in the type definition. Two interface types P and Q are compatible if and only if for every service s : P there exists exactly one service s : Q such that s and s are complementary. That is, in designing component interaction both s and s will be assigned to occur simultaneously at component interfaces of interacting components. We define the predicate Compatible(P, Q) which is true if and only if P and Q are compatible.

• A frame is a black-box with a finite (non-empty) set of interface types, such that no two pairs of interface types of the frame are compatible.

• A Connector type is a tuple (L, M), where L is a link specification and M is a communication style specification. The link specification L is a tuple (F1, P, F2, Q), where P is an interface type of frame F1, Q is an interface type of frame F2 and (P, Q) are compatible interface types. This abstraction allows us to define composition of frames.
We use the notation $P \circ F_1 \square Q \circ F_2$, instead of tuple notation, to introduce a connector type link. The communication style $M$ specifies the type of communication used by the connector to deliver services. There are a number of common communication styles to choose from [29]: procedure call, message passing, remote procedure calls, etc. This paper focuses on the link specification only.

- A frame has an associated architecture, an abstract implementation of it. In general, more than one architecture may correspond to a frame as different implementation versions. If the specification of the architecture of a frame is given by a program implementation then the frame is primitive. A non-primitive component is a composite frame, which includes two or more frames. The architecture specification of a composite frame is a grey-box view of the frame. It includes the black-box definitions of the constituent frames and a specification of connectors used to compose them.

- A template, also called component type, is a tuple $CT = (F, A)$ where $F$ is a frame and $A$ is an architecture of frame $F$. We use the notation $CT_F$ and $CT_A$ to respectively denote the frame and architecture of the component type $CT$.

- A component is an instance of a component type. A component instance $C$ of type $CT = (F, A)$ is the tuple $C = (F', A')$, where $F'$, called component frame, has interfaces that are instances of interface types in the definition of $F$, and $A'$, called the component architecture, has connectors that correspond to the connector types in $A$. We use the notation $C_F$ and $C_A$ to respectively denote the frame and architecture of the component $C$. In general, if $C$ is an instance of $CT$ then $C_F$ is an instance of $CT_F$, and $C_A$ is an instance of $CT_A$. The component frame $C_F$ can have one or more interfaces that are instances of an interface type of $CT_F$. The aggregation of all interface instances in $C_F$, hereafter called interfaces of the instantiated component $C$, provides a black-box view of the component $C$. If $p$, an interface of $C_F$, is an instance of the interface type $P$ of frame $CT_F$, the interface $p$ inherits all the services in the type definition of $P$. That is, in the component $C$ services that are inherited by $p$ are either requests (stimulus) received at $p$ or provides sent out at $p$.

- A connector is an instance of a connector type. It implements the communication style specified in the connector type. We use the symbol $\bowtie$ to denote the link part of a connector. As an example, a connector link of type $P \circ F_1 \square Q \circ F_2$ is $p \bowtie f_1 \bowtie q \bowtie f_2$, if $f_1$ is an instance of $F_1$, $P$ is an interface type in $F_1$, $p$ is an instance of $P$, $f_2$ is an instance of $F_2$, $Q$ is an interface type in $F_2$, and $q$ is an instance of $Q$.

The template definition enables the dynamic configuration of components at instantiation time. This is because the frame definition consists only of interface types and the architecture definition consists only of connection types. When a component is instantiated, multiple instances of each interface type and connection types can be created and linked to create different versions of the frame and the architecture. Different notations are given to the type definitions of: frame($CT_F$), interface($P$), architecture($CT_A$), and connection link($P \circ F_1 \square Q \circ F_2$) than those given to the instances of those entities: $C_F$, $p$, $C_A$, and $p \bowtie f_1 \bowtie q \bowtie f_2$ respectively. This enables the formal specification of dynamic configuration.
3.2 Un-timed components - a formal definition

An un-timed component requests/provides services without being governed by time restrictions. In the rest of this section we simply refer to an un-timed component as component.

Services are modeled as events occurring at the interfaces of a component frame. Therefore, every element of $\Sigma$ is considered an actual event. A service request (stimulus) is an input event representing an information flow from outside the component to the inside. On the other hand, a service provision is an output event representing an information flow from inside the component to the outside. Input and output events are external events. Internal processing of services inside the component is done using internal events. Therefore, $\Sigma$ is divided into a set of input events $\Sigma_{input}$, a set of output events $\Sigma_{output}$, and a set of internal events $\Sigma_{internal}$. Formally, $\Sigma = \Sigma_{input} \cup \Sigma_{output} \cup \Sigma_{internal}$ and $\Sigma_{input} \cap \Sigma_{output} \cap \Sigma_{internal} = \emptyset$. Service requests and responses may include information carried by events. These information are modeled as data parameter values attached to events. We assume a finite set of data parameters $\Lambda$, in which every data parameter can be assigned to one or many events.

A component should be specified as a black-box entity to enable designers to reuse it without knowledge of its internal structure [10]. From the basics explained in Section 3.1 it is clear that we need to formally specify a component type $CT$.

Definition 3.1 Let $CT$ be a primitive component type. The specification of frame $CT_F$ is a tuple $< \Pi, \Sigma, \Lambda, \Xi, \sigma >$, where $\Pi$ is a finite non-empty set of interface-types such that $\forall P \in \Pi$, $\forall Q \in \Pi \bullet Compatible(P,Q)$, $\Sigma$ is a finite set of events, $\Lambda$ is a finite set of data parameters, $\Xi : \Sigma \rightarrow \mathbb{P} \Lambda$ is a function that associates with each event a set of data parameters, and $\sigma : \Pi \rightarrow \mathbb{P} \Sigma$ is a function that associates a finite non-empty subset of external events to each interface-type such that $\forall P, Q \in \Pi, \sigma(P) \cap \sigma(Q) = \emptyset$.

Definition 3.2 Let $CT$ be a composite component type. The specification of $CT_F$ is the specification of its constituent frames, each specified as in Definition 3.1. Compatible interface types of the frames are used to connect the frames and define connector types. Other non-compatible interface types form the set of interface types of $CT_F$. The specification of architecture type $CT_A$ is a collection of connector types, where each connector type link is of the form $P \oplus F_1 \oplus Q \oplus F_2$, where $F_1$ and $F_2$ are two frames in defining $CT_F$ and the interface type $P$ of $F_1$ is compatible with the interface type $Q$ of $F_2$.

Definition 3.3 A component frame $C_F$ is created from the template frame $CT_F$ by specifying for each interface type $P$ in $CT_F$ the number of interfaces ($\#P$) of type $P$ required in $C_F$. If $\#P = n$, we let $C_P = \{p_1, \ldots, p_n\}$ denote the interfaces created. A specification of $C_F$ is $< \Pi_I, \Sigma, \Lambda, \Xi, \sigma >$, where $\Pi_I = \bigcup_{P \in CT_F} C_P$. The $\sigma$ function is extended to interfaces: $\forall p \in P \bullet \sigma(p) = \sigma(P)$. A component architecture $C_A$ from the template architecture $CT_A$ is created by defining $n$ connectors in $C_A$ for each connector type in $CT_A$ if $n$ interfaces have been created in $C_F$ corresponding to the interface type(s) in the connector type.

Example 3.4 Let $CT_F$ be a composite frame whose constituent frames are $F_1 = < \Pi_1, \Sigma_1, \Lambda, \Xi_1, \sigma_1 >$, and $F_2 = < \Pi_2, \Sigma_2, \Lambda, \Xi_2, \sigma_2 >$, where $\Pi_1 = \{X, Y\}$, $\Pi_2 = \{Z, W\}$, $\Sigma_1 = \{e_1, e_2, e_3\}$, $\Sigma_2 = \{\overline{e}_1, \overline{e}_2, \overline{e}_3\}$, $\Lambda = \emptyset$, $\Xi_1(e_1) = \Xi_2(e_2) = \Xi_1(e_3) = \Xi_2(\overline{e}_1) = \Xi_2(\overline{e}_2) = \Xi_2(\overline{e}_3) = \emptyset$, $\sigma_1(X) = \{e_1\}$, $\sigma_1(Y) = \{e_2, e_3\}$, $\sigma_2(Z) = \{\overline{e}_1\}$, $\sigma_2(W) = \{\overline{e}_2, \overline{e}_3\}$. Interface types $X$ and $Z$ are compatible, as well as interface types $Y$ and $W$. Other interface types are not compatible.
and $W$ are compatible. An architecture type $CT_A$ for $CT_F$ is defined below:

$X \oplus F_1 \uplus Z \oplus F_2$

$Y \oplus F_1 \uplus W \oplus F_2$

**Example 3.5** Let $x_1, x_2 : X$, $y_1 : Y$, $z_1 : Z$, and $w_1, w_2 : W$. Let $F'_1$ and $F'_2$ be instances of $F_1$ and $F_2$ respectively. The specification of the composite component frame is given below:

$$F'_1 = < \Pi_{I_1}, \Sigma_1, \Lambda, \Xi_1, \sigma_1 >, F'_2 = < \Pi_{I_2}, \Sigma_2, \Lambda, \Xi_2, \sigma_2 >,$$

where $\Pi_{I_1} = \{x_1, x_2, y_1\}$, and $\Pi_{I_2} = \{z_1, w_1, w_2\}$, $\sigma_1(x_1) = \sigma_1(x_2) = \{e_1\}$, $\sigma_1(y_1) = \{e_2, e_3\}$, $\sigma_2(z_1) = \{\tau_1\}$, $\sigma_2(w_1) = \sigma_2(w_2) = \{\tau_2, \tau_3\}$.

Two possible instance architectures $C_{A_1}$ and $C_{A_2}$ of $CT_A$, are given below:

<table>
<thead>
<tr>
<th>$C_{A_1}$</th>
<th>$C_{A_2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_2 \oplus F'_1 \bowtie z_1 \oplus F'_2$</td>
<td>$x_1 \oplus F'_1 \bowtie z_1 \oplus F'_2$</td>
</tr>
<tr>
<td>$y_1 \oplus F'_1 \bowtie w_1 \oplus F'_2$</td>
<td>$y_1 \oplus F'_1 \bowtie w_2 \oplus F'_2$</td>
</tr>
<tr>
<td>the interfaces $x_1$ and $w_2$ are free.</td>
<td>the interfaces $x_2$ and $w_1$ are free.</td>
</tr>
</tbody>
</table>

Free interfaces $x_1$ and $w_2$ can be used to connect the component to other components.

By a component $C$ we mean the pair $(C_F, C_A)$. The architecture $C_A$ is an abstract implementation of the component frame $C_F$. The behavior of $C$ is the implementation behavior of $C_A$, and is observed at the interfaces of $C_F$. We define the behavior at an interface $p$ of $C_F$ as a set of sequences over $\sigma(p)$. The behavior of component $C$ is the arbitrary interleaving of sequences at the interfaces of $C_F$.

### 3.3 Timed reactive components (TRC) - a formal definition

A reactive component is a component that maintains continuous interaction with its environment through stimulus and response (reaction). A stimulus is an input event and a response is either an output or an internal event. In a timed system, timing information is associated with event occurrences. A timed reactive component (TRC) is a reactive component whose responses are governed by constraints of the following two types: (1) time constraints, and (2) data parameter constraints. First, a response $e$ can be constrained to happen, say at time $t_e$ within a time bound $[l_e, u_e)$. That is, $l_e \leq t_e < u_e$. Second, a response can be enabled or disabled using data parameter constraints. These constraints are logical expressions that evaluate to boolean values based on the values of data parameters carried by the stimulus. Hence, we extend the formal definition of the un-timed component frame to include reactivity and a finite set of constraints.

**Definition 3.6** The frame, architecture, and component definitions of a TRC are obtained by extending respectively the frame, architecture, and component definitions of un-timed system with the parameters $\Theta, \Gamma, \Omega$ as defined below:

$\Theta : \Sigma_{input} \rightarrow \Sigma_{output} \cup \Sigma_{internal}$ is a total function that associates a set of responses to each stimulus, $\Gamma$ is a finite set of timing constraints for the events in $\Sigma$, where each time constraint involves conjuncts of the form $(t(r) \rightarrow t(s)) \circ n$, where $t(.)$ is the time function for event occurrences, $s \in \Sigma$ is a stimulus, $r \in \Sigma$, $r \in \Theta(s)$ is a response to $s$, $\circ \in \{<, \leq, =, \geq, >\}$.
and $n : \mathbb{N}$, and

$\Omega$ is a finite set of constraints for the data parameters associated with the events in $\Sigma$, where each data constraint of an event $s \in \Sigma$ is a predicate defined over the values of the data parameters associated with $s$. If $s$ has $n$ number of responses in $\Theta(s)$ then there must be $n$ number of mutually exclusive data constraints defined over the data parameters of $s$. This ensures that the responses of $s$ are mutually exclusive. A TRC frame $CT_F$ is written as $< \Pi, \Sigma, \Lambda, \Xi, \sigma, \Theta, \Gamma, \Omega >$.

**Definition 3.7** For a TRC component $C$ with frame $C_F =< \Pi_f, \Sigma, \Lambda, \Xi, \sigma, \Theta, \Gamma, \Omega >$ we define the *behavior* at an interface $p$ as a set $S(p)$ of timed sequences, where each sequence $\omega \in S(p)$ contains only stimulus and response events belonging to $\sigma(p)$, and satisfies the following conditions:

- [S1] for every stimulus $s \in \omega, s \in \sigma(p)$, there exists exactly one response $r \in \Theta(s), r \in \sigma(p)$. The stimulus $s$ may occur at many different times in $\omega$; let $s[i]$ denote an occurrence of $s$ in $\omega$ then for every $s[i]$ there exists exactly one response $r[i]$ where $r[i] \in \Theta(s), i : \mathbb{N}, i < \text{number of events in } \omega$. It is possible to have different responses for different occurrences of the same stimulus (based on data constraints in $\Omega$),

- [S2] $t(r[i]) \geq t(s[i])$, where $t(.)$ is the time function for event occurrences and $r[i], s[i]$ denote an occurrence of $s$ and $r$ in $\omega$. Also, $t(s[i]) > t(s[j]) \land t(r[i]) > t(r[j]), i, j : \mathbb{N}, i > j \land i, j < \text{number of events in } \omega$. This means that an event may occur at difference times in the timed sequence where always the time of the later occurrence is greater than the time of the former occurrence of the same event,

- [S3] for every stimulus $s \in \omega$ and response $r \in \Theta(s), t(r)$ conforms to $\Gamma(s, r)$, and

- [S4] for every stimulus $s \in \omega$ and response $r \in \Theta(s)$, if there is a data constraint defined over the data parameters of $s$ then the data constraint is satisfied. If there are many data constraints defined on the data parameters of $s$ then one of them is satisfied.

Notice that [S1] assures predictability, [S2] and [S3] assure timeliness, and [S4] asserts that safety properties are satisfied.

Event names in the sequences of $S(p)$ can be qualified by the name of the interface instance $p$, from which the event originated.

**Definition 3.8** The behavior of a reactive component is the arbitrary interleaving of the behaviors at the interfaces of the component.

### 3.4 Secure-TRC (STRC)- a formal definition

A TRC which has no security restriction will respond to every stimulus received by it. The introduction of security properties at the frame of a TRC will enrich its behavior by forcing (1) an analysis of the stimulus received before processing it internally, and (2) an analysis of the response before sending it. In computer security [6] the *identity* of the entity executing a process is the basis for assigning and checking security access rights. In our discussion, the user identity, henceforth called user, is associated with the component at its instantiation time. All access control to system resources assume that the association is correct. Verifying the correctness of the identity and describing how it is associated to components falls outside the scope of this paper. Ensuring security requires an explicit
definition of an *access control matrix* that defines the *access level* of users to both events and information carried by events.

The formal specification and implementation of the TRC frame is extended to include *security access functions*. The access functions are used by the interfaces of the component to check if the user of the component is authorized to request a service (send a stimulus) and/or receive a service (response) from the TRC. Also, it checks if the user is authorized to view the information carried by the response. If the user is denied access, the stimulus will be ignored. Also, if the user is not authorized to view a data parameter, the parameter will be filtered. The security property of a TRC can be defined in terms of *event-security* and *data-security*.

**Definition 3.9** An interface of a TRC is event-secure if (1) every stimulus event is received from a user who is authorized to trigger the stimulus, and (2) for every response event sent, the user receiving the response is authorized to view the response. An interface of a TRC is data-secure if (1) the TRC user has access rights for the data parameters in every stimulus sent by the user, and (2) for every response sent by the TRC, the user receiving the response has access rights for the data parameters in the response.

**Definition 3.10** A sequence of events at a component interface is secure if and only if it is event-secure and data-secure. A TRC is secure if and only if all event sequences at all its interfaces are secure.

The frame specification of a STRC is defined by extending the frame specification of the TRC with security specifications. We assume that $U$ denotes the set of users. For the sake of simplicity we assume $AC = \{grant, deny\}$ is the set of access rights for events, and $DA = \{read, write\}$ is the set of allowed actions on data.

**Definition 3.11** The frame specification $CT_F$ of a STRC is obtained by extending the tuple $<\Pi, \Sigma, \Lambda, \Xi, \Theta, \Gamma, \Omega>$ with functions $\Upsilon, \Psi$, where $\Upsilon : U \times \Sigma \rightarrow AC$ is the event-security access function that assigns for every pair $(user, event)$ an authorization which is either grant or deny, and $\Psi : U \times \Lambda \rightarrow \mathcal{P} DA$ is a data-security access function that assigns for every pair $(user, data)$ an authorization which is a subset of $DA$. If $\Psi(u, d) = \emptyset$ user $u$ is denied access to data $d$. A STRC component is thus the tuple $<\Pi, \Sigma, \Lambda, \Xi, \Theta, \Gamma, \Omega, \Upsilon, \Psi>$.

In a STRC, we require the behavior $S(p)$ at every interface $p$ of $C_F$ to satisfy the following conditions: for every sequence $\omega \in S(p)$, for every stimulus $s$ in $\omega$, $s \in \sigma(p)$ let $u$ denote the user associated with the component injecting $s$, let $u'$ denotes the user associated with the component which will receive the response, $u, u' \in U$:

- [C1] $\Upsilon(u, s) = grant$ and $\Upsilon(u', \Theta(s)) = grant$, and
- [C2] for every data parameter $d \in \Xi(s)$ and $d' \in \Xi(\Theta(s))$, $\Psi(u, d) = \{read, write\}$ and $\Psi(u', d') = \{read\}$.

**Definition 3.12** A trustworthy component (TTRC) is a STRC whose behavior satisfies the conditions $[S1], [S2], [S3], [S4], [C1], [C2]$. 

[71] FACS'07 Pre-Proceedings
3.5 Substitution of TTRCs

Substituting one component by another is a common design practice in CBD. There are two main reasons for substitution: (1) replace existing component by another one to improve performance or fix defects, and (2) upgrade a component to a newer version that provides more services. Therefore, in both cases, the new component should render at least the same behavior as the substituted one. It can be proven that in order to substitute a TTRC $C_1$ by another component $C_2$, (1) $C_2$ should be a TTRC, and (2) $C_2$ respects the contract of $C_1$. That is, $C_2$ satisfies the following conditions for every stimulus $e \in \Sigma_{C_1}$.

- $e$ is an input event in $\Sigma_{C_2}$.
- $\Theta_1(e)$, the response of $e$, should exist in the set of output or internal events of $C_2$.
- The set of data parameters $\Xi_1(e)$ carried by $e$ should exist in the set of data parameters $\Xi_2$ of $C_2$.
- The time constraints in $\Gamma_1$ of $C_1$ that govern $e$ should imply the set of time constraints $\Gamma_2$ of $C_2$ that govern $e$.
- The data constraints in $\Omega_1$ of $C_1$ that restricts $e$ should exist in the set of data constraints $\Omega_2$ of $C_2$.
- Every user event access right in $\Upsilon_1$ of $C_1$ that is defined for $e$ should exist in $\Upsilon_2$ of $C_2$.
- Every user data access right in $\Psi_1$ of $C_1$ that is defined for the data parameters of $e$ should exist in $\Psi_2$ of $C_2$.

4 Composition of Trustworthy Components

This section discusses the basic issues in the composition of trustworthy components. We present the difference between our model and the compositional theories presented in the literature. Then, we present a compositional rule for TTRCs.

4.1 Composing security and safety

Informally, composition means “gluing together” two or more components to form a new component. A given set of components can be composed in different ways to achieve different results. However, the challenging aspect is to develop a set of rules for a stated property to be preserved in a composition. It should be possible to reason about the properties of the composite component relative to the properties of the constituent components. In this respect composition of components is different from component integration [10].

The main characteristic of a trustworthy composition is that it preserves the credentials of trustworthiness (safety and security) in the composition. This means that if both components $C_1$ and $C_2$ satisfy a trustworthy property $pr$ and their composition results in a component $C_3$ then $C_3$ is trustworthy only if it preserves the property $pr$. In general, if $C_1$ satisfies a property $pr_1$ and $C_2$ satisfies a property $pr_2$ then a trustworthy composition should result in a component $C_3$ such that $C_3$ satisfies both $pr_1$ and $pr_2$, where $pr_1$ can be either a safety or a security property.

In the literature, safety and security properties are formally specified and composed using different methods. This is due to the common consensus that while safety properties are defined as sets of “safe” sequences, security properties cannot be expressed as sets of
sequences\cite{23,32,19}. It is known \cite{1} that safety properties can be preserved in a composition, however some security properties are not preserved by any composition \cite{20}. Hence it was concluded that it would not be possible to neither express safety and security using one formal logic nor use one compositional theory for both safety and security. This implies that different formal methods have to be used for the specification and verification of security independent of safety.

Many security properties have been proposed as information flow properties \cite{12,20,21,22,11,18} which attempt to prevent a low-level user from inferring some thing that is confidential to a high-level user \cite{32}. Many interface security properties that were presented early on were proved to be weak in later research and were replaced by stronger ones. See \cite{12,20,21,22,11,18} for a history of the research related to the introduction of new information flow properties and an account of how they were either proven to be weak subsequently or proven that they failed to preserve security in composition. Most importantly, the use of this type of security properties doesn’t allow combining it with safety properties within one formal specification method so that composition, and verification can be formally achieved \cite{23,32,19}. Finding a single composition rule and a formalism to assure the satisfaction of trustworthiness in composite components has been an open problem until now.

In this Section we propose a composition rule that unifies both access control and interface security models. Access control models restrict access to component services, and validates user requests of authorized users. We apply this restriction at the interfaces of components. We argue that event-security and data-security properties suggested by this paper can be expressed as sets of sequences. Hence, these security properties can be expressed in any mathematical logic in which safety properties are expressed. Therefore, one compositional theory can be used for both safety and security.

**Definition 4.1** Let \( S(p) \) be the set of sequences occurring at the interface \( p \) of \( C_F \). Each sequence \( \omega \in S(p) \) consists of stimuli and responses satisfying the conditions \( S[1], S[2], S[3], S[4] \) in Definition 3.7. Let \#\( \omega \) denote the number of events in a sequence \( \omega, e[i] \) denote the event at the index \( i \) of sequence \( \omega \), \( u \) denote the identity of the user injecting the stimulus \( e[i] \), \( \Theta(e[i]) \) denote the response of \( e[i] \), and \( u' \) denote the identity of the user who will receive the response. The event security property at the interface \( p \) is

\[
\forall \omega \in S, 1 \leq i \leq \#\omega \bullet (\Upsilon(u, e[i]) = \text{grant}) \land (\Upsilon(u', \Theta(e[i])) = \text{grant})
\]

The data security property at the interface \( p \) is

\[
\forall \omega \in S, 1 \leq i \leq \#\omega \bullet \forall d \in \Xi(e[i]), \forall d' \in \Xi(\Theta(e[i])) \bullet (\Psi(u, d) = \{\text{read}, \text{write}\})
\]

\[
\land (\Psi(u', d') = \{\text{read}\})
\]

4.2 Composition of TTRCs

In this section we define the composition of templates \( CT_1 \) and \( CT_2 \).

**Definition 4.2** Let \( CT_1_F =< \Pi_1, \Sigma_1, \Lambda_1, \Xi_1, \sigma_1, \Theta_1, \Gamma_1, \Omega_1, \Upsilon_1, \Psi_1 > \) and \( CT_2_F =< \Pi_2, \Sigma_2, \Lambda_2, \Xi_2, \sigma_2, \Theta_2, \Gamma_2, \Omega_2, \Upsilon_2, \Psi_2 >, \) their corresponding architectures \( CT_1_A \) and \( CT_2_A \) are hidden. The compositional rule defines a unique \( CT_F \) which can have many
architectures $CT_{A}$. The composition $CT_{F} = < \Pi, \Sigma, D, \Xi, \sigma, \Theta, \Gamma, \Omega, \Upsilon, \Psi >$ is given below:

$$
\Pi = \{ P \mid P \in \Pi_1 \land \exists Q \in \Pi_2 \cdot \text{Compatible}(P, Q) \} \cup \{ P \in \Pi_2 \land \exists Q \in \Pi_1 \cdot \text{Compatible}(P, Q) \}
$$

$$
\Sigma = \Sigma_1 \cup \Sigma_2
$$

$$
\Lambda = \Lambda_1 \cup \Lambda_2
$$

$$
\forall e \in \Sigma, \Xi(e) = \{ \Xi_1(e) \mid e \in \Sigma_1 \} \cup \{ \Xi_2(e) \mid e \in \Sigma_2 \}
$$

$$
\forall P \in \Pi, \sigma(P) = \{ \sigma_1(P) \mid P \in \Pi_1 \} \cup \{ \sigma_2(P) \mid P \in \Pi_2 \}
$$

$$
\forall e \in \Sigma, \Theta(e) = \{ \Theta_1(e) \mid e \in \Sigma_1 \} \cup \{ \Theta_2(e) \mid e \in \Sigma_2 \}
$$

$$
\Gamma = \Gamma_1 \cup \Gamma_2.
$$

Notice that we want to retain the constraints at the interfaces that are no more ‘visible’ in order that we can reason about grey-box behavior.

$$
\Omega = \Omega_1 \cup \Omega_2
$$

$$
\forall e \in \Sigma, \forall u \in U, \Upsilon(u, e) = \begin{cases} 
\text{grant,} & \text{if } e \in \Sigma_1 \cap \Sigma_2 \land \Upsilon_1(u, e) = \Upsilon_2(u, e) = \text{grant}; \\
\text{deny,} & \text{if } e \in \Sigma_1 \cap \Sigma_2 \land \Upsilon_1(u, e) = \text{deny} \lor \Upsilon_2(u, e) = \text{deny}; \\
\Upsilon_1(u, e), & \text{if } e \in \Sigma_1 \land e \notin \Sigma_2; \\
\Upsilon_2(u, e), & \text{if } e \in \Sigma_2 \land e \notin \Sigma_1
\end{cases}
$$

$$
\forall e \in \Sigma, \forall d \in \Xi(e), \forall u \in U, \Psi(u, d) = \begin{cases} 
\Psi_1(u, d) \cap \Psi_2(u, d), & \text{if } d \in \Lambda_1 \land \Lambda_2; \\
\Psi_1(u, d), & \text{if } d \in \Lambda_1 \land d \notin \Lambda_2; \\
\Psi_2(u, d), & \text{if } d \in \Lambda_2 \land d \notin \Lambda_1
\end{cases}
$$

$$
\forall P \in \Pi_1, \forall Q \in \Pi_2, \text{if } \text{Compatible}(P, Q) \text{ then there exists a connector type } P \oplus CT_{1P} \oplus Q \oplus CT_{2P} \text{ in } CT.
$$

There could be many architecture types for $CT_{F}$ because not all the interfaces in the resulting connector types should be linked. Also, different component instances can have a different number of connector and interface instances which enables the component to have different possible dynamic architectures.

We assert that the composition rule stated in Definition 4.2 preserves the credentials of trustworthiness: safety and security.

**Theorem 4.3** The composition of two TTRCs results in a TTRC.

The proof is provided in the Appendix.

### 4.3 System definition

A RTRS can be defined as a network of connected TTRCs. The template in Figure 2 shows the syntax for system specification. The syntax starts with the keyword **System** to introduce an identifier for the system, and includes labeled sections described in Figure 2. The section **Component types** defines component templates. It contains three subsections. The subsection **Frame types** defines frame for each component type, the subsection **Connector types** defines frame connector types, and the subsection **Architecture types** defines architecture types for composite components. The section **Components** defines instances of component types declared in the first section. It includes two subsections. In the subsection **Frame instances** instances of frames declared in section Frame types are defined, and in the subsection **Architecture instances** architectures for frame instances, created as instances of architecture types defined in Architecture types are defined. In the section **Users** a finite set
System <name>

Component types
  Frame types:
  Connector types:
  Architecture types:

Components
  Frame instances:
  Architecture instances:

Users:
  Access control
    Event access matrix:
    Data access matrix:

end

Fig. 2. System specification

of client user identities are listed. Only on behalf of them component is requesting or providing services. The section Access control includes two subsections Event access matrix to enforce event security, and Data access matrix to enforce data security. Event security is defined as a partial function $EAM : \mathcal{P}(Users \times \Sigma) \rightarrow AC$, and data security is defined as another partial function $DAM : \mathcal{P}(Users \times \Lambda) \rightarrow \mathcal{P}(AD)$.

5 Related Work

We compare our model with the state of the art component model SOFA 2.0 [7,16,8]. A comparison between SOFA 2.0 and the other component models can be found in [15]. SOFA 2.0 is a hierarchical component model that inherits structure from its ancestor SOFA [26]. The main features of SOFA 2.0 include: (1) a meta-model based design of components, (2) support for dynamic reconfiguration of architectures using predefined patterns that allow adding/removing components and connecting to external services, (3) support for different communication styles by defining connectors as first class components, (4) defining the control part of components using micro-components, and (5) providing design time and runtime environments for the development and deployment of component based systems. In this paper, we compare the relevant formal and structural aspects of our model with their correspondents in SOFA 2.0. Our work differs from SOFA 2.0 fundamentally in that our model is supporting RTRS. The introduction of time brings sophistication to system design and composition. Moreover, our model supports parameterized events while behavioral protocols of SOFA [26] doesn’t support parameters. Specifying parameters is essential for the specification and verification of complex RTRS. Also, our model provides formal foundation for the specification and composition of security properties in components which is not supported by SOFA 2.0. Furthermore, in SOFA, the frame is defined as an aggregation of interfaces, whereas in our model the frame is an aggregation of interface types. Also, the architecture of composite components in SOFA defines connectors while in our model it defines connector types. This allows, in our model, a dynamic configuration of the component’s black-box and internal structure at deployment time benefiting from the available information. This is because it is possible, in our model, to create multiple instances of each interface type and instantiate connectors to each of them at the
time of instantiating an architecture from the design time architecture type selected for a frame. However, SOFA 2.0 defines a static deployment model and allows reconfiguration later. The reconfiguration of components in SOFA 2.0 is not suitable for trustworthy RTRS because (1) removing components affects the availability of services which violates an important requirement of RTRS, (2) adding new components may introduce new security threats. Instead, in our model, we advocate components substitution as a means of facilitating dynamic reconfiguration at runtime. Because we define strict compatibility rules for substitution, properties such as reactivity, timeliness, safety, and security are enforced dynamically. In SOFA 2.0, connectors define end points that can be connected to any interface; however, in our model we specify interface types in the link of the connector type. This enables us to enforce and check compatibility between connected interfaces.

6 Conclusion

In this paper, we have presented one part of our ongoing research work on a formal approach to component modeling for the development of trustworthy RTRS. We have presented a formal foundation for defining and composing hierarchical structure and contract for trustworthy components. We pointed out that safety and security are the two essential credentials that can assure a high degree of trustworthiness. We have shown that using CBD it is possible to formally specify trustworthy components and compose them. This approach leads us to a unified method for the verification of trust using model checking, which has been shown to be a promising method for the verification of safety properties for RTRS. Currently, we are working on the automatic generation of the behavior protocol for trustworthy components from its structure and contract views. We are investigating a model checking approach that first translates the component model to UPPAAL language and uses the model checker in UPPAAL toolset to verify trustworthiness credentials [2].

7 Appendix

In this section we provide proof for Theorem 4.3.

**Proof.** Let $C_1$ and $C_2$, instances of $CT_1$ and $CT_2$ respectively, be two trustworthy components. Their behavior satisfy the properties [S1],[S2],[S3],[S4],[C1], and [C2]. Let $C$ be an instance of $CT_F$, the composition of $CT_{1_F}$ and $CT_{2_F}$ according to Definition 4.2. Let $S_1$ and $S_2$ be behaviors representing the set of all possible observed sequences of $C_1$ and $C_2$ respectively, $S$ be the behavior of the composite component $C$ representing the set of all possible observed sequences of $C$, $S(p)$ be the behavior at an interface $p$ instantiated from interface type $P \in \Pi$ in $C_F$. The proof procedure consists of 3 steps: (I) a proof that the behavior of $C$ satisfies the properties [S1],[S2],[S3], and [S4]. (II) a proof that if $C_1$ satisfies safety property $\tau_1$ and $C_2$ satisfies safety property $\tau_2$ then $C$ satisfies both $\tau_1$ and $\tau_2$ i.e. the composition preserves safety properties, and (III) a proof that the behavior of $C$ satisfies [C1] and [C2] i.e. the composition preserves event security and data security.

We use the following properties in the proof. (Prop.1) From Definition 3.1 every event is associated with only one interface type: $\forall P,Q \in \Pi, \sigma(P) \cap \sigma(q) = \emptyset$, (Prop.2) from Definition 3.2 every interface type in the composite frame belongs only to one frame definition $CT_{1_F}$ or $CT_{2_F}$: $\forall P \in \Pi \bullet (P \in \Pi_1 \lor P \in \Pi_2) \land (P \notin \Pi_1 \cap \Pi_2)$, (Prop.3) from
Definition 3.1 there are no two interface types that are compatible in the composite frame definition $C_F$: $\#Q \in I, \text{Compatible}(P, Q)$.

**Preserving properties [S1],[S2],[S3], and [S4]:** from Definition 3.8, $S$ is constructed from the arbitrary interleaving of the behaviors at the interfaces of $C_F$. From Prop.2, every interface $p$ in $C_F$ is either an interface at $C_1p$, or at $C_2p$. Since the behaviors $S_1(p)$ or $S_2(p)$ for every interface $p$ in $C_1$ and $C_2$ satisfies $[S1],[S2],[S3]$, and $[S4]$ then the behavior $S(p)$ of every interface $p$ at $C_F$ satisfies those properties.

**Preserving Safety:** Let $\tau_1$ and $\tau_2$ be two safety properties, $R_1$ and $R_2$ be the set of all sequences which satisfy $\tau_1$ and $\tau_2$ respectively. From Prop.1, Prop.2, and Prop.3 we have $\forall p \in P, p \in \Pi, \forall \omega \in S(p), \omega \in S_1(p) \lor \omega \in S_2(p)$. This means that $\forall \omega \in S(p), \omega \in R_1 \lor \omega \in R_2$. Hence, $\forall \omega \in S(p), \omega \in R_1 \lor R_2$. Therefore, $S \subseteq R_1 \cup R_2$, and the composition satisfies both $\tau_1$ and $\tau_2$. This shows that the composition preserves safety properties.

**Preserving event security [C1]:** Let $\omega \in S$, $u, u' \in U$ be users and $u[i], u'[i]$ denote the user stimulating the event at index $i$ of $\omega$ and the user receiving the response respectively. From Prop.1, Prop.2, and Prop.3: $\forall \omega \in \Pi, \forall u \in \sigma(P), (e \in \Sigma_1 \lor e \in \Sigma_2) \land (e \notin \Sigma_1 \lor \Sigma_2)$. Therefore, $\forall \omega \in S, \forall i \in N \cdot 1 \leq i \leq \#\omega \cdot (\Xi(u[i], e[i]) = \Upsilon_1(e, u), \Upsilon(u[i]', \Theta(e[i])) = \Upsilon_1(u[i]', \Theta(e[i]) \lor e \in \Sigma_1) \lor (\Upsilon(u[i], e[i]) = \Upsilon_2(u[i]', \Theta(e[i])) \lor e \in \Sigma_2)$. Since $\forall \omega \in S_1, \forall i \in N \cdot 1 \leq i \leq \#\omega \cdot \Upsilon_1(e[i], u[i]) = \text{grant} \land \Upsilon_1(u'[i], \Theta(e[i])) = \text{grant}$ and $\forall \omega \in S_2, \forall i \in N \cdot 1 \leq i \leq \#\omega \cdot \Upsilon_2(e[i], u[i]) = \text{grant} \land \Upsilon_2(u'[i], \Theta(e[i])) = \text{grant}$. Thus, $\forall \omega \in S, \forall i \in N \cdot 1 \leq i \leq \#\omega \cdot \Upsilon(e[i], u[i]) = \text{grant} \land \Upsilon(u'[i], \Theta(e[i])) = \text{grant}$. Therefore, the composition preserves the event-security property.

**Preserving data security [C2]:** From Prop.1, Prop.2, and Prop.3: $\forall \omega \in \Pi, \forall e \in \sigma(P), (\Xi(e) \subseteq \Lambda_1, \Xi(\Theta(e)) \subseteq \Lambda_1 \lor \Xi(e) \subseteq \Lambda_2, \Xi(\Theta(e)) \subseteq \Lambda_2) \land (\forall d \in \Xi(e), \forall d' \in \Xi(\Theta(e)) \cdot d, d' \notin \Lambda_1 \lor \Lambda_2)$. Thus, $\forall \omega \in S, \forall i \in N \cdot 1 \leq i \leq \#\omega \cdot d \in \Xi(e[i]), \forall d' \in \Xi(\Theta(e[i])) \cdot \Psi(u[i], d) = \Psi_1(u[i], d), \Psi(u'[i], d') = \Psi_1(u'[i], d') \land e \in \Sigma_1 \lor (\Psi(u[i], d) = \Psi_2(u[i], d), \Psi(u'[i], d') = \Psi_2(u'[i], d') \land e \in \Sigma_2)$. Since $\forall \omega \in S_1, \forall i \in N \cdot 1 \leq i \leq \#\omega, \forall d \in \Xi(e[i]), \forall d' \in \Xi(\Theta(e[i])) \cdot \Psi_1(u[i], d) = \{\text{read, write} \land \Psi_1(u'[i], d') = \{\text{read}\} \land \forall \omega \in S_2, \forall i \in N \cdot 1 \leq i \leq \#\omega \cdot \Psi_2(u[i], d) = \{\text{read, write} \land \Psi_2(u'[i], d') = \{\text{read}\}$. Hence, $\forall \omega \in S, \forall i \in N \cdot 1 \leq i \leq \#\omega \cdot \Psi(u[i], d) = \{\text{read, write} \land \Psi(u'[i], d') = \{\text{read}\}$. Therefore, the composition preserves the data security property.

References


A CSP architectural model for fault-tolerant systems

David Paulo Pereira and Ana C. V. de Melo

University of São Paulo (USP)
Department of Computer Science
São Paulo - SP - Brazil

Abstract

Architectures based on Coordinated Atomic action (CA action) concepts have been used to build concurrent fault-tolerant systems. This conceptual model combines concurrent exception handling with action nesting to provide a general mechanism for both enclosing interactions among system components and coordinating forward error recovery measures. This paper proposes an architectural model to guide the formal specification of concurrent fault-tolerant systems. This architecture provides built-in CSP (Communicating Sequential Process) processes and pre-defined channels to coordinate exception handling of the user-defined components. As a result, a formal and general architecture supporting software fault-tolerance (including all the exception handling coordination protocols) are ready to be used as users define components with the normal and exceptional behaviors.

Keywords: fault-tolerant architecture, exception handling coordination, CSP

1 Introduction

The more complex to develop and manage systems the more software design faults increase [2], and a significant amount of code is dedicated to error detection and recovery [21]. Exception handling mechanisms provide a clear separation of codes for error recovery and normal behavior, and helps in decreasing code complexity and software design faults [9]. Due to these benefits, exception handling is recognized as a good approach to provide fault-tolerant software [2] and has naturally been applied to software components. Following this approach, the software component must comprise the normal behavior code, in which errors are detected and the corresponding exceptions are raised, and the exception handler code [7], in which the exceptional behavior is treated resulting in the error recovery.

Despite being crucial for software robustness [14], software fault tolerance at the architectural level and, in particular, exception handling flow in that context

1 Email: dpereira@ime.usp.br
2 Email: acvm@ime.usp.br

This paper is electronically published in

Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
has not been widely explored. The development of components comprising normal and exceptional behaviors is still mostly ad-hoc. For each system, a particular architecture and the corresponding mechanisms to coordinate exception propagation are designed. As a result, one can get coordination mechanisms that are error prone, hard to implement and maintain, do not scale up and are not suitable for reuse in different application contexts.

The definition of a formal and general architecture to support software fault-tolerance, by a systematic and standard exception handling coordination, would provide great benefits to the quality of systems based on components. Most works at the architectural level either provide connectors with different semantics in which new architectures can be created, or redefine an architectural model with no formal means. Issarny and Banatre [14] propose exception handling facilities at the architectural level with no formalization: they defined extensions to Architecture Description Languages (ADLs) to handle exceptions within components. An enriched set of connectors has been defined and formalized for Reo [3] in which mobility and dynamic features are included together with the protocols, but the architectural coordination model is not predefined. Guerra [9] described a fault-tolerant architectural model in the C2SADEL [17] based on the concept of idealized Fault-Tolerant Component (iFTComponent) [2] with no coordination predefinition. iFTComponent either gives a normal response if the service request is successfully processed or raises an external exception otherwise. Recent studies [10, 8] propose a fault-tolerant architectural solutions in terms of ADL components and connectors based on the concept of iFTComponent and collaborative exception handling, but no protocols formalization is given.

The present paper defines an architectural model composed of standard processes and communication channels to provide the exception handling coordination. The strategy employed here is based on concurrent and collaborative exception handling defined in Coordinated Atomic action (CA action) conceptual model (a companion paper [18] presents the complete formalization of exception handling coordination protocols in CSP). As with the previous studies, the present work is based on the separation of normal and exceptional behaviors and the cooperative exception handling concepts. Then, besides quality, software reuse and evolution get promoted with the description of complex software systems based on a predefined architecture. Moreover, the formal strategy for specifying fault-tolerant systems helps in the identification of ambiguities, omissions and inconsistencies at the specification level, and the formal verification serves as a fault-prevention mechanism to better eliminate design faults.

The main contributions are on predefining a general architecture for systems based on components through formal means, and predefining an exception handling coordination mechanism. The predefined architecture allows a designer to specify new components based systems (in CSP) from the identification of the main components and their exceptions only; the elements necessary to coordinate the exception mechanisms and their corresponding behavior are all provided by the predefined architecture. Also, due to the formalization means, properties can be proved over the system specification.

Section 2 introduces the CA actions conceptual model, often applied in the de-
development of concurrent and fault-tolerant systems. The concepts and properties of CA actions are formalized in Section 3. Section 4 describes the fault-tolerant architectural model based on CA actions and details the standard communication channels used in coordination. Section 5 shows the steps a designer must follow to specify the system components based on the proposed architecture. Section 6 shows the instantiation of the whole system based on the previously defined components and the proposed architecture. Section 7 presents the related work and review the major contributions of the present work.

2 CA Actions

In concurrent and distributed systems, recovering a single software component fault might not be enough since incorrect or corrupted data could be propagated to other components. Moreover, raising concurrent exceptions may represent a symptom of systemic errors in the system (e.g. network connectivity problems). Looking at these problems, Romanovský [22] proposed a two-level exception handling strategy. First, the exceptions are locally handled, at the component level. If the component cannot handle the exception properly, it must be treated at the architectural level where other components can cooperate in the exception handling.

According to Romanovský [23], exception handling at architectural level in concurrent and component based systems can be structured through atomic actions. In this approach, components cooperate while executing a particular action of the system as much as when exceptions are handled. The concept of atomic action was proposed by Randell [20] and relates to the demarcation and isolation of a sequence of interactions between the software components necessary for the conclusion of a specific task. If an exception is raised inside an atomic action, all of the participating components must cooperate in exception handling activity. Atomic actions are today one of the promising strategies for structuring forward error recovery in cooperating component-based software systems [5].

The coordinated atomic action (CA action) [27] extends the atomic action concept with the notion of transactions. Functional components designed to explicitly cooperate and exchange information in order to complete the action successfully are called participating components. External components are the transactional elements that provide shared data (e.g. database) concurrently used by a number of CA actions. Those components must satisfy the Atomicity, Consistency, Isolation and Durability (ACID) properties due to the concurrent access by participating components.

CA actions provide a sound architecture for specifying fault-tolerant software systems based on participating components that both cooperate in the execution of actions and compete for shared resources described by external components. Some case studies have demonstrated the practical feasibility of the CA action architecture model including a railway control system [4], an industrial production cell [28] and an insulin pump therapy [6].

A CA action starts when all participants are activated and ready to execute, and finishes when all of them have completed their execution. Figure 1(a) shows the normal execution of a CA action (ac1) with two participating components (par1
and \( par_2 \) and one external component \((ext_1)\). If an exception is raised by one or several participating components within a CA action, then a coordinated exception handling mechanism is required. In Figure 1(b), \( par_1 \) raises an exception that is followed by the suspension of \( par_2 \) normal execution. Then, both participating components handle exception in a collaborative way. Since it is cooperative, all the participating components must have corresponding exception handlers for a given exception.

![Diagram](image)

Fig. 1. Conceptual model of CA actions

To deal with concurrent exceptions raised inside a CA action, Campbell [5] proposed the concept of exception resolution tree. This tree includes all exceptions associated with a certain action and imposes a partial order on them: a higher exception in the tree has a handler to deal with any lower-level exception or any combination of them.

![Diagram](image)

Fig. 2. Exception resolution tree

Figure 2 shows an example of exception resolution tree in which leaf nodes represent the system raised exceptions \( e_1 \), \( e_2 \) and \( e_3 \). All the nodes in a higher level represent possible concurrent exceptions. Node \( e_1 \land e_2 \), for example, identifies a resolved exception that must be handled when exceptions \( e_1 \) and \( e_2 \) are concurrently raised. The root node is the universal exception that handles a combination of all concurrent exceptions.

3 CA Actions: Preliminary Definitions

To formally define a general architecture for components with exception handling coordination, we first define some concepts regarding CA actions.
The coordination of exception handling requires definition of both normal and exceptional behaviors. Next sections formalize the main elements required to specify systems based on CA actions concepts.

3.1 CA Actions: Normal Behavior

For systems specification based on CA actions, the participants and external components must be defined, together with the set of actions and relations to denote the interaction among participants, actions and external components:

Definition 3.1 (Normal Model). The normal model of CA actions is formalized as follows:

CA actions If \( n \) is the number of CA actions in the system then \( Ac = \{ ac_i \mid 1 \leq i \leq n \} \) defines the set containing all these actions, denoted as \( ac_i \).

Participating components If \( m \) is the number of participating components then \( Par = \{ par_i \mid 1 \leq i \leq m \} \) defines the set containing all these components, denoted as \( par_i \).

External components If \( r \) is the number of external components then \( Ext = \{ ext_i \mid 1 \leq i \leq r \} \) defines the set containing all these components, denoted as \( ext_i \).

Nesting The function \( fNestedAction : Ac \rightarrow \mathcal{P}Ac \) maps each CA action to the subset of nested CA actions.

Participation The function \( fParticipant : Ac \rightarrow \mathcal{P}Par \) maps each CA action to the subset of participating components.

Allocation The function \( fExternal : Ac \rightarrow \mathcal{P}Ext \) maps each CA action to the subset of external components allocated.

3.2 CA Actions: Exceptional Behavior

In order to achieve software fault-tolerance for concurrent systems based on CA actions, the mechanism of exception handling must be applied. Besides the normal behavior, the designer must also specify all systems exceptions: the ones raised by each participating component and the ones to be handled in an outer context. Moreover, corresponding exception trees for CA actions must be defined. Thus, the introduction of a standardized and formal notation for the specification of CA action exception model is also required.

Definition 3.2 (Exceptional Model). The exceptional model of CA actions is formalized as follows:

Exceptions If \( p \) is the number of exceptions then \( Exp = \{ exp_i \mid 1 \leq i \leq p \} \) defines the set containing all these exceptions, denoted as \( exp_i \).

Raising The function \( fRaise : Par \times Ac \rightarrow \mathcal{P}Exp \) maps each participating component inside a specific CA action to the subset of exceptions raised by a given.

Signaling The function \( fSignal : Ac \rightarrow \mathcal{P}Exp \) maps each CA action to the containing CA action to the subset of external exceptions signaled (propagated).

Concurrent Exception Resolution The function \( fExceptionTree : \mathcal{P}Exp \times Ac \rightarrow \)
**Exp** maps a subset of concurrent raised exceptions in a given CA action to a resolved exception.

### 3.3 Example: CA Actions Formalization

Suppose a system designer specifies a concurrent system composed of two participating components, one external component and two CA actions (illustrated in Figure 3(a)), and identifies three types of exceptions: \( \text{exp}_1, \text{exp}_2 \) and \( \text{exp}_3 \). Figure 3(b) states the exceptional behavior of that system: exception \( \text{exp}_1 \) may be raised during normal execution of participant \( \text{par}_1 \) in the context of the CA actions \( \text{ac}_1 \) and \( \text{ac}_2 \), while \( \text{exp}_2 \) may be raised by participant \( \text{par}_2 \).

![Diagram](image)

**Fig. 3. Example: Conceptual Model**

The system specification based on the conceptual model of CA actions is formalized by defining values to all sets and functions in Definitions 3.1 and 3.2. Besides the relation between actions, participants and exceptions, the relationship among the exceptions themselves must be defined to compose the resolution tree. For this particular system, suppose exception \( \text{exp}_3 \) represents the concurrent raising of exceptions \( \text{exp}_1 \) and \( \text{exp}_2 \) by the participating components \( \text{par}_1 \) and \( \text{par}_2 \), respectively, in the context of both CA actions. In spite of handling each exception separately, \( \text{exp}_3 \) is actually the resolved exception to be handled and must also be introduced to the system. For the complete formalization of the system architecture, the set \( \text{Exp} \) and functions \( \text{fRaise} \), \( \text{fSignal} \) and \( \text{fExceptionTree} \) must also be specified. Table 1 formalizes the normal and exceptional model for the example in Figure 3.

<table>
<thead>
<tr>
<th>Normal Model</th>
<th>Exceptional Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Ac} = {\text{ac}_1, \text{ac}_2} )</td>
<td>( \text{Exp} = {\text{exp}_1, \text{exp}_2, \text{exp}_3} )</td>
</tr>
<tr>
<td>( \text{Par} = {\text{par}_1, \text{par}_2} )</td>
<td>( \text{fRaise}(\text{par}_1, \text{ac}_1) = {\text{exp}_1} )</td>
</tr>
<tr>
<td>( \text{Ext} = {\text{ext}_1} )</td>
<td>( \text{fRaise}(\text{par}_1, \text{ac}_2) = {\text{exp}_1} )</td>
</tr>
<tr>
<td>( \text{fNestedAction}(\text{ac}_1) = {\text{ac}_2} )</td>
<td>( \text{fRaise}(\text{par}_2, \text{ac}_1) = {\text{exp}_2} )</td>
</tr>
<tr>
<td>( \text{fNestedAction}(\text{ac}_2) = \emptyset )</td>
<td>( \text{fRaise}(\text{par}_2, \text{ac}_2) = {\text{exp}_2} )</td>
</tr>
<tr>
<td>( \text{fParticipant}(\text{ac}_1) = {\text{par}_1, \text{par}_2} )</td>
<td>( \text{fSignal}(\text{ac}_1) = \emptyset )</td>
</tr>
<tr>
<td>( \text{fParticipant}(\text{ac}_2) = {\text{par}_1, \text{par}_2} )</td>
<td>( \text{fSignal}(\text{ac}_2) = {\text{exp}_4} )</td>
</tr>
<tr>
<td>( \text{fExternal}(\text{ac}_1) = \emptyset )</td>
<td>( \text{fExceptionTree}({\text{exp}_1, \text{ac}_1}) = {\text{exp}_1} )</td>
</tr>
<tr>
<td>( \text{fExternal}(\text{ac}_2) = {\text{ext}_1} )</td>
<td>( \text{fExceptionTree}({\text{exp}_1, \text{ac}_2}) = {\text{exp}_1} )</td>
</tr>
</tbody>
</table>

**Table 1**

All the formalization above regarding the normal and exceptional behaviors are defined by designers; they represent the system behavior and are part of the problem
solution. However, that definition is not enough to solve the whole problem because a coordination mechanism is needed to deal with the concurrent exceptions. Next section presents an architectural model for that coordination mechanism.

4 The Architectural Model

The CA actions concepts formalized in the previous section is a step forward to provide a standard model for concurrent fault-tolerant systems. Based on those concepts, the normal behavior was defined in terms of action nesting, component participation and allocation, while the exceptional model was defined in terms of exception raising, signalling and resolution. However, in practice, the designer must combine all the normal and exceptional elements together to make them work accordingly to the CA actions concepts. This means that the designer should be responsible for providing an architecture for each particular system, making unrealistic the effective use of CA actions concepts. Instead, to make these concepts practical, it is necessary to define a standard architecture to allow designers applying those concepts in a standard, rigorous and easy way.

For this practical approach, we propose a new CSP [13,24] architectural model for the specification of concurrent fault-tolerant systems. The standard architecture must provide the system components and the communication protocols as presented in the next sections.

4.1 Architectural Elements

The system elements must comprise all components defined by the designer and some extra components responsible for the exception handling coordination. The elements defined by designers are the participant and external components:

- \( \text{Participant}_{\text{par}} \) - Each functional participating component \( \text{par}_j \) defined for the concurrent system must be specified by the designer. It is a user-defined CSP process. However, its internal structure must be composed by built-in sub-processes of type \( \text{Execution}_{\text{par}, \text{ac}, k} \) (rules regarding the specification are described in Section 5.1).

- \( \text{External}_{\text{ext}} \) - Each external component \( \text{ext}_j \) defined for the concurrent system is represented by the built-in \( \text{External}_{\text{ext}} \) process, in which the state data manipulated by the participating components inside a transactional context is kept. Besides serving as data repository, this type of process handles the transactions opened by coordinator processes.

For each participant and external component, a corresponding \( \text{Participant}_{\text{par}} \) and \( \text{External}_{\text{ext}} \) process must be defined. Apart from them, for each action \( \text{ac}_i \), a coordination component must be created to handle the exceptions raised:

- \( \text{Coordinator}_{\text{ac}} \) - In CA actions model, the mechanism for exception handling implies a cooperation between the participating components. It is necessary to instantiate the corresponding built-in \( \text{Coordinator}_{\text{ac}} \) process for each CA action \( \text{ac}_i \) defined for the system. This type of CSP process is responsible for coordinating the activities defined in the exception handling coordination protocol.

- \( \text{CoordinatorState}_{\text{ac}, \text{state}} \) - For each process \( \text{Coordinator}_{\text{ac}} \) defined, a correspond-
ing state and \textit{built-in CoordinatorState}_{ac_i,state} process must also be specified. This process is responsible for maintaining the execution state of the CA action \( ac_i \).

Note that the coordination for each action has been split into two components: \( \text{Coordinator}_{ac_i} \) and \( \text{CoordinatorState}_{ac_i,state} \). This design is more verification driven, once it keeps the control and the state in separate CSP processes. This allows a clear separation of properties related either to control or state, making easier the properties definitions and verification. Moreover, properties involving control and state at the same time can also be defined if both processes are used (this design does not impose limitations on properties definitions).

\subsection{4.2 Communications}

Besides the definition of the system components, the way they are wired must also be defined in the architectural model. This section formalizes how the system components are linked together, by some \textit{built-in} channels, in order to enable the communication between coordinators, participating and external components regarding the cooperative exception handling protocol. Figure 4 shows all connections necessary to coordinate the architectural elements.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4}
\caption{Communications}
\end{figure}

\subsubsection{4.2.1 Coordinator-Participant}

The coordination channel \( a_i p_j \) binds the coordinator process of action \( ac_i \) to the participant process \( par_j \). Through this channel, participants raise exceptions and are requested to suspend their normal execution for cooperative exception handling. Figure 4(a) shows an example consisting of one coordinator and two participants. These CSP coordination channels are formalized as follows.

\begin{definition}[Coordination Channels]
The set of communication channels for the coordinator process of \( ac_i \), used in coordination of exception handling activity, is defined as \( \text{CoordCh}_{ac_i} = \{ a_i p_j \mid 1 \leq j \leq m \land par_j \in f\text{Participant}(ac_i) \} \)
\end{definition}
4.2.2 Coordinator-Participant Synchronization

The synchronization channel \( ac_i \_\text{syn} \) binds the coordinator process of \( ac_i \) to all participant processes which execute in the context of this CA action. This channel is shared by participants and is responsible for synchronizing the initialization and end of the CA action \( ac_i \). It also binds the coordinator process of action \( ac_i \) to the coordinator process of its containing action \( ac_j \) (initialization or end of nested actions must be informed). Moreover, synchronization channel \( ac_i \_\text{syn} \) binds the coordinator process of \( ac_i \) to its corresponding coordination state process. Figure 4(b) shows an example consisting of the nested action \( ac_2 \) with two participants and the containing action \( ac_1 \).

In order to fully formalize the proposed architecture, some definitions regarding synchronization channels are given:

**Definition 4.2.** The function \( f\text{SynCh} : Ac \rightarrow \text{CHANNEL} \) maps CA action \( ac_i \)\(^3\) to the corresponding synchronization channel \( ac_i \_\text{syn} \).

**Definition 4.3 (Synchronization Channels).** The set of communication channels for the \( ac_i \) coordinator process, used in synchronization with coordinator processes of its nested actions, is given by

\[
\text{SynCh}_{ac_i} = \{ ac_j \_\text{syn} \mid i < j \leq n \land ac_i \_\text{syn} = f\text{SynCh}(ac_j) \land ac_j \in \text{fNestedAction}(ac_i) \}
\]

4.2.3 Coordinator-Coordinator State

The state channel \( ac_i \_\text{st} \) binds the coordinator process of \( ac_i \) to its auxiliary state process that maintains the execution state \( state \in \{ \text{STARTED}, \text{STOPPED} \} \) of the corresponding CA action. State \( \text{STARTED} \) denotes the CA action in execution and state \( \text{STOPPED} \) denotes that it has not started yet or has already stopped its execution. When the coordinator of nested action \( ac_i \) receives an abortion request from its containing coordinator, it must request the execution state of \( ac_i \) through channel \( ac_i \_\text{st} \). Figure 4(c) shows an example describing the state channel linking the coordinator process of CA action \( ac_1 \) with its auxiliary state process.

**Definition 4.4.** The function \( f\text{StateCoordCh} : Ac \rightarrow \text{CHANNEL} \) maps a given CA action \( ac_i \) to the corresponding state channel \( ac_i \_\text{st} \).

4.2.4 Coordinator-Coordinator

The abortion channel \( ac_i \_\text{ab} \) binds the coordinator process of \( ac_i \) to the coordinator process of the containing action. Through this channel an abortion request is received indicating a raised exception in the context of the containing CA action. Figure 4(d) shows that coordinator of \( ac_1 \) is bound to the coordinators of nested actions \( ac_2 \) and \( ac_3 \) through channels \( ac_2 \_\text{ab} \) and \( ac_3 \_\text{ab} \), respectively. Some definitions regarding abortion channels are given below:

**Definition 4.5.** The function \( f\text{AbortCh} : Ac \rightarrow \text{CHANNEL} \) maps a CA action \( ac_i \) to the corresponding abortion channel \( ac_i \_\text{ab} \).

**Definition 4.6 (Abortion Channels).** The set of communication channels for coord-

---

\(^3\) In this work the expression \( \text{CHANNEL} \) represents the set containing any CSP communication channel.
dinator process of $ac_i$ used in abortion of nested CA actions is defined as:

$$\text{AbortCh}_{ac_i} = \{ ac_{j-ab} \mid i < j \leq n \land ac_{j-ab} = f\text{AbortCh}(ac_j) \land ac_j \in f\text{NestedAction}(ac_i) \}$$

4.2.5 Coordinator-External Component

The transaction channel $a_i e_j$ binds the coordinator process of $ac_i$ to the external process of $ext_j$. Through this channel, the coordinator process requests the beginning of a new transaction on external process. When CA action ends, a new request is sent to commit or rollback the corresponding opened transaction. Figure 4(e) shows an example consisting of processes for one CA action $ac_1$ and two external components $ext_1$ and $ext_2$. These CSP coordination channels are formalized as follows:

**Definition 4.7** (Transaction Channels). *The set of communication channels for coordinator process of $ac_i$ used in transaction management of external components is defined as:*

$$\text{TransCh}_{ac_i} = \{ a_i e_j \mid 1 \leq j \leq r \land ext_j \in f\text{External}(ac_i) \}$$

4.2.6 Participant-External Component

The update channel $p_i e_j$ binds the participant process of $par_i$ to the external process of $ext_j$. Through this channel, participating components may request or update the external component’s state. Figure 4(f) shows an example consisting of processes representing one participating component $par_1$ and two external components $ext_1$ and $ext_2$.

In order to fully formalize the proposed architecture some definitions regarding update channels are given:

**Definition 4.8.** *The function $f\text{State} : Ext \rightarrow \mathcal{P} \text{Par}$ maps the subset of participating components which query or update the state of a given external component.***

**Definition 4.9** (Update Channels). *The set of communication channels for external process of $ext_j$ used in state management by participating components is defined as:*

$$\text{StateCh}_{ext_j} = \{ p_i e_j \mid 1 \leq i \leq m \land ext_j \in Ext \land par_i \in f\text{State}(ext_j) \}$$

5 Architectural Model Usage

Once the standard architectural model is defined, to specify a new concurrent fault-tolerant system, designers only need to adhere to the model by instantiating the general model with their system elements. They neither need to decide which communications are necessary (they are already defined) nor define the coordination mechanism because they are built-in processes. Figure 5 shows the CSP architectural model for the system introduced in Section 3. Note that all processes and channels have been defined following the proposed architecture.
5.1 Participants Specification

The Participant$_{par_j}$ process must be specified by the system designer using built-in processes which identify execution blocks. As each participating component executes in the context of several nested CA actions, its functional behavior is split into execution blocks. The built-in Execution$_{par_j, ac_i,k}$ process identifies the k-th execution block of the participating component par$_j$ in the context of CA action ac$_i$. The following constants for argument k are defined:

- FIRST - Identifies the first execution block in the context of ac$_i$.
- LAST - Identifies the last execution block in the context of ac$_i$.
- SINGLE - Identifies the single execution block in the context of ac$_i$.

The normal execution of a participating component can be interrupted due to exceptions raised by this or other participants. Whenever an exception is raised, all participants must start their exceptional execution to handle those exceptions and all execution blocks related to a specific CA action must be guarded with a corresponding exceptional execution block. The built-in Suspension$_{par_j, ac_i}$ process represents this exceptional execution block and implements the participant coordination protocol of cooperative exception handling. It guards the execution blocks through CSP interrupt operator $\text{\$}$.

For the example introduced in Section 3, participant processes should be specified as:

\[
\text{Participant}_\text{par}_1 = (\text{Execution}_\text{par}_1, \text{ac}_1, \text{FIRST}; \text{Execution}_\text{par}_1, \text{ac}_1, \text{SINGLE} \triangle \text{Suspension}_\text{par}_1, \text{ac}_1; \\
\text{Execution}_\text{par}_1, \text{ac}_1, \text{LAST}) \triangle \text{Suspension}_\text{par}_1, \text{ac}_1)
\]

\[
\text{Participant}_\text{par}_2 = (\text{Execution}_\text{par}_2, \text{ac}_2, \text{FIRST}; \text{Execution}_\text{par}_2, \text{ac}_2, \text{SINGLE} \triangle \text{Suspension}_\text{par}_2, \text{ac}_2; \\
\text{Execution}_\text{par}_2, \text{ac}_2, \text{LAST}) \triangle \text{Suspension}_\text{par}_2, \text{ac}_1)
\]

Participants par$_1$ and par$_2$ first engage into action ac$_1$, then into the nested action ac$_2$, and finally into ac$_1$.

5.1.1 Specification of Normal Behavior

Figure 6 shows the CA action conceptual model for the example introduced in Section 3 with the interactions between the components identified. These interactions
must be represented as CSP mutual participating events. When CA action $ac_1$ starts, component $par_1$ interacts with $par_2$. This interaction is externally observed by event $a$. When action $ac_2$ is introduced, another interaction is observed, event $b$. Then, $par_1$ executes an internal procedure with no communication between participants, but externally observed by event $c$. Meanwhile, participating component $par_2$ executes an internal procedure (event $d$) and updates the external component’s state with a new value (event $update$). Back to the $ac_1$ context, $par_2$ executes an internal procedure (event $e$) and finally interacts with $par_1$ through event $f$.

In the proposed architectural model we also specify the process $Normal_{par_1,ac_1,k}$ which implements the $k$-th normal execution block for participant $par_1$ in the context of $ac_1$. In the example system, the following specifications would be provided for participant $par_1$:

$Normal_{par_1,ac_1,first} = a \rightarrow SKIP$

$Normal_{par_1,ac_1,single} = b \rightarrow c \rightarrow SKIP$

$Normal_{par_1,ac_1,last} = f \rightarrow SKIP$

The specification of normal behavior for participant $par_2$ includes the built-in process $Update_{par_1,ext_1,state}$ which updates the external component’s state:

$Normal_{par_2,ac_2,first} = a \rightarrow SKIP$

$Normal_{par_2,ac_2,single} = b \rightarrow d \rightarrow Update_{par_2,ext_1,new\_state}$

$Normal_{par_2,ac_2,last} = e \rightarrow f \rightarrow SKIP$

5.1.2 Specification of the Exceptional Behavior

Figures 7(a) and 7(b) show the exceptional interactions between participating components $par_1$ and $par_2$ during cooperative exception handling in the context of CA actions $ac_1$ and $ac_2$, respectively.

During CA action $ac_1$, participating component $par_1$ executes different activities depending on the exception raised: for exception $exp_1$, an internal activity with event $g$ is initially executed; for exceptions $exp_2$ and $exp_3$, other activities involving events $h$ and $l$, respectively, are executed. Then, $par_1$ interacts with $par_2$ afterwards, observed by event $k$. Execution of participating component $par_2$, regarding action $ac_1$, is simpler because it implements a default exceptional behavior for all excep-
tions. It is described by an internal activity (event $j$) and an interaction with $p_{ar1}$ (event $k$).

In this particular example, the exceptional behavior of participating components $p_{ar1}$ and $p_{ar2}$ involves a single interaction, represented by event $e$, in the context of CA action $ac_2$. In the proposed CSP architectural model, we must specify the exceptional behavior of participating component $p_{arj}$ in the context of $ac_i$ using the user-defined process $\text{Exceptional}_{p_{arj},ac_i,exp}$. Although the system designer is free to define the behavior accordingly to the system exceptional requirements, a standard must be followed: the $\text{Exceptional}_{p_{arj},ac_i,exp}$ process must finish with a standard CSP event requesting either a normal or an exceptional end. These terminations have been implemented by built-in processes $\text{RequestNormalEnd}_{p_{arj},ac_i}$ and $\text{RequestExceptionalEnd}_{p_{arj},ac_i}$. For the example system, the following specifications should be provided:

$$\begin{align*}
\text{Exceptional}_{p_{ar1},ac_1,exp} &= \begin{cases} 
g \to k \to \text{RequestNormalEnd}_{p_{ar1},ac_1} & \text{if } exp = exp_1 
\end{cases} \\
\text{Exceptional}_{p_{ar2},ac_2,exp} &= e \to \text{RequestNormalEnd}_{p_{ar2},ac_2} 
\end{align*}$$

and $\text{Exceptional}_{p_{ar2},ac_2,exp} = e \to \text{RequestExceptionalEnd}_{p_{ar2},ac_2}$.

6 Coordinator and External Specification

The previous section described how the designer should specify participant processes within the proposed CSP fault-tolerant architecture. This section is concerned with the specification of other three fundamental processes: $\text{Coordinator}_{ac_i}$, $\text{CoordinatorState}_{ac_i,\text{state}}$, and $\text{External}_{\text{ext}_i}$. These built-in processes implement the exception handling coordination and transaction management protocol and are ready to be instantiated by system designers. The complete CSP specification of the protocol is out of the scope of this paper, once it is confined to present the architectural model (the complete protocol specification is in [18]).

$$\text{Coordinator}_{ac_i} = \begin{cases} 
\text{RecInitRequest CoordCh}_{ac_i,ac_i} & \text{if } ac_i = ac_1 
\text{RecInitRequest CoordCh}_{ac_i,ac_i} & \text{if } ac_i = ac_2 
\triangle \text{RecAbortRequest}_{ac_i} & \text{otherwise} 
\end{cases}$$

$$\text{CoordinatorState}_{ac_i,\text{state}} = \begin{cases} 
\text{ac}_i\_\text{sync} := \text{SyncCh}(ac_i); \text{ac}_i\_\text{state} := \text{StateCoordCh}(ac_i); 
\text{ac}_i\_\text{sync}\_\text{init} \to \text{CoordinatorState}_{ac_i,\text{STARTED}} \triangle \text{RequestCoordCh}_{ac_i,ac_i} 
\text{ac}_i\_\text{sync}\_\text{ok} \to \text{CoordinatorState}_{ac_i,\text{STOPPED}} \triangle \text{RecAbortCoordCh}_{ac_i,ac_i} 
\text{ac}_i\_\text{sync}\_\text{abort} \to \text{CoordinatorState}_{ac_i,\text{STOPPED}} \triangle \text{RecAbortCoordCh}_{ac_i,ac_i} 
\text{ac}_i\_\text{sync}\_\text{handle} \text{req} \to \text{CoordinatorState}_{ac_i,\text{state}} \triangle \text{RecInitCoordCh}_{ac_i,ac_i} 
\text{ac}_i\_\text{state}\_\text{get} \text{state} \to \text{CoordinatorState}_{ac_i,\text{state}} \triangle \text{End} 
\end{cases}$$

The example introduced in Section 3 has two CA actions, two participating components and one external component. In the last section, the processes $\text{Participant}_{p_{ar1}}$ and $\text{Participant}_{p_{ar2}}$, describing these participating components, were specified. In order to complete the specification of the system fault-tolerant architecture, it is necessary to instantiate the remaining fundamental processes and compose them using the parallel CSP operator, as described below:

$$\text{Syst} = \parallel \text{Participant}_{p_{ar1}} \parallel \text{Participant}_{p_{ar2}} \parallel \text{Coordinator}_{ac_1} \parallel \text{Coordinator}_{ac_2} \parallel \text{CoordinatorState}_{ac_1,\text{STOPPED}} \parallel \text{CoordinatorState}_{ac_2,\text{STOPPED}} \parallel \text{External}_{\text{ext}_1}$$
7 Conclusion

Some previous studies have addressed the formalization of the CA action model. Schwier [23] formalizes some properties based on temporal logic using TLA (Temporal Logic of Actions) [16] notation. However, that work does not provide a general framework for the specification of fault-tolerant systems, but specify some properties regarding cooperative exception handling and action termination. The work by Tartanoglu [26] presents a general framework in B [1] for the specification and development of systems based on CA action model. Although it is considered a good option on the formalization of structure and state properties, B is not suitable for the specification of concurrency, synchronisation and interaction patterns between system components. Reo[3] is also a channel based model formalized using a coalgebraic approach. It is, however, concentrated on the formalization of connectors instead of giving a general architectural model.

Castor [12] proposes a formal specification and verification method for fault-tolerant systems, based on object-oriented language Alloy [15]. As that work is concerned with the formalization of exception representation (i.e. internal and external exceptions) and exception flow, the dynamic behavior of the system, in terms of component interaction, is not described in details. More recent studies [10,8] propose a fault-tolerant architectural solution in terms of ADL components and connectors based on the concept of IFTComponent and collaborative exception handling. However, they do not formalize the exception handling coordination protocol.

The main contribution of the present work is on the definition of a simple, formal and standard architecture composed of built-in processes and channels responsible for providing a sound foundation for the specification of fault-tolerant systems in CSP, according to the CA action structuring model. The predefined architecture allows a designer to specify new component based systems (in CSP) by identification of the main components and their exceptions only; the elements necessary to coordinate the exception mechanisms and their corresponding behavior are all provided by the predefined architecture. Also, due to the formalization means, properties can be proved over the system specification. The coordination protocol for cooperative handling of concurrent exceptions and the complete specification for built-in processes are presented in a companion article [18]. The built-in CSP processes have been implemented in CSPm language in order to allow the practical specification and verification of real fault-tolerant and concurrent systems by a system designer. The CSPm specification can be simulated in the ProBE [19] tool or formally verified in FDR [11] tool.

References

Pereira and de Melo


Formalization of component substitutability

Meriem Belguidoum\textsuperscript{1} and Fabien Dagnat\textsuperscript{2}

\textit{Department of Computer Science}
\textit{ENST Bretagne, Technopole Brest-Iroise}
\textit{Brest, France}

\section*{Abstract}
Component-Based Software Engineering (CBSE) is increasingly used to develop large scale software. In this context, a complex software is composed of many software components which are developed independently and which are considered as black boxes. Furthermore, they are assembled and often dependent from each other. In this setting, component upgrading is a key issue, since it enables software components to evolve. To support component upgrading, we have to deal with component dependencies which need to be expressed precisely. In this paper, we consider that component upgrade requires managing substitutability between the new and the old components. The substitutability check is based on dependency and context descriptions. It involves maintaining the availability of previously used services, while making sure that the effect of the new provided services do not disrupt the system and the context invariants are still preserved. We present here a formal definition and a verification algorithm for safe component substitutability.

\textit{Keywords:} Component, Upgrading, Substitutability

\section{Introduction}
Component based software has gained recognition as the key technology for building high quality and large software. In this setting, sharing collections of components has become common practice for component oriented applications. These components are independently produced and developed by different providers and reused as black boxes making it necessary to identify component dependencies to guarantee interoperability.

According to Szyperski’s definition [11], a component is a unit of composition with contractually specified interfaces and explicit dependencies. An interface describes the provided and the required services of a component. Software consists of the assembly of components in an architecture, by binding a required interface of one component to an offered interface of another component.

In this context, upgrading a component is difficult because this component may be used by several software applications. More generally, replacing a component

\textsuperscript{1} Email: meriem.belguidoum@enst-bretagne.fr
\textsuperscript{2} Email: fabien.dagnat@enst-bretagne.fr

This paper is electronically published in
\textit{Electronic Notes in Theoretical Computer Science}
\textit{URL: www.elsevier.nl/locate/entcs}
$C_{old}$ with a component $C_{new}$ in a system $S$ requires that it does not disrupt $S$. This property is often described as substitutability \[5\].

Several techniques exist to ensure substitutability between components, see for example \[12,13\]. All these approaches are built upon the substitution principle of Liskov introduced in \[1\] in the context of object oriented programming. They use the interface type to define a subtyping relation between components and then authorize $C_{new}$ to replace $C_{old}$ only if $C_{new}$ is a subtype of $C_{old}$. Various forms of those types exist, starting with the classical interface type \[10\] and enhancing them with behavioral description such as automata for example \[6\]. Some related research \[12,13\] show that the resulting condition of pure subtyping ensures safety of the replacement but is too restrictive. Recent work \[5\] has shown the limits of this approach and proposes a less restrictive notion of substitutability depending on the context. In this setting, $C_{new}$ may safely replace $C_{old}$ in certain systems. In fact, all interfaces not used in the context are ignored when ensuring the subtyping.

To extend our previous work on the formalization of safe component installation and deinstallation \[2\], we tried to define contextual substitutability to build a safe replacement operation following the previously cited approaches. But it appeared that the resulting rule needs to be enhanced to reach safety. While in other work the new services provided by $C_{new}$ do not have to meet any requirement, in our setting they may conflict with the context requirements.

Generally, replacing a component $C_{old}$ by a new one $C_{new}$ has an effect on the context and to maintain safety, we have to check that effect will not break system invariants. In previously mentioned work, the only effect taken into account is the services that $C_{new}$ provides. Therefore, ensuring substitutability consist in ensuring compatibility of $C_{new}$ provided services with the requirements of component that were previously using $C_{old}$ services. This compatibility can have different contract levels (Syntactical, behavioral, synchronisation and quality of service) as described in \[4\]. This paper advocate adding the verification of component deployment effects (upgrade) on the target system. In the deployment, it appears that upgrade effect must not disturb the target system. For this, we propose a substitution principle ensuring that (1) the new component still provides all the services used in the context (as usual) and that (2) new provided services do not conflict with this context (effect verification). The formalization of this substitutability enables us to provide a safe and flexible replacement operation for our deployment system.

The paper is organized as follows. Section 2 introduces our dependencies description and illustrates it with the example of a mail server in Linux GNU. In Section 3, we present our substitutability approach with a progressive refinement of substitutability definitions. Section 5 describes the substitutability checking algorithm. Section 4 illustrates some substitutability examples. Section 6 discusses related work. Finally, Section 7 concludes and discusses future work.

## 2 Dependency description

In this section, we present the precise definition of the relation between a required and a provided service, either of the same component or of two different components. Such a relation is called a dependency.
Fig. 1. A mail server assembly

Fig. 1 illustrates a simplified architecture of a mail server on a Linux system. It is composed of five components: POSTFIX, an SMTP server playing the role of a Mail Transport Agent (MTA), FETCHMAIL that recovers mails from a distant server like Pop or IMAP using a mail transport protocol, PROCMAIL, a Mail Deliver Agent (MDA) that manages received mail and enables, for example, mail to be filtered. THUNDERBIRD, a mail manager for reading and composing mail called a Mail User Agent (MUA). Finally, ORGANIZER is an inbox organizer, which allows mailing lists, web pages and users e-mails to be managed. Each component is represented in Fig. 1 by a rectangle with the required interfaces (half circles and square brackets on left side) and the provided interfaces (black circles on right side). Requirements in the left hand side of a component, may be of two kinds: (1) software requirements (i.e., services provided by other components), for example libraries (half-circles) or (2) system requirements expressed by comparison of variables with values represented by square brackets, for example requiring a certain amount of free disk space ($FDS \geq 1380\text{ Ko}$).

In this example, the two forms of dependencies, respectively intra-dependencies and inter-dependencies, are represented respectively by lines inside components and links between components (like PROCMAIL and THUNDERBIRD, PROCMAIL provides MDA and THUNDERBIRD requires it). There are three main kinds of dependencies, either mandatory, optional or negative:

- a mandatory dependency (represented by a solid line) is a firm requirement. If it is not fulfilled, installation is not possible. For example, POSTFIX needs a terminal with a specific libraries ($lib1$), an amount of free disk space ($FDS \geq 1380\text{ ko}$), etc.

- an optional dependency (represented by a dashed line) specifies that the component may provide optional services. Such services may not be provided (if their requirements are not fulfilled) without preventing the installation. For example, POSTFIX may provide a service for scanning messages against viruses if the service amavis is available. Otherwise POSTFIX can be installed and provides the MTA.

---

3 To simplify the figure, only some interesting dependencies are represented.
service, but the service AntiVirus is not provided.

- a negative dependency (expressed by a negation) specifies a conflict forbidding installation. The conflict may be due to a service or a component. For example, as presented in Fig. 1, POSTFIX cannot be installed if the component SENDMAIL (another component providing MTA) is already installed in the target system.

Intra-dependencies are defined by the producer of the component and used to perform installation. Inter-dependencies result from installation and are used to perform deinstallation and replacement. The two notions are briefly presented below, more details on these concepts are given in [2].

2.1 Intra-dependencies

The intra-dependency description language uses the concepts of **dependency** and **predicate** defined by the following grammar where $s$ represents the name of a service and $c$ the name of a component:

$$
D ::= P \Rightarrow s \mid D \bullet D \mid D \# D \mid ? D \\
P ::= \text{true} \mid P \land P \mid P \lor P \mid R \\
R ::= [v O \text{val}] \mid \neg s \mid \neg c \mid c.s \mid s \\
O ::= > | \geq | < | \leq | = | \neq
$$

The precise semantics of these operators is described in detail in [2]. Intuitively, a dependency may be the conjunction $\bullet$ or the disjunction $\#$ of two dependencies, an optional dependency $?$ or a simple dependency $P \Rightarrow s$ specifying the requirements $P$ of the service $s$. If these requirements are fulfilled the service $s$ is available. The requirements are expressed in a first order predicate language with five conditions $(R)$ expressing a comparison of an environment variable with a value ([v O val]), a conflict with a service ($\neg s$), with a component ($\neg c$), the requirement of a service provided by a precise component ($c.s$) or any component ($s$). Examples of such predicates appear in Fig. 1 on the required interfaces represented in the left-hand side of a component.

2.2 Inter-dependencies

When a component is installed in a system $S$, each of its requirements is fulfilled by binding it to any existing component of $S$ satisfying the requirement. This binding is what we call an inter-dependency. It is the result of installation and is required to ensure safe deinstallation and replacement. We have chosen to represent inter-dependencies by a dependency graph (see [2] for more details). A node of a dependency graph is an available service $s$ with its provider ($c.s$) and an edge is a pair of nodes $n_1 \rightarrow n_2$ meaning that $n_2$ requires $n_1$. Each edge is labeled (above the arrow) by the kind of dependency, either mandatory $M$ or optional $O$. Fig. 2 presents the dependency graph of the mail server of Fig. 1. We can see that some solid (resp. dashed) lines inside components in the Fig. 1 (intra-dependencies) are reflected in Fig. 2 by solid (resp. dashed) edges. For example, POSTFIX depends on lib1 which is provided by a component C1, so the used service is C1.lib1. This dependency is a mandatory one (solid line in Fig. 2). POSTFIX has also an optional dependency; it
Fig. 2. Dependency graph of the mail server of Fig. 1

can provide an Anti virus (POSTFIX.AntiVirus) if a service amavis provided by a component (for example AMAVIS) is available. This dependency is represented in the graph by a dashed line. The system requirements (like $FDS \geq 1380$) and negative dependencies ($\neg$SENDMAIL) are not represented in the graph. The inter-dependencies between components in Fig. 1 are represented in the graph as mandatory edges between services, for example the service popclient provided by FETCHMAIL is linked with service popclient required by PROCMAIL.

3 What is substitutability?

In this section, we present and analyze progressively the substitutability problem, we propose definitions and rules to check the correctness and safety of substitutability. In general, two forms of compatibility between components can be defined: vertical compatibility and horizontal compatibility. The vertical compatibility is called substitutability, it expresses the requirements that allow the replacement of one component by another ($C_{old}$ by $C_{new}$ in Fig. 3). The horizontal compatibility expresses connexion between a provided service of a component and a required service of another component ($C_{old}$ used by $C_{client}$ in Fig. 3). When substituting the component $C_{old}$ for the component $C_{new}$, we have to ensure that the component $C_{client}$ can use the services provided by $C_{new}$ as it used previously those provided by $C_{old}$ and the new provided services do not conflict with $C_{client}$ and all other client components.

3.1 Substitutability definitions

Following the current trend, we define two kinds of substitutability, one addressing substitutability in a particular context and the other independent of the context. The definitions of strict and contextual substitutability are given below and are inspired by those of Brada in [5].

**Definition 3.1 (Strict substitutability)**

A component $C_{old}$ is **strictly substitutable** for a component $C_{new}$, if the latter can
replace $C_{\text{old}}$ in all contexts.

**Definition 3.2 (Contextual substitutability)**

A component $C_{\text{old}}$ is **substitutable** in a context $Ctx$ for a component $C_{\text{new}}$ if the latter can replace $C_{\text{old}}$ in the context $Ctx$.

Contextual substitutability is related to the context which represents the resources and the architecture of the target system. Ideally, it could be the union of the dependencies of all components (part of the system). The resulting description of the context would be a huge logical term. Its manipulation when deciding whether to authorize a deployment operation would be difficult and expensive (in calculation). Thus, we have chosen instead a safe approximation of the context description. The context definition is presented in [2]. It is summarized as follows:

**Definition 3.3 (Context)**

The *Context* is composed of (1) an environment $E$ storing the values of environment variables (OS, disk space, etc.), (2) a set $C$ of four-tuples $(c, P_s, F_s, F_c)$ storing for each installed component $c$ its provided services $P_s$, forbidden services $F_s$ and forbidden components $F_c$ \footnote{The required services of a component are stored in the dependency graph not in the component tuple.} and (3) a dependency graph $G$ storing the dependencies (the required and the provided services of each component and the relation between them).

### 3.2 Component substitutability

To decide whether a component $C_{\text{new}}$ can substitute a component $C_{\text{old}}$, it is necessary to compare what they provide and what they require. Indeed, the provided (or required) services of $C_{\text{new}}$ can be the same or different from those of $C_{\text{old}}$. We therefore have to study all the possibilities. Fig. 4 depicts the different possible relations between the old and the new set of provided (resp. required) services:

- **case 1**: the set of provided (resp. required) services of $C_{\text{new}}$ is included in the set of provided (resp. required) services of $C_{\text{old}}$.
BELGUIDOUM AND DAGNAT

- case 2: the set of provided (resp. required) services of \( C_{new} \) and \( C_{old} \) are equal;
- case 3: the set of provided (resp. required) services of \( C_{old} \) is included in the set of provided (resp. required) services of \( C_{new} \);
- case 4: the two sets are different from each other and can have some services in common.

![Diagram](image)

**Fig. 4. Comparison according to the old and the new service sets**

There are four cases for provided services combining with four cases for required services leading to sixteen possibilities. To illustrate these cases, we suppose that the component \( C_{old} \) provides the services \( PS_1 \) and \( PS_2 \) and requires the services \( RS_1 \) and \( RS_2 \). Table 1 represents the different forms that the component \( C_{new} \) may have, depending on its provided and required services. Each cell of the table corresponds to numerous possible components and is here represented by one possible component for illustrative purposes only.

<table>
<thead>
<tr>
<th>Requires</th>
<th>Provides</th>
<th>more</th>
<th>same</th>
<th>fewer</th>
<th>different</th>
</tr>
</thead>
<tbody>
<tr>
<td>more</td>
<td></td>
<td>• ( RS_2 ) • ( PS_1 )</td>
<td>• ( RS_2 ) • ( PS_2 )</td>
<td>• ( RS_2 ) • ( PS_3 )</td>
<td></td>
</tr>
<tr>
<td>same</td>
<td></td>
<td>• ( RS_2 ) • ( PS_1 )</td>
<td>• ( RS_2 ) • ( PS_2 )</td>
<td>• ( RS_2 ) • ( PS_3 )</td>
<td></td>
</tr>
<tr>
<td>fewer</td>
<td></td>
<td>• ( RS_2 ) • ( PS_1 )</td>
<td>• ( RS_2 ) • ( PS_2 )</td>
<td>• ( RS_2 ) • ( PS_3 )</td>
<td></td>
</tr>
<tr>
<td>different</td>
<td></td>
<td>• ( RS_2 ) • ( PS_1 )</td>
<td>• ( RS_2 ) • ( PS_2 )</td>
<td>• ( RS_2 ) • ( PS_3 )</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1**
Substitutability possibilities

In fact, the sixteen possible cases can be refined to table 2 below, containing eight possibilities combining only three conditions:

- ensure new requirements (\( NR \)) of \( C_{new} \). For example, in line 1, is \( RS_3 \) satisfied?
- ensure no conflicts (\( NC \)) between the new services of \( C_{new} \) and the system. For example, in column 1, is \( PS_3 \) in conflict with the system?
ensure no previously provided services are necessary for the system (NON). For example, in column 3, is \( PS_2 \) (previously provided by \( C_{old} \) and not provided by \( C_{new} \)) necessarily used?

<table>
<thead>
<tr>
<th>Requires</th>
<th>Provides</th>
<th>more</th>
<th>same</th>
<th>fewer</th>
<th>different</th>
</tr>
</thead>
<tbody>
<tr>
<td>same / fewer</td>
<td>NC</td>
<td></td>
<td></td>
<td>NON</td>
<td>NC+NON</td>
</tr>
<tr>
<td>different / more</td>
<td>NR+NC</td>
<td>NR</td>
<td></td>
<td>NR+NON</td>
<td>NR+NC+NON</td>
</tr>
</tbody>
</table>

Table 2: Substitutability conditions

This table shows the different substitutability conditions on the context. The only cell corresponding to strict substitutability is the empty one. The condition is then that \( C_{new} \) requires the same thing or less than \( C_{old} \) and provides the same services. The seven other cells represent contextual substitutability. Necessary and sufficient conditions (NSC) for strict and contextual substitutability are defined as follows:

**NSC 1 (Strict substitutability)** A component \( C_{old} \) is strictly substitutable for a component \( C_{new} \) iff they provide the same services and \( C_{new} \) has the same or fewer requirements than \( C_{old} \).

**NSC 2 (Contextual substitutability)** A component \( C_{old} \) is substitutable for a component \( C_{new} \) in a context \( Ctx \) iff:
- all the new requirements of \( C_{new} \) are satisfied in \( Ctx \) (NR).
- none of the new provided services is in conflict with \( Ctx \) (NC).
- none of the services provided by \( C_{old} \) not provided by \( C_{new} \) is used necessarily within \( Ctx \) (NON).

Compared to existing substitutability approaches, the condition (NC) is original because it enables to take into account various form of component effects on the context (potential conflicts that can occur due to the new component) and maintaining the safety of the system. In an extension of our system not presented here, we have the specification of non-functional properties. Replacing a component by another may have an impact on the system properties and therefore may be forbidden. An example of such substitution is further discussed in Section 6.

### 3.3 Ensuring substitutability in our context

To ensure substitutability in our system, it is necessary to:
- determine which case is examined,
- evaluate the corresponding conditions among NR, NC and NON.

Using our dependency descriptions presented in section 2 it is easier to calculate and compare provided services than required ones. In our approach, we do not consider requirements because the conditions are described in predicate logic and it is rather complex to compare requirements for each provided service. Therefore,
we check substitutability according to provided services only as follows:

(i) **NR** and **NC**: to check the new requirements (NR) and prevent new conflicts (NC), we reassess the installability condition of the new component \( C_{\text{new}} \). This condition ensures, on the one hand, that all the component requirements are fulfilled (NR) and, on the other hand, that the provided services are not in conflict with context (NC). Therefore, the NR and NC conditions correspond to ensuring installability as presented in [2]: \((Ctx \vdash C_{\text{new}} : D_{\text{new}})\).

(ii) **NON**: this condition is based on the calculation of provided services from the right-hand member of the dependencies. So, for each provided service of \( C_{\text{old}} \) which is not provided by \( C_{\text{new}} \) we have to check that it has no mandatory dependency (i.e., it is a leaf in the dependency graph) or it is only used (directly or indirectly) by optional services (in the graph, all paths coming from it must be optional, see definition 3.4). In fact, it corresponds to the deinstallation requirements of [2].

**Definition 3.4 (Mandatory dependencies (MD))** The set of mandatory dependencies \( (MD) \) of a service \( s \) provided by a component \( c \) \((c.s)\) in a dependency graph \( (G) \) is defined as follows:

\[
MD(G, c.s) = \bigcup \{\{c'.s'\} \cup MD(G, c'.s') | c, s \xrightarrow{M} c'.s' \in G\}
\]

The condition NON can be expressed as follows:

\[
\bigcup \{(MD(G, C_{\text{old}}.s) | s \in (C_{\text{old}}.P_s \setminus C_{\text{new}}.P_s)\} = \emptyset
\]

We summarize the different substitution conditions for the four cases illustrated in Fig. 4 and table 2 as follows:

- providing more (case 3): \( C_{\text{new}} \) is installable (NR+NC);
- providing the same (case 2): \( C_{\text{new}} \) is installable (NR);
- providing less (case 1): \( C_{\text{new}} \) is installable (NR) and services from \( C_{\text{old}}.P_s \setminus C_{\text{new}}.P_s \) are not used necessarily (NON);
- different (case 4): \( C_{\text{new}} \) is installable (NR+NC) and services from \( C_{\text{old}}.P_s \setminus C_{\text{new}}.P_s \) not used necessarily (NON).

## 4 How substitutability is checked?

The substitutability handled in our system is only a contextual one. We have to calculate the context denoted \( Ctx \) without \( C_{\text{old}} \) \((Ctx \setminus C_{\text{old}})\), i.e., simulate the effect of removing from the context the component \( C_{\text{old}} \) with its four-tuple \((C_{\text{old}}, P_s, F_s, F_c)\). Then, we have to check the installability of \( C_{\text{new}} \) in the resulting context \((Ctx \setminus C_{\text{old}})\).

The formal definition of contextual substitutability is presented below:

**Theorem 4.1 Contextual substitutability**

A component \( C_{\text{old}} \) is substitutable for a component \( C_{\text{new}} \) in a context \( Ctx \) if:

- \( C_{\text{new}} \) is installable in \( Ctx \setminus C_{\text{old}} \);
Calculation of \( Ctx' = Ctx \setminus c_{old} \)

**Diagram:**

- Yes: \( C_{new} \) installable in \( Ctx' \)?
- No:
  - Calculation of \( C_{new}.P_s \)
  - Calculation of \( C_{odd}.P_s \)
  - \( C_{old}.P_s \setminus C_{new}.P_s = \emptyset \)
  - NON
  - Yes: \( C_{old} \) is substitutable
  - No: \( C_{old} \) not substitutable

**Fig. 5. Substitutability phases**

- all provided services of \( C_{old} \) which are not provided by \( C_{new} \) must not be used necessarily in the context (NON condition):
  \[ \bigcup \{ (MD(\mathcal{G}, C_{old}.s) | s \in (C_{old}.P_s \setminus C_{new}.P_s) \} = \emptyset \]

Substitutability is checked as depicted in the diagram of Fig. 5. First, the new context \( Ctx' \) is calculated without the old component \( C_{old} \). So, \( Ctx' \) is \( Ctx \) without the set of all provided services of \( C_{old} \) and without its forbidden services and forbidden components. Then, we check whether the new component can be installed using installability rules in the new context, i.e., all \( C_{new} \) requirements are fulfilled in \( Ctx' \) and its provided services does not conflict with \( Ctx' \) (the rules are described in [2]). Once the installation of the new component is possible in the new context, we calculate the effect of its installation in the context from its dependency description using installation rules described in [2], i.e., its provided services, forbidden services, forbidden component and the new dependency graph (it is illustrated in the diagram of the Fig. 5 by the calculation of \( C_{new}.P_s \)).

Since the set of provided services depends on the availability of services in the context, we need to use installation rules to calculate it. The two main phases of substitutability are the installability and the calculation of provided services (installation) which depend on the context description and the component dependency description \((D_1 \bullet D_2, D_1 \# D_2, \) or \(?D)\). The calculation of provided services is not obvious without evaluating the dependency description in the context. Even if the component is installable the provided services depend on fulfilled dependency conditions in the context. Therefore, we present the calculation of provided services depending on the dependency descriptions \((D_1 \bullet D_2, D_1 \# D_2, \) or \(?D)\):

- For \( D_1 \bullet D_2 \), \( D_1 \) and \( D_2 \) must be verified in \( Ctx' \), and the set of provided services is the union of the provided services of \( D_1 \) and those of \( D_2 \). For example,
considering the following description: \(((C_1.S_1 \Rightarrow S_2) \cdot (S_3 [FDS \geq 10] \Rightarrow S_4))\), the installability conditions are:

- \(S_1\) belongs to the set of provided services of \(C_1\) and \(S_2\) is not forbidden in the context and
- \(S_3\) belongs to the set of available services of the context, the condition \([FDS \geq 10]\) is verified and \(S_4\) is not forbidden in the context.

Thus, the provided services are \(C.P_s = \{S_2, S_4\}\) or \(\emptyset\).

- \(D_1\sharp D_2\) is verified if \(D_1\) is verified in \(Ctx\) or \(D_2\) is verified in \(Ctx\). For example, for: \(((C_1.S_1 \Rightarrow S_{text})\sharp(S_3 [FDS \geq 10] \Rightarrow S_{graph}))\), the installability conditions are:
  - \(S_1\) belongs to the set of provided services of \(C_1\) and \(S_{text}\) is not forbidden in the context or
  - \(S_3\) belongs to the set of available services of the context, the condition \([FDS \geq 10]\) is verified and \(S_{graph}\) is not forbidden in the context.

The set of provided services is \(C.P_s = \{S_{text}\}\) or \(\{S_{graph}\}\) or \(\emptyset\).

- \(?D\) is always installable, for example: \(?(C_1.S_1 \Rightarrow S_2)\), the set of provided services is \(C.P_s = \{S_2\}\) if \(S_1 \in C_1.P_s\) and \(S_2\) is not forbidden else \(\emptyset\).

Next, we compare the provided services of \(C_{old}\) with those of \(C_{new}\). \(C_{old}\) is substitutable in two cases: either the set of previously provided services which are no longer provided by \(C_{new}\) is empty or each of these services are not necessarily used by other components in the context.

### 5 Example

Let us illustrate component substitutability by examining two substitutability scenarios. First, the new component provides fewer services. Second, the new component provides more services. The first case may happen for optimizing purposes by replacing one component by another which requires fewer resources and provides fewer services. For example, we replace THUNDERBIRD by SYLPEEDE. We suppose that the service THUNDERBIRD.corrector (a spell checker) is the only service which is not provided by SYLPEEDE. The system must ensure the condition NON i.e., this service is not used by another component. According to the dependency graph of the mail server represented in Fig. 2 of section 2, the service THUNDERBIRD.corrector is a leaf in the graph. Thus, THUNDERBIRD.corrector is not used by another component and NON is ensured. The component THUNDERBIRD can be substituted by SYLPEEDE which provides fewer services if SYLPEEDE is installable, i.e., its required services are available in the context. The required services here are the libraries \((C_6.lib6)\) which means the libraries \(lib6\) provided by any component, for example \(C_6\) (see Fig. 6).

The second example addresses the substitution of a component providing more services. For instance, replacing THUNDERBIRD with the mail user agent of SEAMONKEY which has numerous enhancements, for example: a Chat service. (see Fig. 7).

Checking substitutability corresponds to ensuring the requirements of SEAMONKEY \((C_7.lib7, IRC.ircclient, SPELL.spell − Dict, etc.)\) which are different from those of THUNDERBIRD and ensuring that the additional provided services do not disrupt the
context. The related dependency graph is presented in Fig. 8.

Now, let’s illustrate our main contribution, the two substitution examples of
BELGUIDOUM AND DAGNAT

THUNDERBIRD presented above may not be possible even if conditions on provided services are fulfilled \((C_{old}.P_s \setminus C_{new}.P_s = \emptyset \text{ and NON})\). Indeed, according to the diagram of Fig. 5, we have to verify firstly the installability conditions of SEAMONKEY and SYLPHEED. One of the most important condition is the verification of the effect of the component in the context. If we suppose that the service chat is forbidden in the context, then the condition of the installability of SEAMONKEY is not verified. Therefore the substitution is not possible before comparing provided services of each component.

Finally, we can have invariants in the context that we need to preserve. For example, we may have to preserve the security level of a system by forbidding the installation of any service that can decrease the security level of the system (like the service ftp for example). Let’s suppose a component \(C_{new}\) which provides the service ftp and someone want to replace component \(C_{old}\) by \(C_{new}\). The fact that the service ftp decreases the security level will forbid the replacement of \(C_{old}\) by \(C_{new}\) even if it provides all necessary services and does not require too much.

6 Related work

The issue of component substitutability has already been addressed in literature. We mention here only those which are the closest to our work and summarize the most common approaches dealing with the substitutability problem.

In oriented object programming, the substitution principle of Liskov is a particular definition of subtype which was introduced in [1]. This concept of subtype is founded on the concept of substitutability, i.e., if S is a subtype of T then we can substitute objects of the type S for objects of the type T without deteriorating the desirable properties of a program. However, although the concept of subtype is approved in [10], the rules based on typing are rather restrictive. We therefore choose a more flexible approach that allows us to make choices according to the system and user requirements.

Substitutability is presented in [9] as a relation between the types of the old and the new components. This approach is based on a contract definition which is a consequence of the definition of the component type [8,7]. Substitutability is based on the definition of the compatibility between the component and all the elements with which it interacts. Compatibility is defined in terms of syntactic, semantic, and pragmatic contracts for operations, interfaces, ports and components. Thus, a component A is substitutable for another component B if its compatibility with other components is preserved after the substitution and the new required properties are checked. Our work follows the same principle but it is done on the interface signature only.

The concept of substitutability in [5] is defined for black box components. The principle is to check that the substitution of the component preserves the consistency of the preliminary configuration. The concepts of context of deployment, strict substitutability and contextual substitutability are defined. The representations of component specifications and the deployment context are based on the ENT model (Export, Needs, Ties). The definition of strict substitution is different from ours because it considers that the new component must provide at least the
same thing and requires at most the same thing as the old component (generalization of the needs and specialization of the requirements). In the case of "strict" substitutability there is no check for new required interfaces since those are not supposed to exist and they are "forbidden" by the strict subtyping case. Nevertheless, new provided interfaces are allowed and therefore checked. However, in our work, the verification is done for the new requirements as well as for the new provided services without using subtyping rules. We think that the strict substitutability is not really interesting because the component needed functionalities depend on the environment in which it is used and it does not verify the effects on the context and its invariants.

Despite enhancements in substitutability specification at signature, semantics and protocol levels, we believe that these works do not take into account the effect of the component. Indeed, they do not verify potential conflicts that can occur after substitution, due to new services. Therefore, in our approach we impose more constraints on the new provided services and ensure that they do not conflict with the existing context and its invariants are preserved. For example, when we want to substitute a component which provides \texttt{http} with another providing \texttt{http} and \texttt{ftp} services and the system forbid non secure services like \texttt{ftp}, such a substitution cannot occur. Furthermore, taking into account the potential effect of the new component on the system can be generalized. It is applied not only to conflicts but also to other kinds of effect. For example, the substitutability verification of non-functional properties needs such a mechanisms as the new component may conflicts with the system invariants. Another example of use is the resource consumption. The new component despite being functionally equivalent may not work because it consumes too much resources for the system. For this reason, we have to control the effect of the new component on the context.

7 Conclusion and Future work

In this work, we have presented a formalization of component substitutability. Our formalization is based on dependencies and context descriptions which are also used for installation and deinstallation phases in [2]. It aims at providing a safe and flexible component upgrade. The key concept is the comparison between dependency descriptions of the new and the old component. The comparison concerns provided services and does not take into account required services. We have defined the strict and the contextual substitution and we have concentrated only on contextual one. We have presented an analysis of different substitutability cases and summarized them into three key conditions. These conditions involve checking the installability rule of the new component (verifying requirement and ensuring that provided services will not conflict with the context of the system) and checking the effect of deinstallation of the old component using the dependency graph. A prototype implementing our proposal has been developed in Ocaml. Our objective is to ensure the safety of substitutability without being restrictive by authorizing all cases of substitutability. For example, replacing a component which has a lot of unused services with another which has fewer provided services (only those which are useful) is possible. This substitutability can also depends on a system policy or
property models as described in [3]. We focus on ensuring the safety of the system, i.e., verifying the requirements, the effect of the substitution and preserving context invariants, component and service properties.

Now, we aim to parametrize the substitutability check by policies and properties (for example, if the policy tries to optimize resources we cannot replace a component by another one which requires a lot of resources). Furthermore, we are working on a dependency description extension to express properties on services and components. As future work we aim at extending our system to overcome its two main limitations, which are:

- to substitute a component assembly, we have to calculate the dependency of a composite component using the dependencies of its sub-component.
- in our current approach, components and services are identified by their names. This identity must be extended to include interface type and property information. This means changing from name equivalence to a form of subtyping when determining dependencies between services. In such an approach, we could reuse behavioral substitutability such as [6] for example.

References


Substitutability Relations for Active Components

Youcef Hammal

Department of Computer Science
USTHB University
Algiers, Algeria

Abstract

This paper deals with the checking of active components substitutability at the specification level. We present new subtyping relations which allow us comparing behaviors of previous and new components respectively within interleaving and true concurrency semantics. These relations are based on optimal variants of branching and forth-back bisimulations in order to preserve service availability and correctness properties in highly reactive and complex systems.

Keywords: Component substitutability, Behavioral subtyping, Strong branching bisimulation.

1 Introduction

In recent years, the component-based development has gained widespread use thanks to its many advantages in terms of more productivity and reusability. Indeed, it promotes the design and the implementation of great systems based on distributed and interoperable components, each of which encapsulates one or more services so that the composite systems yield news services to the environment. However on the other hand, analysis and verification of such systems (especially reactive ones) become more difficult with regards to the features of the various components and their architectural links. So, to ensure many basic properties of the software product, both behavioral aspects of components and interaction protocols of combined components have to be specified and analyzed earlier (at design-time) to reduce the number of serious errors at the later stages of development. Nevertheless, most of architecture description languages focus only on component interfaces and ignore the specification of their behaviors. Their semantics are also not formally defined up to achieve the static analysis of composite systems [20].

Furthermore, the removal of any component and its substitution by another one cannot be made easily even if they offer both the same services. The success of this
Hammal

operation is strongly based on the fact that sequences of the previous component (inter)aotions remain in the new context. But those sequences may often not be the same from one component to another one.

Consequently, in order to validate the assembly of the various components, modeling their behaviors and their interaction protocols would be done by means of formal models with suitable combination of both functional and non-functional properties in an appropriate manner that helps us achieve the system analysis in a compositional way. Hence, many formal modeling languages are invented or derived from existing ones (such as Interface Automata [2], I/O Automata [12], StateCharts [19], and process algebras [10,16]) to document behaviors of components and connectors. According to this approach, and since above high level languages are often translated into labeled transition systems, we use them in this paper as a common basic formalism in order to describe active components implementations.

Furthermore, many languages are as well used to capture assumptions about the ways provided and required services of any component should be used when plugged into its environment. For instance, temporal logics [11], and UML interaction diagrams [19] are often used as a specification language to capture safety and liveness properties the component should fulfill. However, when proceeding to compatibility checking between implementation and specification documents, the latter are usually encoded into some kinds of automata (i.e., LTA formulas to Büchi automata [11], Sequence Diagrams to Interface Automata [5,9]). So, we prefer as well use an automata-based formalism to specify properties, thereby making it easier to check the adequacy of any implementation (refined) model with its specification by using the same equivalences which we propose to define substitutability relations.

In this paper, we define and discuss new substitutability relations between active components in reactive and safety-critical systems where involved components have to react to sporadic and periodic stimuli of their environment. As our components can be provided with either interleaving semantics or true concurrency semantics, we will define the substitutability relation in both these two contexts.

It is worth noting that every relation should be safe and preserve services availability so that any new component still fulfills properties which the oldest component used to fulfill. The rest of the paper is structured as follows: in the second section, we present the motivation of our approach and the related work. Next in section 3, we recall branching variants of bisimulation and propose a new fitting one to our setting. Then, we give and discuss new substitutability relations between components in section 4 and relate them in section 5 to the checking of properties to be fulfilled with respect to an interleaving framework. Afterward we present in section 6 a new substitutability relation considering true concurrency semantics. Finally, a conclusion is given in section 7.

2 Motivation and related work

Software systems evolve throughout the product life cycle [3]. That is to say; many of their software components are improved or added since functional and perfor-

---

2 An active component is a mutable shared component [15] which progresses within its own thread.
mance requirements regularly change, bugs are discovered and fixed. However, such substitution operations upon components often lead to two kinds of problems: unavailability of previously provided services and violation of global correctness properties that were previously respected.

In view of these risks and in order to find out the accurate criterion to compare new components against removed ones, it is worthy to recall the Liskov Substitution Principle [15]: “If for each object \( o_1 \) of the type \( S \) there is an object \( o_2 \) of the type \( T \) such that for all programs \( P \) defined in terms of \( T \), the behavior of \( P \) is unchanged when \( o_1 \) is substituted for \( o_2 \) then \( S \) is a subtype of \( T \)”. Accordingly, the substitutability problem can be defined as the verification of two conditions [15,3]:

(i) An updated component of a software system must continue to provide all services offered by its earlier counterpart.

(ii) And previously established system correctness properties must remain valid for the new versions of the software system.

In other terms, the first criterion of containment requires that every behavior of the old component is also a behavior of the new one. The second criterion is about compatibility and stipulates that the obtained assembly satisfies the correctness properties. Indeed, a component model viewed in isolation, can not be meaningfully validated because its correctness properties can only be expressed in respect of its cooperating components. Thus, when considering whether a particular component design is appropriate or not, one must not simply view it in isolation. One must also think about reasonable assumptions related to the intended interactions of the component within its environment.

Consequently, it is not enough that execution paths of a previous component be preserved in the new component behavior which is often augmented with new services. Substitutability checking should go beyond asserting that traces of the previous component are only prefixes of some traces of the latter component.

Obviously, such simulation and traces based substitutability relations [12,17,14] are too weak because they do not deal with the mutable shared objects and are not able to preserve correctness properties which may be violated by the new behaviors that these relations entirely discard. Thus, much work focuses on using failures based preorders and weak bisimulations [4,1,8,21] where new behaviors are only hidden. However we think that these kinds of relations are not strong enough to check faithfully substitutivity of active components whose event-driven and time-constrained behaviors may deeply alter the service availability and correctness properties in reactive systems.

So, we reinforce the equivalence relations such that the Liskov Substitution Principle [15] holds for active components in critical-safety systems. Accordingly, we use transition systems as component modeling formalism in the interleaving semantics context, and we propose new relations that are based on strong branching bisimulation which takes into account the hiding of new added provided services and possibly some old removed required services. Additionally, we use transition systems to specify the correctness properties which we check against components models by exploiting the same bisimulations we used in defining the subtyping relations. Thereafter, we discuss what kind of subtyping relations we should establish between
old and new components with regards to the fulfillment relation one would preserve.

Moreover, we address the issue of substitutability in the case of true concurrency semantics where previous subtyping relations are no longer suitable as explained later. Therefore, we propose an adequate relation based on a new branching variant of forth and back bisimulation upon asynchronous transitions systems.

3 Preliminary Concepts

Because of involved high-level concepts of active components, we deem that their behavioral models would be better depicted by means of high level modeling languages (such as StateCharts). Indeed, these formalisms cope well with specification and modeling of complex and reactive systems both in terms of how components communicate and collaborate and how they conduct their own internal behavior. On the other hand, we should assume that we have an adequate mapping (note it $\left[\cdot\right]$), that is the semantics function that maps the behavioral model of any component $C$ (whatever the modeling language we use) into a labeled transition system $G$ as follows: $\left[\left[C\right]\right] = G = \langle Q, \rightarrow, q_0 \rangle$ where:

- $Q$ is the set of all reachable component configurations (or status) with the starting node $q_0$ as its initial configuration. Note that for sake of simplicity, configurations are referred to as states of component $C$, although configurations are thoroughly different from the concept of states used in high level languages (as StateCharts).

- $\rightarrow \subseteq Q \times \Sigma \times Q$ are transitions between nodes labeled with events of $\Sigma$. This relation can be obviously extended as follows: a compound transition $(q_0 \xrightarrow{\omega} q_n / \omega = (a_1..a_n) \in \Sigma^*)$ is introduced to denote a sequence of basic transitions labeled with actions names of $\Sigma$ such that: $\exists q_1, ..., q_{n-1} \in Q \land q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} ... q_{n-1} \xrightarrow{a_n} q_n$.

- $\Sigma$ is a set of events. Note that the interface signature of $C$ is simply the subset $\Sigma_I \subseteq \Sigma$ that contains interactions of $C$ with its environment, each of which is related either to a provided service or a required service. However, actions from the set $\Sigma \setminus \Sigma_I$ are internal operations which the component carries out without any cooperation from its environment.

We recall that when substituting any component, its execution paths should be preserved by the new component behavior which can be augmented with new actions. Thus, traces of the previous component are prefixes of some traces of the latter component. Nevertheless, a subtyping relation based on traces is too weak because it is not able to take into account deadlock behaviors. Hence, many works focus on using failures based preorders. However, we think that these kinds of relations are not strong enough to check faithfully substitutivity of active components which behaviors may change deeply w.r.t. old ones. Indeed, such preorders are not the most discriminatory powerful and do not fit well with many cases where new component signature does not preserve some specific required services since those ones do not affect in any way the capabilities of the new component in terms of provided services. Thus, we recall below definitions of existing bisimulation variants [16,7,6] and introduce a new one we use in defining our substitutability relations.

Let $G_1$ and $G_2$ be two labeled transition systems (also referred to as graphs for more conciseness): $G_i = (Q_i, \rightarrow_i, \Sigma_i, q_{i0})$ for $i=1,2$ and let $\Sigma$ denote the set $\Sigma_1 \cup \Sigma_2$. 

FACS'07 Pre-Proceedings
Definition 3.1 A (strong) bisimulation is a binary relation $R \subseteq Q_1 \times Q_2$, such that for every $(p, q) \in Q_1 \times Q_2$, $pRq$ iff:

- $\forall a \in \Sigma, \forall p' \in Q_1 : p \xrightarrow{a} p'$ implies $\exists q' \in Q_2 : q \xrightarrow{a} q'$ and $p' R q'$,
- and conversely, $\forall a \in \Sigma, \forall q' \in Q_2 : q \xrightarrow{a} q'$ implies $\exists p' \in Q_1 : p \xrightarrow{a} p'$ and $p R q'$.

Two graphs $G_1$ and $G_2$ are bisimilar ($G_1 \approx G_2$) when their initial nodes are strongly bisimilar ($q_1^0 \approx q_2^0$).

This relation is too strong and does not take into account the unobservable action $\tau$ which is important to model new services to be hidden. So, we make use of the branching variant of bisimulation [7] considering the branching structure even in the presence of $\tau$-actions in such a way that internal decisions are preserved.

Definition 3.2 A branching bisimulation is a binary relation $R \subseteq Q_1 \times Q_2$, such that for every $(p, q) \in Q_1 \times Q_2$, $pRq$ iff:

- $\forall a \in \Sigma \cup \{\tau\}, \forall p' \in Q_1 : p \xrightarrow{a} p'$ then either: $a = \tau$ and $p' R q$, or $\exists q_1, q' \in Q_2 : q \xrightarrow{a} q'$ and $p R q_1$ and $p' R q'$,
- and conversely, $\forall a \in \Sigma \cup \{\tau\}, \forall q' \in Q_2 : q \xrightarrow{a} q'$ then either: $a = \tau$ and $p R q'$, $\exists p_1, p' \in Q_1 : p \xrightarrow{\tau} p_1 \xrightarrow{a} p'$ and $p_1 R q$ and $p' R q'$.

Two graphs $G_1$ and $G_2$ are branching bisimilar ($G_1 \approx_{br} G_2$) when their initial nodes are branching bisimilar ($q_1^0 \approx_{br} q_2^0$).

Two states $p$ and $q$ are equivalent if they can perform the same observable actions even though interleaved with $\tau$-actions. If some state $p'$ can be reached via $\tau$-steps from $p$ (see fig.1(a)), then it should be also equivalent to the state $q$, i.e. $p'$ remains able to achieve all actions the state $q$ can do either directly or via $\tau$-steps. So the capabilities (services) of an old component may sometimes be only preserved indirectly in the new component such that some of these services become unavailable at some intermediate points along a chain of $\tau$-steps.

Note that $\tau$-action is not under the sole control of the component and requires the cooperation of its environment. When plugging the new component instead of the old one in a composite system, new actions are used to yield new services. As
these new interactions may be synchronous, they may lead to unavailability of old services until they finish.

Accordingly, this relation does not comply well with the substitutability constraints between active components by allowing previous services to be interleaved with \( \tau \)-actions without compelling intermediate states to directly provide the previous actions. To overcome this drawback in a context of mutable shared objects, we introduce our new bisimulation which is stronger than the classical branching bisimulation in respect of services availability even throughout \( \tau \)-progress.

**Definition 3.3** A strong branching bisimulation is a binary relation \( R \subseteq Q_1 \times Q_2 \), such that for every \( (p, q) \in Q_1 \times Q_2 \), \( pRq \) iff:

- \( \forall a \in \Sigma \cup \{\tau\}, \forall p' \in Q_1 : p \xrightarrow{a} p' \) then either: \( a = \tau \) and \( p'Rq \), or \( \exists q' \in Q_2 : q \xrightarrow{a} q' \) and \( p'Rq' \),
- and conversely, \( \forall a \in \Sigma \cup \{\tau\}, \forall q' \in Q_2 : q \xrightarrow{a} q' \) then either: \( a = \tau \) and \( pRq' \), or \( \exists p' \in Q_1 : p \xrightarrow{a} p' \) and \( p'Rq' \).

Two graphs \( G_1 \) and \( G_2 \) are strongly branching bisimilar (\( G_1 \approx_{sbr} G_2 \)) when their initial nodes are strongly branching bisimilar (\( q_1^0 \approx_{sbr} q_2^0 \)).

Note that two states \( p \) and \( q \) are equivalent if they can perform directly the same observable actions in order to reach pairs of states always equivalent. Furthermore, if some state \( p' \) can be attained by one \( \tau \)-step from \( p \) then it should be also equivalent to the state \( q \) and preserve the same capabilities (see fig.1(b)). This strong constraint is due to the fact that the unobservable action is not an internal one under the control of the only involved component. Indeed in our setting, the unobservable action represents some new services that are not visible to clients of the previous component. So even though \( \tau \) is performed these clients should always be able to use the same services as those which were available before the \( \tau \)-progress.

As the strong branching bisimulation is weaker than the strong bisimulation and stronger than the classical branching one, we deduce from equivalence hierarchies given in [16,6] the following one (which proof is similar to that of proposition 5.6):

**Proposition 3.4** \( \approx \Rightarrow \approx_{sbr} \Rightarrow \approx_{br} \).

### 4 Substitutability in interleaving context

Our approach for handling substitutability among components is based on the idea that capabilities of a previous component should be preserved and may be augmented in the new one. Thus, the component signature (including required and provided services) has to be modified. Nevertheless, a question arises about what kind of services should be preserved or modified and how this modification could be achieved.

In our opinion, the main concern is to preserve and augment provided services of the component in such a way it continues fulfill any request related to previous services from its clients. In fact, a well-suited upgrading requires to always preserve the set of provided services even if they are augmented. However on the other hand, enhancements operated on the internal implementation may lead to the removal of some required services which become unnecessary as well as the addition of new
ones needed to realize these improvements.

Besides this, it is also possible to include internal (and unobservable) improvements on how these services are carried out. Thus, some required services can be removed as well as other required services can be appended in the new component. Indeed, some improvements can be operated on the internal behavior of this component such that some previous required services are no longer required in the new component. In addition, a number of new services would be claimed from its environment to achieve these improvements.

So what are the consequences of such assertions on the definition of substitutability? Obviously the observable behavior of the new component would include new paths containing additional provided and required services and discard all previous paths including removed required services.

Consequently, we distinguish two main cases:

4.1 Case 1: The new component adds new provided/required services without removing any required services

We obviously discard the strong bisimulation because it does not take into account the unobservable action $\tau$ which models all new services that have been added. Instead, we use others kinds of bisimulation taking care of the unobservable actions like branching bisimulation variants.

Let $C_1$, $C_2$ be two components. $\Sigma(C_i)$ is the set of events raised by a component $C_i$ ($i=1,2$). Some events of $\Sigma(C_i)$ are internal actions, whereas some others depict interactions with its environment (to provide or require services).

$[[C_i]]$ is the transition system modeling the behavior of $C_i$. It is derived from its high level models (for instance, StateCharts, Petri nets, etc.).

In order to compare two components, we use the hiding operator (denoted by $H$) as defined in process algebras to hide actions related to new services making them unobservable from outside. However, unobservable events in our setting are different from classical internal actions because the latter events are completely under control of the component, whereas the former events are under control of both the involved component and its environment. Hence, new services are unobservable only by some clients who view the enhanced component as it was the old one and thus still use only its old services. Consequently, we consider here only hidden actions depicting claimed old services and discard internal controlled events of a component which do not affect its external behavior w.r.t. availability of services.

Let $N = \Sigma(C_2) - \Sigma(C_1)$ be the set of new services. $H_N([[\cdot]])$ denotes the behavior model of $C$ where all events of $N$ are relabeled to $\tau$.

$$H_N(e) = \begin{cases} e & \text{if } e \not\in N \\ \tau & \text{if } e \in N \end{cases}$$

The basic idea of our comparison criteria is that old services which were available at some states in the old component should not be affected by the performance of new (hidden) actions in the new component and so have to remain available independently from them (i.e., before and after the $\tau$-action).

Hence, we give below the optimal substitutability relation which is faithful
enough in terms of preservation of capabilities for active components:

**Definition 4.1** \( C_2 \preceq_{sbr} C_1 \) if and only if \( H_N([\lfloor C_2 \rfloor]) \approx_{sbr} [\lfloor C_1 \rfloor] \).

The main target of this definition is to ensure \( C_2 \) to behave as \( C_1 \) (in respect of an external client) whichever the state that the component \( C_2 \) has reached.

Every time \( C_1 \) is able from a state \( s_1 \) (fig. 1(b)) to interact with its environment (by offering or demanding a service \( x \)) and reach some stable state \( s_2 \), \( C_2 \) should be also capable to do the same interaction \( x \) and to reach an equivalent state of \( s_2 \) even though after performing an unobservable activity. In this case, the intermediate states \( \{q_1', q_1''\} \) in \( C_2 \) should be strongly equivalent to the starting point \( s_1 \) of \( C_1 \) by offering the same actions as \( s_1 \). Indeed, unobservable events in \( C_2 \) do not affect its capability at state \( q_1' \) or \( q_1'' \) to simulate the oldest component \( C_1 \) which remains in its starting state \( s_1 \). So the intermediate states (namely, \( q_1 \), \( q_1' \) and \( q_1'' \)) should preserve directly all services offered or required at the state \( s_1 \).

Besides the above strong bisimulation, the weaker version of our strong substitutability relation is the following one which agrees with the subtyping relations proposed in [8,21].

**Definition 4.2** \( C_2 \preceq_{br} C_1 \) if and only if \( H_N([\lfloor C_2 \rfloor]) \approx_{br} [\lfloor C_1 \rfloor] \).

Hence, \( C_2 \) behaves as \( C_1 \) in respect of \( C_1 \) clients with intermittent loss of service availability at some intermediate states (fig. 1(a)).

Every time \( C_1 \) is able from a state \( s_1 \) to interact with its environment (by offering or demanding a service \( x \)) and reach some stable state \( s_2 \), \( C_2 \) is also capable to do the same interaction \( x \) and to reach an equivalent state of \( s_2 \) even though after performing unobservable activities. In this case, the intermediate states in \( C_2 \) are equivalent to the starting point \( s_1 \) of \( C_1 \) by offering the same actions but interleaved with \( \tau \)-actions. As these states are stable, this means that \( C_2 \) may refuse at these points to provide old services to a client which still requires them.

### 4.2 Case 2: The new component adds new provided/required services with removing some previous required services

Herein, we distinguish in \( \Sigma(C) \) two subsets:

- \( \Sigma_P(C) \) denotes the subset of events related to the provided services of \( C \).
- \( \Sigma_R(C) \) denotes the subset of events related to the required services of \( C \).

Let \( C_2 \) be the new component added instead of \( C_1 \). Let \( \alpha_P \), \( \alpha^1_R \), \( \alpha^2_R \) denote the following subsets:

- \( \alpha_P = \Sigma_P(C_2) - \Sigma_P(C_1) \), including new provided services.
- \( \alpha^1_R = \Sigma_R(C_2) - \Sigma_R(C_1) \), including required services added in the new component.
- \( \alpha^2_R = \Sigma_R(C_1) - \Sigma_R(C_2) \), including required services removed from the new component interface \(^3\).

Firstly, we propose to give the weak definition of substitutability in this case:

\(^3\) Removed actions may be some bugs or deprecated operations.
Definition 4.3 $C_2 \preceq_{\text{ibr}} C_1$ if and only if $H'_{\alpha^2_R}(\lceil [C_1] \rceil) \approx_{\text{ibr}} H_{\alpha_P \cup \alpha^2_R}(\lceil [C_2] \rceil)$.

In the first component $C_1$ model, we hide the required services removed from $C_2$. However in the second component model, we conceal the new provided and required services. The hiding operation in the two components is the same one, generating thus only $\tau$-actions as invisible actions. However, the hiding actually should not be the same in the two components because we are interested in the availability of old services only in the new component. In fact, since removed services from the old component $C_1$ are required services which are no more used by its clients, we can deduce that these actions are under the exclusive control of $C_1$ when considering the new framework. Therefore, these removed actions have to be hidden into internal actions ($i$-action). Whereas in $C_2$, hidden services are only concealed to clients of the old component and remain needed by new component clients which cooperate to carry out them. For this purpose, we define a new branching bisimulation which behaves strongly with respect to $\tau$-actions but behaves as a classical branching bisimulation with respect to $i$-actions.

Definition 4.4 A strong $i$-branching bisimulation is a binary and symmetric relation $R \subseteq Q_1 \times Q_2$, such that for every $(p, q) \in Q_1 \times Q_2$, $pRq$ iff:

- $a = \tau$ or $a = i$ and $p' \in Q_1 : p \xrightarrow{a} p'$ then either
  - or either at least one of the following cases holds:
    - $\exists q', q_1 \in Q_2 : q \xrightarrow{i} q_1 \xrightarrow{a} q'$ and $pRq_1$ and $p'Rq'$,
    - $\exists q' \in Q_2 : q \xrightarrow{a} q'$ and $p' \in Q_1 : p \xrightarrow{a} p'$.

Two graphs $G_1$ and $G_2$ are strongly $i$-branching bisimilar ($G_1 \approx_{\text{sibr}} G_2$) when their initial nodes are strongly $i$-branching bisimilar.

Two states are equivalent if they lead to new equivalent states via a same action even if interleaved with $i$-actions, but when one component achieves a $\tau$-step then its source and target states should be able to do the same actions as the equivalent state on the other component.

Hence, we give the new strong definition of substitutability in this second case:

Definition 4.5 $C_2 \preceq_{\text{sibr}} C_1$ if and only if $H'_{\alpha^2_R}(\lceil [C_1] \rceil) \approx_{\text{sibr}} H_{\alpha_P \cup \alpha^2_R}(\lceil [C_2] \rceil)$.

Note that $H'_{\alpha^2_R}(\lceil [C_1] \rceil)$ renames into $i$-action all actions of $\alpha^2_R$ in $C_1$ whereas $H_{\alpha_P \cup \alpha^2_R}(\lceil [C_2] \rceil)$ renames into $\tau$-action all actions of $(\alpha_P \cup \alpha^2_R)$ in $C_2$.

Example: in the figure 2(a), we hide the actions that do not exist both in the two graphs. The obtained graphs $G_1$ and $G_2$ are branching bisimilar but are not strongly branching bisimilar. To make the two graphs strongly branching bisimilar, the node $q_2$ in $G_2'$ should preserve the same capabilities (dashed lines) as the equivalent state $p_0$ in $G_1$ (see fig.2(b)).

5 Compatibility Checking

A formal description of any system is given both in terms of detailed descriptions of its components and assumptions about their intended interactions. We would
then perform adequacy checking between components behaviors as described in implementation models and interaction assumptions given as a specification model.

For this purpose, we adapt to our component-based framework a verification approach given in [13] where properties are seen as processes which are combined with the system model by a classical parallel operator. Then, the result has to remain equivalent to the system. However, our approach is more close to the refinement orientation and then takes a component specification as a global model pattern which any implementation should always match. This approach seems less complex to deal with than seeing the specification as some individual fragments of the intended behavior which we want the implementation to embed.

Accordingly, we introduce a definition of a more fitting synchronization product between graphs\(^4\) which can progress in an interleaving manner for the performance of the actions of some subset \(\Gamma\) but should synchronize for the other actions (\(\in \Gamma = \Sigma \setminus \Gamma\)). Let \(G' = < Q', \leadsto, \Sigma', q_0'> \mid i=1,2\) be two transition systems.

**Definition 5.1** Let \(\Gamma\) be a subset of \(\Sigma\). A \(\Gamma\)-synchronization product \(\otimes_{\Gamma}\) of two graphs \(G_1\) and \(G_2\) yields a new transition system \(G = < Q, \leadsto, \Sigma, q_0 >\) such that:

- \(Q = Q_1 \times Q_2\)
- \(q_0 = (q_0^1, q_0^2)\)
- \(\Sigma = \Sigma_1 \cup \Sigma_2\)

and \(\leadsto = \{ (p_1, p_2) \xrightarrow{a} (q_1, q_2) / a \notin \Gamma \land (p_1 \xrightarrow{a} q_1) \in \leadsto^1 \land (p_2 \xrightarrow{a} q_2) \in \leadsto^2 \}\)

\(\cup \{ (p_1, p_2) \xrightarrow{a} (q_1, q_2) / a \in \Gamma \land (p_1 \xrightarrow{a} q_1) \in \leadsto^1 \lor (p_2 \xrightarrow{a} q_2) \in \leadsto^2 \}\)

We can derive from \(\Gamma\)-Synchronization product two operators as follows:

- A full synchronization product \((\otimes_f)\) when \(\Gamma = \emptyset\). This means that the two graphs have to synchronize for all actions.
- A \(\{\tau\}\)-synchronization product \((\otimes_{\tau})\) when \(\Gamma = \{\tau\}\). Herein, we do not require the synchronization of the two graphs for the special action \(\tau\).

The next proposition states the compositionality of branching bisimulation variants w.r.t. the product operator. Thus, when plugging an updated component instead of the old one, the new composite system is equivalent to the previous one.

\(\footnote{Similar to the parallel operator of CSP [10]}\)
Proposition 5.2 \( \forall C, [\{C_1\}] \sim [\{C_2\}] \implies [\{C\}] \otimes \tau [\{C_1\}] \sim [\{C\}] \otimes \tau [\{C_2\}] \) for all equivalence relations \( \sim \in \{\approx, \approx_{sbr}, \approx_{br}\} \).

Proof. Because of space limitation, we prove the proposition for the case of \( \approx_{br} \) which is the weaker variant of the branching bisimulations. Nevertheless, the proposition can be proved similarly for the strongest ones by employing the same method.

We should prove that the relation \( R = \{((p, q), (p, r)) \in ([\{C\}] \otimes \tau [\{C_1\}]) \times ([\{C\}] \otimes \tau [\{C_2\}]) : q \approx_{br} r \} \) is a branching bisimulation. Assume: \( (p, q) \xrightarrow{a} (p', q') \) with \( (p, q) R(p, r) \). Hence, \( q \approx_{br} r \) by definition of \( R \). We distinguish three cases:

- **case 1:** \( a \notin \Gamma \). Then, \( p \xrightarrow{a} p' \) and \( q \xrightarrow{a} q' \). Since \( q \approx_{br} r \), we obtain: \( \exists r_1, r' \in [\{C_1\}] : r \xrightarrow{a} r_1 \xrightarrow{a} r' \) and \( q \approx_{br} r_1 \) and \( q' \approx_{br} r' \). Hence, \( (p, r) \xrightarrow{a} (p, r_1) \xrightarrow{a} (p', r') \) and \( q \approx_{br} r_1 \) and \( q' \approx_{br} r' \). Therefore, \( (p, q) R(p, r_1) \) and \( (p', q') R(p', r') \).

- **case 2:** \( a \in \Gamma \setminus \{\tau\} \). Then, \( p \xrightarrow{a} p' \) and \( q = q' \) or \( q \xrightarrow{a} q' \) and \( p = p' \). When \( q = q' \), we get \( (p', q) R(p', r) \) because \( q \approx_{br} r \). When \( q \xrightarrow{a} q' \) and \( p = p' \), we obtain: \( \exists r_1, r' \in [\{C_1\}] : r \xrightarrow{a} r_1 \xrightarrow{a} r' \) and \( q \approx_{br} r_1 \) and \( q' \approx_{br} r' \). Hence, \( (p, r) \xrightarrow{a} (p, r_1) \xrightarrow{a} (p', r') \) such that \( q \approx_{br} r_1 \) and \( q' \approx_{br} r' \). Therefore, \( (p, q) R(p, r_1) \) and \( (p', q') R(p', r') \).

- **case 3:** \( a = \tau \). Then, \( p \xrightarrow{a} p' \) and \( q = q' \) or \( q \xrightarrow{a} q' \) and \( p = p' \). When \( q = q' \), we get \( (p', q) R(p', r) \) since \( q \approx_{br} r \). When \( q \xrightarrow{a} q' \) and \( p = p' \), we obtain: \( q' \approx_{br} r \) (by definition of \( \approx_{br} \)). Hence, we deduce that: \( (p, q') R(p, r) \).

Symmetrically, for any transition \( (p, r) \xrightarrow{a} (p', r') \) with \( (p, q) R(p, r) \), we can prove the above three cases.

Therefore, we deduce that \( R \) is a branching bisimulation. \( \Box \)

We say that a component \( C \) implementation **strongly fulfills** \( \models \) its specification \( (S_{pec}) \) if the full synchronization product of their behavioral models (resp. \( [[C]] , [[S_{pec}]]) \) is strongly bisimilar to the implementation model \( ([C]) \).

**Definition 5.3** \( C \models S_{pec} \text{ if } \Sigma(C) = \Sigma(S_{pec}) \land [[S_{pec}]] \otimes f [[C]] \approx [[C]]. \)

We say also that a component \( C \) implementation **fulfills** (resp. weakly fulfills) its specification \( (S_{pec}) \) if the \( \{\tau\}\)-synchronization product of their behavioral models is strongly branching bisimilar (resp. simply branching bisimilar) to the \( C \) model.

**Definition 5.4** \( C \models_{sbr} S_{pec} \text{ if } \Sigma(C) = \Sigma(S_{pec}) \cup \{\tau\} \land [[S_{pec}]] \otimes \tau [[C]] \approx_{sbr} [[C]]. \)

**Definition 5.5** \( C \models_{br} S_{pec} \text{ if } \Sigma(C) = \Sigma(S_{pec}) \cup \{\tau\} \land [[S_{pec}]] \otimes \tau [[C]] \approx_{br} [[C]]. \)

The following proposition tells us that when the full synchronization product of two graphs is strongly bisimilar to the first graph then their \( \{\tau\}\)-synchronization product is always strongly branching bisimilar to the first graph. Formally,

**Proposition 5.6** \( C \models S_{pec} \text{ implies } C \models_{sbr} S_{pec} \text{ implies } C \models_{br} S_{pec}. \)

Proof. We have to prove that: \( [[S_{pec}]] \otimes f [[C]] \approx [[C]] \text{ implies } [[S_{pec}]] \otimes \tau [[C]] \approx_{br} [[C]]. \)

**First implication:** The bisimulation \( \approx \) can be given as the relation \( R_0 = \{((p, q), q) \in ([S_{pec}] \otimes f [[C]]) \times [[C]] : (p, q) \approx q \}. \) We prove now that the relation \( R_1 = \{((p, q), q) \in ([S_{pec}] \otimes \tau [[C]]) \times [[C]] : (p, q) \in R_0 \} \) is a strong branching bisimulation.
Assume that: \((p, q) \xrightarrow{a} (p', q')\) with \((p, q)R_1q\). By definition of \(R_0\) and \(R_1\), we get: \((p, q) \in \left(\left[\left[S_{pec}\right]\right] \otimes_f \left[\left[C\right]\right]\right)\) and \((p, q) \approx q\). Hence, \(\forall a \in \Sigma \cup \{\tau\}, p \xrightarrow{a} p'\) and \(q \xrightarrow{a} q'\). Since \(\exists q' \in \left[\left[C\right]\right]: q \xrightarrow{a} q'\), and since \((p, q) \approx q\) we obtain \((p', q') \approx q'\). By definition 5.1, \(\left(\left[\left[S_{pec}\right]\right] \otimes_f \left[\left[C\right]\right]\right) \subseteq \left(\left[\left[S_{pec}\right]\right] \otimes_{\pi} \left[\left[C\right]\right]\right)\). So, \((p', q') \in \left(\left[\left[S_{pec}\right]\right] \otimes_{\pi} \left[\left[C\right]\right]\right)\), and thus, we obtain \((p', q')R_1q'\).

Conversely, for any transition \(q \xrightarrow{a} q'\) with \((p, q)R_1q\), we can similarly deduce \(\exists (p', q') \in \left(\left[\left[S_{pec}\right]\right] \otimes_{\pi} \left[\left[C\right]\right]\right): (p, q) \xrightarrow{a} (p', q')\) and \((p', q')R_1q'\).

Therefore, \(R_1\) is a strong branching bisimulation.

**Second implication:** The bisimulation \(\approx_{sbr}\) can be given as the relation \(R_2 = \{(p, q), sbr \in \Sigma \setminus \{\tau\}, (p, q) \approx_{sbr} q\} \times \{(p, q) \approx_{sbr} q\}\). We prove now that the relation \(R_2\) is also a branching bisimulation.

Assume that: \((p, q) \xrightarrow{a} (p', q')\) with \((p, q)R_2q\). Hence, \((p, q) \approx_{sbr} q\) by definition of \(R_2\). We explore the two cases related to the branching bisimulation:

**case 1:** \(a \in \Sigma \setminus \{\tau\}\). Then, \(p \xrightarrow{a} p'\) and \(q \xrightarrow{a} q'\). Since \((p, q) \approx_{sbr} q\), we obtain: \((p', q') \approx_{sbr} q'\). Hence, \((p', q')R_2q'\).

**case 2:** \(a = \tau\). Following the definition 5.1, we get either \(p \xrightarrow{\tau} p'\) or \(q \xrightarrow{\tau} q'\). If \(q = q'\) then we deduce from \((p, q) \approx_{sbr} q\) that \((p', q') \approx_{sbr} q'\) and thus, \((p', q')R_2q'\). On the other hand, if \(p = p'\) then we deduce from \((p, q) \approx_{sbr} q\) that \((p', q') \approx_{sbr} q'\) and thus, \((p', q')R_2q'\).

Conversely, for any transition \(q \xrightarrow{a} q'\) with \((p, q)R_2q\), we prove the two cases related to the branching bisimulation:

**case 1:** \(a \in \Sigma \setminus \{\tau\}\). Since \((p, q) \approx_{sbr} q\), then, \(\exists (p', q') \in \left(\left[\left[S_{pec}\right]\right] \otimes_{\pi} \left[\left[C\right]\right]\right): (p, q) \xrightarrow{a} (p', q')\) such that \((p', q') \approx_{sbr} q'\). Thus, \((p', q')R_2q'\).

**case 2:** \(a = \tau\). Following the definition 5.1, we get \((p, q) \xrightarrow{\tau} (p', q')\). Since \((p, q) \approx_{sbr} q\) then \((p', q') \approx_{sbr} q'\). Thus, \((p', q')R_2q'\).

Therefore, we deduce that \(R_2\) is a branching bisimulation. \(\square\)

Below we present some results about compatibility checking between old and new components w.r.t. interaction assumptions with their environment.

**Case 1:** *without removing any old required services.*

When \(C_2\) is a strong subtype of \(C_1\) then whenever \(C_1\) strongly fulfills any specification then \(C_2\) also does. However when \(C_2\) is a weak subtype of \(C_1\) then the fulfillment of the specification by \(C_2\) is weakened.

**Theorem 5.7** If \(C_2 \preceq_{br} C_1\) then \(C_1 \models S_{pec} \implies H_N(C_2) \models_{br} S_{pec}\).

**Proof.** Following the definition 5.3, \(C_1 \models S_{pec}\) implies \([\left[S_{pec}\right]] \otimes_f \left[\left[C_1\right]\right] \approx \left[\left[C_1\right]\right]\). With regards to definition 5.5, we have to prove that: \([\left[S_{pec}\right]] \otimes_{\pi} H_N(\left[\left[C_2\right]\right]) \approx_{br} H_N(\left[\left[C_2\right]\right])\). Following proposition 5.6, we obtain: \([\left[S_{pec}\right]] \otimes_f \left[\left[C_1\right]\right] \approx \left[\left[C_1\right]\right]\) implies \([\left[S_{pec}\right]] \otimes_{\pi} \left[\left[C_1\right]\right] \approx_{sbr} \left[\left[C_1\right]\right]\). However, \(C_2 \preceq_{br} C_1\). This means that \(H_N(\left[\left[C_2\right]\right]) \approx_{br} \left[\left[C_1\right]\right]\). Since \(H_N(\left[\left[C_2\right]\right]) \approx_{br} \left[\left[C_1\right]\right]\), we have also: \([\left[S_{pec}\right]] \otimes_{\pi} \left[\left[C_1\right]\right] \approx_{br} \left[\left[C_1\right]\right] \approx_{br} H_N(\left[\left[C_2\right]\right])\). Therefore, to prove \([\left[S_{pec}\right]] \otimes_{\pi} H_N(\left[\left[C_2\right]\right]) \approx_{br} H_N(\left[\left[C_2\right]\right])\), it suffices to prove: \([\left[S_{pec}\right]] \otimes_{\pi} \left[\left[C_1\right]\right] \approx_{br} H_N(\left[\left[C_2\right]\right])\). This property is straightforwardly deduced from proposition 5.2 stating the compositionality of \(\approx_{br}\). \(\square\)

**Theorem 5.8** If \(C_2 \preceq_{sbr} C_1\) then \(C_1 \models S_{pec} \implies H_N(C_2) \models S_{pec}\).
Proof. Following the definition 5.3, \( C_1 \equiv S_{pec} \) implies \( [S_{pec}] \otimes f [C_1] \approx [C_1] \). With regards to definition 5.4, we have to prove that: \( [S_{pec}] \otimes \tau H_N([C_2]) \approx_{sbr} H_N([C_2]) \). Following proposition 5.6, we obtain: \( [S_{pec}] \otimes f [C_1] \approx [C_1] \) implies \( [S_{pec}] \otimes \tau [C_1] \approx_{sbr} [C_1] \). However, \( C_2 \not\approx_{sbr} C_1 \) which means: \( H_N([C_2]) \approx_{sbr} [C_1] \). Since \( H_N([C_2]) \approx_{sbr} [C_1] \), we have also: \( [S_{pec}] \otimes \tau [C_1] \approx_{sbr} [C_1] \approx_{sbr} H_N([C_2]) \). Therefore, to prove \( [S_{pec}] \otimes \tau H_N([C_2]) \approx_{sbr} H_N([C_2]) \), it suffices to prove: \( [S_{pec}] \otimes \tau [C_1] \approx_{sbr} [S_{pec}] \otimes \tau H_N([C_2]) \). This property is straightforwardly deduced from proposition 5.2 stating the compositionality of \( \approx_{sbr} \). □

Case 2: with removing of some old required services.

Theorem 5.9 If \( C_2 \not\approx_{ibr} C_1 \) then \( H_{\alpha_2}([C_1]) \models_{sbr} S_{pec} \implies \text{H}_{\alpha_2 \cup \alpha_1}([C_2]) \models_{br} S_{pec} \).

Theorem 5.10 If \( C_2 \not\approx_{sibr} C_1 \) then \( H_{\alpha_2}([C_1]) \models_{sbr} S_{pec} \implies \text{H}_{\alpha_2 \cup \alpha_1}([C_2]) \models_{sbr} S_{pec} \).

The same remarks hold in this case but here we require the fulfillment of \( S_{pec} \) by \( H_{\alpha_2}([C_1]) \) and not \( C_1 \) because some of required services will no more be used in \( C_2 \). The proofs are similar to those of theorems 5.7 and 5.8.

6 Substitutability relations in true concurrency context

When we handle the substitutivity of components with regards to true concurrency semantics, we should use a suitable behavioral model for this purpose.

6.1 Why Non-interleaving models?

On comparing the two StateCharts \( C_1 \) and \( C_2 \) of figure 3 with regards to an interleaving semantics, we can say they are equivalent even modulo the strong bisimulation. However, let us consider the case when “a” is a synchronous action. Thereafter, if the first StateChart is blocked on execution of “a” because of an unprepared cooperating component then it remains able to execute “b” once it receives its trigger event. Indeed, according to the semantics of StateCharts [19] the actions “a” and “b” are independent and thus achieved concurrently, each of which within its orthogonal region. However in the second StateChart, these two actions are causally ordered. Consequently, in the first case, the component \( C_1 \) is capable to react to the two stimuli at the same time whereas the second one can only handle them sequentially so that if it is blocked on a first synchronous action, it can not immediately handle the second stimulus even though this is critical.

Which kind of behavioral model can capture these subtleties? It is obvious that a non-interleaving model (like asynchronous transition system [18]) is a suitable one to distinguish between the two above schemes as depicted below. Then, another question arises about what observation criterion can be used to distinguish between these two graphs. As the classical bisimulations are not suitable, we introduce a new variant of the forth-back bisimulation which is more appropriate for this setting.
An asynchronous transition system (ATS) is a quintuple $G = (Q, \rightarrow, \Sigma, q_0, I)$ where $G = (Q, \rightarrow, \Sigma, q_0, I)$ is a labeled transition system, and $I \subseteq \Sigma \times \Sigma$ is a symmetric and non reflexive (independence) relation such that:

- $e \in \Sigma \implies \exists q, q' \in Q, (q, e, q') \in \leftrightarrow$
- $(q, e, q_1) \in \leftrightarrow \land (q, e, q_2) \in \leftrightarrow \implies q_1 = q_2$
- $e_1 \lor e_2 \land (q, e_1, q_1) \in \leftrightarrow \land (q, e_2, q_2) \in \leftrightarrow \implies \exists u, (q_1, e_1, u) \in \leftrightarrow \land (q_2, e_2, u) \in \leftrightarrow$
- $e_1 \lor e_2 \land (q, e_1, q_1) \in \leftrightarrow \land (q_1, e_2, u) \in \leftrightarrow \implies \exists q_2, (q_1, e_2, q_2) \in \leftrightarrow \land (q_2, e_1, u) \in \leftrightarrow$

A branching forth-back bisimulation is a symmetric relation $\approx_{fb}$ such that $p \approx_{fb} q$ if and only if $\forall p', a : p \xrightarrow{a} p'$ implies $\exists q', a : q \xrightarrow{a} q'$ and $p' \approx_{fb} q'$,

In order to take into account the unobservable actions, we introduce our new branching variant of the forth-back bisimulation.

A branching forth-back bisimulation is a symmetric relation $\mathcal{R} \subseteq Q_1 \times Q_2$, such that for every $(p, q) \in Q_1 \times Q_2$, $p \mathcal{R} q$ if:

- $\forall a \in \Sigma \cup \{\tau\}, \forall p' : p \xrightarrow{a} p'$ then either $a = \tau$ and $p' \mathcal{R} q$ or $\exists q' : q \xrightarrow{\tau} q_1 \xrightarrow{a} q'$ and $p' \mathcal{R} q_1$ and $p' \mathcal{R} q'$,
- and $\forall a \in \Sigma \cup \{\tau\}, \forall p' : p' \xrightarrow{a} p$ then either $a = \tau$ and $p' \mathcal{R} q$ or $\exists q' : q' \xrightarrow{\tau} q_1 \xrightarrow{a} q$ and $p' \mathcal{R} q_1$ and $p' \mathcal{R} q'$.

Two graphs $G_1$ and $G_2$ are branching forth-back bisimilar ($G_1 \approx_{bfb} G_2$) when their initial nodes are branching forth-back bisimilar.

### 6.2 Substitutability Relations

We still use the hiding operator $(H)$ to conceal events related to additional services in new components making them unobservable from outside. We recall always that new unobservable events are different from internal ones because while the latter events are completely under control of the component, the former events are under control of both the involved component and its environment. Hence, new services are unobservable only by some clients who view the improved component as it was the old one and thus still use only its old services. So, we consider below only hidden actions which depict new services and ensure that they do not affect the availability of old services of the new component in a true concurrency framework.

Let $N = \Sigma(C_2) \setminus \Sigma(C_1)$ be the set of new services. $H_N([C])$ denotes the behavior model of $C$ where all events of $N$ are relabeled to $\tau$. Now, we give below an optimal substitutability relation which is faithful enough in terms of preservation.
of capabilities of active components in a true concurrency framework:

Definition 6.4 $C_2 \preceq_{bfb} C_1$ iff $H_N([|C_2|]) \approx_{bfb} [|C_1|]$.

The main target of this definition is to ensure $C_2$ to behave as $C_1$ with preservation of the independence relation between concurrent actions whichever the state that $C_2$ has reached. Every time $C_1$ is able from a state $s_1$ to interact with its environment (by offering or demanding a service $x$) and reach (or return back to) some stable state $s_2$, $C_2$ should be also capable to do the same interaction $x$ and to reach (or return back to) an equivalent state of $s_2$ even though after performing an unobservable activity. In this case, the intermediate states in a $\tau$-sequence in $C_2$ should be equivalent to the starting point $s_1$ of $C_1$ by preserving all actions offered at the state $s_1$. Indeed, unobservable events in $C_2$ have not to exclude any action that $C_1$ can perform from the equivalent state $s_1$. Note that independent actions in $C_1$ remain independent in $C_2$ so that their interleaved paths remain illustrated by diamond-shaped forms even if they are combined with $\tau$-actions (see fig. 4).

Example: Let $C_3$ be a new component we want to substitute to $C_1$ (see fig. 3). The new action $c$ is hidden in the graph ($[|C_3|]$) which we can easily prove that it is branching forth-back bisimilar to the behavioral graph of $C_1$ (see fig. 4).

However, these two graphs ($[|C_1|]$ and $[|C_3|]$) are not branching forth-back bisimilar to the graph obtained from the parallel combination of $C_2$ and the action $c$ (see fig. 4) because $C_2$ is already not equivalent to $C_1$ via $\approx_{bfb}$.

7 Conclusion

This paper presents a practical approach of substitutability checking that is based on strong assumptions of service availability and correctness properties mainly in reactive systems. We have dealt with this issue with respect to the two semantics; interleaving and true concurrency semantics. In all cases, the removed services in new components are concealed; however our subtyping relations preserve the availability of old services even though the control is in some intermediate states along a chain of $\tau$-steps. Furthermore, we show that new components continue preserve properties that old components fulfill when interacting with its environment. In the true concurrency framework, we exploited the history preserving relations to
strenthen the service availability in one component even though some of its orthogonal regions (internal tasks) are blocked at doing some synchronous actions.

For future work, we deem it is important to consider timing constraints in substitutability relations and to analyze to what extent the different kinds of interactions may influence the compatibility of constrained components.

References


Residual for Component Specifications

Jean-Baptiste Raclet

IRISA
Campus de Beaulieu
F-35042 Rennes Cedex, France

Abstract

We address the problem of component reuse by describing a quotient operation. Starting from the specifications of the behaviors of the component and of the desired overall system, this operation computes the residual specification characteristic of the systems that, when composed with the given component, satisfy the overall specification. This problem is solved when behaviors are given by modal or acceptance specifications and when composition allows mixed product and internalization of events. We show on an example how weak form of liveness constraint may be taken into account by this technique.

Keywords: Component-based design, Behavioral interface, Modal and acceptance specifications, Residual of specifications, Behavioral reuse.

1 Introduction

In current component platforms, a component is equipped with an interface which lists the signature of the services that the entity offers. This light description is sufficient to enable component reuse. However, it provides no guarantee that the reused component will interact suitably with its environment and critical behavioral mismatch such as deadlock may occur.

In this paper, we investigate the extension of component interfaces to behavioral descriptions in order to provide techniques to reason about component reuse at a behavioral level rather than at a signature level. More precisely, we study the following issues: can a component, the behavior of which is described in its interface by the specification $S_1$, be used to build a system satisfying a global specification $S$? If so, what are the components that, when composed with the reused component, constitute a composite system satisfying $S$?

These problems can be seen as kinds of supervisor synthesis with the main difference that the reused component (corresponding to the plant in control theory) is a black-box. Indeed, a component must be reusable from the description of its

---

1 Email: Jean-Baptiste.Raclet@irisa.fr

This paper is electronically published in

Electronic Notes in Theoretical Computer Science

URL: www.elsevier.nl/locate/entcs
behavior in its interface and not from its implementation which is unknown as it may have been developed by a third party.

Behavioral reuse of components can also be related to some works in equation solving. This problem introduced in [11] consists in solving the equation $S_1 \times X \simeq S$ with $S_1$ called the context, $S$ a global specification and $\simeq$ a trace equivalence relation. Solutions for this problem were proposed for various models of specification: finite automata [11,5], finite state machine [14] (with inclusion of traces as equivalence relation), CCS or CSP processes [12,10] (with bisimulation as equivalence relation) or input/output automata [8].

In this paper, we introduce modal specifications and acceptance specifications as intuitive formalisms for behavioral interface description. From the expressiveness point of view, they allow to state some forms of liveness properties. For each of this formalism, a quotient operation is defined to address the problem of behavioral reuse of a component as computing the residual specification $S/S_1$. We study two kinds of composition between components: synchronous composition and mixed product with internalization of events. This paper is organized as follows: after short preliminaries, section 3 introduces modal specification. The notion of residual specification for behavioral reuse is formalized in section 4. Then, the quotient of modal specification is proposed in section 5. Expressivity is improved thanks to acceptance specifications in section 6. Then, section 7 is devoted to the quotient operation corresponding to the use of mixed product with internalization of events as composition operation. An example and a hint for some line of future work conclude the paper.

2 Preliminaries

We use $\Sigma$ to denote the universe of events. $\Sigma^*$ denotes the set of all finite length event sequences, called traces, with $\epsilon$ the zero length trace. A language $L$ over $\Sigma$ is a subset of $\Sigma^*$. For $u \in L$, we let $L_u$ be the set of events $a$ such that the trace $u$ followed by $a$ (noted $u.a$) belong to $L$. The operations $\cap$ and $\cup$ over languages correspond to the set-theoretic intersection and union. The complement of the language $L$ over $\Sigma$, noted $\neg L$, is the set $\Sigma^* \setminus L$. Given a trace $w \in \Sigma'$ and a subset of events $\Sigma \subseteq \Sigma'$, the projection of $w$ on $\Sigma$ denoted $\Pi_\Sigma(w)$ is the trace obtained from $w$ by erasing the events not belonging to $\Sigma$. A trace $u$ is said to be a prefix of a trace $w$ (noted $u \preceq w$), if $w = uv$. Given a language $L$ its prefix closure $\hat{L}$ consists of all the prefixes of all the traces in $L$. A language $L$ is said prefix-closed when $L = \hat{L}$.

3 Modal specifications

In this section we introduce modal specification as a formalism to specify sets of languages:

3.1 Modal specification and their models

Modal automata are standard finite automata with modalities "may" or "must" on transitions. They were originally used in [9] to study the refinement of actions. We
now introduce modal specifications which generalize the use of modalities on events to non-necessarily regular languages:

**Definition 3.1** A modal specification \( S \) over \( \Sigma \) is a triple \( \langle L, \text{must}, \text{may} \rangle \) where \( L \) is a prefix-closed language over \( \Sigma \) and \( \text{must}, \text{may} : L \to P(\Sigma) \) are partial functions that type events: for \( u \in L \),

- \( a \in \text{may}(u) \) means that the event \( a \) is allowed after the trace \( u \);
- \( a \in \text{must}(u) \) means that the event \( a \) is required after the trace \( u \);
- \( a \notin \text{may}(u) \) (often denoted \( a \in \neg\text{may}(u) \)) means that \( a \) is forbidden after \( u \).

For consistency, the following conditions are required for all \( u \in L \):

(C1) \( \text{must}(u) \subseteq \text{may}(u) \);
(C2) \( \text{may}(u) = L_u \).

The consistency condition (C1) expresses that the events required after a trace \( u \) of the specification must also be allowed.

In the sequel the elements of the tuple corresponding to the modal specification \( S \) could be denoted \( L(S), \text{must}(S) \) and \( \text{may}(S) \). The prefix-closed language \( L(S) \) may be called the *support* of \( S \). The set of modal specifications over \( \Sigma \) is denoted \( \mathcal{MS}(\Sigma) \).

A model of a modal specification is a language. The definition of the validation relation is the following:

**Definition 3.2** A prefix-closed language \( C \subseteq \Sigma^* \) is a model of the modal specification \( S \in \mathcal{MS}(\Sigma) \), noted \( C \models S \), if:

- \( C \subseteq L(S) \);
- for all \( u \in C \), \( \text{must}(S)(u) \subseteq C_u \subseteq \text{may}(S)(u) \).

The interpretation of this definition is the following: every trace \( u \) of a model \( C \) is also a trace of \( S \) and the events available in \( C \) after a trace \( u \) are all the required events \( (\text{must}(u) \subseteq C_u) \) and none of the forbidden events \( (C_u \subseteq \neg\text{may}(u)) \) after the trace \( u \) in the specification.

**Example 3.3** The specification \( \top = \langle \Sigma^*, \text{must}, \text{may} \rangle \) with, for all \( u \in \Sigma^* \), \( \text{must}(u) = \emptyset \) and \( \text{may}(u) = \Sigma \), admit every language over \( \Sigma \) as model. \( \bot = (\emptyset, \emptyset, \emptyset) \) has no model.

Modal specifications are ordered by the following relation:

**Definition 3.4** \( S_1 \leq S_2 \) if and only if:

- \( L(S_1) \subseteq L(S_2) \) and
- \( \forall u \in L(S_1), \begin{cases} \text{may}(S_1)(u) \subseteq \text{may}(S_2)(u) \\ \text{must}(S_1)(u) \supseteq \text{must}(S_2)(u) \end{cases} \)

**Remark 3.5** Any prefix-closed language \( C \) can be viewed as a modal specification where \( \text{may}(u) = \text{must}(u) = C_u \) for all trace \( u \in C \). Hence \( C \models S \) if and only if \( C \leq S \).
Proposition 3.6 \( S_1 \leq S_2 \) if and only if every model \( C \) of \( S_1 \) is also a model of \( S_2 \).

When \( L(S) \) is regular, \( S \) can be viewed as the unfolding of a modal automaton [9] all of whose states are final. The logical fragment equivalent to modal automata has been identified in [6]. It is a fragment of the mu-calculus called the conjunctive mu-calculus as it includes conjunctions and greatest fix-points along with diamond and box modalities. It is strictly less expressive than mu-calculus as neither disjunction nor eventualities can be stated.

3.2 Pseudo-modal specifications

For technical reasons, we shall consider an extension of the class of modal specification called pseudo-modal specification and denoted \( p\mathcal{MS} \). They are specifications where the consistency condition (C1) of the definition 3.1 is relaxed for some traces \( u \):

Definition 3.7 A pseudo-modal specification \( pS \in p\mathcal{MS}(\Sigma) \) is a triple \( \langle L, \text{must}, \text{may} \rangle \) where \( L \) is a prefix-closed language over \( \Sigma \) and \( \text{must}, \text{may} : L \to \mathcal{P}(\Sigma) \) are partial functions that type events with no consistency constraint between \( \text{may}(u) \) and \( \text{must}(u) \).

A trace \( u \) of \( pS \) such that \( \text{must}(u) \notin \text{may}(u) \) is said incoherently specified.

The relations \( = \) and \( \leq \) for pseudo-modal specifications are the same as for modal specifications (cf. def. 3.2 and 3.4). Hence, if \( \text{must}(u) \notin \text{may}(u) \) then \( u \) can’t belong to a language \( C \) model of \( pS \) because this would imply, for \( a \in \text{must}(u) \) and \( a \notin \text{may}(u) \), on one hand that \( u.a \in C \) (as \( a \in \text{must}(u) \)) and, on the other hand, that \( u.a \notin C \) (as \( a \notin \text{may}(u) \)). This remark give the intuition of a reduction of a pseudo-modal specification into a modal specification with respect to its set of models; we let \( \rho : p\mathcal{MS} \to \mathcal{MS} \) be the operation such that:

Proposition 3.8 Either a pseudo-modal specification \( pS \) has no model or there exists a largest modal specification \( \rho(pS) \) smaller than \( pS \), and \( \rho(pS) \) has the same models as \( pS \).

The modal specification \( \rho(pS) \) is obtained from the pseudo-modal specification \( pS \) by application of the following steps:

(i) basis: we let \( R \) be a copy of \( pS \);

(ii) we let \( U \) be the set of traces \( u \) incoherently specified in \( R \); we remove \( U \) from \( L(R) \); for all trace \( v \) such that \( v.a = u \) with \( u \in U \), we remove \( a \) from \( \text{may}(R)(v) \) to enforce the consistency condition (C2);

When \( a \in \text{must}(R)(v) \), \( v \) becomes incoherently specified in \( R \). Thus, we repeat this step until there is no more incoherently specified trace in \( L(R) \);

(iii) \( \rho(pS) \) is built from \( R \): \( L(\rho(pS)) = \widehat{L(R)} \) and for all \( u \in L(\rho(pS)) \), \( \text{may}(\rho(pS))(u) = \text{may}(R)(u) \) and \( \text{must}(\rho(pS))(u) = \text{must}(R)(u) \).

3.3 Lattice of modal specifications

The set of modal specifications \( \mathcal{MS} \) equipped with the partial order \( \leq \) is a complete distributive lattice (hence a bounded lattice) where the meet \( S_1 \land S_2 \) is the reduction
of the pseudo-modal specification $S_1 \& S_2$ over $L(S_1) \cap L(S_2)$ with:

$$\forall u \in L(S_1) \cap L(S_2), \begin{cases} 
    \text{may}(S_1 \& S_2)(u) = \text{may}(S_1)(u) \cap \text{may}(S_2)(u) \\
    \text{must}(S_1 \& S_2)(u) = \text{must}(S_1)(u) \cup \text{must}(S_2)(u)
\end{cases}$$

and the join $S_1 \lor S_2$ is the modal specification over $L(S_1) \cup L(S_2)$ with:

$$\begin{cases} 
    \text{if } u \in L(S_1) \cap L(S_2), \text{may}(S_1 \lor S_2)(u) = \text{may}(S_1)(u) \cup \text{may}(S_2)(u), \\
    \quad \text{must}(S_1 \lor S_2)(u) = \text{must}(S_1)(u) \cap \text{must}(S_2)(u) \\
    \text{if } u \in L(S_1) \setminus L(S_2), \text{may}(S_1 \lor S_2)(u) = \text{may}(S_1), \text{must}(S_1 \lor S_2)(u) = \text{must}(S_1)(u) \\
    \text{if } u \in L(S_2) \setminus L(S_1), \text{may}(S_1 \lor S_2)(u) = \text{may}(S_2), \text{must}(S_1 \lor S_2)(u) = \text{must}(S_2)(u).
\end{cases}$$

**Proposition 3.9** For all modal specification $S \in MS(\Sigma)$, we have:

$$S = \bigvee \{C \mid C \models S\}$$

### 4 Residual specification for behavioral reuse

In the sequel, a component is a pair $(C, S)$ such that $C \models S$ with $C$ called the implementation and $S$ the specification of the component. Reusing a component $(C_1, S_1)$ to realize a global system specified by $S$ amounts to exhibit a residual specification $S/S_1$ so that any component $(C_2, S/S_1)$ is such that the composition of $C_1$ with $C_2$ (noted $C_1 \otimes C_2$) satisfies $S$. In component-based design, components are regarded as black-box. As a result, the implementation $C_1$ of the component to be reused is unknown and its composition with the possible components $(C_2, S/S_1)$ must realize $S$ whatever the implementation $C_1$ of $S_1$ could be. Thus the characteristic property of a residual operation for behavioral reuse of a component is the following:

**Proposition 4.1** $C_2 \models S/S_1 \iff \forall C_1. [C_1 \models S_1 \Rightarrow C_1 \otimes C_2 \models S]$.

In the next section, we establish this proposition when $S$ and $S_1$ are modal specifications and when $C_1$ and $C_2$ are composed using a synchronous product.

### 5 Quotient of modal specifications for behavioral reuse

The synchronous product of languages $C_1$ and $C_2$ over the same alphabet $\Sigma$ corresponds to the set theoretic intersection $C_1 \cap C_2$. We generalize the synchronous product to modal specifications:

**Definition 5.1** The synchronous product of $S_1$ and $S_2$ is the modal specification $S_1 \otimes S_2$ over $L(S_1) \cap L(S_2)$ with:

$$\forall u \in L(S_1) \cap L(S_2), \begin{cases} 
    \text{must}(S_1 \otimes S_2)(u) = \text{must}(S_1)(u) \cap \text{must}(S_2)(u) \\
    \text{may}(S_1 \otimes S_2)(u) = \text{may}(S_1)(u) \cap \text{may}(S_2)(u)
\end{cases}$$
Remark 5.2  

(i) This operator is monotonic over the order relation $\leq$:  

$$S_1 \leq S_2 \Rightarrow (S \otimes S_1 \leq S \otimes S_2 \text{ and } S_1 \otimes S \leq S_2 \otimes S)$$

(ii) If $C_1, C_2 \subseteq \Sigma^*$ are viewed as modal specifications (cf. remark 3.5) then $C_1 \otimes C_2 = C_1 \cap C_2$.

Now, to define the quotient operation, we start from the observation that, for any prefix-closed languages $L$, $M$, and $N$: $L \cap M \subseteq N \iff L \subseteq S (N \cup \neg M)$ where:

$$\downarrow X = \{u \in \Sigma^* \mid \forall v \quad v \preceq u \Rightarrow v \in X\}$$

denote the prefix interior of a set $X$; it is an interior operation giving the greatest prefix-closed subset of the given set, when such a subset exists and the empty set otherwise. This remark is used to define the support of the modal specification $S/S_1$. Now in order to define the typing functions $\text{may}(S/S_1)$ and $\text{must}(S/S_1)$, we proceed by case inspection:

Definition 5.3 The quotient of the modal specifications $S$ and $S_1$ is the pseudo-modal specification $S/S_1$ over $\downarrow (L(S) \cup \neg L(S_1))$ with:

(i) for all $u \in L(S/S_1) \cap L(S_1)$:

(a) if $a \in \text{must}(S)(u) \cap \text{must}(S_1)(u)$ then $a \in \text{must}(S/S_1)(u)$ and $a \in \text{may}(S/S_1)(u)$;

(b) if $a \in \text{must}(S)(u) \cap \neg \text{must}(S_1)(u)$ then $a \in \text{must}(S/S_1)(u)$ but $a \notin \text{may}(S/S_1)(u)$;

(c) if $a \in (\text{may}(S)(u) \setminus \text{must}(S)(u))$ then $a \in \text{may}(S/S_1)(u)$;

(d) if $a \in \text{mustnot}(S)(u) \cap \text{mustnot}(S_1)(u)$ then $a \in \text{may}(S/S_1)(u)$;

(e) if $a \in \text{mustnot}(S)(u) \cap \text{may}(S_1)(u)$ then $a \in \text{mustnot}(S/S_1)(u)$;

(ii) if $u \in (L(S/S_1) \setminus L(S_1))$ then $\text{must}(S/S_1)(u) = \emptyset$ and $\text{may}(S/S_1)(u) = \Sigma$.

Now we give for each possible case, an intuitive interpretation of the resulting modality assuming that $C_1 \models S_1$ and we intend to have $C_2 \models S/S_1$ and $C_1 \otimes C_2 \models S$:

(i) first, when $u \in L(S/S_1) \cap L(S_1)$:

(a) $a$ is required in the global specification $S$ that is $u.a$ must belong to $C_1 \otimes C_2 = C_1 \cap C_2$. As $a$ is guaranteed in $S_1$, $u.a \in C_1$ for all $C_1 \models S_1$ with $u \in C_1$; thus $u.a$ must belong to $C_2$ to always have $u.a \in C_1 \otimes C_2$: $a \in \text{must}(S/S_1)(u)$ and $a \in \text{may}(S/S_1)(u)$.

(b) $a$ is required in the global specification $S$ but as $a \notin \text{must}(S_1)(u)$, there are some $C_1 \models S_1$ such that $u.a \notin C_1$; hence, for all $C_2$, $C_1 \otimes C_2$ can’t be a model of $S$. As a result, the trace $u$ must be incoherently in $S/S_1$ and we let $a \in \text{must}(S/S_1)(u)$ but $a \notin \text{may}(S/S_1)(u)$ to model this inconsistency.

(c) $a$ is allowed in the global specification $S$ and $u.a$ may belong to $C_1 \otimes C_2$. Thus, whether or not $u.a$ belongs to $C_1 \models S_1$, $u.a$ can belong to $C_2$ without violating the specification $S$. Hence: $a \in \text{may}(S/S_1)(u)$.

(d) $a$ is forbidden in the global specification $S$ and in $S_1$ thus, whether or not $u.a \in C_2$, we have $u.a \notin C_1 \otimes C_2$ which is conform to $S$. Hence: $a \in \text{may}(S/S_1)(u)$. 


According to prop. 5.4, Lemma 5.5

\[ \forall \{ C \otimes C \} \]

\[ \text{we forbid } a \text{ in } S/S_1: a \in \mathit{mustnot}(S/S_1)(u). \]

As a result, when \( C_2 \models S/S_1, u.a \notin C_1 \otimes C_2 \) which is conform to \( S \).

(ii) if \( u \in (L(S/S_1) \setminus L(S_1)) \): as \( u \notin C_1 \), \( u \) may belong to \( C_1 \otimes C_2 \). As a result, \( S/S_1 \) is relaxed after the trace \( u \) by taking \( \mathit{must}(S/S_1)(u) = \emptyset \) (nothing is required) and \( \mathit{may}(S/S_1)(u) = \Sigma \) (every event is allowed).

The adjoint operation of this quotient operation is the synchronous product of definition 5.1:

**Proposition 5.4** If \( S, S_1 \) and \( S_2 \) are modal specifications over \( \Sigma \) then:

\[ S_1 \otimes S_2 \leq S \iff S_2 \leq \rho(S/S_1) \]

To prove proposition 4.1 for modal specifications and synchronous product, we need the following lemma:

**Lemma 5.5** \( \forall \{ C \otimes C' \mid C \models S \} = \forall \{ C \mid C \models S \} \otimes C' \)

**Proposition 5.6** If \( S \) and \( S_1 \) are modal specifications over \( \Sigma \) then:

\[ C_2 \models S/S_1 \text{ iff } \forall C_1, [C_1 \models S_1 \Rightarrow C_1 \otimes C_2 \models S] \]

**Proof.** (\( \Rightarrow \)) According to prop. 5.4, if \( C_2 \models S/S_1 \) (that is \( C_2 \models \rho(S/S_1) \)) then \( C_2 \otimes S_1 \leq S \). Moreover as \( C_1 \models S_1 \) then \( C_1 \otimes C_2 \leq S_1 \otimes C_2 \). As a result, \( C_1 \otimes C_2 \leq S \) that is \( C_1 \otimes C_2 \models S \).

(\( \Leftarrow \)) If for all \( C_1 \) such that \( C_1 \models S_1 \) we have \( C_1 \otimes C_2 \models S \) then:

\[ \forall \{ C_1 \otimes C_2 \mid C_1 \models S_1 \} \leq S \]

Thus, by lemma 5.5, \( \forall \{ C_1 \mid C_1 \models S_1 \} \otimes C_2 \leq S \) i.e. \( S_1 \otimes C_2 \leq S \) (by prop. 3.9).

According to prop. 5.4, \( C_2 \leq S/S_1 \) hence \( C_2 \models S/S_1 \). \( \square \)

As previously pointed out, the disjunction is not included in the logical fragment equivalent to modal automata. Therefore particular liveness properties can’t be stated in this framework. For instance, let us consider the following "progressive" property: "any stimulus a is followed by at least b1 or b2 as reaction". This can’t be specified with a modal specification: \( b_1 \) and \( b_2 \) can’t belong to \( \mathit{must}("a") \) because this would request that every stimulus \( a \) is followed by both \( b_1 \) and \( b_2 \); \( b_1 \) and \( b_2 \) can’t also belong to \( \mathit{may}("a") \) because the language such that the stimulus \( a \) is followed by no reaction would be a model of the specification.

A trace \( u \) in a modal specification specifies any situation where the system is ready to engage in a set of events \( X \), if and only when \( \mathit{must}(u) \subseteq X \subseteq \mathit{may}(u) \). This set of "acceptance" set is thus given by:

\[ \mathit{Acc}(u) = \{ X \in \mathcal{P}(\Sigma) \mid \mathit{must}(u) \subseteq X \subseteq \mathit{may}(u) \} \]

By definition this set is closed under union, intersection and convexity (that is given \( X, Y \in \mathit{Acc}(u) \) and a set \( Z \) such that \( X \subseteq Z \subseteq Y \) then \( X \sqcup Y, X \cap Y \) and \( Z \in \mathit{Acc}(u) \)) and may and must modalities may be recovered as \( \mathit{may}(u) = \bigcup_{X \in \mathit{Acc}(u)} X \) and \( \mathit{must}(u) = \bigcap_{X \in \mathit{Acc}(u)} X \).
Thus, for example if \( \text{may}(u) = \{b_1, b_2\} \) and \( \text{must}(u) = \emptyset \), we obtain \( \text{Acc}(u) = \{\emptyset, \{b_1\}, \{b_2\}, \{b_1, b_2\}\} \). If we want to specify that at least \( b_1 \) or \( b_2 \) occur, the specified acceptance set should be \( \{\{b_1\}, \{b_2\}, \{b_1, b_2\}\} \) which is no longer closed by intersection. According to this example, closure by intersection should be relaxed to deal with such "progressive" properties. Trees labeled by acceptance set closed by union and convexity have been studied in [7]. In the next section, we propose a quotient operation for acceptance specifications with no closure constraint over the acceptance set.

6 Improving expressivity with acceptance spec.

We generalize the previous framework presented for modal specifications to acceptance specifications:

6.1 Acceptance specifications and their models

Definition 6.1 An acceptance specification \( S \) is a pair \( S = \langle L, \text{Acc} \rangle \) where \( L \) is a prefix-closed language over \( \Sigma \) and \( \text{Acc} : L \rightarrow \mathcal{P}(\mathcal{P}(\Sigma)) \) is a map associating each trace \( u \in L \) to its acceptance set. For consistency, we require for all trace \( u \in L \):

\[ \text{(C1)} \quad \text{Acc}(u) \neq \emptyset \]
\[ \text{(C2)} \quad u.a \in L \text{ if and only there exists at least one set } X \in \text{Acc}(u) \text{ such that } a \in X \]

The condition (C2) can be rephrase in \( L_u = \bigcup_{X \in \text{Acc}(u)} X \). The set of acceptance specifications over \( \Sigma \) is denoted \( \mathcal{AS}(\Sigma) \).

Definition 6.2 A prefix-closed language \( C \subseteq \Sigma^* \) is a model of the acceptance specification \( S \in \mathcal{AS}(\Sigma) \), noted \( C \models S \), if:

\[ \bullet \quad C \subseteq L(S); \]
\[ \bullet \quad \text{for all } u \in C, \text{ } C_u \in \text{Acc}(S)(u). \]

Example 6.3 The specification \( \top = \langle \Sigma^*, \text{Acc} \rangle \) with, for all \( u \in \Sigma^* \), \( \text{Acc}(u) = \mathcal{P}(\Sigma) \), admit every language over \( \Sigma \) as model. \( \bot = \langle \emptyset, \emptyset \rangle \) has no model.

Remark 6.4 \( \text{Acc}(u) = \emptyset \) is different from \( \emptyset \in \text{Acc}(u) \). The first situation is a violation of a consistency condition whereas the second reports that some models of the specification can perform no event after the trace \( u \).

Definition 6.5 The order relation on acceptance specifications is given by inclusion of both corresponding languages and acceptance sets:

\[ S_1 \leq S_2 \text{ iff } L(S_1) \subseteq L(S_2) \text{ and } \forall u \in L(S_1), \quad \text{Acc}(S_1)(u) \subseteq \text{Acc}(S_2)(u) \]

Remark 6.6 Any language \( C \) can be viewed as an acceptance specification with \( \text{Acc}(u) = C_u \) that is its acceptance set is a singleton for all trace \( u \in C \). Hence \( C \models S \) if and only if \( C \leq S \).

6.2 Pseudo-acceptance specifications

Definition 6.7 A pseudo-acceptance specification \( pS \in p\mathcal{AS}(\Sigma) \) is a pair \( \langle L, \text{Acc} \rangle \) where \( L \) is a language over \( \Sigma \) and \( \text{Acc} : L \rightarrow \mathcal{P}(\mathcal{P}(\Sigma)) \) is a map associating each
trace $u$ to its set of acceptance with no consistency constraint over $Acc$.
A trace $u$ of $p\mathcal{S}$ is said incoherently specified when $Acc(u) = \emptyset$.

Every pseudo-acceptance specification $p\mathcal{S}$ can be reduced to an acceptance specification $\rho(p\mathcal{S})$ that admits the same models, by iteration of the following steps:

(i) basis: we let $R$ be a copy of $p\mathcal{S}$;
(ii) we let $U$ be the set of traces $u$ incoherently specified in $R$; we remove $U$ from $L(R)$; for all trace $v$ such that $v.a = u$ with $u \in U$, we remove from $Acc(R)(v)$ all sets containing the letter $a$ to enforce the consistency condition $(C2)$;

When $a \in \bigcap_{X \in Acc(R)(v)} X$, $v$ becomes incoherently specified in $R$. Thus, we repeat this step until there is no more incoherently specified trace in $L(R)$;
(iii) $\rho(p\mathcal{S})$ is built from $R$: $L(\rho(p\mathcal{S})) = \widehat{L(R)}$ and for all $u \in L(\rho(p\mathcal{S}))$, $Acc(\rho(p\mathcal{S}))(u) = Acc(R)(u)$.

6.3 Lattice of acceptance specifications

The set of acceptance specifications $\mathcal{AS}$ equipped with the partial order $\leq$ is a complete distributive lattice (hence a bounded lattice) where the meet $S_1 \land S_2$ is the reduction of the pseudo-acceptance specification $S_1 \& S_2$ whose support is $L(S_1) \cap L(S_2)$ and with $Acc(S_1 \& S_2)(u) = Acc(S_1)(u) \cap Acc(S_2)(u)$. The join $S_1 \lor S_2$ is defined over $L(S_1) \cup L(S_2)$ by:

\[
\begin{align*}
\text{if } u \in L(S_1) \cap L(S_2), & \quad Acc(S_1 \lor S_2)(u) = Acc(S_1) \cup Acc(S_2) \\
\text{if } u \in L(S_1) \setminus L(S_2), & \quad Acc(S_1 \lor S_2)(u) = Acc(S_1) \\
\text{if } u \in L(S_2) \setminus L(S_1), & \quad Acc(S_1 \lor S_2)(u) = Acc(S_2)
\end{align*}
\]

6.4 Quotient of acceptance specifications

We define quotient of acceptance specifications such that proposition 4.1 is verified with synchronous product as component composition:

**Definition 6.8** The synchronous product of the acceptance specifications $S_1$ and $S_2$ is the acceptance specification $S_1 \otimes S_2$ over $L(S_1) \cap L(S_2)$ with for all $u \in L(S_1) \cap L(S_2)$:

\[Acc(S_1 \otimes S_2)(u) = \{X_1 \cap X_2 \mid X_1 \in Acc(S_1)(u) \text{ and } X_2 \in Acc(S_2)(u)\}\]

This operation has for adjoint the following quotient operation:

**Definition 6.9** The quotient of the acceptance specification $S$ and $S_1$ is the pseudo-acceptance specification $S/S_1$ over $\downarrow (L(S) \cup \neg L(S_1))$ with, for all $u \in L(S/S_1) \cap L(S_1)$: $Acc(S/S_1)(u) = \{Y \in P(S) \mid \forall X \in Acc(S_1)(u), X \cap Y \in Acc(S)(u)\}$ and for all $u \in (L(S/S_1) \setminus L(S_1))$, $Acc(S/S_1)(u) = P(S)$.

**Proposition 6.10** If $S$, $S_1$ and $S_2$ are acceptance specifications over $\Sigma$ then:

\[S_1 \otimes S_2 \leq S \iff S_2 \leq \rho(S/S_1)\]
Similarly to the proof for modal specifications, we use the previous result to establish the characteristic property of a residual operation for behavioral reuse of a component:

**Proposition 6.11** If \( S \) and \( S_1 \) are acceptance specifications then:

\[
C_2 \models S / S_1 \text{ iff } \forall C_1 : [C_1 \models S \Rightarrow C_1 \otimes C_2 \models S]
\]

As previously briefly noticed, acceptance specifications strictly subsume modal specifications. Indeed, consider the two following transformations:

**Definition 6.12** Let \( S = (L, \text{must}, \text{may}) \in \mathcal{MS} \) and \( S' = (L', \text{Acc'}) \in \mathcal{AS} \):

- \( j : \mathcal{MS} \to \mathcal{AS} \)
  
  \[j(S) = (L, \text{Acc}) \text{ with } \text{Acc}(u) = \{ X \in \mathcal{P}(\Sigma) \mid \text{must}(u) \subseteq X \subseteq \text{may}(u) \}\]

- \( k : \mathcal{AS} \to \mathcal{MS} \)
  
  \[k(S') = (L', \text{must}', \text{may}') \text{ with } \begin{cases} \text{may}'(u') = \bigcup_{X \in \text{Acc'}(u')} X \\ \text{must}'(u') = \bigcap_{X \in \text{Acc'}(u')} X \end{cases}\]

We have: \( k \circ j = \text{Id} \) but \( j \circ k \neq \text{Id} \). Quotient operations for modal specifications and acceptance specifications can be related:

**Proposition 6.13** The quotient operation for modal specifications is a particularization of the quotient operation for acceptance specifications.

**Proof.** Given \( S \) and \( S_1 \) two modal specifications, we let \( S' \) be the acceptance specifications obtained by quotienting \( j(S) \) and \( j(S_1) \) using definition 6.9. Then the modal specification \( k(S') \) is identical to the one obtained by quotienting \( S \) and \( S_1 \) using definition 5.3. \( \square \)

Synchronous product is a very restrictive form of composition. In proposition 6.11, the component \((C_2, S / S_1)\) restrict the behavior of all possible \((C_1, S_1)\) so that the composite system \(C_1 \otimes C_2\) realizes \(S\). In the next section, we investigate an approach where the component \((C_2, S / S_1)\) may also contribute directly to the realization of the specification \(S\): given \(S\) a global specification over \(\Sigma\) and \((C_1, S_1)\) a component to be reused over \(\Sigma_1\), the events belonging to \(\Sigma \setminus \Sigma_1\) are realized by the component \((C_2, S / S_1)\). Thus, we now consider residual of specifications when component composition corresponds to mixed product [4]. We also consider internalization of event that is \(C_1\) and \(C_2\) may evolve without being observed externally.

7 Using mixed product with internalization of events as component composition

We first recall the definition of mixed product, restriction and expansion of languages:

**Definition 7.1** \(\ast\) Given \(C_1\) and \(C_2\) two languages respectively over \(\Sigma_1\) and \(\Sigma_2\), the mixed product of \(C_1\) and \(C_2\) is the language:

\[C_1 \bowtie C_2 = \{ w \in (\Sigma_1 \cup \Sigma_2)^* \mid \Pi_{\Sigma_1}(w) \in C_1 \text{ and } \Pi_{\Sigma_2}(w) \in C_2 \}\]
• Given $C$ a language over $\Sigma'$ and $\Sigma \subseteq \Sigma'$, the restriction of $C$ to the alphabet $\Sigma$ is the language:
$$C_{|_{\Sigma}} = \{ u \in \Sigma^* \mid u = \Pi_{\Sigma}(w) \text{ with } w \in C \}$$

• Given $C$ a language over $\Sigma$ and $\Sigma \subseteq \Sigma'$, the expansion of $C$ to the alphabet $\Sigma'$ is the language:
$$C_{|_{\Sigma'}} = \{ w \in \Sigma'^* \mid \Pi_{\Sigma}(w) \in C \}$$

Now, we generalize these operations for acceptance specifications:

7.1 Mixed product of acceptance specifications

**Definition 7.2** The mixed product of the acceptance specifications $S_1 \in AS(\Sigma_1)$ and $S_2 \in AS(\Sigma_2)$ is the acceptance specification $S_1 \cap S_2$ over $L(S_1) \cap L(S_2)$ with for all $u \in L(S_1) \cap L(S_2)$:
$$\text{Acc}(S_1 \cap S_2)(w) = \{ (X_1 \cup (\Sigma_2 \setminus \Sigma_1)) \cap (X_2 \cup (\Sigma_1 \setminus \Sigma_2)) \mid X_1 \in \text{Acc}(S_1)(\Pi_{\Sigma_1}(w)) \text{ and } X_2 \in \text{Acc}(S_2)(\Pi_{\Sigma_2}(w)) \}$$

When $\Sigma_1 = \Sigma_2$ the definition of the synchronous product is retrieved.

The mixed product of acceptance specifications can be related in a general way to the synchronous product:

**Definition 7.3** Given $\Sigma \subseteq \Sigma'$ and $S \in AS(\Sigma)$, the $\tau$-expansion of $S$ to $\Sigma'$ is the acceptance specification $\tau_{\Sigma'}(S)$ over $(L(S))_{|_{\Sigma'}}$ with:
$$\text{Acc}(\tau_{\Sigma'}(S))(w) = \{ Y \cup (\Sigma' \setminus \Sigma) \mid Y \in \text{Acc}(S)(\Pi_{\Sigma}(w)) \}$$

This operation consists in saturating the element of each acceptance set with all events of $(\Sigma' \setminus \Sigma)$. Thus, mixed product of acceptance specifications is reduced to synchronous product (cf. definition 6.8):

**Proposition 7.4** Given $S_1 \in AS(\Sigma_1)$ and $S_2 \in AS(\Sigma_2)$:
$$S_1 \cap S_2 = \tau_{\Sigma_1 \cup \Sigma_2}(S_1) \otimes \tau_{\Sigma_1 \cup \Sigma_2}(S_2)$$

The restriction operation adjoint of the $\tau$-expansion is the following:

**Definition 7.5** Given $\Sigma \subseteq \Sigma'$ and $S' \in AS(\Sigma')$, the II-restriction of $S'$ to $\Sigma$ is the acceptance specification $\Pi_{\Sigma}(S')$ over $(L(S'))_{|_{\Sigma}}$ with:
$$\text{Acc}(\Pi_{\Sigma}(S'))(w) = \{ Y \mid Y \cup (\Sigma \setminus \Sigma') \in \bigcap \{ \text{Acc}(S')(w) \mid w \in L(S') \text{ and } \Pi_{\Sigma}(w) = u \} \}$$

**Proposition 7.6** Given $S \in AS(\Sigma)$ and $S' \in AS(\Sigma')$ with $\Sigma \subseteq \Sigma'$,
$$\tau_{\Sigma'}(S) \leq S' \iff S \leq \Pi_{\Sigma}(S')$$

To deal with internalization of event, we now define the restriction of an acceptance specification to a sub-alphabet:
7.2 Restriction of acceptance specification

**Definition 7.7** Given \( \Sigma \subseteq \Sigma' \) and \( \mathcal{S} \in \mathcal{AS}(\Sigma') \), the restriction of \( \mathcal{S} \) to \( \Sigma \) is the acceptance specification \( \mathcal{S}_{\mid \Sigma} \) over \( (L(\mathcal{S}))_{\mid \Sigma} \) with:

\[
\text{Acc}(\mathcal{S}_{\mid \Sigma})(u) = \bigcup \{ \{ X \cap \Sigma \mid X \in \text{Acc}(\mathcal{S})(w) \text{ and } X \cap \Sigma \neq \emptyset \text{ when } X \neq \emptyset \} \mid w \in L(\mathcal{S}) \text{ and } \Pi_{\Sigma}(w) = u \}\]

**Definition 7.8** Given \( \Sigma \subseteq \Sigma' \) and \( \mathcal{S}' \in \mathcal{AS}(\Sigma) \), the expansion of \( \mathcal{S}' \) to \( \Sigma' \) is the acceptance specification \( \mathcal{S}'_{\mid \Sigma'} \) over \( (L(\mathcal{S}'))_{\mid \Sigma'} \) with:

\[
\text{Acc}(\mathcal{S}'_{\mid \Sigma'})(w) = \{ X \mid X \cap \Sigma \in \text{Acc}(\mathcal{S}')(\Pi_{\Sigma}(w)) \} \cup \{ X \mid X \neq \emptyset \text{ and } X \subseteq (\Sigma' \setminus \Sigma) \}
\]

This operation consists in allowing after each trace \( u \) of \( \mathcal{S}' \) finite sequences of events in \( \Sigma' \setminus \Sigma \).

**Proposition 7.9** Given \( \mathcal{S} \in \mathcal{AS}(\Sigma') \) and \( \mathcal{S}' \in \mathcal{AS}(\Sigma) \) with \( \Sigma \subseteq \Sigma' \),

\[ \mathcal{S}_{\mid \Sigma} \leq \mathcal{S}' \iff \mathcal{S} \leq \mathcal{S}'_{\mid \Sigma'} \]

7.3 Adjoint of the mixed product with internalization of events of acceptance specification

From the previous propositions, we can deduce:

\[
(S_{1 \cdot \tau} S_2)_{\mid \Sigma} \leq S \iff S_{1 \cdot \tau} S_2 \leq S_{\mid (\Sigma_1 \cup \Sigma_2)} \quad \text{by prop.7.9}
\]

\[
\iff \tau_{\Sigma_1 \cup \Sigma_2}(S_1) \otimes \tau_{\Sigma_1 \cup \Sigma_2}(S_2) \leq S_{\mid (\Sigma_1 \cup \Sigma_2)} \quad \text{by prop.7.4}
\]

\[
\iff \tau_{\Sigma_1 \cup \Sigma_2}(S_2) \leq S_{\mid (\Sigma_1 \cup \Sigma_2)} / \tau_{\Sigma_1 \cup \Sigma_2}(S_1) \quad \text{by prop.6.10}
\]

\[
\iff S_2 \leq \Pi_{\Sigma_2}(S_{\mid (\Sigma_1 \cup \Sigma_2)} / \tau_{\Sigma_1 \cup \Sigma_2}(S_1)) \quad \text{by prop.7.6}
\]

Similarly to the proof for modal specifications in the synchronous case, we use the previous equivalence to establish the characteristic property of a residual operation for behavioral reuse of a component when the product of components is the mixed product with internalization of events:

**Proposition 7.10** Given \( \mathcal{S} \in \mathcal{AS}(\Sigma) \), \( \mathcal{S}_1 \in \mathcal{AS}(\Sigma_1) \) and \( \Sigma_2 \) such that \( \Sigma \subseteq \Sigma_1 \cup \Sigma_2 \):

\[
\forall \mathcal{C}_1. \mathcal{C}_1 \models \mathcal{S}_1 \Rightarrow (\mathcal{C}_1 \cdot \tau \mathcal{C}_2)_{\mid \Sigma} \models \mathcal{S} \iff \mathcal{C}_2 \models \Pi_{\Sigma_2}(S_{\mid (\Sigma_1 \cup \Sigma_2)} / \tau_{\Sigma_1 \cup \Sigma_2}(S_1))
\]

8 An example

This example is inspired from [3]. In this paper, component interfaces are designed via interface automata. We refine the intended behavior thanks to acceptance automata; the quotient of acceptance specifications defined in 6.9 can be adapted when the support of the specification is a regular prefix-closed language [13].

- The goal is to build a system satisfying the following specification \( \mathcal{S} \) over the alphabet \( \Sigma = \{ \text{msg, ok, fail} \} \):
To realize \( S \), we aim at reusing a component satisfying the following specification \( S_1 \) describing the behavior of a communication channel:

\[
\begin{array}{c|c}
\text{Acc}(S_1) & \text{msg} \\
0 & \{\text{msg}\} \\
1 & \{\text{send}\} \\
2 & \{\text{ack}, \text{nack}\} \\
3 & \{\text{send}\} \\
4 & \{\text{ack}, \text{nack}\} \\
5 & \{\text{ok}\} \\
6 & \{\text{fail}\}
\end{array}
\]

\( S_1 \) is defined over the alphabet \( \Sigma_1 = \{\text{msg, send, ack, nack, fail, ok}\} \). Note that loss of message is allowed in \( \text{Acc}(S_1)(2) \) which is not the case in \( \text{Acc}(S_1)(4) \).

- We let \( \Sigma_2 = \{\text{ack, nack, send}\} \). As \( \Sigma \setminus \Sigma_1 = \emptyset \), \((C_2, S/S_1)\) will restrict the behavior of \((C_1, S_1)\) to enforce \( S \). We have: \( \tau_{\Sigma_1 \cup \Sigma_2}(S_1) = S_1 \) as \( \Sigma_2 \subseteq \Sigma_1 \).

- We compute \( S_1^{\uparrow}(\Sigma_1 \cup \Sigma_2) \):

\[
\begin{array}{c|c}
\text{Acc}(S_1^{\uparrow}(\Sigma_1 \cup \Sigma_2)) & \text{msg} \\
0' & \{\text{msg}\} \\
1' & \{\text{ok}\}
\end{array}
\]

- \( \text{Acc}(S_1^{\uparrow}(\Sigma_1 \cup \Sigma_2))(0') = \{\text{msg}\}, \{\text{msg}\} \cup X, X \) with \( X \subseteq \Sigma_2 \) and \( X \neq \emptyset \).
- \( \text{Acc}(S_1^{\uparrow}(\Sigma_1 \cup \Sigma_2))(1') = \{\text{ok}\}, \{\text{ok}\} \cup X, X \) with \( X \subseteq \Sigma_2 \) and \( X \neq \emptyset \).

- Then we compute the quotient \( S_1^{\uparrow}(\Sigma_1 \cup \Sigma_2)/\tau_{\Sigma_1 \cup \Sigma_2}(S_1) \) (in the following figure, only transitions labeled by required events are depicted):
\[ \text{Acc}(00') = \{ \{ \text{msg} \} \cup X \mid X \subseteq \{ \text{ok, fail, ack, nack, send} \} \}; \]
\[ \text{Acc}(11') = \{ \{ \text{send} \} \cup X \mid X \subseteq \{ \text{ok, fail, ack, nack, msg} \} \}; \]
\[ \text{Acc}(21') = \{ \{ \text{ack, nack} \} \cup X \mid X \subseteq \{ \text{ok, fail, msg, send} \} \}; \]
\[ \text{Acc}(31') = \{ \{ \text{send} \} \cup X \mid X \subseteq \{ \text{msg, ok, fail, ack, nack} \} \}; \]
\[ \text{Acc}(41') = \{ \{ \text{ack} \} \cup X \mid X \subseteq \{ \text{ok, fail, msg, send} \} \}; \]
\[ \text{Acc}(51') = \{ \{ \text{ok} \} \cup X \mid X \subseteq \{ \text{msg, fail, ack, nack} \} \}; \]
\[ \text{Acc}(61') = \emptyset. \]

• Last, we apply the operation \( \Pi_2 \) on the acceptance specification equivalent to the previous pseudo-acceptance specification. The result is the following:

\[ \text{Acc}(a) = \{ \{ \text{send} \}, \{ \text{send, ack} \}, \{ \text{send, nack} \}, \{ \text{send, ack, nack} \} \}; \]
\[ \text{Acc}(b) = \{ \{ \text{ack, nack} \}, \{ \text{ack, nack, send} \} \}; \]
\[ \text{Acc}(c) = \{ \{ \text{send} \}, \{ \text{send, ack} \}, \{ \text{send, nack} \}, \{ \text{send, ack, nack} \} \}; \]
\[ \text{Acc}(d) = \{ \{ \text{send} \}, \{ \text{send, ack} \}, \{ \text{send, nack} \}, \{ \text{send, ack, nack} \} \}; \]
\[ \text{Acc}(e) = \{ \{ \text{ack} \}, \{ \text{ack, send} \} \}; \]
\[ \text{Acc}(\top) = \mathcal{P}(\Sigma_2). \]

We remark that a sent message that has been acknowledged negatively must then be acknowledged positively (in state \( e \), \textit{ack} is required and \textit{nack} is forbidden).

## 9 Conclusion

In this paper, we have studied the problem of behavioral reuse of a component as the computation of a residual specification. We have introduced modal specification and acceptance specification as formalisms to specify component behavior. They allow
to address restricted forms of liveness. Quotient of mu-calculus formulas was investigated in [1]. Mu-calculus is quite expressive but the complexity of the proposed quotient operation is double exponential in the size of the tree automata equivalent to the quotiented formulas. In contrast, our solutions using the automata-based version of modal and acceptance specifications are polynomial [13]. Furthermore, to our knowledge, our approach is the first to consider components as black box in equation solving: the equation is solved for a given set of possible implementations characterized by a specification $S_1$.

Future works concern the application of these techniques to the component adaptation problem. In particular, these techniques seems suited when detection of mismatch between components is performed thanks to the description of the properties the system should verify [2]. Moreover modal and acceptance specifications are sets equipped with a lattice structure and a monoid structure with a residual operation, adjoint of a commutative product operation ie. are residuated lattices. We are interested in a more precise characterization of the underlying algebraic structure of the sets of modal and acceptance specification in order to develop the basis of an algebraic theory of components adaptation and reuse.

Acknowledgment: The author would like to thank Eric Badouel and Philippe Darondeau for many suggestions and discussions.

References

Multiple Concern Adaptation for Run-time Composition in Context-Aware Systems

Javier Cámara, Carlos Canal, Gwen Salaün

Dept. of Computer Science, University of Málaga, Spain.
Emails: jcamara@lcc.uma.es, canal@lcc.uma.es, salaun@lcc.uma.es

Abstract
Context-Aware computing studies the development of systems which exploit context information (e.g., user location, network resources, time, etc.), which is specially relevant in mobile systems and pervasive computing. When these systems are built assembling pre-existing software components (COTS), the composition process must be able to solve potential interoperability problems, adapting component interfaces. In addition, the composition must be adapted to the execution conditions of such systems, which are likely to change at run-time, affecting component behaviour. This work presents an approach to the flexible composition of possibly mismatching behavioural interfaces in systems where context information can vary at run-time. Our approach enables composition at run-time, enabling dynamic changes in composition according to context changes. Furthermore, our approach simplifies the specification of composition/adaptation by keeping Separation of Concerns, and is able to handle context-triggered adaptation policies.

Keywords: Component-based Software Development, Run-time Composition, Model-based Adaptation, Separation of Concerns

1 Introduction

In recent years, software systems engineering has evolved from the development of applications from scratch, to the paradigm known as Component-Based Software Development (CBSD) [23], where third-party, pre-existing software components known as Commercial-Off-The-Shelf or COTS are selected and assembled in order to build fully working systems. The main advantage of CBSD is that it promotes component reuse, saving time and money in system development. Due to the Black-box nature they exhibit, these components are equipped with public interfaces to access their functionality. However, most of the time a COTS component cannot be directly reused as is, requiring adaptation when composed with the rest of the system due to possible interoperability problems with other components.

Software Adaptation [6] is a field characterised by the modification of component interactions through the use of special components called adaptors [27], which

---

1 This work has been partially supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and project P06-TIC-02250 funded by the Andalusian local Government.

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

are capable of enabling components with mismatching interfaces to interoperate. These are automatically built from an abstract specification of how mismatch can be solved (i.e., adaptation mapping or contract), based on the description of component interfaces. Particularly, recent research efforts [1, 8, 21, 19] concentrate on behavioural interoperability, extending interfaces with a description of the protocol they follow, and ensuring correctness and termination of component interactions.

Traditionally, the context of reuse of a component used to be more or less static (e.g., spreadsheets, banking systems, etc.), but the advent of mobile and pervasive computing has given rise to a whole new breed of systems where execution conditions are likely to change at run-time (e.g., time, user location, resource availability, etc.). Although Context-Aware computing [9] has broadly studied the development of systems exploiting context information, it does not deal with the specificities of component-based systems. Component-based, context-aware systems must be able to reflect environmental changes affecting system behaviour, altering the composition at run-time.

This work advocates for the flexible (i.e., modifiable at run-time) interaction between an arbitrary number of components depending on the current state of the execution of the system. This approach serves a double purpose: On one hand, it adapts the composition to the changing environmental conditions or context of the system. On the other hand, it works out the potential incompatibilities among components. This work develops and formalises the seminal ideas sketched in [4], extending previous work [5] which focuses on run-time composition and adaptation techniques. Run-time composition is an essential feature of our proposal since it avoids the costly generation of the full adaptor, and its recomputation when the system changes (e.g., addition of a new service).

Moreover, we reduce the complexity of mapping specification by enabling separating concerns [12], breaking down the specification of adaptation into multiple context facets which express the different concerns which may affect system behaviour. Furthermore, our proposal is able to deal with adaptation policies which may depend on context changes (i.e., context-triggered actions), an important issue in Context-Aware computing which remains obviated by previous proposals in adaptation [10].

The rest of this paper is structured as follows: Section 2 presents a Wireless Medical Information System used to illustrate the different issues described in the remaining sections. Section 3 describes our run-time composition/adaptation proposal. Section 4 describes some related work. Finally, Section 5 draws up conclusions and further work.

2 Case Study: Wireless Medical Information System

In order to illustrate the different issues addressed in this paper, we describe a Wireless Medical Information System based on a real-world example, although simplified here for the sake of clarity. As it can be observed in Figure 1, the system consists of a client-server application which systematically processes the clinical information related to patients in a medical institution. There is a central server with a DBMS installed which is queried remotely from PDAs. Handheld devices and server are
The client PDA must be able to work with three user profiles which have different privileges: while Staff can access a restricted set of information (e.g., administrative info for attendants), Doctors and Nurses can access also medical information, and prescribe specific treatments for any given patient in the case of doctors. When a nurse applies a treatment previously prescribed by a doctor on a specific patient, the actions and/or medicines administrated must be entered in the application (treatment logging).

It is important to maintain the application operative on the PDA continuously, hence a lightweight DBMS component has been incorporated to each PDA, enabling the user to work locally whenever the wireless signal is lost (local mode). Moreover, since the storage on a PDA is very limited, only treatment prescriptions and logging are stored in the local DBMS. Patient information is retrieved from Radio Frequency Identification (RFID) [26] tags fixed on patient bracelets when in local mode. This is achieved through an RFID reader incorporated on each PDA. Every time the client on the PDA returns from local to remote mode, it is mandatory to synchronise the data stored locally with the central DBMS. This process must be automatically conducted by the application.

The client PDA is being reused from a legacy system which does not take into account user profiles, hence the appropriate restrictions must be applied at the composition level in order to limit the access rights to the DBMS as informally sketched above. Likewise, this client is built to work with a DBMS, independently of its location (the client does not know about the existence of the local DBMS nor the RFID reader), requiring adaptation to the new characteristics of the system.
3 Separation of Concerns for Run-time Composition and Adaptation

In this section, we first introduce our component and environment model. Second, we describe a graphical notation for our mapping which features separation of concerns, simplifying the specification of adaptation. Finally, we detail the process used for composition.

3.1 Component and Environment Model

Since this work deals with the behavioural interoperability level, we have to extend component interfaces with a description of the protocols they follow. In order to do so, we use Labelled Transition System (LTS) descriptions, which take the set of messages (both offered and required) in the signature of a component as input alphabet. Many automata-based languages can be used to describe behavioural interfaces (e.g., Interface Automata, UML State Diagrams, etc.). We chose LTSs because of their simplicity and expressiveness, and also because they are widely used for design and formalisation purposes. In addition, this notation is particularly suited to be used as input for the algorithms presented in this work since it gives explicit information about the states of components. Moreover, it is user-friendly since its graphical representation is straightforward, in contrast with other notations such as process algebras:

**Definition 3.1** [LTS] A Labelled Transition System is a tuple \((A, S, I, F, T)\) where:
- \(A\) is an alphabet formed by a set of events (\(a! / a?\), emissions and receptions able to synchronise),
- \(S\) is a set of states,
- \(I \in S\) is the initial state,
- \(F \subseteq S\) are final states, and
- \(T \subseteq S \times A \times S\) is the transition function.

**Definition 3.2** [Execution Environment] An Execution Environment \(E = \{e_1, \ldots, e_n\}\) is the set of environmental signals (or signals, for short) which do not belong to any particular component.

**Example 3.3** Figure 2 depicts the different protocols for the components in our case study: The CLIENT component can log an user in/out (\(loginDoctor! / loginNurse! / logout!\)), request the insertion of a given treatment on the database (\(prescTreat!\)), log the administration of a treatment (\(logTreat!\)), or...
request some information to the server \texttt{query!}, returned by \texttt{response?}. It is worth noticing that the client grants the same privileges to all users. The \textit{DBMS} and \textit{LOCAL DBMS} components have analogous actions, with the exception that the latter only accepts \texttt{prescTreat!?/logTreat!?} requests, and \texttt{synch?}, which triggers a synchronisation process with the DBMS components. This synchronisation is effectively performed by \texttt{update!}/\texttt{update?} on both component interfaces. Finally, the \textit{RFID READER} component first has to be enabled (\texttt{enable!!}). Subsequently, it can receive a \texttt{read?} command, returning the requested information on \texttt{data!}. In our case study, when the wireless signal is found or lost, we will consider the pair of signals $E = \{\text{connected!}, \text{disconnected!}\}$ for our execution environment.

The composition in this system must take into account a couple of different concerns: (i) User profile. The client we are reusing does not distinguish user privileges, therefore we must provide the means to restrict user privileges based on user profiles (e.g., a \texttt{prescTreat!} should not be issued to the DBMS unless a doctor user is logged in -\texttt{loginDoctor!}-). (ii) Wireless coverage. Working in connected mode, queries are issued to the DBMS, but when in local mode, a \texttt{query!} request must be issued to the \textit{RFID READER} component.

Independently of the different concerns to be considered for the composition, there are interoperability issues to be solved relative to the different interfaces of our case study:

- Name mismatch occurs when a particular process is expecting a particular input event, and receives one with a different name (e.g., \textit{CLIENT} sends \texttt{query!} while \textit{DBMS} is expecting \texttt{request?!}).
- 1-to-many interaction is given if one or more events on a particular interface have not an equivalent in the counterpart’s interface. If we take a closer look at the \textit{CLIENT} and \textit{RFID READER} interfaces, it can be observed that while the client is just sending \texttt{query!} when it wants to read some data, the reader is expecting two messages (\texttt{enable?} and \texttt{read?!}). While the latter actually requests the data to the reader, the former has no correspondence on the \textit{CLIENT} interface. Hence, the composition process has to solve this mismatch by making the reader evolve independently, through the reception of \texttt{enable?} before each \texttt{read?!} request.

Finally, composition must also consider the synchronisation of local and remote DBMS, triggering \texttt{synch?} when the system recovers connectivity (\texttt{connected!!}).

3.2 Mapping

A mapping establishes correspondences or bindings between operations on the different component signatures in order to make interactions explicit and solve possible mismatch between them. When this correspondence is fixed or static, the specification of the mapping is relatively simple [2], but it gets more complicated in systems which require the modification of these correspondences at run-time depending on the current state of context. Moreover, there is an additional complexity in the specification of changes in the context derived from all the possible combinations of factors which may influence the execution of the system. In order to tackle this problem, the mapping described in this work features Separation of Concerns, a divide-and-conquer strategy which makes the problem easier to manage by break-
To begin with, correspondences are specified in our mapping by denoting communication among several components and the environment. For that we use synchronisation vectors [7]:

**Definition 3.4** [Synchronisation Vector] A synchronisation vector (or vector for short) for a set of components \( C_{i \in \{1, \ldots, n\}} = (A_i, S_i, I_i, F_i, T_i) \), and an execution environment \( E \) is a tuple \( \{l_1, \ldots, l_n, e_1, \ldots, e_m\} \) with \( l_i \in A_i \cup \{\varepsilon\} \), \( e_j \in \{1, \ldots, m\} \) \( \in E \cup \{\varepsilon\} \), where \( \varepsilon \) means that a particular component or signal does not participate in a synchronisation.

A vector may involve any number of components and/or signals and does not require interactions on the same names of events as it is the case in process algebras [16,15]. To identify component messages in a vector, their names are prefixed by the component identifier (\( \langle c : \text{prescTreat}!, l : \text{prescTreat}? \rangle \)), whereas signals are not prefixed, e.g., (\( \langle \text{connected}!, l : \text{synch}? \rangle \)). Moreover, in a vector, all the components which do not participate in an interaction may be removed to simplify the notation.

Communication expressed by vectors affects the state of context. To keep track of changes in context we use vector expressions. These are predicates over vectors, meaning that given a specific vector, a vector expression can either match it or not:

**Definition 3.5** [Vector Expression] A vector expression is a tuple \( \langle l_1, \ldots, l_n \rangle \) where each \( l_i \in \{1, \ldots, n\} \) is an expression designating one or several events/signals. Expressions may contain the following wildcards:

- \( * \) designates a sequence of 0 or more characters to be used either on the event prefix or identifier. For instance, \( d: \text{login}? \) designates in our case study both \( d: \text{login}? \) and \( l: \text{login}? \).
- \( .. \) designates 0 or more events/signals. Hence, a vector expression such as \( \langle \text{connected}!, .. \rangle \) would match on \( \langle \text{connected}! \rangle \) or \( \langle \text{connected}!, l : \text{synch}? \rangle \), for instance.

We describe a mapping through an incremental specification, focusing on the different facets or concerns involved in the composition. Each concern is represented as a context facet, where the changes between the different states of that particular context facet are triggered either by component messages or signals designated by vector expressions. Specifically, when an expression matches on the vector which is currently being applied in the composition, the transition is triggered. The order of events and signals does not need to coincide both in the expression and the vector for them to match.

**Definition 3.6** [Context LTS] A Context LTS for a set of components \( C_{i \in \{1, \ldots, n\}} = (A_i, S_i, I_i, F_i, T_i) \), and an execution environment \( E \) is a tuple \( (A_c, S_c, I_c, F_c, T_c) \) specified over a set of vectors \( V \) where: \( A_c \) is an alphabet (set of vector expressions specified over \( A_i \) and \( E \) ), \( S_c \subseteq Id \times V \) is a set of context states, \( I_c \in S_c \) is the initial state, \( F_c \subseteq S_c \) are final states, and \( T_c \subseteq S_c \times A_c \times S_c \) is the transition function. \( Id \) stands for a set of identifiers.

**Definition 3.7** [Context Facet] A Context Facet is is a tuple \( (CLTS, Vc) \) where:
• CLTS is a context LTS defined for a given set of components and an execution environment.

• $V_c$ is a set of vectors used by CLTS context states.

Vectors will only be taken into account for the composition when associated to the facet’s current state. In addition, the mapping may contain a set of global vectors which are not associated to any particular context facet and are always considered for composition. Moreover, vectors in different facets may share the same identifier (in such a case we refer to them as vector declarations). This characteristic is used to be able to specify a modification over a part of the behaviour of the system which has already been specified by overriding it. Facets have a precedence order assigned, hence the declaration of a vector in a facet with higher precedence overrides a lowest precedence declaration. All facets have a different precedence order higher than 0. Global vectors have a precedence order $p = 0$, and may be overridden by vectors on facets.

Definition 3.8 [Mapping] A context mapping built over components $C_i \in \{1, \ldots, n\} = (A_i, S_i, I_i, F_i, T_i)$, is defined as a tuple $(CF_j \in \{1, \ldots, m\}, Vg, P)$ where:

- $CF_j \in \{1, \ldots, m\} = (\text{CLTS}_j, \text{Vc}_j)$ is a set of context facets.
- $Vg$ is a set of vectors global to all context facets and states in the mapping.
- and $P = \{p_1, \ldots, p_m\}, p_j \in \{1, \ldots, m\}, \forall p_k, p_l \in P \ p_k \neq p_l \land p_j > 0$ is a list of precedence orders, one for each context facet.

It is worth noticing that there may be cases in which we may need to assign a specific precedence order to just one vector. This could be expressed by adding the precedence order in the vector declaration (e.g., $v_{\text{conn}}[3] = \left( \text{connected}!, 1 : \text{synch}! \right)$).

Example 3.9 Figure 3 depicts context facets for the mapping we use for our case study. These form the mapping along with the precedence orders $P = \{p_{\text{WC}} = 1, p_{\text{UP}} = 2\}$, and the following set of global vectors $Vg$:

- $v_{\text{had}} = \langle c: \text{loginDoctor}!, d: \text{login}\rangle$
- $v_{\text{resp}} = \langle c: \text{response}?, d: \text{response}! \rangle$
- $v_{\text{lin}} = \langle c: \text{loginNurse}!, d: \text{login}\rangle$
- $v_{\text{log}} = \langle c: \text{logout}!, d: \text{logout}! \rangle$
- $v_{\text{ptr}} = \langle c: \text{prescTreat}!, d: \text{prescTreat}\rangle$
- $v_{\text{qry}} = \langle c: \text{query}!, d: \text{request}! \rangle$
- $v_{\text{up}} = \langle c: \text{logTreat}!, d: \text{logTreat}\rangle$
- $v_{\text{up}} = \langle l: \text{update}!, d: \text{update} \rangle$

It can be observed that while the set of global vectors specifies the general behaviour of the system, the different facets modify composition according to the part of the context they are concerned about:

- WIRELESS COVERAGE (WC) is concerned about the local/remote operation mode. This facet specifies alternatives for the global vectors in the following way:
  - When the state of this facet is LOCAL, treatment prescription and logging are performed on the local DBMS (vectors $v_{\text{ptr}}$ and $v_{\text{ltr}}$, respectively). Likewise, queries for patient data are performed on the RFID reader (vectors $v_{\text{qry}}$, $v_{\text{ebl}}$ and $v_{\text{resp}}$). Notice that $v_{\text{ebl}}$ will help to solve the 1-to-many interaction problem described in our case study, by making the RFID reader evolve independently whenever it is ready to receive enable?.
The state LOCAL – DB – UPDATED is similar to LOCAL, although it is entered when the local database is modified. In addition to the vectors described for local mode, \( v_{\text{sync}} \) is included for local-remote DBMS synchronisation. This state is introduced in the mapping in order to avoid unnecessary synchronisations (when the local database has not been updated) which can cause additional network traffic.

**USER PROFILE (UP)** modifies the functionality of the system according to the current user profile: \( v_{\text{ptr}} \) restricts treatment prescription in this facet (this vector appears in all states except DOCTOR, which are entitled to enter prescriptions). Treatment logging is similarly restricted for non-nurses by \( v_{\text{ltr}} \).

### 3.3 Composition

Once we have described the inputs to our approach, we will detail the process followed for composition and adaptation. First, we illustrate the selection of active vectors for a particular state of the global context (i.e., the combination of the active states of all context facets). Second, we sketch the approach used to ensure correct termination of the system. Finally, we describe the composition algorithm, illustrating it with an execution trace coming from our case study.
3.3.1 Selection of an Active Vector Set

For each state of the global context, there is a vector set that describes the possible interactions among the components. In order to select those vectors, namely active vectors, we define the function active, which takes as input mapping \( M = (CF_{j \in \{1,...,n\}, Vg, P}) \) and the list of current states for the facets cstates.

It returns the set of active vectors for the current state of the global context (including global vectors which have not been overridden). In order to do that it makes use of the \( \cup \) operator, which returns the set of couples \((v, p)\) of the different sets given as input, where \( p \) is the highest precedence for \( v \). For instance, for the sets \( A = \{(v_1, 1), (v_2, 1), (v_3, 2)\} \) and \( B = \{(v_1, 2), (v_2, 0)\} \), \( A \cup B = \{(v_1, 2), (v_2, 1), (v_3, 2)\} \).

Function id returns the identifier of context state \( s \).

\[
active(M, cstates) = \\
\{v \mid (v, p) \in \{(v, 0) \mid v \in Vg\} \cup \{(v, P[j]) \mid v \in V, v \in Vc_{j \in \{1,...,n\}}, \exists s \in S_{c_j} \ s = cstates[j] \land (id(s), V) \in s\} \}
\]

\( A \cup B = \{(v, p) \mid (v, p) \in A \cup B \land \forall (v', p') \in (A \cup B) \backslash \{(v, p)\}, v = v', p \geq p'\} \)

Example 3.10 In order to illustrate how active vectors are selected for a given state of the context, we use the mapping for our case study depicted in Figure 3.

We focus on the particular vector \( v_{ltr} \), declared as \( v_{ltr} = \langle c: logTreat!, d : logTreat? \rangle \) in the global set of vectors (the logTreat! message is issued to the remote DBMS), \( v_{ltr} \) is defined as \( v_{ltr} = \langle c: logTreat!, l : logTreat? \rangle \) in the WIRELESS COVERAGE facet (the operation is performed on the local DBMS), and as \( v_{ltr} = \langle c: logTreat! \rangle \) in the USER PROFILE (the operation is not performed, since the client request logTreat! corresponds to no action on the rest of the components). We also consider that the set of current states in facets are \( C = \{DOCTOR, LOCAL\} \).

Focusing on WIRELESS COVERAGE, we can observe that since \( v_{ltr} \) is associated to the LOCAL state, the declaration on this facet overrides the global declaration. Similarly, since \( v_{ltr} \) is associated to DOCTOR and the precedence of the USER PROFILE is higher, the currently dominant declaration is again overridden. Finally, the operation is not performed since the prevailing declaration is \( v_{ltr} = \langle c : logTreat! \rangle \). This is consistent with our example since doctors are not allowed to enter administrated treatments on the application. To sum up, we keep the vector with the highest precedence in case there are several vectors identified similarly. The complete set of active vectors for this particular state of the global context is:

\[
\begin{align*}
v_{ln} &= \langle c: loginNurse! \rangle & v_{resp} &= \langle c: response?, r: data! \rangle & v_{lo} &= \langle c: logout!, l: logout? \rangle \\
v_{ld} &= \langle c: loginDoctor! \rangle & v_{qry} &= \langle c: query!, r: read? \rangle & v_{lt} &= \langle c: logTreat! \rangle \\
v_{conn} &= \langle connected! \rangle & v_{en} &= \langle r: enable? \rangle & v_{up} &= \langle l: update!, l: update? \rangle \\
\end{align*}
\]

\( v_{adr} = \langle c: prescTreat!, l: prescTreat? \rangle \)

3.3.2 Ensuring Correct Termination

Adapting interfaces at the protocol level implies not engaging the system into deadlocking executions. A deadlock state is a state which is not final and in which a process cannot evolve. A system deadlocks when all its constituent components are blocked because at least one of them is in a deadlock state. Since deadlock removal cannot be performed before the application of the adaptor as in approaches which
generate full adaptor descriptions [2,10], we have to ensure the existence of one termination state for the system before every application of a vector. Hence, if we cannot find a sequence of vectors to be applied leading to a global termination state for the system after the application of vector \( v \), we do not apply that vector.

The vector to be applied at a specific moment is selected from a set of applicable vectors (i.e., active vectors which in addition can make the system evolve in a given moment). Function \( \text{applicable}_V \) returns the set of applicable vectors from a set of vectors \( V \) for the list of current states associated to components \( C_i \). Note that for our purposes, we will select applicable vectors from the set of currently active vectors:

\[
\text{applicable}_V (\text{states}, C_i, V, E) = \{ v \mid v \in V, (\forall l \in v) ((s_j \in \{1, \ldots, n\}) \in S_j, s_j = \text{states}[j], (s_j, l, s_j') \in T_j, l \neq e) \lor (l \in E)) \}
\]

In order to keep track on the evolution of the system, function \( \text{next}_\text{states} \) computes the states of the components involved in the composition after the application of an specific vector in the current state of the components:

\[
\text{next}_\text{states} ((l_1, \ldots, l_n), \text{states}, T_i, E) = [s'_1, \ldots, s'_n]
\]

where \( \forall i \in \{1, \ldots, n\}, \exists (s_i, l, s'_i) \in T_i, s_i = \text{states}[i], l \neq e, l \notin E \)

In order to know if the search process has found a goal state (i.e., a global termination state), we check if all the components have reached their final state. We define the function \( \text{final} \) as:

\[
\text{final} (\text{states}, F_i) = \text{states}[1] \in F_1 \land \ldots \land \text{states}[n] \in F_n
\]

Considering the nature of our search problem, the use of an informed search algorithm looks like a good strategy in order to find potential solutions efficiently. Specifically, we make use of the A* algorithm [14], a particular best-first search strategy which determines the minimum cost path from a given node \( n \) to a goal state by expanding the most promising candidate paths first. However, we have to provide guidance information for this search. This is achieved by defining a heuristic estimation function \( h(n) \) of the cost of arriving from the current state of the components to a global termination state in the composition. In order to determine the heuristic estimation to be used:

- We define the minimum distance from a specific state \( s \) in a component LTS \( C \) to a final state \( d(s, C) \), as the minimum number of events which have to be applied to traverse the LTS up to a final state. Figure 4 depicts a sample LTS with states tagged with minimum distances to final states. This distance is computed using a variant of the Floyd-Warshall algorithm [18].
- Given a set of components, \( C_i \in \{1, \ldots, n\} = (A_i, S_i, I_i, F_i, T_i) \), a set of current states
states, \( i \in \{1, \ldots, n\} \), \( \text{states}[i] \in S_i \), we define the minimum global distance to a final state for the whole system as:

\[
D(\text{states}, C_i) = \sum_{i=1}^{n} d(\text{states}[i], C_i)
\]

- The heuristic used to inform about how good the application of a specific vector \( v \) is, can be expressed as \( h(v, \text{states}, C_i) = D(\text{next states}(v, \text{states}, T_i), C_i) \)

The heuristic estimate \( h(n) \) is admissible in our case (i.e., it never overestimates the actual cost from node \( n \) to a goal). This is because the distance function \( d \) that we have defined always returns the minimum of the distances from a state in the LTS to a final state, resulting in a lower bound of the estimation. This admissibility guarantees A* to return an optimal solution, if one exists [11].

3.3.3 Composition process

Figure 5 sketches the run-time composition process we propose (for the formal definition of the composition process refer to Appendix A):

(i) The set of active vectors dependent on the current states of the different facets of the context is selected. This selection is performed as described in 3.3.1.

(ii) Run-time composition should avoid to engage into execution branches that may lead to deadlock situations. At this stage the state of the components is checked, and if all of them are in a final state, the composition finishes. Otherwise, the composition engine attempts to select an applicable vector \( v \) which may lead to a global correct termination state for the system. If such vector does not exist, the composition process ends as well.

(iii) Vector \( v \) is processed. First, the engine receives the emissions specified in \( v \). Notice that the engine operates reversing the direction of communication with respect to the events specified in vectors. Next, the engine sends the receptions specified in \( v \). After processing both emissions and receptions the state of the components is updated accordingly. Finally, environmental signals are received by the engine as well, and the state of context facets is updated by matching vector expressions on context facet LTSs with the vector being
(iv) Finally, if the state of the global context has changed, the set of active vectors is updated according to the new state of the context, and composition continues.

**Example 3.11** In order to illustrate the composition process, we describe in Figure 6 a sample execution trace for the composition engine in our case study:

The initial state of the global context is $C = \{\text{STAFF, LOCAL}\}$. The RFID READER is waiting to be enabled, so the composition engine applies $v_{\text{ebl}}(WC)$, making the component evolve independently. Next the client makes a query for patient data ($c : \text{query}!$), and the engine applies $v_{\text{qry}}(WC)$, and $v_{\text{resp}}(WC)$ subsequently as data is returned.

The client requests a treatment prescription, which is received by the composition engine, but it is obviated since the current user profile is not allowed to perform that operation. Hence, $v_{\text{ptr}}(UP)$ is applied.

The user logs in as doctor on the local DBMS through the application of $v_{\text{lid}}(WC)$. This causes a change in the state of context facet $UP$. The new state of the global context is $C = \{\text{DOCTOR, LOCAL}\}$ and active vectors are selected again. The client requests a new treatment prescription, which in this case is effectively performed through the application of $v_{\text{ptr}}(WC)$. The update of the local DBMS causes a change in the state of context facet $WC$. The new state of the global context is now $C = \{\text{DOCTOR, LOCAL } \text{DB - UPDATED}\}$.

The PDA recovers the wireless network signal at this stage. This causes the application of vector $v_{\text{synch}}(WC)$, which triggers the synchronisation of local and remote DBMS. The new state of the global context is now $C = \{\text{DOCTOR, REMOTE}\}$. Subsequently, the LOCAL DBMS causes the application of $v_{\text{up}}$ in order to perform the effective synchronisation process.

The client requests a new treatment prescription, which in this case is performed on the remote DBMS, since coverage is now available. This is achieved through the application of $v_{\text{ptr}}(Vg)$.
• Finally, the user logs out, $v_{lo}(Vg)$ is applied and the composition finishes correctly.

4 Related Work

In Context-Aware Computing applications can discover and take advantage of contextual information (such as user location, time, resource availability, etc.). Although this topic has been broadly studied and the usefulness of this technology has been demonstrated [9], this paradigm does not explicitly deal with the composition and adaptation of software entities within the system.

Regarding separation of concerns, different proposals in the field of Aspect-Oriented Programming have been put forward related to the adaptation of applications. For instance, in [24], Tanter et al. supply support for an aspect language with constructs which adequate the behaviour of aspects to the state of context. The notion of context supported refers to a set of attributes or variables in the application and their value at some specific point in time (context snapshots). Vanderperren et al. present in [25] an extension for the JAsCo programming language [22] which allows the triggering of aspects describing their applicability in terms of a sequence or protocol of previously matched run-time events. These approaches do not deal with the different issues related to the composition of entities (including interface adaptation), providing only a way to extend aspect behaviour according to context information in a static way.

In [17], Mukhija and Glinz describe a contract-based adaptive software architecture which deals with the adaptation of applications at run-time according to their execution environment. While this approach supports the notion of composition, it does not deal with the different issues related to protocol adaptation. For instance, if a component is going to be replaced by a new version, both must conform to the same interface. Moreover, contracts define alternative configurations for the composition according to different states of the context. Likewise, Braione and Picco [3] propose a calculus to specify contextual reactive systems separating the description of behaviours and the definition of contexts in which some actions are enabled or inhibited.

Our approach goes beyond [17] and [3] by allowing a separate representation of the different concerns involved in the composition, which is automatically handled by the composition engine. Moreover, our proposal takes contextual information into account while integrating components with mismatching interfaces, since vectors defined in our mapping notation can work behavioural mismatch out.

Cubo et al. describe in [10] an adaptor-based approach to context-aware adaptation. However, the state of context depends exclusively on the exchange of messages among components during execution. Hence, while this proposal can work out behavioural mismatch situations, adaptation policies depending on other type of context information (i.e., environment) is not supported. Compared to our proposal, [10] does not support separation of concerns. As a consequence, every possible state of the context has to be manually specified by the developer writing the mapping, increasing the complexity of its specification. Moreover, the adaptor generation process does need to consider every possible state of the system (not only context). This implies that the adaptor is no longer valid if new context information or com-
ponents are added or removed at run-time, requiring the costly generation of a new adaptor. On the contrary, our approach does not require any recomputation in case of changes to the system (context information or components), since composition and adaptation are generated and conveniently modified according to the description given in the mapping at run-time.

5 Conclusions

In this paper we have presented an approach to the composition and adaptation of mismatching components in systems where its behaviour may be affected by the execution environment. Our approach applies composition at run-time rather than generating a full adaptor off-line, and simplifies the specification of adaptation applying separation of concerns to the specification of the adaptation mapping. The proposed approach adapts the composition to the changing context of the system and works out potential incompatibilities among components, while taking into account context-triggered actions.

Regarding future work, a first perspective is reconfiguration of the system. While the nature of the mapping and the compositional process we have presented enables the transparent modification of the system, this work does not currently deal with the specifics of the reconfiguration process which takes place after the addition or removal of new context information or components as the system is running. Mapping or component update must be performed only at specific safe points, since the modification of this information at any other point could harm the correct execution of the system. The same applies to context changes during already running transactions, which should be able to execute correctly. A potential solution to this problem is delimiting the boundaries of transactions and delaying the application of context changes until they end.

Our main perspective is to implement this proposal as a composition engine, using Aspect-Oriented Programming (AOP) [13], where unlike in traditional platforms and languages, a particular system can be modified without altering its (base) code. This is achieved by separately specifying modifications as aspects, and a description of their relation with the current system. Then the AOP environment weaves or composes aspects and base code into a coherent program. This weaving process can be performed at different stages of the development, ranging from compilation-time to run-time [20] (dynamic weaving). In this dynamic approach, the virtual machine or interpreter running the code is aware of aspects and controls the weaving process. Hence, aspects can be applied and removed at run-time in a transparent way. Dynamic AOP will enable us to shape up the composition engine as aspects able to: (i) intercept communication (i.e., service invocations) between components; (ii) apply the composition process introduced in this proposal wrt. the adaptation mapping in order to make the right message substitutions; (iii) forward the substituted messages to their recipients transparently.
References


A Composition Algorithm

Function \textit{select\_vector} returns a single applicable vector non-deterministically chosen from \( V \), whose application can lead to a final state for the whole system (function \textit{exist\_final} corresponds to the search process described in 3.3.2, returning true if a correct global termination state exists for the system after the application of \( v \)):

\[
\text{select\_vector}(\text{states}, C, V, E) = \begin{cases} 
  v & \text{if } v \in \text{applicable}_V(\text{states}, C, V, E) \neq \emptyset \\
  v^\perp & \text{otherwise (no vector applicable)}
\end{cases}
\]

We define the functions \textit{emissions}, \textit{receptions}, and \textit{signals} which return the set of emissions, receptions, and signals respectively, of any given synchronisation vector. Note that functions \textit{emissions} and \textit{signals} take an execution environment \( E \) as input in order to discriminate actual signals from component emissions:

\[
\text{emissions}(\{ l_1, \ldots, l_n \}, E) = \{ e \mid l_i = e! \land e \notin E \} \\
\text{receptions}(\{ l_1, \ldots, l_n \}) = \{ r \mid l_i = r^? \} \\
\text{signals}(\{ l_1, \ldots, l_n \}, E) = \{ s \mid l_i = s! \land s! \in E \}
\]

The function \textit{match}(v, ve) returns true if the supplied vector expression \( ve \) matches with vector \( v \), being used to update the state of the different context facets according to the vector which is currently being applied within the composition process.

\begin{algorithm}
\caption{runtime\_composition}
\begin{algorithmic}
\State \text{states} := \{ l_1, \ldots, l_n \}
\State \text{cstates} := \{ i_1, \ldots, i_m \}
\State \text{started} := false // composition start condition
\State \text{CVS} := \text{active}(\text{CF}_j, \text{cstates}, P, Vg)
\State v := \text{select\_vector}(\text{states}, C, CVS, E)
\While{v \neq v^\perp \land \sim\text{final}(\text{states}, F_i) \lor \sim\text{started}}
\State \text{started} := true
\State \text{Em} := \text{receptions}(v)
\State \text{Rec} := \text{emissions}(v, E)
\State S_g := \text{signals}(v, E)
\Repeat {\{receptions\}}
\State r^? \mid r \in \text{Rec}_j, j \in \{1, \ldots, n\}, s_j = \text{states}[j], (s_j, r^?, s'_j) \in T_j
\State \text{states}[j] := s'_j // update of component states
\State \text{Rec} := \text{Rec} \setminus \{ r \}
\Until{\text{Rec} = \emptyset}
\Repeat {\{emissions\}}
\State e! \mid e \in \text{Em}_j, j \in \{1, \ldots, n\}, s_j = \text{states}[j], (s_j, e!, s'_j) \in T_j
\State \text{states}[j] := s'_j // update of component states
\State \text{Em} := \text{Em} \setminus \{ e \}
\Until{\text{Em} = \emptyset}
\Repeat {\{signals\}}
\State \text{signal} \mid \text{signal} \in \text{Sg}
\State \text{Sg} := \text{Sg} \setminus \{ \text{signal} \}
\Until{\text{Sg} = \emptyset}
\For{all } j \in \{1, \ldots, m\}, s_{c_j} = \text{cstates}[j], (s_{c_j}, ve, s'_{c_j}) \in T_{c_j}
\If{match(v, ve)}
\State \text{cstates}[j] := s'_{c_j} // update context facet states
\EndIf
\EndFor
\State \text{CVS} := \text{active}(\text{CF}_j, \text{cstates}, P, Vg)
\State v := \text{select\_vector}(\text{states}, C, CVS, E)
\EndWhile
\end{algorithmic}
\end{algorithm}

This algorithm is an extension of the work described in [5], where further details can be found about its correctness, termination and prototype implementation.

---

Câmara, Canal and Salaën
The STSLib Project: Towards a Formal Component Model Based on STS

Fabricio Fernandes Fabricio.Fernandes@emn.fr Jean-Claude Royer Jean-Claude.Royer@emn.fr

OBASCO project
École des Mines de Nantes-INRIA, LINA
4 rue Alfred Kastler, 44307 Nantes cedex 3, France

Abstract

We present the current state of our STSLib project. This project aims at defining an environment to formally specify and execute software components. One important feature is that our components are equipped with a protocol description, namely a Symbolic Transition System. These descriptions glue together a protocol with guards and input/output notations and a data type part. These sophisticated protocols are well-suited to the design of concurrent and communicating systems but verification remains a difficult challenge. We expect to narrow the gap between the design level and the programming level by providing a runtime support for STS. We give in this paper the main objectives of the STSLib project and overview the current implementation. We address here the component model description, a specific approach to verify these systems and the operational interpreter to execute them. These features are illustrated on a cash point case study.

Keywords: Software component, Behavioural protocol, Symbolic transition system, Verification, Java code generation.

1 Introduction

Software engineering is still evolving in several main directions. One first direction is to provide a better modularization and a separation of concerns. Examples are the numerous work around software architecture, component based programming and aspect oriented programming. A second and old preoccupation has been to provide formal semantics to models and programming features as well as verification means. Associated to this there is the need for tools and automation when possible. A perfect illustration of these trends is trusted software components [23]. Quoting this paper we are particularly concerned with the high road: “the high road is intended to lead components with fully proved properties. The ambition of this goal

1 This work was partly supported by the STREP AMPLE project (www.project-ample.net) and the CAPES program from Brazil.
2 Email: Fabricio.Fernandes@emn.fr
3 Email: Jean-Claude.Royer@emn.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
implies that it’s more long-term, and that its realization must start with relatively
fine-grain (but practically critical) components such as library of classes.”

Following this road we are focusing on the formal specification of concurrent
software components and the generation of Java code from these specifications.
Furthermore we consider mixed specifications, that are mixing protocol and data
type descriptions. The STSLib project aims at providing a framework to define
formal components. We want a powerful and concise formalism for both dynamic
behaviour (control, concurrency, and communications) and data parts with precise
semantics. We expect something between process algebras with values [18,9], but
in a more visual way, and UML Statecharts, but simpler and more rigorous. We
need to connect this formalism with usual verification means (general prover, model-
checkers) but also to develop complementary ways to check the specifications. We
aim at a Java code translation which would be as automated as possible, that is a real
runtime support not only a specification simulator. The objective is to define a Java
tool support allowing the formal design of software components and their execution.

One first originality is the formalism we use for atomic components which is a mix
of protocol description and algebraic data type: the Symbolic Transition System
notion [28] (or STS for short). We develop a specific way to check the components
based on an extension of the synchronous product and the configuration graph
computation. This is applied to a cash point case study and we describe some
results related to our verification experiments. Currently we are also defining a
Java interpreter to execute the component descriptions. Component data parts are
translated into imperative Java code and a runtime support implements a n-ary
rendezvous allowing the synchronization of primitive and composite components.

The outline of this paper is the following. Section 2 will present related work.
The next Section is devoted to a brief introduction to the STS formalism, and the
communication diagram. These notions are illustrated in the cash point case study.
Section 4 describes some experiments we have made about the verification of this
case study. Section 5 shows the general guidelines and principles to define a real
Java runtime support for our components. Lastly we conclude and discuss future
work.

2 Related Work

Related work may be classified into environments dedicated to the formal specifica-
tion of concurrent systems, Java related tools or libraries and automatic translation
into Java programming source code.

Before these, we may mention related work to software components and pro-
tocols. Java/A is an architectural programming language which generates Java
code [7]. It may be viewed as a step ahead from ArchJava [1] which defines an ar-
chitectural extension of Java. Java/A integrates the notions of component, required
and provided interface, port, connector and has some means to verify component
assemblies. One important feature is that they have protocols inside ports which
express the ordering of messages. The language is equipped with tools: namely a
compiler and a model-checker for protocol consistency. The authors give a formal-
ization of the abstract component model in terms of transition systems and states
as algebras. They also prove a consistency result for assemblies which provides the basis for reasoning on assemblies and port compatibility. The semantic model uses a states-as-algebras approach for representing the internals of components and assemblies, and I/O-transition systems for describing the observable behaviour. Note that the semantics do not distinguish between simple components and composite components. The semantics of both is a component which is defined through its internal, algebraic state space and the declaration of the ports it offers. In our case we may distinguish the semantics of a composite from the semantics of a simple component, but we can also abstract away this structural information. Our model supports some simple forms of dynamic reconfiguration but Java/A provides a real support for dynamic port changes. The behaviour of a component is given by an I/O-transition system and a state operator maps each control state to a data state which is an algebra over the state signature of the component. The labels of the transitions are either the internal label or I/O-labels corresponding to the messages sent and received via ports. Our semantic model is close to this. Removing some differences about ports and protocols, we may consider that our configuration graph is an integration of the I/O automata and the state space of Java/A in a unique automaton. However one important difference is that our model of STS allows guards which are not possible in Java/A and this complicates the semantics models and the checking. However due to the close relation between both semantic models it is possible to adapt the results of Java/A to our context.

Some other relevant references are: [14,6,19], but none provide a tool support from specification to code. [14] provides a finite labelled transition model for behavioural interface of components and an automatic way to check their compatibility. The originality is that they consider an optimistic hypothesis and redefine the way to compose dynamic systems. In [19] the authors propose a way to model-check Java components by extracting a model of its environment. This can be seen as a variation of compositional model-checking but verifying specific properties of individual components. The behavioural model of [6] is a subset of the STS model since it has only restricted data types and assignment actions. Additionally it provides a model to encode dynamic configuration of components and this is applied to ProActive Java code. These two last references focused on behaviour extraction from component code while we rather focus on a Java code synthesis approach.

CADP [16] (“Construction and Analysis of Distributed Processes”) is considered as a good representative of the verification tools. CADP is a toolbox for the design of communication protocols and distributed systems based on the ISO language LOTOS. The CADP tool box provides a rich set of tools: equivalence checking, model-checkers for various temporal logics and mu-calculus and verification algorithms (enumerative verification, on-the-fly verification, symbolic verification using binary decision diagrams, etc). One interesting thing is that the tool box allows specification simulation (CAESAR tool). This simulation is based on C code for the data part and Petri nets for the concurrent and synchronization parts. Our current environment can not compete with it on traditional verification and model-checking, we rather propose some complementary tools and approaches. Another difference is that we do not provide a simulator but a real framework to execute software components automatically built from formal descriptions.
Related work about Java tools are the LTSA system and the JCSP library. The book [20] provides methods to link Finite State Process (FSP) and Java constructions. FSP is a recent process algebra originally proposed to design software architectures and based on CSP. It provides the classic construction to define processes and to compose them. FSP does not provide an interpreter of process algebras and verification tools are rather limited in performances and in functionality compared to CADP. JCSP is a pure Java class library designed by P. Welch and P. Austin and provides a base range of CSP primitives and a rich set of extensions, see [32] for more details. One main interest is that it conforms to the CSP model of communications and there is a long experience of tools and many practical case studies. The Java monitor threads model is rather easy to understand however it is more difficult to use it safely as soon as examples are not small ones. Thus JCSP is indeed a safer alternative than the built-in monitor model of Java threads. It uses explicit shared channels to synchronize processes. To relate our model with these existing approaches, we have no explicit channel and processes, not limited to two, synchronize on any service execution not only on read and write operations. Our prototype is not strictly based on CSP but may be viewed as an operational framework for a LOTOS like model of concurrency.

Considering Java source code generation, the constructive approach of Coglio and Green [12] is relevant. In this paper the authors think that a constructive approach, generating code from specifications, can be a valuable alternative to usual code verification. The usual way is to verify legacy code by a post-hoc method of proving certain properties, or possibly functional correctness. But the combinatorial difficulty of a post-hoc approach has generally prevented the community from being able to prove full functional correctness, i.e. that the program actually does what is intended. The goal of the authors is to provide a proof of functional correctness of the code with respect to its specification. The automated generation of such a proof, along with the code, is guided by the availability of the code generation/design process. A specification-first approach is made and use a user-friendly domain-specific notations which simplify code and proof generation. However, the input specification language is domain-specific and precludes certain features found in general-purpose languages (e.g. no recursion, no concurrency). The second point is that the target Java Card language is a subset of Java, and in our case concurrency is an essential feature.

Another related approach is [17] which consider translation of SDL specification into Java. The general principle is to map the SDL constructions to the Java ones. Since SDL is rather a complex language, the translation has to cope with many details not relevant here. The asynchronous signal in SDL are simply translated into one way void method and uses CORBA as communication platform. Basically our model provides a complex synchronization mechanism on top of which asynchronous communications are simple to implement. This experiment lacks of details about the generation of the Java code for the data part which is thus difficult to compare with our approach. JTN2 [3] is an extension of JTN an object-oriented, formal, visual notation for designing concurrent Java applications. JTN is devoted to modelling large scale Java applications and the generated code may be JavaBeans, EJB or JINI. A JTN2 component exposes a well defined interface and uses method call and
channel connectors to communicate with others. This existing proposal does not provide explicit tools for verification or code generation. However, the context is similar to our work, two main differences are that we explicit dynamic behaviour of component with STS and we target pure Java code.

3 The Component Model

Our current component model is a subset of the KADL model described in [10,26]. This model builds on the ADL ontology [22]: architectures or configurations made of components with ports, and connections between component ports. We call our structures components but precisely we are describing component types that can be instantiated. The features we are discussing in this paper are the definition and the use of symbolic transition systems to model software components both at the specification and programming levels. KADL provides a rich set of modal operators to synchronize components but in STSLib we restrict it to a small core subset. There are two categories of components: primitives and composites. A primitive component is described by the way of an STS and composite components are reusable compositions of components (i.e. architectures) represented as communication diagrams. We first give the formal definitions of STS, configuration graph and synchronous product.

3.1 Formal Definition of Symbolic Transition Systems

An STS is a dynamic behaviour coupled with a data type description. An Algebraic Data Type (ADT for short) is given for each STS, and transitions from this STS use the operations defined in the ADT. The operations semantics are described using algebraic axioms. Algebraic specifications abstract concrete implementation languages such as Java, C++, or Python (more details about these notations may be found in [5]). In [30] we present a partially automated approach which enables one to derive the operations, their profiles and parts of the axioms from the STS. We here consider an approach with simple rules which is easier to translate into programming source code. A signature (or static interface) \( \Sigma \) is a pair \((\mathcal{S}, \mathcal{F})\) where \( \mathcal{S} \) is a set of sorts (type names) and \( \mathcal{F} \) a set of function names equipped with profiles over these sorts. If \( R \) is a sort, then \( \Sigma_R \) denotes the subset of functions from \( \Sigma \) with result sort being \( R \). \( X \) is used to denote the set of all variables. From a signature \( \Sigma \) and from \( X \), one may obtain terms, denoted by \( T_{\Sigma,X} \). The set of closed terms (also called ground terms) is the subset of \( T_{\Sigma,X} \) without variables, denoted by \( T_{\Sigma} \). An algebraic specification is a pair \((\Sigma, Ax)\) where \( Ax \) is a set of axioms between terms of \( T_{\Sigma,X} \).

**Definition 3.1** [STS] An STS is a tuple \((D, (\Sigma, Ax), S, L, s^0, T)\) where: \((\Sigma, Ax)\) is an algebraic specification, \( D \) is a sort called sort of interest defined in \((\Sigma, Ax)\), \( S = \{s_i\} \) is a finite set of states, \( L = \{l_i\} \) is a finite set of event labels, \( s^0 \in S \) is the initial state, and \( T \subseteq S \times T_{\SigmaBoolean,X} \times Event \times T_{\SigmaD,X} \times S \) is a set of transitions.

Events denote atomic activities that occur in the components. Events are either: i) hidden (or internal) events: \( \tau \), ii) silent events: \( l \), with \( l \in L \), iii) emissions: \( l e \), with \( e \in \Sigma \), or iv) receipts: \( l ? x : R \) with \( x \in X \). Internal events denote internal
actions of the components which may have an effect on its behaviour yet without being observable from its context. Silent events are pure synchronizing events, while emissions and receptions naturally correspond, respectively, to requested and provided services of the components. STS transitions are tuples \((s, \mu, \epsilon, \delta, t)\) for which \(s\) is called the source state, \(t\) the target state, \(\mu\) the guard, \(\epsilon\) the event and \(\delta\) the action. Each action is denoted by a term with variables. In forthcoming figures, transitions will be labelled as follows: \([\mu] \epsilon / \delta\).

### 3.2 Configuration Graphs

The semantics of STS is formalized using configuration graphs. They are obtained applying jointly the unfolding of receptions and the reduction of ground terms to their normal forms, noted with the \(\downarrow\) operator.

**Definition 3.2** [Unfolding] The unfolding of an STS \((D, (\Sigma, Ax), S, L, s^0, T)\), in \(v^0 \in T_{\Sigma D}\), is the STS \((D, (\Sigma, Ax), S', L, (s^0, v^0)), T')\). The sets \(S' \subseteq S \times D\) and \(T'\) are inductively defined by: \((s^0, v^0) \in S'\) and for each \((s, v) \in S'\):

- if \((s, \mu, \tau, \delta, t) \in T\) and \(\mu(v) \vdash \text{true}\) then \(s' = (t, \delta(v) \downarrow) \in S'\) and \((s, v, \text{true}, \tau, \text{Sel}_D, s') \in T'\);
- if \((s, \mu, t, \delta, t) \in T\) and \(\mu(v) \vdash \text{true}\) then \(s' = (t, \delta(v) \downarrow) \in S'\) and \((s, v, \text{true}, t, \text{Sel}_D, s') \in T'\);
- if \((s, \mu, l\epsilon, \delta, t) \in T\) and \(\mu(v) \vdash \text{true}\) then \(s' = (t, \delta(v) \downarrow) \in S'\) and \((s, v, \text{true}, l\epsilon(v) \downarrow, \text{Sel}_D, s') \in T'\) and \((s, v, \text{true}, l\epsilon(v) \downarrow, \text{Sel}_D, s') \in T'\).

Pairs \((s, v)\) are configurations where \(s\) is the control state. Let \(d\) be an STS, its unfolding in a \(v^0\) term, \(G(d, v^0)\), is called a configuration graph. A configuration graph is a particular STS without receipt, where guards are all equal to \(\text{true}\), emission terms are in normal form and actions are do-nothing actions denoted by \(\text{Sel}_D\).

### 3.3 STS synchronous product

We extend the synchronous product originating from [4] to cope with STS. A synchronization vector is a tuple of ports, one for each component, which denotes a synchronization between the transitions associated to the port labels. The special port \(\tau\), is used when a component does not synchronize. Two components synchronize at some transition if their respective labels are synchronous (i.e. belong to the vector) and if the label offers are compatible. Offer compatibility follows simple rules: type equality and emission/reception matching.

**Definition 3.3** [Synchronous Product] The synchronous product (or product for short) of two STS \(d_i = (D_i, (\Sigma_i, Ax_i), S_i, L_i, s_i^0, T_i), i \in \{1, 2\}\), relatively to a synchronization vector \(V\), denoted by \(d_1 \otimes_V d_2\), is the STS \((D_1 \times D_2, (\Sigma_1, Ax_1) \times (\Sigma_2, Ax_2), S, L_1 \times L_2, s^0, T)\), where the sets \(S \subseteq S_1 \times S_2\) and \(T \subseteq S \times T_{\Sigma_{\text{Bool}, X}} \times (\text{Event}_1 \times \text{Event}_2) \times T_{\Sigma D,X} \times S\) are inductively defined by the rules:
As the reader may see it this product defines an STS with pairs of states and pairs of events. This synchronous product is extended to a n-ary product and to any depth.

3.4 Primitive and Composite

A component defines an interface which is a set of ports with a possible communication: emission, receipt or star and the associated types. We do not use the usual classification of required and provided ports since this is is only meaningful in a strict client-server context. In our case we can define more complex interactions (n-ary and symmetric rendezvous and star communication) and this terminology is not always relevant here. A primitive component is described by the way of an STS (the dynamic and the data part) as we defined it in Definition 3.1. A composite component is an assembly of sub-components (primitive or composite) and a set of communications. A simple meta-model is described in Figure 1.

A composite is also a kind of STS: i) its data type is the free product of the data type sub-components, and ii) its dynamic part is built from the synchronous product, as defined in Definition 3.3 and the synchronization vectors.

3.5 The Till Specification

To illustrate our model, we will consider the FM’99 cash-point service benchmark [31]. The system is composed of several tills which can access a central resource.
containing the detailed records of customers’ bank accounts. A till is used by inserting a card and typing in a Personal Identification Number (PIN) which is encoded by the till and compared with a code stored on the card. After successfully identifying themselves to the system, customers may make a cash withdrawal. Here we only present some parts: the till behaviour and the overall architecture. A comprehensive report is [27] which provides the full KADL specifications, verification results and a full LOTOS specification. A component interface defines the visible ports of the component. As an example the ports of the till are: `insertCard` to insert the user card, `giveCard` to eject the card, `pin` to enter the PIN, `getSum` to enter the desired cash amount, `cash` to get money, `add` to allow an operator to add money to the till available amount, `rec` to receive a message from the connection, and `send` to send a message. The dynamic behaviour of the till is depicted in Figure 2.

A transition such as `cash !sum(self) : Money / giveCash(self)` means that the till emits a sum of money and during this transition the `giveCash` operation updates the information of the till data type. Some axioms of the Till data type are given in Figure 3. We do not give the Till interface and the full ADT here by lack of space.

Transitions have the form `[guard] event / action`, where `guard` is a predicate on `self` and possibly received values which has to yield true for the transition to be fireable. `event` is a communication event, a communication port name together with reception variables denoted using `?` or emission terms denoted using `!`. `action` is the action of the ADT to be done when the transition is fired.

STS are (possibly non-deterministic) symbolic labelled finite transition systems which have appeared under different forms in the literature [18,9]. STS provide an expressive and abstract means to describe symbolically dynamic behaviours. The main interest with these transition systems is that (i) using received variables and guards in transitions, they control the system size and shape, and (ii) using an open
term in states \((\text{self})\), they define equivalence classes (one per state) and hence strongly relate the dynamic and the static (algebraic) representation of a data type.

\[
\begin{align*}
\text{VARIABLE} & \ a, \text{sum}: \text{Money} ; \ c: \text{Card} ; \ \text{code}: \text{PinNumber} ; \ \text{today}: \text{Date} ; \ \text{cpt} : \text{Natural} ; \ \text{self}: \text{Till} \\
\text{AXIOM} & \\
\text{# generator for till} \\
\text{newTill} & : \text{Money Card PinNumber Money Date Natural} \rightarrow \text{Till} \\
\text{# card selector} \\
\text{card} & : \text{Till} \rightarrow \text{Card} \\
\rightarrow & \text{card}(\text{newTill}(a,c,\text{code},\text{sum},\text{today},\text{cpt})) \rightarrow c \\
\text{# guard to check PIN code} \\
\text{fail} & : \text{Till} \rightarrow \text{boolean} \\
\rightarrow & \text{fail}(\text{self}) \rightarrow \\
& \text{and}\left(\text{not}(\text{equals}(\text{crypt}(\text{code}(\text{self})), \text{codecard}(\text{card}(\text{self})))), \text{supLarge}(\text{counter}(\text{self}), \text{three})\right) \\
\text{giveCash} & : \text{Till} \text{int} \rightarrow \text{Till} \\
\rightarrow & \text{giveCash}(\text{newTill}(a, c, \text{code}, \text{sum}, \text{today}, \text{cpt}), \text{sum1}) \rightarrow \\
& \text{newTill}(\text{sub}(a, \text{sum}), \text{updateDailyLimit}(\text{card}(\text{self}), \text{sum}, \text{today}), \text{code}, \text{sum}, \text{today}, \text{cpt})
\end{align*}
\]

Fig. 3. Some Axioms and Signatures of the Till Data Type.

3.6 Composite Component

The graphical definition of a composite component or a hierarchy of components is based on a composition diagrams enriched with communication notations. We reuse the UML class diagram notation with aggregation relations (white diamonds) to denote concurrent composition of sub-components into a composite. The UML aggregation notation has been chosen (in place of the composition notation for example) since the sub-components of a composite (and more generally the components of an architecture) usually have independent life-cycles. The composition diagram seems also better than the UML component structure notion since we consider a global behavioural dynamic part not one associated to each port as in UML or Java/A and we have specific semantics. We use the usual UML roles on aggregation relations to identify components and extend this notation using a range operator (noted \([i..j] \)). A component interface may be associated with a composition by exporting some events of its sub-components. Hence, composites are components too. Communication diagrams are composition diagrams complemented with glue rules (see Fig. 4). Communication notations are links between component ports, sometimes with an explicit modal operator above. For instance, the second link from Database to BankInterface labelled by XOR \([1..N] \) means that the \text{db.reply} port is connected to each \text{bi.get} port and these connections are mutually exclusive. That is a client-server communication from the database to the \(i^{th}\) interface, only one synchronization is possible at a time.

3.7 Specific Features of the Model

In this subsection, we briefly present other original features of our component model. To enrich KADL we introduced the star notation (\(*\)), its use is similar to the ! emission symbol. This mechanism allows us to define a computable quantification over transitions with emitted values. For instance, if we want to describe that the database can reply several tuples of informations we may write a transition as in the left part of Figure 5. This transition stands for a set of transitions which is different depending of the current state of the STS, see the right part. This is used
to code choices whose set of values can be computed by a function (\texttt{gener} in our example).

A second point is that communications are viewed as a first class concern in our model. Thus we have an explicit notation for communication links which is translated into synchronization vectors. A synchronization vector is a tuple of component events which are synchronous in the architecture. These vectors are used to compute the global behaviour of a composite component which gives an STS view to composites too. They are also used in the runtime support to define locks for synchronizing the thread (see Sect. 5). Another point is that synchronization is not restricted to port or event naming as in LTSA or LOTOS. We allow ports to communicate as soon as they are connected in the communication diagram and if they have compatible offers. Often design or formal component languages (except Java/A) do not fully decouple the behavioural description from the communications, for instance LOTOS, FSP or UML. To reuse components, they have to be synchronized into various environments and there is no reason for port or event naming to be a global knowledge. Thus to fight against name mismatch the two classic solutions are renaming (as in FSP) or component adapter. We think that a solution based on synchronization vectors is most general since it does not need code modification or any additional programmable entity. We do not explicitly provide a primitive way to define connectors, however, as explained in [26], a connector can
be defined as a normal component connecting other components.

A last and important question is the semantics of our model. We have two responses to this question. The first approach uses algebraic data types. The principle is to give an algebraic interpretation of the state machine and of the concurrency and communication mechanisms. Such semantics were studied in conjunction with the use of the PVS system, see [2, 30]. The main drawback of a theorem proving approach is the lack of automation, even if in the context of PVS useful automated strategies and model-checking algorithms exist. However, this approach is the only one which is able to cope with difficult problems, for example an unknown number of components.

The second semantic approach we develop in [28, 26] is based on the notion of configuration graph, i.e. a possible infinite state machine resulting from the unfolding of an STS. Figure 6 gives an overall picture showing how to relate classic model-checking and the STS specific verification mean. This diagram uses two transformations, the synchronous product and the unfolding of STS. To cope with concurrency and communication, we define the synchronous product to STS (see Definition 3.3). This extension of the synchronous product of automata keeps inside the result the structure of the composite. Thus we have not only states and transitions but composite states, composite transitions, composite events and so on. This is valuable to get an exact understanding about the events and the conditions occurring in a complex system. In Figure 6, the path (a) takes several STS and produces a configuration graph, that is the way we will illustrate in Section 4. The other way (b) is related to a more classical model-checking approach. In a first setting, let us assume that the global computation of the system is needed. Both ways a) and b) are equivalent from a theoretical expressive power but from a practical point of view time and space may be different. First, it is undecidable to know which way will be the most efficient in the general case. The space problem is the following: the final configuration result may have a manageable size however one of its component may be too wide or infinite. In this case classic model-checking will fail. Here the boundedness property may be critical to know if a configuration graph is finite and it may also provide a more or less precise measure of its size. Now if we consider on-the-fly model-checking, this technique checks a property and only builds the required part of the graph. Such technique improves efficiency but it is also possible to apply it in the context of STS, thus avoiding the global computation of the synchronous product and the complete unfolding. We have successfully experimented several examples which are possible to process with our approach but did not succeed with CADP or Spin, see [27, 26]. One example of this is our cash point example and the next section draws some conclusions on our experiments.

The STSLib specific verification approach may be viewed as a valuable, yet complementary technique, to other existing ones: classic model-checking, abstractions, infinite system approaches and the use of theorem provers.

4 Verifications with STSLib

The STSLib API is an implementation of the STS concept with the following functionalities. It supports the definition of the dynamic and the data type part
Fig. 6. Relating STS and Model-Checking of an STS. Such an STS allows guards, emissions and receipts (n-ary, one-way, and multiple), receipt on guards and the * notation. Architecture can be built from existing components and the synchronous product can be computed. This produces a structured STS allowing analysis of complex architectures. STSLib is able to compute the configuration graph associated with an STS. It provides a uniform definition of STS and configuration graphs thus the system may mix in various ways these notions. For instance, we can compute the configuration graph of an STS and synchronize it with another STS. Some simple verification means have been implemented: deadlock, state reachability and trace computation. Properties to check are expressed as Java predicates of the Java data part class. A boundedness checking was designed, it implements a general algorithm however currently restricted to STS with a set of integer counters as data types. Examples and uses of this prototype may be found in [21,28]. We already applied successfully our approach (boundedness and configuration graph) to several case studies: a simple flight reservation system, several variants of the bakery protocols, the slip protocol, several variants of a resource allocator, and a cash point service. We have developed this prototype in Python, about 4000 lines of code and efficiency was not our primary goal. We are currently rewriting it in Java 1.5 under Eclipse to get better performances, a wider diffusion and to add nice graphical interfaces.

In the sequel, we illustrate some experiments done on the cash-point example. These tests have been done to illustrate the use of the Python prototype. Sometimes it is not efficient but one known solution to get better results is to verify the property on the fly without computing the global product.

We compute the global STS with up to $N = 4$ tills (nearly an architecture with 20 components) and then we calculate the configuration graph for some set of values. The global synchronous product gives an abstract and concise view of the dynamic system. Such a view is useful to early check some errors in the dynamic behaviours especially related to the event synchronizations or communications. One may also check reachability of some configurations and to produce a graphic trace describing the events and the precise data value context. We verified that after a swallowCard (see Till Figure 2) the only outgoing transition is a clock transition which means that the system has a livelock. We verified that states with only one clock transition are exactly targets of a swallowCard event. But these cases are
only due to the fact that the till keeps the client card after three successive wrong PINs (there was a lack in the requirements). We also checked three additional properties: the PIN counter is equal to three after a `swallowCard`, the database amount and the till amount are always greater or equal than zero. We check these properties for \( N = 1, 2 \) and small values for the other variables.

Our second objective was to prove that the system ensures an exclusive access to any bank account (which is a safety property). In the following verifications, we used the fact that abstracting one component of a composition defines an abstraction of the product. We check the part corresponding to the database and bank interfaces and abstract the rest of the system. We define a component devoted to the simulation of the tills, the clients and the communication links. A bad situation would be two clients with the same account number withdrawing via two distinct interfaces. First we remark that the database contains the client accounts and the informations related to communications. We observe that the associated `Informations` type is equivalent to `List[Natural x Ident] x List[Natural x Money]`. From this, we apply the method defined in [28]. The principle is to keep the same system as above except the data type of the database which is redefined to only operate on `List[Natural x Ident]`. Using this decomposition, we prove that the property yields with \( \text{MAX}=3 \) and \( N=2 \), account number up to 10, \( \text{MAX}=2 \) and \( N=3 \), account number up to 4, and without a specific value for the max of accounts, see Table 1.

<table>
<thead>
<tr>
<th>N</th>
<th>MAX</th>
<th>account number</th>
<th>Configuration graph size</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>10</td>
<td>(4561, 24580)</td>
<td>405.</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>10</td>
<td>(17961, 145960)</td>
<td>10727</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>(2351, 9978)</td>
<td>120</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>(19461, 107292)</td>
<td>10419</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>(1895, 7290)</td>
<td>75.</td>
</tr>
</tbody>
</table>

Abstraction techniques such as [11,8,13] may be used in our context, but currently with a manual transformation. Some abstractions are simple to perform on our STS either on the dynamic part or the data part, a comprehensive analysis is under study. For example, we want to check that an existing card is either owned by the proper client or by its connected till or lost. This safety property was proved by abstracting the data of the system into the card identity which is also the client id. The global product has been done for \( N = 1, 2 \) and 3 without choosing effective numbers for the other parameters. The configuration graphs are bounded and the property is checked using an ad-hoc procedure.

A design of this case study has been done with LOTOS and the CADP toolbox. The LOTOS description of the processes appeared in [27]. The LOTOS description is closed to our STS description and an automated translation is even possible. However CADP needs to bound data types, we use really strict bounds and we cannot compute the BCG representation (internal LOTOS representation) even
Fig. 7. Implementation of the Till Primitive Component

with one client and one till.

5 The Interpreter Mechanism

Our long term objective is also to provide a Java compiler which is able to translate STS, both the state machine part and the data part, and architectures into Java code. Currently we defined an experimental interpreter, this section describes its principles.

5.1 Translation of a Primitive Component

We detail our hypotheses related to the description of primitive components in Java. A global picture of our intra-component implementation is depicted in Figure 7. It represents the different elements defining a primitive component. The representation of the finite state machine is described in a .sts files which contains the states, the transitions and some names. These names represent the guards, the events, the receipt variables, the emitters and the actions. The data part is a Java class implementing the formal data type part. The exact role of the class is to give a real implementation, with methods, of the names occurring in the state machine part. Thus both parts are glued thanks to a normalized Java interface. An emitter will be a pure function which computes the emitted value in a given state of the component. Similarly, a guard is a boolean function that implements a condition. So a primitive component results from the combination of a protocol and existing Java code, more precisely, a passive Java class implementing a specific Java interface. Each primitive component is implemented with an active object (thread in Java) in charge of both the STS protocol execution and the call of the passive object implementing the component data part.

5.2 The Data Part Class

Our STSLib library provides a Data class which is the inheritance root for the data parts of the primitive components. This class defines some general services: to create, to copy, to compare, and to view the representation of the data part. We have defined a LL(1) grammar (we use ANTLR [24]) for the STS dynamic part and STSLib is able to generate the interface and a skeleton of the data class from the STS dynamic description, then the user has to fill it. However, the code can be automatically generated from an explicit and formal description [25,29,17]. In the
sequel of this section, we sketch the translation we are experimenting on. One first thing to understand is the link between the verification approach with configuration graph and the real runtime support. A configuration graph memorizes the various states (control states and data values) thus a pure functional approach is required. However, a real program is usually imperative, thus we choose to transform our ADT into an imperative Java class. This class is equipped with some pure functional deep copy and equality which are used by the configuration graph computation to get the right semantics. The Figure 8 represents a situation for a simple transition but extensions are straightforward. This picture shows that new configurations in the verification world correspond to temporal changes in the Java runtime context.

The specification process is the following: the user generates, from the STS dynamic part, a skeleton of the ADT with the signatures, then it fills the axioms part and finally a code generator produces the full Java class and its interface. Our automatic translation relies on some hypotheses about the ADT part: i) there is only one generator called \texttt{newT}, where \texttt{T} stands for the STS name and the sort of interest of the ADT, ii) there is one selector associated to each argument of the generator, iii) conditional axioms are assumed to be oriented into left-to-right rewriting rules, and iv) left part of the conclusion has a simple form as in functional languages: either \( f(x_{i \leq i \leq n}) \) or \( f(\texttt{newT}(x_{0 \leq j \leq m}), x_{0 \leq j \leq m}) \), where \( x_i, x_j \) are variables. Another grammar has been defined for the axiom data part, it is a LL(2) one. A parser and an AST builder have been built, and on top of it we have implemented a Java code generator. To cope with the Java syntax and the imported data types, we use a dictionary of translation. This dictionary is filled with methods which translate functional calls of the ADT into the equivalent Java expression. A function \texttt{simpleTranslation} has the responsibility to walk through the AST of an expression and to build the corresponding Java string. It copes with the \texttt{this} argument, the dotted notation, variables and field selectors translation. See, for example, the \texttt{fail} expression in Figure 3 and its translation in Figure 9. The translation rules for an axiom are the following: i) equations in conditions are translated into terms equalities, ii) conditions give the test part of an \texttt{if} structure, iii) algebraic terms in equations are translated thanks to the \texttt{simpleTranslation} function. The delicate part is the translation of the axiom conclusion. The left term is traversed and a variable context is built to identify variable and field access occurrences. The translation of the right part depends on the fact that it is an observer, a constructor term or a generator call. In the two first cases, a translation with \texttt{simpleTranslation} is done. In the last case we have to identify the arguments to assign to the object.
fields, and to translate these arguments, see the giveCash example in Figure 9.

```java
// fail : [Till] -> boolean
public boolean fail() {
    return (! (crypter(this.code()) == this.card().code()))
            && (this.counter() => 3));
}

// giveCash : [Till] -> Till
public void giveCash() {
    this.amount = (this.amount() - this.sum());
    this.card = this.card().updateDailyLimit(this.sum(), this.date());
}
```

Fig. 9. Translation Examples

Our experimental generator relies on some hypotheses but the original thing was to generate full imperative Java code from data type description. The current hypotheses are a prefixed grammar and the lack of operator overloading. These features do not carry technical difficulties, they only complicate the grammar. Another restriction we are able to relax is the mono-generator constraint since we have already investigated this problem in [25, 29]. This previous work explored object-oriented class generation representing a data type with several generators. Currently our translation process preserves the axiom ordering. We have to investigate a less strict approach allowing more general left conclusion terms and a support to check axiom exclusivity (for instance using critical pairs).

5.3 Translation of a Composite

Our library provides a Composite Java class which defines a list of sub-components, the locations of the dynamic and data parts, the internal connections, with the modal operator and the external connections. From the internal connections and the modal operator, a list of synchronization vectors is computed. These vectors serve to build the locks as needed to manage the rendezvous mechanism. In addition to this, the composite defines a scope which may hide or export some ports (and the associated events) to outside. It has a similar role than the hiding operator of process algebras. A parser and a loader have been designed for the composite structure but yet restricted to simple architectures. This raises the issue of defining a global context class which memorizes the component (primitive or composite) already loaded in the current session.

5.4 The Synchronization Mechanism

Architecture or component assembly rely on primitive and composite components and a glue mechanism to synchronize them. We choose to implement the concurrent composition of STS based on the Java realization of the n-ary rendezvous we have at the formal specification level. The composition construction takes as input several components (STS or composites) and synchronization vectors that bind their events and configures STS runtime support in such a way that STS execution conforms to the semantic model. The direct consequences are that each STS has its own execution thread and that all STS have to be synchronized depending
on synchronization vectors. In this implementation, a primitive component corresponds, at runtime, to a unique thread and a composite component corresponds to a collection of interacting threads. The rendezvous mechanism is used to synchronize primitive components as well as composite components thus achieving a true compositional model at the runtime level. Here we will present an overview of the actual mechanism. More details about its Java implementation may be found in [15]. The principle is based on two synchronization barriers, the first one is used to permit all the participants to reach the rendezvous and the second one is used to leave the rendezvous. The barrier is implemented in a classic manner with the monitor facilities of Java. This solution uses a central arbiter but each synchronization vector has an associated lock object. These locks are responsible for checking the various conditions to enter in a rendezvous and own the entry and exit barriers. Even if the arbiter is a shared object, two independent synchronizations can be processed in the same time since we have independent monitor associated to each locks. This implementation provides an interpreter supporting rendezvous and allowing dynamic changes of STS, data parts or even components (obviously with some care in stopping and restarting components). In this runtime interpreter, reflexivity is used to glue protocols and data parts. In the future compiler version, protocols will do direct call to the data parts methods. However, a major problem will be the distribution of the shared objects and the limitation of remote communications. We partially addressed this problem with the introduction of conflict sets and locks and we will feature the balance between synchronous and asynchronous communications.

6 Conclusion and Future Work

The STSLib project aims at providing a powerfully way to design software components with protocol descriptions. One strong preoccupation is to narrow model for verification and programming code. Our environment currently proposes the STS notation, dynamic and algebraic parts, and composite description using communication diagrams. Our approach considers a specific verification mean which acts on the symbolic representation of the behaviour rather than on a finite state approximation. However we think that openness and interfacing with other verification tools is an essential aspect of such an environment support. Currently several tools have been implemented. The verifications allow to compute synchronous product and configuration graphs. We also study a true runtime support: generation of interfaces and Java class skeletons, generation of ADT signatures and full translation of a simple ADT into imperative Java code. Lastly our environment defines an implementation of the rendezvous mechanism which is able to synchronize components.

There are many things to do before to get a friendly usable environment. One important task on the verification side is to implement and define abstraction techniques. The second task is to elaborate a concrete syntax for hierarchical components and to implement a Java compiler based on our experimental interpreter. Another future perspective is to prove the translation process into Java code and the correctness of our rendezvous mechanism.
References


An Open System Operational Semantics for an Object-Oriented and Component-Based Language

Jasmin Christian Blanchette and Olaf Owe

Department of Informatics
University of Oslo
Oslo, Norway
{jasminb,olaf}@ifi.uio.no

Abstract

Object orientation and component-based development have both proven useful for the elaboration of open distributed systems. These paradigms are offered by the Creol language. Creol objects are concurrent, each with its own virtual processor and internal process control, and communicate using asynchronous (non-blocking) method calls. This provides the efficiency of message passing systems, while keeping the structuring benefits of methods and object-oriented programming. Conditional processor release points provide a high-level synchronization mechanism that allows us to combine active and reactive behavior. A Creol component can be a single (concurrent) object or a collection of objects, together with a number of interfaces, and counterfaces, defining provided and required interaction and semantical behavior. Creol’s semantics is defined formally using operational semantics and Hoare logic. An operational semantics lets us simulate an entire system, where all components are known in advance; in contrast, Hoare logic, together with class invariants and communication histories, lets us reason locally about a method body, without needing access to the implementations of the other classes. To bridge the gap between these two semantics, we introduce a history-based operational semantics for open systems. This new semantics can be used as an intermediate step for proving that Creol’s Hoare logic is sound and complete with respect to the language’s operational semantics. The approach can easily be adapted to other component-based languages where communication is done by message passing or by method interaction.

Keywords: Operational semantics, open distributed systems, communication histories, object orientation.

1 Introduction

Component-based system design directly supports the role of autonomous objects in distributed architectures. This useful paradigm can be combined with the structuring mechanisms of object orientation, as done within the Creol framework [10,11,12,13,14]. Creol objects are concurrent, each with its own virtual processor and internal process control, and communicate using asynchronous (non-blocking) method calls. This provides the efficiency of message passing systems, while keeping the structuring benefits of methods and object-oriented programming, notably late

1 This work is in the context of the EU project IST-33826 CREDO (http://credo.cwi.nl/).

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
binding and inheritance. A Creol object, together with its interfaces, constitutes an autonomous unit that can act and react in a distributed setting. More complex components are formed by combining several concurrent objects together and defining interfaces that describe and control the component’s visible behavior. Creol’s notion of cointerface allows specification of required and provided interfaces.

The goal of Creol is to develop a formal framework for reasoning about dynamic and reflective modifications in open distributed systems, where objects may be dispersed geographically, ensuring reliability and correctness of the overall system. The Creol language is high-level, imperative, and object-oriented. The language’s semantics is defined formally using a small-step operational semantics expressed in rewriting logic [11]. This semantics forms the core of the Creol interpreter, which is written in Maude [7]. Using Maude’s extensible rewrite strategies, its search capabilities, and its model checker, we can test Creol programs in various ways [12,13].

The Creol interpreter allows us to simulate a closed distributed system, where all the initial components are known in advance. On the other hand, it does not let us execute a component without providing and implementing an environment with which it can interact. At the reasoning level, this limitation is addressed by the Hoare logic developed by Dovland et al. [10]. The Hoare logic allows us to prove that an invariant holds for a given class. A system-wide invariant can be constructed from the class invariants using a compositional rule. The invariants may refer to a mythical communication history that records the object creations and method calls that have taken place in the system [8].

In this paper, we introduce an “open system” operational semantics that incorporates the class invariants and the communication history that characterize Creol’s Hoare logic. One benefit of this approach is that it moves these techniques from the syntax-driven world of Hoare logic to the more fundamental semantics level. We then have the full power of mathematics and of formal tools like Maude at our disposal to analyze individual Creol classes.

Once we have an open system operational semantics, we can use it as a stepping stone towards the development of a Hoare logic. For Creol, where a Hoare logic already exists, it could be used to prove that the Hoare logic is sound and complete with respect to Creol’s reference interpreter. The proof would proceed in two steps: (1) Prove that the closed system semantics and the open system semantics are equivalent, modulo the way they represent the environment. (2) Prove that the Hoare logic is sound and complete with respect to the open system semantics.

We focus on the class aspect of the Creol language, and limit ourselves to the basic communication and synchronization model of Creol, omitting the notions of interface, inheritance, self-reentrance, and dynamic update, as well as typing and specification. The approach can then easily be adapted to other languages where communication is done by messages passing or method interaction. We assume throughout that Creol programs are syntactically correct and well-typed.

The rest of this paper is organized as follows: Section 2 reviews the syntax of the main Creol statements. Section 3 presents a closed system operational semantics for the core language. Section 4 introduces the open system semantics and connects it to the closed semantics and to Hoare logic. Section 5 considers related work. Finally, Section 6 summarizes the paper.
2 The Creol Language

Creol is a strongly-typed object-oriented language that supports interfaces, inheritance, and polymorphism. Classes are the fundamental structuring unit, and all interaction between objects occurs through method calls. Each object executes on its own virtual processor, leading to increased parallelism in a distributed system. Classes are equipped with class parameters, as in Simula.

Objects are uniquely identified and communicate using asynchronous method calls. When an object \( A \) calls a method \( m \) of an object \( B \), it first sends an invocation message to \( B \) along with arguments. Method \( m \) executes on \( B \)'s processor and sends a reply to \( A \) once it has finished executing, with return values. Object \( A \) may continue executing while waiting for \( B \)'s reply. Object identities may be passed around, and thanks to Creol’s interface concept, method calls are type-safe [14]. In an object-oriented system, asynchronous method calls arguably offer a more natural interaction model than shared variables and message passing, while avoiding the delays associated with synchronous method calls [11].

The other main distinguishing feature of Creol is its reliance on explicit processor release points. Since there is only one processor per object, at most one method \( m \) may execute at a given time for a given object; any other method invocations must wait until \( m \) finishes or releases the processor using the \texttt{await} statement. This ensures that while a method is active, no other processes can access the object’s attributes, leading to a programming and reasoning style reminiscent of monitors [5].

The syntax of the Creol statements relies on a few basic syntactic entities. The set \texttt{Ident}, with typical elements \( c, l, m, x, y \), consists of alphanumeric tokens that start with a letter, excluding keywords. The set \texttt{BExp}, with typical element \( B \), consists of Boolean expressions such as \( i \geq n \). The set \texttt{Exp}, with typical element \( e \), consists of expressions of any type, including Boolean expressions, arithmetic expressions, and object references. The keyword \texttt{self} refers to the current object, and the implicit parameter \texttt{caller} identifies the caller of a method, allowing type-safe call-backs. The set \texttt{Guard}, with typical element \( g \), includes \texttt{BExp} and otherwise contains the reply guard \( ? \), the release guard \texttt{wait}, and the conjunction \( g_1 \& g_2 \).

The set \texttt{Stmt} of statements, with typical element \( S \), contains these constructs:

\[
\begin{align*}
  x & := e \quad \text{assignment} \\
  x & := \texttt{new} \ c(e_1, \ldots, e_n) \quad \text{object creation} \\
  l!x.m(e_1, \ldots, e_n) & \quad \text{asynchronous invocation} \\
  l?(y_1, \ldots, y_p) & \quad \text{asynchronous reply} \\
  \texttt{await} \ & g \quad \text{conditional wait} \\
  \texttt{if} \ B \ & \texttt{then} \ S_1 \ \texttt{else} \ S_2 \ \texttt{fi} \quad \text{conditional statement} \\
  \texttt{return} \ & e_1, \ldots, e_n \quad \text{return statement} \\
  S_1; S_2 & \quad \text{sequential composition}
\end{align*}
\]

The object creation statement creates a new instance of class \( c \). The expressions \( e_1, \ldots, e_n \) are assigned to class parameters. If the class has a parameterless \texttt{run} method, this method executes immediately.

Asynchronous method calls consist of an invocation and a reply. The invocation can be seen as a message from the caller to the called method, with arguments...
corresponding to the method’s input parameters. The reply is a message from the called method, containing the return values. The label \( l \) uniquely identifies the method call. For convenience, Creol also provides the classic synchronous (blocking) method call \( x.m(e_1, \ldots, e_n; y_1, \ldots, y_p) \) as an abbreviation for \( t!x.m(e_1, \ldots, e_n); t?(y_1, \ldots, y_p) \), where \( t \) is a fresh label name. Here each \( e_i \) acts as an actual input parameter to the method, and each \( y_j \) acts as an actual output parameter.

The statement `await` \( g \) releases the processor if the guard \( g \) evaluates to false and reacquires it at some later time when \( g \) is true. The guard \( l? \) evaluates to true if and only if a reply for the asynchronous call identified by \( l \) has arrived. The `wait` guard evaluates to false the first time it is encountered, resulting in a processor release, and evaluates to true from then on.

The conditional statement, the return statement, sequential composition, and assignment behave essentially like their Java equivalents.

**Example 2.1** The following code initiates two asynchronous calls, releases the processor while waiting for the replies, and retrieves the return values:

```javascript
var result1 : int, result2 : int;

l1!server.request(); l2!server.request();

await l1? & l2?;

l1?(result1); l2?(result2)
```

Without the `await` statement, the program would block on the reply statements \( l1?(result1) \) and \( l2?(result2) \) until the method invocations have terminated.

### 3 An Operational Semantics for Creol

The operational semantics of Creol is defined using rewriting logic (RL) [11], which can be seen as a generalization of structural operational semantics [16]. A rewrite theory is a triple \( \mathcal{R} = (\Sigma, E, R) \), where the signature \( \Sigma \) defines the function symbols of the language, \( E \) defines equations between terms, and \( R \) is a set of rewrite rules. When modeling computational systems, we represent a state configuration by a multiset of terms of given types. These types are specified algebraically in the equational logic \( (\Sigma, E) \), the functional sublanguage of RL.

The dynamic behavior of a system is expressed by rewrite rules, which describe how a part of a configuration can evolve in one transition step. A rule \( p \rightarrow q \ [\text{if} \ c] \) allows an instance of the pattern \( p \) to evolve into the corresponding instance of the pattern \( q \) if the (optional) side condition \( c \) is met. Rewrite rules are applied modulo \( E \) to complete terms or to subterms. Rules can be applied simultaneously on non-overlapping subterms; as a result, RL is implicitly concurrent.

The operational semantics for Creol consists of 11 rewrite rules that model concurrent execution, object creation, and inter-object communication. It also relies on equations to perform auxiliary tasks. The rewrite rules have the form

\[
\text{subconfiguration}_1 \rightarrow \text{subconfiguration}_2 \ [\text{if} \ \text{condition}]
\]

where \( \text{subconfiguration}_1 \) is a subset of the current configuration. In our setting, a configuration is a multiset of Creol objects, Creol classes, and messages reflecting either method invocations or replies. Typically, each subconfiguration consists of a
single object, and possibly a message, reflecting that Creol objects are autonomous.

In a system configuration, Creol objects are represented by terms of the form

\[ o : c \mid \text{Pr: } S, \text{LVar: } \beta, \text{Att: } \alpha, \text{PrQ: } P, \text{MsgQ: } Q, \text{LabCnt: } k \],

where \( o \in OId \) is a unique object identity, \( c \) is the object’s class, \( S \) is the active process’s code, \( \beta \in \text{State} \) is the active process’s local variables, \( \alpha \in \text{State} \) is the current state of the object’s attributes, \( P \in \mathcal{P}(\text{State} \times \text{Stmt}) \) is a queue of suspended processes, \( Q \in \mathcal{P}(\text{Msg}) \) is the incoming message queue, and \( k \in \mathbb{N} \) is a counter used to generate unique label values for asynchronous calls.

The set \( \text{State} \), with typical elements \( \sigma, \alpha, \beta \), consists of mappings from variables to values. For example, \([x \mapsto 1][y \mapsto 2]\) denotes the state in which \( x = 1 \) and \( y = 2 \). The concatenation \( \alpha \beta \) of two states \( \alpha \) and \( \beta \) gives precedence to \( \beta \) for variables defined by both. The function \( \{ e \}_\sigma \) returns the value of an expression list \( \bar{e} \) in a state \( \sigma \). The notation \( \bar{x} \) stands for the comma-separated list \( x_1, \ldots, x_n \). The empty list is written \( \epsilon \). The set \( \text{Value} \), with typical elements \( v, w \), includes the Boolean constants \texttt{true} and \texttt{false}, numeric constants, and object identities.

Creol classes are represented by terms of the form

\[ c : \text{Class} \mid \text{Param: } \bar{x}, \text{Att: } \alpha, \text{Mtd: } M, \text{ObjCnt: } n \],

where \( c \) is the class name, \( \bar{x} \) is the list of class parameters, \( \alpha \) is the list of class attributes with initial values, \( M \in \mathcal{P}(\text{Mtd}) \) is a set of methods, and \( n \in \mathbb{N} \) is a counter used to generate unique object identities.

Creol objects interact by exchanging messages. Invocation messages have the form \texttt{Invoke}(\( o, k, m, \bar{v} \)), where \( o \) is the calling object, \( k \) is the sequence number (label value) associated with the method call, \( m \) is the called method, and \( \bar{v} \) is a list of input arguments to \( m \). Reply messages have the form \texttt{Reply}(\( k, \bar{v} \)), where \( k \) is the sequence number for the method call and \( \bar{v} \) is a list of return values. When messages are passed around, the receiver object \( o' \) is specified by appending to \( o \).

Rewrite Rule R1 (Assignment)

\[ o : c \mid \text{Pr: } x := e; S, \text{LVar: } \beta, \text{Att: } \alpha \]

\[ \text{if } x \in \beta \text{ then } o : c \mid \text{Pr: } S, \text{LVar: } \beta[x \mapsto \{ e \}_\alpha \beta], \text{Att: } \alpha \]

\[ \text{else } o : c \mid \text{Pr: } S, \text{LVar: } \beta, \text{Att: } \alpha[x \mapsto \{ e \}_\alpha \beta] \] \text{fi}

The assignment statement evaluates the expression \( e \) in the compound state \( \alpha \beta \) and stores that value in \( x \). If \( x \) is a local variable, we update the local state \( \beta \); otherwise, we update the object state \( \alpha \). In the style of Full Maude [7], the fields that are not used by a rule are omitted in the rule.

Rewrite Rule R2 (If Statement)

\[ o : c \mid \text{Pr: if } B \text{ then } S_1 \text{ else } S_2 \text{ fi; } S, \text{LVar: } \beta, \text{Att: } \alpha \]

\[ \text{if } \{ B \}_\alpha \beta \text{ then } o : c \mid \text{Pr: } S_1; S, \text{LVar: } \beta, \text{Att: } \alpha \]

\[ \text{else } o : c \mid \text{Pr: } S_2; S, \text{LVar: } \beta, \text{Att: } \alpha \] \text{fi}

If \( B \) evaluates to \texttt{true}, the Creol if statement expands to its \texttt{then} branch; otherwise, it expands to its \texttt{else} branch. Notice that the first \texttt{if} construct in the rule above is a Creol statement, while the second \texttt{if} is a conditional expression in RL.
Rewrite Rule R3 (Guard Crossing)

\[ \langle o ; c \mid Pr: \text{await } g; S, \text{LVar: } \beta, \text{Att: } \alpha, \text{MsgQ: } Q \rangle \]
\[ \rightarrow \]
\[ \langle o ; c \mid Pr: S, \text{LVar: } \beta, \text{Att: } \alpha, \text{MsgQ: } Q \rangle \]
if \( \text{enabled}(g, \alpha \beta, Q) \)

An \text{await} statement whose guard evaluates to true is simply skipped. The \text{enabled} predicate is defined recursively using equations:

\[ \text{enabled}(B, \sigma, Q) \triangleq \{ B \}_\sigma \]
\[ \text{enabled}(\text{wait}, \sigma, Q) \triangleq \text{false} \]
\[ \text{enabled}(?!, \sigma, \emptyset) \triangleq \text{false} \]
\[ \text{enabled}(?!, \sigma, Q \cup \{ \text{Invoke}(...) \}) \triangleq \text{enabled}(?!, \sigma, Q) \]
\[ \text{enabled}(?!, \sigma, Q \cup \{ \text{Reply}(k, \bar{v}) \}) \triangleq k = \{ l \}_\sigma \lor \text{enabled}(?!, \sigma, Q) \]
\[ \text{enabled}(g_1 \& g_2, \sigma, Q) \triangleq \text{enabled}(g_1, \sigma, Q) \land \text{enabled}(g_2, \sigma, Q) \]

Rewrite Rule R4 (Process Suspension)

\[ \langle o ; c \mid Pr: \text{await } g; S, \text{LVar: } \beta, \text{Att: } \alpha, \text{PrQ: } P, \text{MsgQ: } Q \rangle \]
\[ \rightarrow \]
\[ \langle o ; c \mid Pr: \epsilon, \text{LVar: } \emptyset, \text{Att: } \alpha, \text{PrQ: } P \cup \{ \langle \text{await } \text{clearWait}(g); S, \beta \rangle \}, \text{MsgQ: } Q \rangle \]
if \( \neg \text{enabled}(g, \alpha \beta, Q) \)

If the next statement to execute is an \text{await} statement whose guard is not enabled, the active process is put on the process queue, together with its local variables. The \text{clearWait} auxiliary function replaces any occurrence of \text{wait} in the guard with \text{true}, so that a process that was suspended because of \text{wait} may become active again.

Rewrite Rule R5 (Process Activation)

\[ \langle o ; c \mid Pr: \epsilon, \text{LVar: } \beta, \text{Att: } \alpha, \text{PrQ: } \{ \{ S', \beta' \} \} \cup P, \text{MsgQ: } Q \rangle \]
\[ \rightarrow \]
\[ \langle o ; c \mid Pr: S', \text{LVar: } \beta', \text{Att: } \alpha, \text{PrQ: } P, \text{MsgQ: } Q \rangle \]
if \( \text{ready}(S', \alpha \beta', Q) \)

While a process is suspended, other processes that are ready may be activated. A reply statement is \text{ready} only if the reply message has arrived, and an \text{await} statement only if the guard is enabled, while other statements are always ready. The list \( S_1; \ldots; S_n \) is ready whenever \( S_1 \) is ready. Maude’s facilities for associative, commutative, and identity (ACI) matching allow \( \{ \{ S', \beta' \} \} \) to match any process in \( \text{PrQ} \).

Rewrite Rule R6 (Object Creation)

\[ \langle o ; c \mid Pr: y := \text{new } c'(...); S, \text{LVar: } \beta, \text{Att: } \alpha \rangle \]
\[ \langle c' \mid \text{Class } | \text{Param: } \bar{x}, \text{Att: } \alpha', \text{ObjCnt: } n \rangle \]
\[ \rightarrow \]
\[ \langle o ; c \mid Pr: y := c'\#n; S, \text{LVar: } \beta, \text{Att: } \alpha \rangle \]
\[ \langle c' \mid \text{Class } | \text{Param: } \bar{x}, \text{Att: } \alpha', \text{ObjCnt: } n + 1 \rangle \]
\[ \langle c'\#n : c' \mid Pr: \text{self.run()}, \text{LVar: } \emptyset, \text{Att: } \alpha'[\bar{x} \mapsto \{ \bar{e} \}_\alpha][\text{self} \mapsto c'\#n], \text{PrQ: } \emptyset, \text{MsgQ: } \emptyset, \text{LabCnt: } 0 \rangle \]

A \text{new} statement creates an instance of a given class. The new object’s identity is \( c'\#n \), where \( c' \) is the class name and \( n \) a sequence number that identifies this object among \( c' \) instances. The new object is set up with the class parameters
and attributes of class $c'$. The Pr field is initialized with a synchronous call to run to launch the object’s active behavior. In the parent object, creating an object is viewed as an assignment of $c'\#n$ to a variable. In the instantiated class, the object counter is incremented to ensure that object identities remain unique.

**Rewrite Rule R7 (Asynchronous Invocation)**

\[
\langle o : c \mid Pr: \text{run} \rangle 
\xrightarrow{\text{Asynchronous Invocation}}
\langle o : c \mid Pr: l \rightarrow \text{run} \rangle
\]

Asynchronous method calls lead to the creation of an invocation message that is sent to the called object. Each method call originated by a given object is identified by a unique sequence number $k$. This number is assigned to the local variable $l$, which corresponds to the label $l$. A call is uniquely identified by the pair $(o, k)$.

**Rewrite Rule R8 (Transport of Message)**

\[
\langle o : c \mid \text{MsgQ: } Q \rangle
\]

\[
\xrightarrow{\mu \text{ to } o}
\langle o : c \mid \text{MsgQ: } Q \cup \{\mu\} \rangle
\]

At some unspecified point after an invocation or reply message $\mu$ has been sent, the recipient receives it. Rewrite Rules R7 and R8 allow message overtaking—messages might arrive in a different order than they were sent. Again, ACI matching applies.

**Rewrite Rule R9 (Method Binding)**

\[
\langle o : c \mid \text{MsgQ: } \{\text{Invoke}(o', k, m, \bar{e})\} \cup Q \rangle
\]

\[
\xrightarrow{\langle o' : \text{Class} \mid \text{Mtd: } M \rangle}
\langle o : c \mid \text{MsgQ: } \{\text{bind}(o', k, m, \bar{e}, M)\} \cup Q \rangle
\]

A pending invocation message gives rise to a new pending process. The \text{bind} function fetches method $m$ from the method set $M$ and returns a $\langle S, \beta \rangle$ pair storing the code and initial state of the process. The rule does not consider base classes; method binding with multiple inheritance in Creol is treated in Johnsen et al. [14].

**Rewrite Rule R10 (Method Return)**

\[
\langle o : c \mid Pr: \text{return } \bar{e} \rangle
\]

\[
\xrightarrow{\langle o : c \mid Pr: \text{epsilon} \rangle}
\langle o : c \mid \text{Reply}(\{\text{label}\}_\beta, \{\bar{e}\}_\beta) \text{ to } \{\text{caller}\}_\beta \rangle
\]

The \text{return} statement sends a reply message to the caller along with the values of the output parameters. Reply messages are eventually received by the calling object and put into its incoming message queue using Rewrite Rule R8.

**Rewrite Rule R11 (Asynchronous Reply)**

\[
\langle o : c \mid Pr: l?(\bar{y}) \rangle
\]

\[
\xrightarrow{\langle o : c \mid Pr: \text{epsilon} \rangle}
\langle o : c \mid \text{Reply}(k, \bar{v}) \rangle
\]

\[
\langle o : c \mid \text{Pr: } \bar{y} := \bar{v} \rangle
\]

Replace the output parameters with their values.
A statement $t?(\bar{y})$ may proceed only if the corresponding reply message has arrived, which we can find out by looking for a reply message numbered $k$, where $k$ is $t$’s value. The output parameter values stored in the reply are assigned to $\bar{y}$.

4 An Alternative Semantics for Open Systems

While the operational semantics presented in the previous section correctly captures the behavior of a closed system, it doesn’t directly cater for open systems, in which objects don’t have access to each other’s implementations. This means that we have no satisfactory way to simulate the activity of a single process taken in isolation once we abstract away the environment with which it communicates (the other processes executing in the same object and the other objects in the system). It also means that there’s no direct way to derive a Hoare logic from the operational semantics.

4.1 Definition of the Open System Semantics

In this section, we will define an alternative version of Creol’s operational semantics that focuses on the execution of a single process, mimicking a Hoare logic. The new “open system” operational semantics uses a communication history to abstract away the environment. This semantics reuses Rewrite Rules R1–R3 and R11 from the previous section, because these rules involve no interaction between objects or between processes within an object. Rewrite Rules R4–R10 are replaced with a new set of rules that operate on the history. The table below compares the closed system semantics of Section 3 with the open system semantics introduced here.

<table>
<thead>
<tr>
<th>Process Suspension</th>
<th>Closed System</th>
<th>Open System</th>
</tr>
</thead>
<tbody>
<tr>
<td>R4</td>
<td>R4'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process Activation</th>
<th>Closed System</th>
<th>Open System</th>
</tr>
</thead>
<tbody>
<tr>
<td>R5</td>
<td>R5'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object Creation</th>
<th>Closed System</th>
<th>Open System</th>
</tr>
</thead>
<tbody>
<tr>
<td>R6</td>
<td>R6'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Asynchronous Invocation</th>
<th>Closed System</th>
<th>Open System</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7</td>
<td>R7'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transport of Message</th>
<th>Closed System</th>
<th>Open System</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
<td>R8'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Binding</th>
<th>Closed System</th>
<th>Open System</th>
</tr>
</thead>
<tbody>
<tr>
<td>R9</td>
<td>R9'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Return</th>
<th>Closed System</th>
<th>Open System</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10</td>
<td>R10'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment Activity</th>
<th>Closed System</th>
<th>Open System</th>
</tr>
</thead>
<tbody>
<tr>
<td>R12</td>
<td>R12'</td>
<td></td>
</tr>
</tbody>
</table>

From Hoare logic we borrow the concept of a communication history [8]. The communication history records the creation of objects and the messages that are exchanged between objects in a distributed system. More formally, a history is a finite sequence of communication events:

- $[o\rightarrow o'.\text{new } c(\bar{v})]$ object creation
- $[o\rightarrow o'.m(\bar{v})]^k$ asynchronous invocation
- $[o\rightarrow o'.m(\bar{v}; \bar{w})]^k$ asynchronous reply

For invocation events, $\bar{v}$ stores the values passed to the method; for reply events, $\bar{v}$ stores the return values. For both types of event, $k$ is the sequence number of the method call. For object creation events, $\bar{v}$ stores the actual class parameters. The history represents a snapshot of the system’s execution at a given point and is
therefore finite. When designing or analyzing a complex system, we often want to
to know the possible histories for that system, to deduce safety properties about it [10].

In the new semantics, Creol objects have the form

\[ \langle o : c \mid \text{Pr: } S, \text{LVar: } \beta, \text{Att: } \alpha, \text{MsgQ: } Q, \text{LabCnt: } k \rangle. \]

Since we concentrate on one process’s execution, we now omit the PrQ field. On
the other hand, the object’s attribute set now includes a distinguished \( \mathcal{H} \) attribute
that stores the system’s communication history. The history could also have been
stored in a separate field, or as a separate object, but making it an attribute will
simplify the definition of Hoare logic formulas. The MsgQ field is redundant now
that we record the history; we keep it because Rewrite Rules R3 and R11 rely on it.
Also, some of the rewrite rules will refer to the class invariant \( I_c \), which is expected
to hold at startup and whenever the processor is released. This invariant is derived
from the semantic specification supplied by the programmer in the class declaration
for \( c \) and in the interface declarations for the interfaces implemented by \( c \).

**Rewrite Rule R6′ (Object Creation)**

\[ \langle o : c \mid \text{Pr: } y := \text{new } c'(\bar{e}); S, \text{LVar: } \beta, \text{Att: } \alpha \rangle \]

\[ \langle o : c \mid \text{Pr: } y := o'; S, \text{LVar: } \beta, \text{Att: } \alpha[\mathcal{H} \mapsto \{\mathcal{H}\}_\alpha \leftarrow \{\alpha \rightarrow \alpha'.\text{new } c'([\bar{e}]_{\alpha\beta})\}] \]

\[ \text{if } o' \notin \text{objectIds}(\{\mathcal{H}\}_\alpha) \]

With the open system semantics, an object creation statement allocates a fresh
object identity \( o' \) and extends the history \( \mathcal{H} \) with an object creation event. The
new object is now part of the implicit environment embodied by \( \mathcal{H} \).

**Rewrite Rule R4′ (Process Suspension and Reactivation)**

\[ \langle o : c \mid \text{Pr: } \text{await } g; S, \text{LVar: } \beta, \text{Att: } \alpha, \text{MsgQ: } Q, \text{LabCnt: } k \rangle \]

\[ \langle o : c \mid \text{Pr: } S, \text{LVar: } \beta, \text{Att: } \alpha', \text{MsgQ: } \text{replies}(\{\mathcal{H}\}_\alpha', o), \]

\[ \text{LabCnt: } \text{nextLabel}(\{\mathcal{H}\}_\alpha', o) \]

\[ \text{if } \neg enabled(g, \alpha\beta, Q) \land \text{release}(I_c, \alpha, \alpha', \beta) \]

\[ \land \text{enabled}([\text{clearWait}(g), \alpha' \beta, \text{replies}(\{\mathcal{H}\}_\alpha', o)) \]

If the next statement is \( \text{await } g \) and the guard \( g \) is not enabled, the process is
suspended and wakes up in a different state in which the guard is enabled. The
class attributes, including the history \( \mathcal{H} \), might have changed in the meantime; this
is modeled by replacing \( \alpha \) with \( \alpha' \). In addition, the MsgQ and LabCnt fields are
updated to reflect the new history.

The function \( \text{replies}(h, o) \) returns a set of pending reply messages corresponding
to the pending replies encoded in the history \( h \). The constraint \( \text{release}(I_c, \alpha, \alpha', \beta) \)
restricts the values that the attributes \( \alpha' \) may take. It is defined as follows:

\[ \text{release}(I_c, \alpha, \alpha', \beta) \triangleq \{\mathcal{H}\}_\alpha \preceq \{\mathcal{H}\}_\alpha' \land \text{wf}(\{\mathcal{H}\}_\alpha') \land \{I_c\}_\alpha \Rightarrow \{I\}_\alpha' \]

\[ \land \text{pending}(\{\mathcal{H}\}_\alpha', \{\text{caller}\}_\beta, \{\text{self}\}_\alpha, \{\text{label}\}_\beta) \]

Informally, the new history \( \{\mathcal{H}\}_\alpha' \) must be an extension of the original history \( \{\mathcal{H}\}_\alpha \),
it must be well-formed, the class invariant \( I_c \) should still hold if it held before the
release, and the call that released the processor should still be pending after the
processor release. The well-formedness predicate is defined below:
A communication history is well-formed if new objects have unique identifiers and if method invocations and replies match. We also require that a pair \((o,k)\) uniquely identifies a method call originating from an object \(o\). Well-formedness expresses program-independent properties of the history.

**Rewrite Rule R7’ (Asynchronous Invocation)**

\[
\langle o : c \mid \text{Pr: } !x.m(\bar{e}); S, \text{LVar: } \beta, \text{Att: } \alpha, \text{LabCnt: } k \rangle \rightarrow \langle o : c \mid \text{Pr: } S, \text{LVar: } \beta[l \leftarrow k], \text{Att: } \alpha[H \mapsto \{\mathcal{H}\}_\alpha \rightarrow [o \rightarrow \{x\}_{\alpha \beta}.m(\bar{e})\}_{\alpha \beta}] , \text{LabCnt: } k + 1 \rangle
\]

Asynchronous method calls lead to an extension of the history with a new invocation event. Similarly, returning from a method extends the history with a reply event:

**Rewrite Rule R10’ (Method Return)**

\[
\langle o : c \mid \text{Pr: } \text{return } \bar{e}; S, \text{LVar: } \beta, \text{Att: } \alpha \rangle \rightarrow \langle o : c \mid \text{Pr: } \epsilon, \text{LVar: } \beta, \text{Att: } \alpha[H \mapsto \{\mathcal{H}\}_\alpha \rightarrow \text{replyEvent}(\{\mathcal{H}\}_\alpha, o, \{\text{caller}\}_\beta, \{\text{label}\}_\beta, \{\bar{e}\}_\alpha) \rangle
\]

The auxiliary function \(\text{replyEvent}\) calculates the reply event by inspecting the history. If \([o' \rightarrow o.m(\bar{v})]^k \in h\), then \(\text{replyEvent}(h, o, o', k, \bar{w})\) is \([o' \rightarrow o.m(\bar{v}; \bar{w})]^k\).

**Rewrite Rule R12’ (Environment Activity)**

\[
\langle o : c \mid \text{Pr: } S, \text{Att: } \alpha, \text{MsgQ: } Q \rangle \rightarrow \langle o : c \mid \text{Pr: } S, \text{Att: } \alpha[H \mapsto h], \text{MsgQ: } \text{replies}(h, o) \rangle
\]

\[
\text{if } \text{interleave}(I_e, o, \alpha, h)
\]

Rewrite Rule R12’ lets us extend the history in a nondeterministic way with events originating from the environment at any point during the execution of a process. The new history \(h\) must abide by the following rules, expressed by the \(\text{interleave}\) predicate: The environment may only append events to the history, it must preserve the well-formedness of the history, it may not invalidate the invariant \(I_e\), and it may not produce events that \(o\) can produce. This is formalized as follows:

\[
\text{interleave}(I_e, o, \alpha, h) \triangleq \{\mathcal{H}\}_\alpha \leq h \land \text{wf}(h) \land \{I_e\}_\alpha \Rightarrow \{I_e\}_\alpha[\mathcal{H} \rightarrow h] \\
\land \{\mathcal{H}\}_\alpha/out_o = h/out_o
\]

In the above, \(out_o\) denotes the set of events that originate from \(o\), and \(h/E\) denotes the longest subsequence of \(h\) that consists exclusively of events belonging to \(E\).

Because some of the rules presented here use variables that do not occur in their left-hand side, they cannot be used directly to test or simulate a Creol component. One solution would be to alter the rewrite rules so that they accept user-supplied data along with the Creol program. An alternative is to define a custom evaluation strategy in Maude that instantiates the unbound variables using random data [13].
4.2 Example: An Internet Bank Account

We will consider a `NetBankAccount` class that models a simplistic Internet bank account. In a real-world scenario, the user would log into the Internet bank, perform some deposits and payments, and log out. The deposits and payments take place during the night, and if there is not enough money in the account, the payment is delayed. In Creol, this would be modeled using asynchronous calls:

```
account := new NetBankAccount;
l1! account.deposit(50);
l2! account.payBill(80);
l3! account.deposit(50)
```

Because method overtaking is allowed, the bank might receive the deposit and payment requests in any order. Furthermore, to prevent the user from going overdrawn, the bank would first process the deposits, then pay the bill. The `NetBankAccount` class achieves synchronization using `await` and relies on Creol’s implicit mutual exclusion for processes in the same object. The class declaration follows:

```creol
class NetBankAccount
begin
  var balance : int := 0
  op deposit(amount : nat) is
    balance := balance + amount;
  return
  op payBill(amount : nat) is
    await balance ≥ amount;
    balance := balance − amount;
  return
  spec balance ≥ 0 ∧ balance = sumDeposits(H) − sumPayments(H)
end
```

In the class declaration, the `spec` clause specifies an invariant that should hold initially and whenever the processor is released. Intuitively, `NetBankAccount` guarantees that the balance will always be nonnegative and that it always equals the difference between the deposits and the payments that have been performed so far. The `sumDeposits` and `sumPayments` functions are defined recursively on histories, by inspection of reply events. Here is the definition of `sumDeposits`:

```
sumDeposits(ε) ≜ 0
sumDeposits(h ↘ [o'←self.deposit(a)]k) ≜ sumDeposits(h) + a
sumDeposits(h ↘ ν) ≜ sumDeposits(h) [otherwise]
```

Using the open system semantics, we can verify the class invariant. The invariant holds initially, because at that point the balance is 0 and `sumDeposits(H) = sumPayments(H) = 0`. We must prove that `deposit` and `payBill` preserve the invariant. Let us first verify `deposit`. We must consider an arbitrary `NetBankAccount` object in a state where the class invariant holds just before executing `deposit`’s body, and show that the invariant still holds when the method is finished. Let $h_0$ and $b_0$ be the initial values of $H$ and $balance$, respectively, such that the invariant holds.
Ignoring Rewrite Rule R12’ (Environment Activity), which has no impact on the invariant, we only need to consider one execution:

\[
\langle o : c \mid Pr: balance := balance + amount; return \epsilon, \\
Att: \alpha[H \mapsto h_0[balance \mapsto b_0]] \rangle
\]

\[
\frac{R_{11}}{\frac{R_{10'}}{\frac{R_{10}}{\frac{R_{10'}}{\frac{R_{10}}{\frac{R_{10}}{R_{10'}}}}}}}
\]

Clearly, if the invariant holds for \( H = h_0 \) and \( balance = b_0 \), it also holds for \( H = h_0 \sim [\text{caller} \mapsto o.\text{deposit}(a_0)] \) and \( balance + a_0. \)

Let’s now turn to \texttt{payBill}. If there is enough money to perform the payment, the \texttt{await} is skipped and the reasoning is similar to what we did for \texttt{deposit}. Otherwise, there is too little money and the payment must wait, leading to this execution:

\[
\langle o : c \mid Pr: \text{await} balance \geq amount; balance := balance - amount; return \epsilon, \\
Att: \alpha[H \mapsto h_0[balance \mapsto b_0]] \rangle
\]

\[
\frac{R_{14'}}{\frac{R_{8}}{\frac{R_{3}}{\frac{R_{14'}}{\frac{R_{3}}{\frac{R_{14'}}{\frac{R_{3}}{R_{14'}}}}}}}}
\]

Rewrite Rule R4’ suspends and reactivates the process. When the process is reactivated, \( H \) and \( balance \) might have changed; their new value is denoted \( h_1 \) and \( b_1 \), respectively. Furthermore, we may assume that the invariant holds, and from the \texttt{await} guard, we know that \( balance \geq amount \), that is, \( b_1 \geq a_0 \). From there, it’s easy to prove that the invariant holds at the end of the method’s execution.

Because the open system semantics focuses on a single process executing in an unspecified environment, an open system configuration will always contain exactly one object executing one process. Dovland et al. [10] describe a method for composing objects, including restrictions on the class invariants to account for asynchronous communication, that can be used unchanged for our semantic setting.

### 4.3 Connection to the Closed System Semantics

The closed system and the open system operational semantics are fairly similar: Some rewrite rules are common to both semantics, and for the others there is an almost one-to-one correspondence between the rules of the two semantics. This makes it easy to detect inconsistencies between them.

If we wanted to prove that the open system semantics is a safe approximation of the closed system semantics, we could proceed as follows: We assume that we have valid class invariants (with respect to the closed system semantics augmented by an implicit history [12]) for all the classes appearing in an arbitrary Creol program, and show that each possible closed system behavior is also possible in the open system semantics, proceeding by case on the Creol statements.

To illustrate this, we will sketch the proof for \texttt{await g}. If \( g \) is enabled, R3 applies for both semantics, so there is nothing to prove. Otherwise, R4 moves the
Blanchette and Owe

active process \( \Pi \) to \( \Pr_Q \), then other processes are allowed to run in the object, and finally \( R5 \) reactivates \( \Pi \) and removes the \textit{await} statement from \( \Pr \). Activity in other objects can be interwoven into this sequence of rewrite rules. In the open semantics, the behavior of \( \text{await} \ g \) with \( g \) disabled is captured by \( R4' \) alone. Like \( R4/R5 \), it removes \textit{await} \( g \) from the beginning of the statement list, and it simulates activities in other objects by allowing nondeterministic extension of the history variable \( \mathcal{H} \). The attributes are assigned random values to reflect activity within the object while the process was suspended, and the \( \text{MsgQ} \) and \( \text{LabCnt} \) fields are updated based on \( \mathcal{H} \) to the values they would have had in the corresponding closed system configuration.

We must now show that the side conditions of \( R4' \) are weak enough to model any possible behavior of \( R4/R5 \). \(^2\) (i) \( R4' \) requires that \( g \) is initially disabled but that \( \text{clearWait}(g) \) is enabled after \( \Pi \) has been reactivated. By inspecting \( R4 \) and \( R5 \), we can prove that this will always be the case in the closed semantics. (ii) \( R4' \) specifies that the new history is an extension of the old history. This obviously holds for the implicit history of the closed semantics. (iii) \( R4' \) requires the new history to be well-formed. This can be proved by induction on the length of a closed system computation. (iv) \( R4' \) requires the invariant to hold when \( \Pi \) is reactivated if it held when it was suspended. This holds by hypothesis. (v) \( R4' \) requires the call that initiated \( \Pi \) to be still pending. It suffices to observe that only \( \Pi \) could have sent the missing reply message, which cannot have happened since \( \Pi \) was suspended.

4.4 Connection to Hoare Logic

With the open system semantics in place, we can interpret Hoare logic formulas as follows: A partial correctness formulas \( \{P\} S \{Q\} \) is valid if and only if the state \( \alpha'\beta' \) satisfies the postcondition \( Q \) for all executions of the form

\[
\langle o : c \mid \Pr : S ; \text{LVar} : \beta , \text{Att} : \alpha \rangle \rightarrow \langle o : c \mid \Pr : S' ; \text{LVar} : \beta' , \text{Att} : \alpha' \rangle
\]

where the initial state \( \alpha\beta \) satisfies the precondition \( P \). Since the history \( \mathcal{H} \) is stored in the object as an attribute, \( P \) and \( Q \) may refer to \( \mathcal{H} \).

Consider the following Hoare axiom schema for \textit{await wait}:

\[
\{ \forall h, \bar{a} : \text{releaseReq}(h, \bar{a}) \Rightarrow Q[h/\mathcal{H}][\bar{a}/\bar{A}] \} \text{ await wait } \{ Q \}
\]

The \text{releaseReq} assertion is modeled after the \text{release} predicate from Section 4.1:

\[
\text{releaseReq}(h, \bar{a}) \triangleq \mathcal{H} \leq h \land \text{wf}(h) \land I_c(\mathcal{H}, \bar{A}) \Rightarrow I_c(h, \bar{a}) \land \text{pending}(h, \text{caller, self, label})
\]

The relationship between the axiom schema for \textit{await wait} and Rewrite Rule \( R4' \) can be made more obvious by encoding the semantics of the \textit{await wait} statement in terms of the following simultaneous random assignment statement \([10]\):

\[
\mathcal{H}, \bar{A} := \text{some } h, \bar{a} : \text{releaseReq}(h, \bar{a})
\]

This statement assigns arbitrary values \( h, \bar{a} \) to the history \( \mathcal{H} \) and the other mutable attributes \( \bar{A} \), such that the condition \text{releaseReq}(h, \bar{a}) \) is true. Clearly, the above random assignment is equivalent to Rewrite Rule \( R4' \) (with \textit{wait} as the guard).

\(^2\) To prove full equivalence between the two semantics, we would also need to show that the side conditions are strong enough to disallow any behavior that is not possible with \( R4/R5 \), assuming that the invariants completely capture the communication behavior of the classes involved.
The Hoare axiom schema for random assignment is \( \forall \bar{y} : P \Rightarrow Q[\bar{y}/\bar{x}] \) for fresh \( \bar{y} \). Using it, we derive the axiom schema given above for \textbf{await wait}. In general, to develop a history-based Hoare logic from a traditional closed system operational semantics, we follow these steps: (1) Specify an open system semantics that abstracts away the environment using a history. (2) Develop a Hoare logic for the language’s sequential subset. (3) Reformulate the open semantics as an encoding in terms of the language’s sequential subset augmented with random assignment. (4) Mechanically derive a Hoare logic from this encoding.

The Hoare logic is essentially a reformulation of the open system semantics at the syntactic level. The strength of the reasoning system depends on the strength of the class invariant, since the open system semantics relies on class invariants to determine the possible results of release points.

5 Related Work

The two main interaction models for distributed processes are remote method invocation (RMI) and message passing [3]. RMI is the approach adopted by Java and typically leads to unnecessary waiting in a distributed setting; moreover, Java’s thread concept forces the programmer to choose between reduced parallelism (using \texttt{synchronized}) and shared-variable interference, and makes reasoning highly complex [1]. Synchronous message passing also results in unnecessary delays. Asynchronous message passing, as popularized by the actor model [2], is very flexible but lacks the structure and discipline of object-oriented method calls; moreover, actors have no direct notion of inheritance or hierarchy. Creol’s release points improve on the efficiency of future variables, found in several languages [6,17]. Johnsen et al. [11] provide a more thorough review of alternative communication models.

The open semantics introduced here is inspired by Dovland et al. [10], who devised an encoding of the Creol language in a nondeterministic sequential language called SEQ, from which they derived a Hoare logic. Our presentation retains the history-based flavor of that work. Histories have been used before both to define the denotational semantics of a concurrent language [15] and to facilitate program verification [8]. The idea of recasting the notion of history from the syntactic level to the semantic level is inspired by de Roever et al. [9], who conduct their soundness and completeness proofs at the semantic level (in their case, on Floyd inductive assertion networks) and carry these proofs over to the syntactic level of Hoare logic.

6 Conclusion

The Creol language supports component and object orientation in a high-level and natural way by means of concurrent objects with processor release points and asynchronous methods calls. The language’s operational semantics is defined using rewrite rules, which form the core of the language’s interpreter.

In this paper, we introduced an “open system” operational semantics for Creol that defines the behavior of a single method execution seen in isolation, using a communication history to abstract away the environment. The semantics can be seen as the missing link between the Creol interpreter and Hoare logic, bringing the
concept of a communication history to the semantic level. Because the open system semantics is expressed in rewriting logic, it is straightforward to detect inconsistencies with the interpreter, by comparing the rewrite rules. From the new semantics, we can easily derive a history-based Hoare logic that is sound and complete by construction. The construction of such a proof system would be significantly simpler than in Dovland et al. [10], since the communication history is explicitly captured by the semantics. The open semantics can also serve as a semantic foundation for studying language extensions, including different network models such as Reo [4].

We have shown in this paper how to construct an “open system” semantics from a more conventional “closed system” operational semantics. This approach can easily be adapted to other component-based languages where communication is done by messages passing, method interaction, or both.

Acknowledgment

We want to thank Willem-Paul de Roever and Martin Steffen for valuable insights into the nature of semantics, as well as Marcel Kyas, Mark Summerfield, and the anonymous reviewers for suggesting textual improvements.

References

Resource-Oriented Design Framework for Embedded System Components

Jin-Hyun Kim, Jae-Hwan Sim and Jin-Young Choi

Department of Computer and Communication Engineering
Korea University, Seoul, Korea
{jhkim,jhsim,choi}@formal.korea.ac.kr

Abstract
To implement functionally and timely correct embedded systems, it is essential to consider both hardware and software behavior simultaneously. This paper presents an embedded system design framework, called resource-oriented design, in which embedded system components are incrementally developed in two behavioral aspects; resource-independent model (RIM) and resource-oriented model (ROM). The former embedded system model describes embedded system behavior in terms of functionality, and the latter model specifies software behavior that is restricted by hardware resource constraints. The software behavior models in those two models are based on a formal and concise hardware behavior model so as to achieve software behavior model in compliance with hardware's behavior. The hardware and embedded software behavior we define is oriented to an interaction between hardware and software. The advantage of our framework is to gain two software behavior models, functional aspect and resource-constrained aspect, such that those two models are consistent in each other and they are in compliance with hardware behavior. For the specification and verification of resource-oriented models, we use ACSR (Algebra of Communicating Shared Resource) and VERSA (Verification Execution and Rewrite System for ACSR).

Keywords: Formal Methods, Embedded System, Resource-Based, Design Methodology

1 Introduction
An embedded system is a hardware and software interacting system such that behavior of them must be in accordance with each other. To construct embedded software component in accord with hardware’s behavior, a correct and complete hardware requirement for software needs to be given in the view of software engineer, and such a hardware requirement originated from hardware engineers can be given in a formal description, and shows hardware’s behavior that not only interacts with software but also restricts the behavior of software.

To create a formal hardware and software description that are in accord with each other, codesign[3,8] provides a design framework where an embedded system is captured in a unique system design without discriminating hardware and software. The system design is partitioned into respectively a specific detailed hardware and software design for implementation of them. However, such codesigned software models cannot be often implemented correctly and completely due to hardware and software interaction problems, such as communicating timing. The behavior of embedded software, in addition, generated from a codesigned system can often differ from the behavior of original software model because
of resource constraints where processes are limited to the use of resources. Thus, it is necessary to capture and reason the behavior of embedded software that is restricted by not only communicating time but also resource constraints.

Our primary goal is to provide a design framework in which capturing a behavior of embedded software is based on a dynamic resource behavior. The behavior of resource is originated from hardware’s behavior that interacts with software or restricts the behavior of software in an execution of embedded system. Thus, the behavior of resource is captured in two aspects; functionality and constraints. The functionality of resource consists of hardware’s own functionality related to software’s functionality and the interaction between hardware and software. The constraints of resource upon software’s functionality is depicted in terms of time and the availability of resource. The timed behavior of embedded software often depends on not only communicating time but also the availability of shared resource in embedded software execution[6]. Thus, the behavior of resource describing hardware’s constraints upon software’s behavior consists of the communicating time of hardware and software and the availability of shared resource.

Based on such resource models, embedded software can gradually be captured in accord with the behavior of hardware. First, the behavior of embedded software can be captured in functional aspects by hardware and software’s functions and interaction. The focus of hardware’s behavior in the functional view is to capture hardware’s actions that impact to embedded software’s behavior. The interaction between hardware and software is composed of interrupts and addressable memory (we call it interface memory). The interrupts in our view are originated from hardware and used to synchronize software periodically or sporadically. The addressable memory, called interface memory, is accessible from both hardware and software and used to move data between hardware and software. The model of resource we present here is based on hardware and software’s interaction in the execution of embedded system. Second, the functional model of embedded software can be extended with hardware timing constraint, such as communicating time. Moreover, it incorporates the availability of resource in software’s execution by capturing the state of resource over software reaction to hardware stimulus.

With such concise resource models, an embedded system is captured in two system views; Resource-Independent Model (RIM) and Resource-Oriented Model (ROM)[5]. RIM is the composition of functional resource model and software behavior model while ROM is the composition of constrained resource model and software model. In RIM, software behavior model can be given without hardware constraints, such as time and availability. It is useful when a commonly used software over different hardware platforms should be designed into a behavioral model. Meanwhile, ROM can help us capture software’s behavior that is oriented to a specific hardware platform because the behavior of software is rather restricted by communication timing and the availability of resource than the functionality of hardware when we have a hardware platform in mind. Thus, ROM is useful when software functional model needs to be extended with hardware specific properties, such as communication time and the availability of a hardware.

The rest of this paper is organized as follows: In next Section 2, some related works are discussed, and then, we define behavior models of embedded system components in Section 3. In Section 4, we present a resource model that captures a hardware’s specific behavior and constraints. After that, we introduce two embedded system models that are composed of behavior models of embedded system components in Section 5. In Section 6,
we explain our design framework by giving an example of embedded system. We discuss a formal verification in our design framework in Section 7, and we finally conclude this paper in Section 8.

2 Related Works

Embedded system models can be divided into structural model and behavioral model[2]. Software developers are usually even more concerned about how these structural elements behave dynamically as the system runs. Douglass[2] suggests a behavioral description using statechart[4]. However, He does not discuss how to define hardware properties, such as time, resource restriction, in the embedded system model.

The notion of a resource provides a convenient abstraction mechanism[7] that can describes the behavior of hardware in the view of software. A resource can be captured in two aspects; preemptive [6] and quantitative [7,10]. Lee and Ben-Abdallah suggests a description language, called ACSR(Algebra of Communicating Shared Resource), in which a real-time system is designed with focusing on a set of communicating processes that use shared resource for execution and synchronize with one another. The nature of resource in ACSR is preemptive by a higher priority process during an execution of processes. Thus, a system is captured in terms of the behavior of process that is restricted by limit resources and a scheduling of processes for shared resources, then such processes do not care for the function or the action of resource. The nature of resource that Lee and Sokolsky present in PACSR(Probability ACSR)[7,10] differs from the resource of ACSR in that a process can acquire a resource with a given probability so that PACSR allow to reason quantitatively about a system’s behavior. In this paper, we present a more dynamic resource that can changes its state according to stimulus from environment and embedded software, including time. A resource we present here captures hardware’s behavior in terms of software, and is modeled by the interaction between hardware and embedded software in their execution and communication.

3 Embedded System Behavior Models

In this section, we present three behavior models that can constitute an embedded system; software behavior model, software-oriented hardware model, and resource model. Software Behavior Model($M_S$) defines the functionality of software including its reaction to hardware events, such as hardware interrupts and time tick. Software-Oriented Hardware Model($M_{SH}$) defines a hardware functional behavior over events in embedded software engineer’s view. The hardware functional behavior is oriented to hardware and software interaction. Resource Model($M_R$) captures the behavior of hardware over events and time in the execution of embedded system. In this resource model, we focus on hardware constraints upon embedded software, such as hardware and software’s communication time. Thus, the behavior of hardware in a resource model describes by not only the functionality of hardware but also timing and availability constraints. Resource Model differs from Software-Oriented Hardware Model in that Resource Model is captured after the decision of hardware platform. Embedded software is often implemented after hardware is developed and a software functional requirement needs to be delivered before hardware’s development is finished. Then, software engineers can design embedded software in as-
pect of functionality using Software-Oriented Hardware Model because Software-Oriented Hardware Model presents the aspect of hardware functionality.

In this paper, the behavior of hardware and embedded software in Fig. 1 is oriented to the interaction of them. We assume that an embedded system is a tuple \((M_S, S_\Omega, M_{SH})\), where \(M_S\) is a software behavior model, \(S_\Omega\) is interface memory, and \(M_{SH}\) is a hardware behavior model. In this embedded system model, embedded software interfaces directly with hardware through interface memory (we call it interface memory because it is shared by hardware and software, such as CPU register and addressable memory). The interaction between hardware and software is achieved as follows: A hardware unit \((M_{SH})\) interrupts CPU running software system when it wants to order a software routine to process hardware data and completes its data writing on interface memory. The interrupted CPU finds out a corresponding interrupt handler in its interrupt vector table (IVT), and then, it initiates interrupt service routine (ISR) based on an address from IVT. ISR just sends a message to the real-time OS in order to initiate the corresponding task. The initiated task reads hardware input data from interface memory, processes them, and writes the processed data into interface memory in a specific time. Hardware reads the software-processed data from interface memory in a specify time and computes another operations according to the software-processed data and its own state. In firmware, an ISR directly calls a software function or procedure when hardware interrupts CPU running an idling function. Thus, the first software routine initiated by a hardware interrupt is an interrupt handler. Other software routines, task, processes, function, and procedure are initiated by such interrupt handlers. These software routines process hardware data from interface memory based on their own state, and hardware units also perform their functions based on data in interface memory and their own state recoded in register.

3.1 Embedded Software Behavior Model

Software Behavior Model defines the functionality of software including its reaction to hardware events. If a hardware event, hardware interrupt or time tick, initiates an interrupt service routine (ISR), the ISR calls software routines that would handle a hardware input data. The software routine transforms the configuration of interface memory by processing data or calls other software routines. Finally, a software routine calls a special task, called IDLE, to wait for a next hardware event. Software Behavior Model is defined as follows:

**Definition 3.1 Software Behavior Model** \((M_S)\) is a tuple \((S_S, E^I_S, F_S, Q_S, \theta)\). \(S_S\) is a set

![Fig. 1. Embedded System Model](image-url)
of software states where $S_Ω \subseteq S_S$, and $E^I_{\Omega}$ is a set of interrupts from hardware. $F^I_{S} : \ S_S \rightarrow S_S \times F_S$ is a set of software routines, and $θ : E^I_{\Omega} \rightarrow F_S$ is a function from hardware events to a software routine, called interrupt service routines, where $∀q ∈ F^I_S, ∀s ∈ S_S, ∃f ∈ F_S.q(s) = f$.

In $F^I_S$, IDLE is a special function such that $∀s ∈ S_S, I(s) = (s, I)$.

The software behavior model defines software’s reaction to hardware interrupts, software routine calls, and interface memory read and write. The embedded software routines can be task, process, function, procedure, or interrupt handler. When a hardware unit interrupts CPU running embedded software, CPU looks at an interrupt vector table ($E^I_{\Omega} \rightarrow Q$) in order to find out a corresponding interrupt handler. An interrupt handler calls another software routine ($q(s) = f$), such as tasks. A software routine initiated by interrupt handler starts its data process using the state of interface memory and another software routine ($F^I_S : S_S \rightarrow S_S \times F_S$). In a completion of embedded software execution, a special function, IDLE, is executed for a next hardware event.

Fig. 2 shows a behavioral view of embedded system behavior. Hardware units and embedded software components in an embedded system transmit their data via addressable memory (we call it interface memory). A hardware port connected to interface memory moves the software processed data into hardware registers to allow a hardware circuit to control the signal system and the gate system in Fig. 2. In the execution of embedded software, hardware interrupts CPU that runs embedded software, and an interrupt handler calls a corresponding task. The task can directly modify interface memory or indirectly modify it via another function and procedure, utilizing its local memory area to process data.

3.2 Embedded Hardware Behavior Model

To capture hardware’s behavior in the running of embedded system, a hardware and software’s interaction using hardware’s state, communication data, and event would be focused on. The information transmitted through interface memory are about hardware’s state and data to be processed by embedded software. **Software-Oriented Hardware Model**, an
abstract hardware behavior model, is defined as follows:

**Definition 3.2 Software-Oriented Hardware Model**\( (M_{SH}) \) is a tuple \((S_{SH}, E^I_H, E^O_H, E_I^S, \delta_{SH}, I_{SH})\), where \(S_{SH}\) is a set of abstract hardware states, and \(E^I_H\) is a set of events from the environment. \(E^O_H\) is a set of events to the environment, \(E_I^S\) is a set of interrupt to embedded software, \(\delta_{SH} : S_{SH} \times E^I_H \rightarrow S_{SH} \times E^O_H \times E_I^S\) is a transition function, and \(I_{SH}\) is a set of initial hardware states.

In Software-Oriented Hardware Model, an event from its environment initiate a hardware module so as to execute a sequence of functions based on both its own state and the value of data from embedded software in interface memory. The software-processed data is written in interface memory\((S_{\Omega} \in S_{SH})\) with the state of software(if software records its state in interface memory). A hardware module reads such software-processed data via port and bus connected to interface memory. During running of hardware unit, it can change its state and the value of interface memory, generate events to the environment, and occur a hardware interrupt for software’s initiation\((\delta_{SH} : S_{SH} \times S_{\Omega} \times E^I_H \rightarrow S_{SH} \times E^O_H \times E_I^S)\).

Hardware’s behavior is often subject to physical time because it physically controls devices. Therefore, we define a time-related function, called duration function\((D)\).

**Definition 3.3 Duration Function** \(D : E \rightarrow N\), where \(E \subseteq E^I_S\)

Most of hardware modules makes a synchronization of each hardware modules by a specific signal. In running of embedded system, a hardware module interacting with embedded software often requires embedded software to finish all of its operations in a given time. A deadline is a time by which a task must complete[11]. However, Duration we define here is a particular time interval from the start of software operation to the completion of all of software operations. Duration\((D)\) is from a hardware event, interrupt, to a duration time.

### 4 Resource-Based Embedded System Models

Resources in computer system are physical and logical elements that application software needs in processing data. For instance, some embedded software needs physical elements, power and bandwidth, to meet QoS and CPU, memory, and I/O devices, to process its data. Some embedded software needs a logical element, data structure, semaphore, message queue, mailbox, network data packet served by operating systems for accessing to such hardware devices.

Resource in embedded system can be defined as a hardware-controlled object that interacts with embedded software. The hardware-control object consists of interface memory for the communication between hardware and software and interrupts from hardware. Interface memory is dedicated to data incorporating the hardware and software’s behavior as well as communicating information. The interrupt initiating the activity of software is originated from hardware and interface memory has the value of data controlled by hardware and software. The reason why the hardware-controlled object is regarded as a resource is as follows: The configuration of interface memory at one time describes, in terms of software, the behavior of hardware at that time, and it can also include the value of data to be processed by software. The configuration of interface memory changes along with hardware behavior, and embedded software reacts to interrupt from hardware and executes
its operations according to the configuration of interface memory. That is, interface memory is controlled by hardware and moves hardware’s state to software. Moreover, it also delivers data of hardware to embedded software to process them. Thus, interface memory is no doubt regarded as a resource. Moreover, the behavior of hardware shown by interface memory can be regarded as one of resource properties. In short, we add a behavioral aspect shown on interface memory to resource properties. The resource of embedded system we consider here can be classified into hardware resource and software resource. A hardware resource is a hardware object that interacts with software or supports software execution. For instance, CPU, memory, I/O ports, motors, sensors are hardware resource. Software resource is a software object that provides hardware management, such as read/write external devices. A software resource also provides application software with data to be processed and a bunch of functions supporting software’s execution. The software resource differs from application software in that it continuously reacts to its environment and keeps reacting to software’s requests and hardware’s interrupts. Thus, it can be regarded as a hardware component in a view of application software.

The nature of resource in embedded systems can characterized by three properties. First, a resource in embedded system has a specific interactive behavior with software. The behavior of resource reflects in a software engineer view hardware behavior that interacts with software. Using interface memory, embedded software can grasp hardware behavior and input data that it needs in its computation. In addition to interface memory, a hardware interrupt and time tick is often used for software to acknowledge hardware behavior in the sense that it can know when to start its operation in hardware and software interaction. Thus, the behavior of resource can be consist of hardware state over hardware events and input data to software. Second, the behavior of resource in embedded system is often timed-constrained so that it sometimes requires software to finish its computation in a given time. Hardware interpreted by resource to software controls external and physical devices, such as sensor, motor, and so on. Such hardware-controlled devices often put real-time constraints upon the behavior of embedded software, and such timing-constraints is also interpreted to software via interface memory. Thus, timing-constraints interpreted by interface memory is incorporated into a property of resource. Finally, the availability of resource also restricts the behavior of embedded software. An execution of software routine can be delayed by not acquiring its necessary resource in a specific time, and resources that embedded software needs in its execution are often hardware resources, such as network packet and ports, including data from hardware. Thus, The software execution may be dependent on the availability of a resource.

To define the availability of hardware resource, we define a resource access function as follows:

**Definition 4.1 Availability Function** $\Lambda :$

$$(S_S \cup S_{SH}) \rightarrow \lambda \times \{ACCESS, DENIED\},$$

where $\lambda \in S_\Omega$

$\Lambda$ is a function that is used to find out a resource’s state, whether it is accessible or not by a software routine. That is, an embedded software routine can never control a resource including interface memory when the resource is locked or preempted by another software routine.

We define the behavior of resource as resource model, incorporating our duration function and resource availability function($\Lambda$) into our software-oriented hardware behavior.
Fig. 3. Resource-Based Embedded System Model

Definition 4.2 Resource Model \( M_R \) is a tuple \( (M_{SH}, D, \Lambda) \)

In Resource Model, we can figure out the following hardware’s properties related to software’s functionality.

- Hardware’s events that initiate embedded software routine.
- Time that is a duration when an embedded software routine can run.
- Hardware state and data when an embedded software routine can access to hardware.

5 Resource-Based Models for Embedded Software

Fig. 3 presents two embedded software models; \textbf{Resource-Independent Model}(RIM) and \textbf{Resource-Oriented Model}(ROM)[5] that are based on two hardware behavior model; software-oriented hardware model and resource model. The hardware models can become software requirements by which embedded software’s behavior is designed and restricted in functionality, timing and availability aspect.

RIM is a software behavior model that is based on hardware functionality, consisting of software behavior model and software-oriented hardware model. RIM is defined as follows:

Definition 5.1 \textbf{Resource-Independent Model}(RIM) is a tuple \( (M_S, M_{SH}) \).

RIM defines the interaction and communication between hardware and software, the functionality of software, and the behavior of hardware based on the interaction. RIM does not consider the resource constraints in respect of timing and availability. The purpose of RIM is to overview embedded system behavioral requirement by capturing hardware and software interaction in their running. In embedded system development, a software-oriented hardware model is provided by hardware engineers after the hardware functional verification. And then, software engineers can construct its necessary operations into RIM with guided by the software-oriented hardware behavior model and system requirement.
that says software requirement. The constructed RIM is used to verify software behavior in their software behavior model in respect of functional aspects of an embedded system.

ROM is a software behavior model that is based on the resource model, and it consists of a software behavior model and a resource model for requiring the property of timing and availability for software behavior. ROM is defined as follows:

**Definition 5.2 Resource-Oriented Model (ROM)** is a tuple \((M_S, M_R)\).

ROM defines software behavior restricted by hardware constraints, such as timing and availability, as a property of resource model. The software behavior in ROM can be verified against such hardware constraints. The resource model in ROM is originated from hardware engineers. They prove its timing properties using hardware timing verification techniques. And then, the hardware model can be abstracted in a software engineer view for providing hardware constraints to software engineers, and software engineers can build their software behavior model based on the software-oriented hardware behavior model and verify it against the hardware model.

Fig. 4 shows the resource-oriented development for embedded system. In this development, the requirement of embedded system is firstly partitioned into respectively hardware and software requirement. A software-oriented hardware model abstracted from hardware’s requirement is constructed so as to be delivered for software requirement analysis of software engineer, in which embedded software’s functionality is validated in companion with hardware’s functionality for checking if embedded software’s behavior is in accord with hardware’s in the view of execution of their interaction and communication. In software requirement specification, there are two requirement specifications about software’s behavior; software’s functionality and its interaction with hardware. The requirement specification with respect to software’s interaction behavior describes pre-conditions and post-conditions of software to hold true, and actions to be performed before and after the execution of hardware and software’s interaction. After a hardware design completion through timing verification, a resource model abstracted from hardware design is build, and then, an embedded software model modeled in a resource-oriented software behavior model is configured to be in accord with hardware’s timing and availability shown in a resource model.
6 Modeling Embedded System Components with ACSR

To show a way how to apply resource-oriented design models to the embedded system design, a construction of embedded system model using ACSR (Algebra of Communicating Shared Resource) [6] is illustrated in this section. ACSR is a timed process algebra based on the synchronization model of CCS that includes features for representing synchronization, time, temporal scopes, resource requirements, and priorities.

In resource-oriented models, there are software behavior model, software-oriented hardware model, resource model, Resource-Independent Model, and Resource-Oriented Model. The first three models are embedded system component models, and the last two models are embedded system models. To explain each property of resource-oriented models, we present an example of a signal and gate control system.

Fig. 2 shows a signal and gate control system, in which the color of signal is determined by a software process and the gate is controlled by a hardware circuit. The color of signal is determined by software in a way that it starts its computation when receiving a periodic interrupt from hardware, and it cyclically turns on one of signals of red, yellow, or green. The gate is controlled by a hardware circuit according to the state of signal. The gate closes down when the red signal is turned on, and it opens up when the green signal is turned on. In an exceptional case, if an emergency signal is occurred from the environment, the red signal is forced to be turned on, and the gate must shut down regardless of whether this system is in any states.

6.1 Software Behavior Model

A software behavior model captures the behavior of embedded software consisting of software’s reaction to a hardware interrupt and a sequence of software function calls during its computation.

**Example 6.1** The signal system controlled by embedded software is captured with software behavior model. To simply define states of system, we use an abbreviated form, \((\text{Var} \mapsto 1)\), meaning that the only variable \(\text{Var}\) in a tuple is value 1 and any other variables are value 0. The software behavior model of signal system is a tuple \((S_S, E^I_S, F_S, Q_S, \theta)\), where

- \(S_S = \{(\text{Green} \mapsto 1), (\text{YellowGreen} \mapsto 1), (\text{YellowRed} \mapsto 1), (\text{Red} \mapsto 1)\}\),
- \(E^I_S = \{\text{Emergency, Go}\},\)
- \(F_S = \{(\text{ChangeYellowToGreen},\text{ChangeGreenToYellow}), \text{ChangeYellowToRed}, \text{ChangeGreenToRed}, \text{ChangeRedToYellow}, \text{ErrorHandle}\}\),
- \(\text{ChangeYellowToGreen} = \{(\text{YellowGreen} \mapsto 1) \rightarrow ((\text{Green} \mapsto 1), \text{IDLE}), (\text{otherwise}) \rightarrow ((\text{otherwise}), \text{ErrorHandle})\}\),
- \(\text{ChangeGreenToYellow} = \{(\text{Green} \mapsto 1) \rightarrow ((\text{YellowRed} \mapsto 1), \text{IDLE}), (\text{otherwise}) \rightarrow ((\text{otherwise}), \text{ErrorHandle})\}\),
- \(\text{ChangeYellowToRed} = \{(\text{YellowRed} \mapsto 1) \rightarrow ((\text{Red} \mapsto 1), \text{IDLE}), (\text{otherwise}) \rightarrow ((\text{otherwise}), \text{ErrorHandle})\}\),
- \(\text{ChangeRedToYellow} = \{(\text{Red} \mapsto 1) \rightarrow ((\text{YellowGreen} \mapsto 1), \text{IDLE}), (\text{otherwise}) \rightarrow ((\text{otherwise}), \text{ErrorHandle})\}\),
- \(\text{ChangeGreenToRed} = \{(\text{Green} \mapsto 1) \rightarrow ((\text{Red} \mapsto 1), \text{IDLE}), (\text{otherwise}) \rightarrow ((\text{otherwise}), \text{ErrorHandle})\}\),
- \(\text{ErrorHandle} = \{(\text{otherwise}) \rightarrow ((\text{Red} \mapsto 1), -)\}\)
A state of software can be captured by a composition of values of software variables at a specific time. The state transition can be enabled by assigning a set of new values to software variables during its computation. However, ACSR includes no explicit syntax that is able to capture software’s state and computation. By this reason, we define several event types for specifying software’s state and operations.

In Table 1, the receiving of event captures a hardware interrupt occurrence. The operations of read and write of a device describe software’s access to external hardware devices, such as motor, and sensor. Those operations need a particular consideration, such as hardware’s timing and availability constraints, so we would capture those constraints in resource model.

Example 6.2 Signal Control Embedded Software in ACSR

<table>
<thead>
<tr>
<th>State Transition</th>
<th>ACSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>F : S \rightarrow S x F'</td>
<td>F = (UPDATE_X,Y1,1,1)F'</td>
</tr>
<tr>
<td>Condition Evaluation if (X == Y) A else B</td>
<td>(X,Y1,Y1,1)A + B</td>
</tr>
<tr>
<td>Receive Event X E</td>
<td>(E1,1)</td>
</tr>
<tr>
<td>Send Event X E1</td>
<td>(E1,1)</td>
</tr>
<tr>
<td>Read X from Device</td>
<td>(READ_X,Y1,1)</td>
</tr>
<tr>
<td>Write X to Device</td>
<td>(WRITE_X,Y1,1)</td>
</tr>
<tr>
<td>Function Call in Process A P()</td>
<td>A = P</td>
</tr>
</tbody>
</table>

Table 1: State and Operations in ACSR
Kim, Sim, and Choi

+ (Green_IS_1,1).(YellowGreen_IS_0,1).(Red_IS_0,1).(YellowRed_IS_0,1)
  .ChangeGreenToYellow
  + SW_ERROR_A;
EmergencyMode = (Green_IS_1,1).(Yellow_IS_0,1).(Red_IS_0,1).(Pass_IS_0,1)
  .ChangeGreenToRed
  + SW_ERROR_A;
ChangeYellowToGreen = ('UPDATE_Green_1,1).('UPDATE_YellowGreen_0,1)
  .('UPDATE_Red_0,1).('UPDATE_YellowRed_0,1)
  .SBM_TCS
  + SW_ERROR_A;
ChangeGreenToYellow = ('UPDATE_Green_0,1).('UPDATE_YellowGreen_1,1)
  .('UPDATE_Red_0,1).('UPDATE_YellowRed_0,1)
  .SBM_TCS
  + SW_ERROR_A;
...
SW_ERROR_A = ('SW_ERROR,1).SW_ERROR_A;

6.2 Software-Oriented Hardware Model

A software-oriented hardware model captures hardware’s behavior in interaction with software. Therefore, it includes only information that software needs in the execution of software. A hardware component receives an event from the environment, changes its state based on states of both hardware and interface memory, and generates events to the environment and to its relevant embedded software component. The state of hardware is inherently related to the execution of embedded software. To specify a hardware component in ACSR, both states($S_{SH}$ and $S_{Ω}$) are interpreted in ACSR as follows:

First, states in the type $S_{SH}$ are captured by the process of ACSR. For instance, states of gate can consist of opened, opening, closed, closing, and warning closing, and these states are captured with the process of ACSR. Second, the state of interface memory ($S_{Ω}$) is captured in the early defined software operations with the event of ACSR.

Example 6.3 Gate Control Hardware System in ACSR

HBM_GC = OPENED ;
OPENED = (tick,1).('closing,1)
  .(Green_IS_1,1).(Yellow_IS_0,1).(Red_IS_0,1).(YellowRed_IS_0,1)
  .('Close,1).('Go,1).CLOSING
  + ('HW_ERROR,1).HW_ERROR_A
  + (Emergency,1).('Close,1).('Go,1).CLOSING
  + OPENED;
CLOSING = (down,1)
  .(Green_IS_0,1).(YellowGreen_IS_0,1).(Red_IS_0,1).(YellowRed_IS_1,1)
  .('Go,1).CLOSED
  + ('HW_ERROR,1).HW_ERROR_A
  + CLOSING;
CLOSED = (tick,1).('opening,1)
  .(Green_IS_0,1).(YellowGreen_IS_0,1).(Red_IS_1,1).(YellowRed_IS_0,1)
  .('Open,1).('Go,1).OPENING
  + ('HW_ERROR,1).HW_ERROR_A
  + CLOSED;
OPENING = (up,1)
  .(Green_IS_0,1).(YellowGreen_IS_1,1).(Red_IS_0,1).(YellowRed_IS_0,1)
  .('Go,1).OPENED
  + ('HW_ERROR,1).HW_ERROR_A
  + (Emergency,1).('Close,1).('Go,1).CLOSING
  + OPENING;
GATE_SENSOR = (opening,1).('up,1).GATE_SENSOR
  + (closing,1).('down,1).GATE_SENSOR
  + GATE_SENSOR;

In Example 6.3, the functionality of gate control system is captured in ACSR. The gate starts its closing operation when it is opened(OPENED) and receives a signal tick from
a timer. If the signal of green is on and the others are off, it continues its actual closing operation with sending a signal GO to the signal control software(′Go, 1)\text{. CLOSING}′.

6.3 Resource Model

A resource model in resource-oriented design captures a hardware behavior including hardware constraints put upon embedded software components. The constraints we include into a resource model are timing and availability in the use of hardware.

The model of ACSR corresponding to a resource model would capture not only the behavior of a hardware component but also explicitly aspects of time and availability of it. An action in ACSR represents the consumption of named resource for one time unit because the execution of an action is subject to the availability of the named resources and the contention for resource is arbitrated according to the priorities of competing actions[6]. The action 0 represents the passage of one time unit without consuming any resources, that is, idling for one time unit. However, ACSR does not provide a specific way to describe an exclusive use of a resource without consuming time. A time behavior in ACSR is a sequence of actions, in which a sequence of events may appear between any two actions, but event’s actions consume no time. In short, ACSR provides no way to capture an action of resource without consuming time because ACSR is designed to describe an exclusive use of a resource that must be in companion with a timing consuming. The resource-oriented design we discuss here pursues a modeling of a resource including more explicit and specific behavior of a hardware component. That is, only an exclusive use of a resource in modeling a resource can be captured in the specification of ACSR. Meanwhile, the resource-oriented design pursues capturing not only the exclusive use of a resource but also a specific impact of resource behavior on the behavior of software.

Example 6.4 A Resource Model for a Gate System in ACSR

\[
\text{TIMER} = (\text{′tick, 1} \text{). } \text{TIMER} + \text{′}: \text{TIMER;}
\]

\[
\text{GATE\_SENSOR} = (\text{′closing, 1} \text{). } \text{GATE\_SENSOR}
+ (\text{′down, 1} \text{). } \text{GATE\_SENSOR}
+ \text{′}: \text{GATE\_SENSOR;}
\]

\[
\text{Green\_0} = (\text{′ACCESS\_Green, 1} \text{). } (\text{′Green\_IS\_0, 1} \text{). } \text{Green\_0}
+ (\text{′ACCESS\_Green, 1} \text{). } (\text{UPDATE\_Green\_1, 1} \text{). } \text{Green\_1}
+ \text{′}: \text{Green\_0 ;}
\]

\[
\text{Green\_1} = (\text{′ACCESS\_Green, 1} \text{). } (\text{′Green\_IS\_1, 1} \text{). } \text{Green\_1}
+ (\text{′ACCESS\_Green, 1} \text{). } (\text{UPDATE\_Green\_0, 1} \text{). } \text{Green\_0}
+ \text{′}: \text{Green\_1 ;}
\]

In Example 6.4, two aspects of a hardware component are highlighted by a resource model: a time consuming of an hardware’s activity and an access to a hardware component. For instance, \text{TIMER} occurs a signal tick every ten time-unit. The gate takes three time-unit to complete its opening or closing after its operation starts. The shared memory indicating the color of signal is protected by an access control using a signal \text{ACCESS\_Green}, so an software component can refer or update the variable of signal color only when a shared hardware component permits its accessibility to be given to the software component.

The last two embedded system models, RIM and ROM, are constructed by a composing basic models embedded system component. A software behavior model and a software-oriented hardware model compose a RIM. After that, the software-oriented hardware model is configured in order to include hardware constraints, such as timing and availability of
a hardware component interacting with software components. The configured software-oriented hardware model is a resource model we present here, and such a resource model is used for software engineers to construct a detailed design of embedded software.

7 Formal Verification in ACSR

The model of ACSR can be verified with VERSA[1]. VERSA is a tool that assists in the algebraic analysis of real-time specification written in ACSR. VERSA can also be used to analyze ACSR. An important technique that VERSA supports is an equivalence checking of ACSR process. In ACSR, if a design specification is equivalent to its requirement specification, then the design specification can be considered correct. The property to be checked in RIM is whether there is no contradiction in the interaction between hardware and software in terms of functionality. ROM, in addition, is verified in aspects of a timed behavior restricted in using a limited resource.

RIM can be formally verified with the following verification property.

\[ \text{SYST}_n \approx \pi \tau^\infty \]

The \( \tau^\infty \) means that it consumes no time and resources, so the question means that the system constantly is bisimilar[9] to an infinite sequence of idle action \( \tau^\infty \) without the consuming of time and the use of resource. If VERSA says “yes”, a modeled system always succeeds synchronization correctly using their communication events. In other words, the functionality of a system in an interaction of hardware and software is correct.

ROM can be formally verified with the following verification property.

\[ \text{SYST}_n \approx \pi \emptyset^\infty \]

The \( \emptyset^\infty \) indicates that a system consumes time in using a resource. Models in a ROM are captured with timing and availability constraints so a verification property for a ROM also asks a question if there is no contradiction in an interaction of hardware and software not only consuming time but also using a limited resource.

8 Conclusions

A resource in computer system is a classical notion to abstract a component used in software computation. An embedded system is often limited in using a resource so that a system requirement includes a specification with respect to such limited resources.

In this paper, we present an embedded system design framework, called resource-oriented design, in which an embedded system is captured in the view of hardware behavior and constraints. This paper presents three embedded system component models, software behavior model, software-oriented model, and resource model. The software behavior model capture the behavior of a software component in respect of its functionality in an interaction of hardware and software software. The software-oriented model abstracts hardware’s behavior in the view of software behavior, and the resource model incorporates hardware resource constraints, such as timing and availability. For a system view, such embedded system component models can be composed into RIM and ROM. Using RIM, each component of embedded system is verified against the functionality of embedded system. Using ROM, aspects of timing and availability can be verified in the execution of embedded
Using resource-oriented design, an embedded system can be developed gradually. In embedded system industry, hardware is first developed, and embedded software is implemented on the hardware. Moreover, software components are necessarily reused in other hardware platforms. Thus, the reused embedded software components need to be designed independently from hardware platform. By means of a software behavior model that resource-oriented design provides, an embedded software component in requirement can be analyzed without a complete hardware model and can be captured to be reused over various hardware platforms. And the software behavior model is formally analyzed in companion with hardware behavior model by RIM in the view of functionality.

A requirement of embedded software needs to be developed into a detailed software component design. The resource-oriented design provides a detailed software component model that is originated and extended directly from a software requirement models by evolving a software behavior model based on an hardware-constraint model, called resource model. Moreover, a detailed embedded system model, ROM, provides a framework in which a detailed software design can be analyzed in companion with hardware constraints captured in a resource model.

Currently, we have an investigation on a formal language that is suited for the resource-oriented design.

References

Process Algebra with Local Communication

Muck van Weerdenburg

Eindhoven University of Technology
Department of Mathematics and Computer Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract
In process algebras like μCRL and ACP communication is defined globally. In the context of component-based architectures one wishes to define subcomponents of a system separately, including communication within that subcomponent. We define a process algebra with an operator for local communication that facilitates component-based architectures. Besides being compositional, this language is aimed to be a more practical language (with respect to closely related languages) and also allows for straightforward modelling of synchronous as well as asynchronous behaviour.

Keywords: process algebra, local communication, true concurrency, compositionality, synchrony

1 Introduction
In modelling systems, component-based architectures are a natural way of separating different parts (and subparts) of a system and specifying how these parts relate to each other by means of, for example, communication. Especially in the case of larger systems (i.e. real-life systems and complex protocols), component-based modelling is essential to avoid losing overview of the model. We introduce a new process algebra called LoCo which aim it is to be a practical language supporting such hierarchical modelling. Besides the asynchronous behaviour seen in most languages, this algebra also supports the modelling of synchronous behaviour. This allows for the modelling of, for example, electronic circuits (from which one wants to abstract away from the relatively small delays) or easy multiway communications.

The main difference between LoCo and most other process algebras is the fact that it has a local communication mechanism and multiactions. Local communication means that one can specify communication between components precisely where it is relevant, whereas global communication means that one has to specify the communication for the whole system in one place. Multiactions are basically multisets of actions that occur at the same time (without communicating). With these concepts we also get a straightforward way to model synchronous behaviour.

1 E-Mail: M.J.van.Weerdenburg@tue.nl

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
To illustrate the differences between global and local communication we consider Figure 1. Here a system with two of components, A and B, that desire to communicate via actions $s$ and $r$ is depicted. Now say that in another part of the system there is a components C that, for some purpose, also uses an action $r$ (possibly because C is actually just B but in a different context). In Figure 1(a) it is illustrated that it is not possible to simply define these communications on a global level. There is no way for component A to avoid communicating with C instead of B. The only way to avoid such mistakes is to rename some of the actions (e.g. $r$ in A to $r_A$ etc.). With local communication one can simply specify that a given communication is only meant for certain parts of the system. This is illustrated on the right in Figure 1(b).

![Diagram](image)

(a) Global communication  
(b) Local communication

Fig. 1. Global vs. local communication

Primarily, the language LoCo was developed as a basis for mCRL2 \(^2\) [13,15], successor of $\mu$CRL [14], where an important motivation was to be able to straightforwardly express Petri nets [17] in process algebra. In Petri nets the firing of a transition consist of taking tokens from some places and, at the same time, putting tokens in others. This was the initial reason to introduce multiactions, which in turn resulted in the addition of local instead of global communication. After adding this local communication it became clear that it made the language a truly compositional process algebra. This led us to reversing the thought process: local communication is introduced to make the language compositional and it is this local communication that leads to the introduction of multiactions (as is explained below).

It is due to this origin that LoCo seems to be more closely related to other attempts to link Petri nets and process algebra than to process algebras developed for component-based modelling. Also the fact that the latter are often inspired by CCS [20] or the $\pi$-calculus [22] plays a role. These languages, such as CaSE [23], IP-calculus [10] and PiLar [11], have local communication, but it is restricted to actions ($a$) communicating with their counterparts ($\overline{a}$) only. Or, as in PADL [6], they use strict synchronisation on actions. In such cases, communication (or synchronisation to a single action) is tightly linked to the parallel operators and therefore does not require multiactions. It does, however, restrict the freedom in modelling.

---

\(^2\) mCRL2 is basically LoCo with a slightly different syntax, timing and a (more or less) fixed data algebra.
and complicates constructions as multiway communication. Other process algebras with local communication or synchronisation, such as LOTOS [16] and the Interworkings language [19], also have this strong connection between communication and parallelism.

In LoCo we have separated the concepts of parallelism and communication and linked them by means of multiactions. This also gives a more natural intuition to the operators. Parallel composition now means just putting two components in parallel; the execution of actions in one component is independent of the other component. Only by applying a special operator for communication one creates links between components. To make this possible, multiactions are needed. The parallel composition results in multiactions as a consequence of components executing actions at the same time. Creating a link between two (or more) components is done by applying the local communication operator. This way actions within the multiactions result in a communication. It is also much easier to model multiway communication in this way, in contrast to algebras where communication and parallelism are as one. In these languages one has to specify communication per pair of components (and pair of pair of components etc.) in such a way that the result is in fact a multiway communication. That is, if it is even possible to do so only by means of communication.

This separation of parallelism and communication is also seen in ACP [3] and the algebra from [2], which describe an ACP [4] approach to combining process algebra and Petri nets. However, in [3] the communication used by the local communication operator is still defined globally and thus limits compositionality. Because of this one cannot use a single component multiple times in the same system (in different contexts) without renaming its actions. On the other hand, [2] uses a renaming operator to apply local communication, but in a setting with data this is not really a preferable choice. It is more intuitive for renaming to disregard data and for communication to require that data parameters (of actions) are equivalent.

The Petri Box Calculus [8,7,9] gives a more Petri-net-based approach to combining both formalisms. Although the communication is done in the CCS style, multiactions are used here. Because of these multiactions multiway communication becomes feasible, but it remains cumbersome due to the type of communication. A component needs to know that it is going to participate in a multiway communication or one has to introduce a special component that takes care of it. The same holds for SCCS [21] and MEIJE [1], which add synchrony to CCS.

What is not encountered in any of the above algebras with multiactions is a restriction operator that only allows certain actions. Only in [2] one might consider it present in the blocking (or encapsulation) operator (the inverse of restriction), but blocking on its own is not sufficient in practice. The reason to add the restriction operator is that it can often be the case that one only wishes to allow a small number of multiactions of a process with a much greater total amount of different multiactions. Take for example the parallel composition of \( n \) actions, which results in \( 2^n - 1 \) different multiactions. If one only wants to allow the multiaction in which each action is present, blocking would require one to specify \( 2^n - 2 \) multiactions (instead of just one with our restriction operator).

In Section 2 we introduce the syntax of LoCo for which we give a semantics in
Section 3. An axiomatisation for LoCo is given in Section 4 to facilitate algebraic reasoning. As abstraction and data often play an important role in systems, we discuss these in Sections 5 and 6, respectively. Some examples are given in Section 7 to illustrate LoCo by modelling a Petri net and a simple compositional system.

2 Syntax

We describe the elements of our algebra informally. A detailed definition of the syntax can be found in [28] (without data) and [27] (with data).

The basic elements of processes are multiactions. Multiactions are bags (or multisets) of actions, from the set of actions $\mathcal{N}_A$, that execute together. We write a multiaction of actions $a$, $b$, and $c$ as $\langle a, b, c \rangle$ (or $\langle b, a, c \rangle$ as order has no meaning in bags). Often we write multiactions that consist of only one action without brackets (i.e. $a$ instead of $\langle a \rangle$). We can combine such multiactions with the common operators $\cdot$ and $+$ to form a sequence of multiactions or a nondeterministic choice between multiactions, respectively. The special case of the empty multiaction $\langle \rangle$ is called a silent step, which we often write as $\tau$. To denote inaction or deadlock we write $\delta$.

For parallel composition we have the merge $\parallel$ which interleaves and/or synchronises multiactions. A communication operator $\Gamma$ allows explicit specification of (two or more) actions that communicate with each other (e.g. $\Gamma_{\{a|b\rightarrow c\}}(\langle a, b, b \rangle + \langle a \rangle)$), meaning that $a$ and $b$ communicate to $c$ in $\langle a, b, b \rangle + \langle a \rangle$, is equivalent to $\langle c, b \rangle + \langle a \rangle$.

To limit the behaviour of a process, it has been common to define which actions are not allowed. However, the number of multiactions we want to prohibit can increase exponentially with the number of parallel processes; putting $n$ actions in parallel results in $2^n - 1$ different multiactions. Therefore we added a restriction operator $\nabla$ that specifies precisely which multiactions are allowed, by a set $V$ of action sequences (e.g. $V = \{a\}$ or $V = \{b|c,d,c|e\}$). If one wishes that in the parallel composition of $a, b$ and $c$ action $a$ does not execute synchronised with another action and $b$ and $c$ must synchronise, one can write $\nabla_{I(a,b|c)}(a \parallel b \parallel c)$, which behaves as $a \cdot (b, c) + (b, c) \cdot a$.

The blocking operator $\partial_H$ (commonly referred to as encapsulation operator) prohibits actions in its set parameter $H$ from executing (e.g. $\partial_{I(a)}(a+b\cdot\langle a,c \rangle)$, which behaves as $b \cdot \delta$), the hiding operator $\tau_I$ makes actions in $I$ invisible (e.g. $\tau_{I(a)}(\langle a, b \rangle)$ becomes $\langle b \rangle$) and the renaming operator $\rho$ renames actions (e.g. $\rho_{I(a \rightarrow b)}(a)$ becomes $b$). Finally we have process references with which we can write definitions such as $P = a \cdot P$, which denotes the process that can do infinitely many $a$ actions.

Table 1, where $\mathcal{V} = \{a_1|\ldots|a_n : a_1, \ldots, a_n \in \mathcal{N}_A\}$ the set of possible action-name combinations, $\mathcal{C} = \{a_1|\ldots|a_n \rightarrow b : a_1, \ldots, a_n, b \in \mathcal{N}_A\}$ the set of possible communications and $\mathcal{R} = \{a \rightarrow b : a, b \in \mathcal{N}_A\}$ the set of all renamings, contains a summary of the above.

Note that this syntax allows one to write sets ($R$) that can contain elements with the same left hand side (e.g. $\{a \rightarrow b, a \rightarrow c\}$). This should not be possible as the meaning of these sets are meant to be functions. Therefore we put the restriction on this syntax that in the sets described by $R$ no left hand side of an element may be the same as the left hand side of another.
### Table 1

**LoCo Syntax**

<table>
<thead>
<tr>
<th>Single actions (from $\mathcal{N}_A$, also $\langle a \rangle$, $\langle b \rangle$, ...)</th>
<th>Multiactions (containing single actions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$, $b$, $c$, ...</td>
<td>$\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$, ...</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Deadlock/inaction</td>
</tr>
<tr>
<td>$\tau$</td>
<td>Silent step (also $\langle \rangle$)</td>
</tr>
<tr>
<td>_+ + _</td>
<td>Alternative composition</td>
</tr>
<tr>
<td>_· _</td>
<td>Sequential composition</td>
</tr>
<tr>
<td>_∥ _</td>
<td>Merge/Parallel composition</td>
</tr>
<tr>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>$P$, $Q$, ...</td>
<td>Process references</td>
</tr>
<tr>
<td>$\nabla_V(_)$</td>
<td>Restriction operator ($V \subseteq V$)</td>
</tr>
<tr>
<td>$\Gamma_C(_)$</td>
<td>Communication operator ($C \subseteq C$)</td>
</tr>
<tr>
<td>$\partial_H(_)$</td>
<td>Blocking operator (Encapsulation, $H \subseteq \mathcal{N}_A$)</td>
</tr>
<tr>
<td>$\tau_I(_)$</td>
<td>Hiding operator ($I \subseteq \mathcal{N}_A$)</td>
</tr>
<tr>
<td>$\rho_R(_)$</td>
<td>Renaming operator ($R \subseteq R$)</td>
</tr>
<tr>
<td>$P =$</td>
<td>Process definition (with $P$ a process reference)</td>
</tr>
</tbody>
</table>

For the set $C$ a similar restriction holds. Specifically, left hand sides must be disjoint, meaning that $\{a|b \to c, d|b \to e\}$ is not allowed as $b$ occurs in both left hand sides. Otherwise communication applied to $\langle a, b, d \rangle$ could result in either $\langle c, d \rangle$ or $\langle a, e \rangle$, which we consider to needlessly complicate the communication (such behaviour can better be explicitly modelled by the user, in our opinion).

Some additional notation is used in this document for ease of reading. Instead of writing the sequence of terms $t_1, t_2, \ldots, t_n$ (e.g. the actions in a multiaction) we often write $t$. We also write $\alpha, \beta$ etc. instead of the multiactions like $\langle a \rangle$.

Now that we have introduced our syntax, we will have a look at some examples of LoCo processes. Process $M = (\text{coin} \parallel \text{button}) \cdot \text{product} \cdot M$ models a simple vendor machine that waits for a user to insert a coin and press a button (in any order) and then gives a product. This process has the same behaviour as $M' = (\text{coin} \cdot \text{button} + \text{button} \cdot \text{coin} + \langle \text{coin}, \text{button} \rangle) \cdot \text{product} \cdot M'$, where the third alternative indicates that it is possible to insert a coin at the same time as pressing the button. Another example is $\nabla_{\{a,b\}}(\tau_{\{s_{ab}\}}(\Gamma_{\{s_a|s_b \to s_{ab}\}}(A \parallel B)))$, with $A = a \cdot s_a$ and $B = s_b \cdot b$, which models two separate processes $A$ and $B$ that have to synchronise such that $a$ happens before $b$. This process has the same behaviour as $a \cdot b$ (in branching bisimulation semantics [25]).

### 3 Operational Semantics

For the definition of the semantics of LoCo we need some auxiliary notation (for precise definitions, see [28]). First of all, we will use the set of all multiactions $\mathcal{A} = \{\langle a \rangle : a \in \mathcal{A}\}$ and we use $|$ to combine multiactions (i.e. $\langle a \rangle | \langle b \rangle$ is just $\langle a, b \rangle$). To be able to reason about terms of our language, we introduce some sets and notations. The set $V_P$, with elements $x, y, \ldots$, consists of process variables. For
the set of LoCo terms (described by) \( T_P \) we have elements \( t, u, \ldots \) and process-closed terms \( p, q, \ldots \) in \( T_{pc} \) (terms that do not have any process variables or references in them).

In Table 2 we give the operational semantics of LoCo. We use the standard transition relations \( \rightarrow \) and \( \rightarrow \checkmark \), and assume we have a set \( E \) containing process definitions. Note that the semantics of the operators \( \nabla_V, \Gamma_C, \partial_H, \tau_I \) and \( \rho_R \) is given separately.

<table>
<thead>
<tr>
<th>( \frac{t \to \checkmark}{a} )</th>
<th>( \frac{t \to t'}{\alpha} )</th>
<th>( \frac{\tau}{\emptyset} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t + u )</td>
<td>( u + t )</td>
<td>( t \cdot u )</td>
</tr>
<tr>
<td>( t \parallel u )</td>
<td>( u \parallel t )</td>
<td>( t \parallel u )</td>
</tr>
<tr>
<td>( t \parallel u, \alpha )</td>
<td>( u \parallel t, \alpha )</td>
<td>( t \parallel u, \alpha )</td>
</tr>
<tr>
<td>( t \parallel u, \alpha )</td>
<td>( u \parallel t, \alpha )</td>
<td>( t \parallel u, \alpha )</td>
</tr>
<tr>
<td>( \frac{P \to \checkmark}{P \to t} )</td>
<td>( P \to t' )</td>
<td></td>
</tr>
</tbody>
</table>

Table 2

LoCo Semantics

The definition of the operators \( \nabla_V, \Gamma_C, \partial_H, \tau_I \) and \( \rho_R \) in the semantics requires some functions that perform the needed transformations or checks on multiactions. To start with the blocking operator \( \partial_H \), we need to test whether or not an action in \( H \) occurs in a multiaction. We do this by converting such a multiaction \( \alpha \) to a set \( \alpha \{ \} \) (i.e. if \( \alpha \) is \( \langle a, b, c, c \rangle \), then \( \alpha \{ \} \) will be \( \{a, b, c\} \) and then taking the intersection of \( \alpha \{ \} \) and \( H \), which gives all actions that occur in both \( \alpha \) and \( H \).

Restriction operator \( \nabla_V \) needs to check whether or not a given multiaction occurs in its set \( V \) (or is \( \tau \) or \( \emptyset \), which is always allowed). Because the set \( V \) does not contain multiactions but action sequences of the form \( a|a|b \), we convert \( V \) to \( V\{ \} \), which is defined by \( \{a_1, \ldots, a_n : a_1|\ldots|a_n \in V\} \).

To apply renaming, defined by \( R \) in \( R\{ \} \), to a multiaction \( \alpha \), we write \( R \bullet \alpha \). For example, if we apply \( R = \{a \to b, b \to a\} \) to \( \langle a, b, b, c \rangle \), we get \( R \bullet \langle a, b, b, c \rangle = \langle R(a), R(b), R(b), R(c) \rangle = \langle b, a, a, c \rangle \).

With hiding \( \tau_I \), we need to remove all actions in \( I \) from multiactions. We introduce a special function \( \theta_I \) for this purpose. Thus, \( \theta_{\{a,b\}}(\langle d, a, b, a, c \rangle) \) would result in \( \langle d, c \rangle \) and \( \theta_{\{a\}}(\langle a, a \rangle) \) in \( \emptyset \) (or \( \tau \)).
For the communication operator we need a somewhat more complex definition. We introduce a communication function \( \gamma_C \) that takes a multiaction and finds all occurrences of left hand sides in \( C \) and replaces those occurrences with the corresponding right hand side. To be somewhat more precise (note that we implicitly use the commutativity of bags in this definition):

\[
\gamma_C(\langle a_1, \ldots, a_n \rangle | \alpha) = \langle c \rangle | \gamma_C(\alpha) \quad \text{if} \quad a_1 | \ldots | a_n \rightarrow c \in C
\]

\[
\gamma_C(\alpha) = \alpha \quad \text{if there is no such} \quad a_1, \ldots, a_n
\]

So, if \( C = \{ a|b \rightarrow a, c|d \rightarrow b \} \), then \( \gamma_C(\langle a, a, a, b, b, c, c, d \rangle) \) results in \( \langle a, a, a, b \rangle \).

Note that the extra condition on \( C \) (discussed in Section 2) is required to make \( \gamma \) a true function. That is, \( \gamma_C(\alpha) \) is a unique multiaction, which follows from the fact that if an action can participate in two possible communications, then these have to be equivalent due to the restriction on \( C \) (e.g. \( a \) can communicate with the first or the second \( b \) of \( \langle a, b, b \rangle \), but either way the effect is the same).

With these auxiliary functions the semantics of LoCo is completed with the rules from Table 3.

To be able to compare and calculate with processes, we need to know when two processes are equal (i.e. have the same behaviour). We therefore use the default notion of (strong) bisimilarity [20,24] and write \( LoCo \models p \leftrightarrow q \) (or just \( p \leftrightarrow q \)) to denote that \( p \) and \( q \) are (strongly) bisimilar. And as the rules in Table 2 and Table 3 are in the path format [5], we have that bisimulation \( \leftrightarrow \) is a congruence with respect to all operators.

4 Axioms

We introduce the axiomatisation in Table 4 for the semantics given in the previous section. The axioms allow for more straightforward reasoning in certain cases. It also shows that LoCo has a reasonably elegant algebraic structure similar to those of other process algebras.

If we can derive \( q \) from \( p \) with the axioms (in the ordinary equational sense), we
Table 4  
LoCo Axioms

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MA1</td>
<td>( a \vdash { a } )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MA2</td>
<td>( \tau \vdash { } )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MA3</td>
<td>( { a, b } \vdash { b, a } )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>( x + y \vdash y + x )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>( x + (y + z) \vdash (x + y) + z )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>( x + x \vdash x )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>( (x + y) \cdot z \vdash x \cdot z + y \cdot z )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>( (x \cdot y) \cdot z \vdash x \cdot (y \cdot z) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>( x + \delta \vdash x )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td>( \delta \cdot x \vdash \delta )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM1</td>
<td>( x \parallel y \vdash x \parallel y \parallel x + x \parallel y )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM2</td>
<td>( \alpha_\delta \parallel x \vdash \alpha_\delta \cdot x )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM3</td>
<td>( \alpha_\delta \cdot x \parallel y \vdash \alpha_\delta \cdot (x \parallel y) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM4</td>
<td>( (x + y) \parallel z \vdash x \parallel z + y \parallel z )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM5</td>
<td>( \alpha_\delta \cdot x \parallel \beta_\delta \vdash (\alpha_\delta \beta_\delta) \cdot x )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM6</td>
<td>( \alpha_\delta \cdot (\beta_\delta \cdot x) \vdash (\alpha_\delta \beta_\delta) \cdot x )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM7</td>
<td>( \alpha_\delta \cdot x \parallel (\beta_\delta \cdot y) \vdash (\alpha_\delta \beta_\delta) \cdot (x \parallel y) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM8</td>
<td>( (x + y) \parallel z \vdash x \parallel z + y \parallel z )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM9</td>
<td>( x \parallel (y + z) \vdash x \parallel y + x \parallel z )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM10</td>
<td>( \alpha_\delta \parallel \beta_\delta \vdash \alpha_\delta \beta_\delta )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD1</td>
<td>( \tau_I (\delta) \vdash \delta )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD2</td>
<td>( \tau_I (\alpha) \vdash \theta_I (\alpha) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD3</td>
<td>( \tau_I (x + y) \vdash \tau_I (x) + \tau_I (y) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD4</td>
<td>( \tau_I (x \cdot y) \vdash \tau_I (x) \cdot \tau_I (y) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RD1</td>
<td>( \rho_R (\delta) \vdash \delta )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RD2</td>
<td>( \rho_R (\alpha) \vdash R \cdot \alpha )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RD3</td>
<td>( \rho_R (x + y) \vdash \rho_R (x) + \rho_R (y) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RD4</td>
<td>( \rho_R (x \cdot y) \vdash \rho_R (x) \cdot \rho_R (y) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GD1</td>
<td>( \Gamma_C (\delta) \vdash \delta )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GD2</td>
<td>( \Gamma_C (\alpha) \vdash \gamma_C (\alpha) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GD3</td>
<td>( \Gamma_C (x + y) \vdash \Gamma_C (x) + \Gamma_C (y) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GD4</td>
<td>( \Gamma_C (x \cdot y) \vdash \Gamma_C (x) \cdot \Gamma_C (y) )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With \( a, b \in N_A, \alpha, \beta \in \mathbb{A}, \alpha_\delta, \beta_\delta \in \mathbb{A} \cup \{ \delta \} \) and \( x, y \in V_P \).

write \( \text{LoCo} \vdash p \equiv q \) (or just \( p \equiv q \)). And if we do so, we obviously want the same to hold for bisimulation, which is stated in the following theorem.

**Theorem 4.1** Let \( p, q \in T_{pc} \). The axiomatisation of LoCo is sound with respect to strong bisimulation (i.e. \( \text{LoCo} \vdash p \equiv q \Rightarrow \text{LoCo} \vdash p \equiv q \)).

**Proof.** This proof is very straightforward and therefore not given here. However, it can be found in [28].

The other way around is also a desired property.

**Theorem 4.2** Let \( p, q \in T_{pc} \). The axiomatisation of LoCo is complete with respect to strong bisimulation (i.e. \( \text{LoCo} \vdash p \equiv q \Rightarrow \text{LoCo} \vdash p \equiv q \)).

**Proof.** The full proof can be found in [28].

Because we also wish to use recursive processes, we will (at least) need some extension to the axioms given in Table 4. We extend \( \vdash \) with the following rule:
\[
P = t \in E \\
P \models t
\]
From the rules for process definitions in Section 3 the soundness of this rule clearly follows.

5 Abstraction

If we want \( \tau \) (or empty multiset \( \langle \rangle \)) to be a “real” silent step, we want to be able to remove \( \tau \) where its presence cannot be determined. Due to the nature of multiactions, it is already the case that if \( \tau \) is synchronised with (i.e. executed at the same time as) some action \( \alpha \), the \( \tau \) “disappears” (applying axiom \( MA2 \) to \( \tau|\alpha \) gives \( \langle \rangle|\alpha \) and with \( CM10 \) we get just \( \alpha \)). A multiaction is not just a multiset of actions, but a multiset of observable actions. There is always the possibility that unobservable actions happen at the same time. Note that this behaviour of the synchronisation operator is not to be confused with the behaviour of the communication operator in, for example, ACP, which would deadlock (because \( \tau \) cannot communicate). Also note that this behaviour is the same as seen in, for example, the Petri Box Calculus and similar to that in SCCS (apart from the different interpretation of this multiaction identity).

However, this is not the only place where we wish to hide these silent steps. In these cases, strong bisimulation is no longer suitable and we will use (rooted) branching bisimulation \([25,26]\). For this form of equivalence, we also need a matching (i.e. sound and complete) axiomatisation. Fortunately, the axioms given before are still sound, but to make the axiomatisation complete again (i.e. to have axioms that reflect the behaviour of \( \tau \)) it is sufficient to add the following two axioms, as in \([12]\).

\[
T1 \quad x \cdot \tau \equiv x \\
T2 \quad x \cdot (\tau \cdot (y + z) + y) \equiv x \cdot (y + z)
\]

Theorem 5.1 The axiomatisation of LoCo is sound with respect to (rooted) branching bisimulation.

Proof. In \([28]\) one can find the soundness proofs of the axioms \( T1 \) and \( T2 \). The other axioms of LoCo, which have already been proven to be sound with respect to \( \equiv \), do not need to be proven again, as \( \equiv \subset \equiv_{rb} \) holds.

Theorem 5.2 The axiomatisation of LoCo is complete with respect to rooted branching bisimulation.

Proof. The proof is similar to that in \([12]\).

6 Data

In many systems, data plays an essential role and is therefore a necessity for any practical language. We add data by adding parameters to actions. So, instead of actions \( a, b \) etc. we now also have actions like \( a(1), b(true, [c_0, c_1(1,2), c_0]) \) and
b(4, error). The precise data expressions that are allowed as parameters are defined by a data algebra \( \mathcal{A} \). All we need to know about it is that it contains a number of data types, one of which is the boolean type (with the default constants \( t, f \) and operators). Detailed requirements are given in [27], as well as a more detailed definition of syntax and semantics with data.

We also need the summation \( \sum_{d:D} p \), the conditional operator \( b \rightarrow p \) and data parameters for process references. The behaviour of \( \sum_{d:D} p \) is the same as the alternative composition of all \( p[e/d] \) (i.e. \( p \) with every unbounded occurrence of \( d \) replaced by \( e \)), for each \( e \) in data type \( D \). Conditional \( b \rightarrow p \) behaves as \( p \) or deadlock if the boolean condition \( b \) is equivalent to \( t \) or \( f \), respectively. With process references with data we can write definitions such as

\[
P(n : \mathbb{N}) = \sum_{m : \mathbb{N}} m < n \rightarrow a(m),
\]

meaning that, for example, process reference \( P(5) \) is equivalent to \( a(0) + a(1) + a(2) + a(3) + a(4) \).

In short, the extensions to our syntax is as follows:

- \( a(\ldots), b(\ldots), \ldots \) Single action with data parameters
- \( \sum_{d:D} - \) Summation over variable \( d \) of type \( D \)
- \( \alpha \rightarrow - \) Conditional operator
- \( P(\ldots), Q(\ldots), \ldots \) Process references with data parameters
- \( P(d : D, \ldots) = - \) Process definition with parameters

These extensions must also be reflected in the semantics and axioms. Because of the fact that actions can now have data parameters, the semantics of operators \( \nabla_V \), \( \partial_H \), \( \tau_I \) and \( \rho_R \) must be reformulated such that data will not affect their behaviour (i.e. data is ignored). For the communication function we will only mention that, with data, we wish actions can only communicate if, and only if, they have equivalent parameters. Also, the rules for single actions and process references must be extended to include data parameters. As these changes are rather trivial, we will not give them here.

The additional rules and axioms are as given in Table 5 and Table 6. Note that we assume to have capture avoiding substitution and alpha conversion in \( \alpha \).

**Table 5**

Additional LoCo Semantics for Data

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{t \alpha \rightarrow \checkmark}{b \rightarrow t \alpha \rightarrow \checkmark} \mathcal{A} \models b )</td>
<td>( b \rightarrow t \alpha \rightarrow \checkmark )</td>
</tr>
<tr>
<td>( \frac{t \alpha \rightarrow \checkmark}{b \rightarrow t \alpha \rightarrow t'} \mathcal{A} \models b )</td>
<td>( b \rightarrow t \alpha \rightarrow t' )</td>
</tr>
<tr>
<td>( \frac{t[e/d] \alpha \rightarrow \checkmark}{\sum_{d:D} t \alpha \rightarrow \checkmark} \mathcal{A} \models e \in D )</td>
<td>( \sum_{d:D} t \alpha \rightarrow \checkmark )</td>
</tr>
<tr>
<td>( \frac{t[e/d] \alpha \rightarrow \checkmark}{\sum_{d:D} t \alpha \rightarrow \checkmark} \mathcal{A} \models e \in D )</td>
<td>( \sum_{d:D} t \alpha \rightarrow t' )</td>
</tr>
</tbody>
</table>

Of course, also with these extensions we want to have a sound and complete axiomatisation, as stated in the following theorems. Note that the completeness of the axiomatisation now depends on the completeness of derivability in the data algebra \( \mathcal{A} \). As we do not explicitly consider this algebra, we say that our axiomatisation is relatively complete. This means that it is complete if we have completeness of derivability in \( \mathcal{A} \).
Table 6
Extra LoCo Axioms for Data

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C1$</td>
<td>$t \rightarrow x \equiv x$</td>
</tr>
<tr>
<td>$SUM1$</td>
<td>$\sum_{d:D} x \equiv x$</td>
</tr>
<tr>
<td>$C2$</td>
<td>$f \rightarrow x \equiv \delta$</td>
</tr>
<tr>
<td>$SUM3$</td>
<td>$\sum_{d:D} p \equiv \sum_{d:D} p + p[e/d] \text{ with } e \in D$</td>
</tr>
<tr>
<td>$V6$</td>
<td>$\nabla V(\sum_{d:D} p) \equiv \sum_{d:D} \nabla V(p)$</td>
</tr>
<tr>
<td>$SUM4$</td>
<td>$\sum_{d:D} (p + q) \equiv \sum_{d:D} p + \sum_{d:D} q$</td>
</tr>
<tr>
<td>$D6$</td>
<td>$\partial H(\sum_{d:D} p) \equiv \sum_{d:D} \partial H(p)$</td>
</tr>
<tr>
<td>$SUM5$</td>
<td>$(\sum_{d:D} p) \cdot y \equiv \sum_{d:D} (p \cdot y)$</td>
</tr>
<tr>
<td>$T6$</td>
<td>$\tau I(\sum_{d:D} p) \equiv \sum_{d:D} \tau I(p)$</td>
</tr>
<tr>
<td>$SUM6$</td>
<td>$(\sum_{d:D} p)[y] \equiv \sum_{d:D} (p[y])$</td>
</tr>
<tr>
<td>$R6$</td>
<td>$\rho R(\sum_{d:D} p) \equiv \sum_{d:D} \rho R(p)$</td>
</tr>
<tr>
<td>$SUM7$</td>
<td>$(\sum_{d:D} p)[y] \equiv \sum_{d:D} (p[y])$</td>
</tr>
<tr>
<td>$G6$</td>
<td>$\Gamma C(\sum_{d:D} p) \equiv \sum_{d:D} \Gamma C(p)$</td>
</tr>
<tr>
<td>$SUM7'$</td>
<td>$x(\sum_{d:D} q) \equiv \sum_{d:D} (x</td>
</tr>
</tbody>
</table>

With $x, y \in V_P$ and $p, q \in T_{pc}$.

**Theorem 6.1** The axiomatisation of LoCo with data is sound with respect to (rooted) branching bisimulation.

**Proof.** This proof is very straightforward and therefore not given here. However, it can be found in [27].

**Theorem 6.2** The axiomatisation of LoCo with data (without $T1$ and $T2$) is relatively complete with respect to strong bisimulation.

**Proof.** This proof is similar to completeness proof in [18]. Also, the full proof can be found in [27].

**Theorem 6.3** The axiomatisation of LoCo with data is relatively complete with respect to rooted branching bisimulation.

**Proof.** The proof is similar to that in [12].

7 Examples

To illustrate the use of LoCo, we look at the following examples.

7.1 Petri net

The (coloured) Petri net in Figure 2 describes a little system that takes tokens (natural numbers) from place $P_1$ one at a time, performs a calculation on the token, and places it in $P_4$. We verify this behaviour by considering an interpretation of this Petri net in LoCo. We assume a true concurrency semantics for the Petri net. Although an interleaving semantics would make no difference here due to the structure of the example, the translation to LoCo itself does not put this restriction on the system.

Plates are modelled by recursive processes parameterised with the contents of such a place. For places $P_1$, $P_2$, $P_3$ and $P_4$ this is a bag of natural numbers and for place $Q$ a bag of $\mathbb{I}$ (with $\mathbb{I} = \{1\}$). Transitions are modelled by recursive processes without parameters (as transitions are memoryless) consisting of a single multiaction. Actions $r_x$ and $s_x$ are used to model the receiving respectively sending
of a token by a place or transition \( X \). These actions correspond to the incoming and outgoing arrows, respectively. If there are two such incoming or outgoing arrows, the actions are labelled with the corresponding number from Figure 2. Thus transition \( \text{Enter} \) has an action \( r^1_{\text{enter}} \) with which he can receive a token from place \( P_1 \).

The whole system is a parallel composition of the places and transitions enclosed by communication (modelling the links between places and transitions), hiding and restriction. Note that places \( P_1 \) and \( P_4 \) are enclosed by a blocking operator because they do not have an incoming or outgoing link, respectively. Also note that the behaviour of places is somewhat restricted here for simplicity; normally it would be possible to atomically take several tokens from one place, as well as putting new ones in it.

\[
P_i(b : \text{Bag}(\mathbb{N})) = \sum_{n \in \mathbb{N}} r_{p_i}(n) \cdot P_i(b \cup \{n\}) + \sum_{n \in \mathbb{N}} n \in b \rightarrow s_{p_i}(n) \cdot P_i(b \setminus \{n\})
\]

\[
Q(b : \text{Bag}()) = \sum_{i=1} r_q(i) \cdot Q(b \cup \{i\}) + \sum_{i=1} (i \in b) \rightarrow s_q(i) \cdot Q(b \setminus \{i\})
\]

\[
\text{Enter} = \sum_{n \in \mathbb{N}} r^1_{\text{enter}}(n) | r^2_{\text{enter}}(1) | s_{\text{enter}}(n) \cdot \text{Enter}
\]

\[
\text{Calc} = \sum_{n \in \mathbb{N}} r_{\text{calc}}(n) | s_{\text{calc}}(f(n)) \cdot \text{Calc}
\]

\[
\text{Exit} = \sum_{n \in \mathbb{N}} r_{\text{exit}}(n) | s^1_{\text{exit}}(n) | s^2_{\text{exit}}(1) \cdot \text{Exit}
\]

\[
PN(\text{in} : \text{Bag}(\mathbb{N}), \text{out} : \text{Bag}(\mathbb{N})) = \nabla \{c\}(\Gamma\{r^1_{\text{enter}} \rightarrow c, s_q | r^2_{\text{enter}} \rightarrow c, s_{\text{enter}} | r_{\text{calc}} \rightarrow c, s_{\text{calc}} | r_{\text{exit}} \rightarrow c, s^1_{\text{exit}} | r_{\text{exit}} \rightarrow c, s^2_{\text{exit}} \rightarrow c\} \left( \partial\{r_{p_1}\}(P_1(\text{in})) \parallel P_2(\emptyset) \parallel P_3(\emptyset) \parallel \partial\{s_{p_4}\}(P_4(\text{out})) \parallel Q(\{1\}) \parallel \text{Enter} \parallel \text{Calc} \parallel \text{Exit} \right)
\]

With basic expansion of the parallel operators and application of communication, hiding and restriction, we get the following result.

\[
PN(\text{in} : \text{Bag}(\mathbb{N}), \text{out} : \text{Bag}(\mathbb{N})) = \sum_{n : \mathbb{N}} n \in \text{in} \rightarrow \tau \cdot PN(\text{in} \setminus \{n\}, \text{out} \cup \{f(n)\})
\]

As one can see, and could have expected, the behaviour of the system is the
same as that of places $P_1$ and $P_3$ connected by transition $Calc$, which simply takes a number $n$ from $P_1$ and puts $f(n)$ in $P_4$.

### 7.2 Components

In Figure 3, a system $C$, which checks whether components $S_1$ and $S_2$ return the same result for a given input, is depicted. Both components $S_1$ and $S_2$ take an integer as input and return an integer as output. For computation they use components $Mul$ and $Plus$, that multiply and add two integers, respectively. In addition, $S_1$ also uses $One$, that can always return a 1 on its output. Component $Cmp$ takes two integers and returns $t$ if they are equal (and $f$ otherwise). Note that all computations occur instantaneous; a component produces output at the same time it takes it input.

![Fig. 3. A simple compositional system](image)

All components are straightforwardly implemented as follows. Incoming arrows of a component $X$ correspond to actions $r_X$ and outgoing arrows correspond to actions $s_X$. If a component has more that one of such actions, then they are numbered from top to bottom as seen in Figure 3. Note that the individual components’ specifications are completely self contained.

\[
\begin{align*}
One &= s_{one}(1) \cdot One \\
Mul &= \sum_{x \in \mathbb{Z}} \sum_{y \in \mathbb{Z}} r_{mul}^1(x)r_{mul}^2(y)s_{mul}(x + y) \cdot Mul \\
Plus &= \sum_{x \in \mathbb{Z}} \sum_{y \in \mathbb{Z}} r_{plus}^1(x)r_{plus}^2(y)s_{plus}(x + y) \cdot Plus \\
S_1 &= \nabla_{r_1[s_1]}(\rho_{s_{mul} \rightarrow s_{s_1}}(r_{mul}^1 \cdot r_{mul}^2 \cdot s_{mul} \cdot s_{plus} \cdot c)) \cdot \cdot \cdot (One \parallel Plus \parallel Mul)) \\
S_2 &= \nabla_{r_2[s_2]}(\rho_{s_{plus} \rightarrow s_{s_2}}(r_{plus}^1 \cdot r_{plus}^2 \cdot s_{mul} \cdot s_{plus} \cdot c)) \cdot \cdot \cdot (Mul \parallel Plus)) \\
Cmp &= \sum_{x \in \mathbb{Z}} \sum_{y \in \mathbb{Z}} r_{cmp}^1(x)r_{cmp}^2(y)s_{cmp}(x = y) \cdot Cmp \\
C &= \nabla_{r_c[s_c]}(\rho_{s_{cmp} \rightarrow s_{c}}(r_{cmp}^1 \cdot r_{cmp}^2 \cdot s_{mul} \cdot s_{plus} \cdot c)) \cdot \cdot \cdot (S_1 \parallel S_2 \parallel Cmp)) \\
\end{align*}
\]

We can derive the following for $S_1$, $S_2$ and $C$. 
\[ S_1 = \sum_{x \geq 0} r_{s_1}(x) | s_{s_1}(x \cdot (x + 1)) \cdot S_1 \]
\[ S_2 = \sum_{x \geq 0} r_{s_2}(x) | s_{s_2}(x \cdot x + x) \cdot S_2 \]
\[ C = \sum_{x \geq 0} r_c(x) | s_{cmp}(t) \cdot C \]

It is not possible to simply move the communication operators to the top level as would be required in language that have global communication instead of local communication. If one would do so, it is impossible for, say, component One to know that he is communicating with the right Plus component. At least one instance of both the Plus and Mul components need to be changed to avoid conflicts in action names.

Although in this example the consequences of global communication are not extremely problematic, with bigger systems this becomes quite bothersome. Also, components \( S_1 \) and \( S_2 \) are typically developed independently. This means that one has to change the internals of \( S_1 \) and \( S_2 \) in order to use them safely in one system with global communication. It is clear that this is contrary to the ideas of component-based modelling.

### 8 Conclusion

We have introduced the process algebra \( LoCo \) that is truly compositional due to its local communication operator and the use of multiactions. It has a formal syntax, semantics and a sound and complete axiomatisation. We have included two small examples to illustrate the compositionality of \( LoCo \) and the ease with which Petri nets can be modelled in it.

As this work is mainly a basis for the mCRL2 language, future work will be continued in this context. This includes the addition of time, formal translations of Petri nets to mCRL2 and adapting existing proof techniques (such as those used with \( \mu \)CRL, for example) to the new setting.

### References


Checking Equivalence for Reo Networks

Tobias Blechmann\textsuperscript{a,1} Christel Baier\textsuperscript{a,1}

\textsuperscript{a} Chair for Algebraic and Logical Foundations of Computer Science
Technical University of Dresden
Dresden, Germany

Abstract

Constraint automata have been used as an operational model for component connectors described in the coordination language Reo which specifies the cooperation and communication of the components by means of a network of channels. This paper addresses the problem of checking equivalence of two Reo networks. We present a compositional approach for the generation of a symbolic representation of constraint automata for Reo networks and report on an implementation that realizes a partitioning splitter technique for checking bisimulation equivalence for Reo networks. Using a special operator on our symbolic data structure enables efficient treatment of the rich labeled transitions in constraint automata. In order to show the power of this approach we then present some benchmarks.

Keywords: Reo, bisimulation, coordination, partitioning splitter, symbolic model checking

1 Introduction

Reo is a channel-based exogenous coordination language\cite{1}. It was invented to provide the glue code for describing how component instances communicate with each other and how they are coordinated. Since then it has been used to model many different complex systems. The Reo point of view is that a system consists of component instances which execute at different locations and communicate through connectors. The main idea is to define a small set of simple channels and their behavior. More complex connectors then can be constructed through composition of these simple channels. During design of such coordination protocols questions like whether two specifications have the same observable behavior or whether one is an abstraction of another one arise naturally and frequently. Therefore we use constraint automata which are a special kind of labeled transition systems introduced by\cite{2}. Constraint automata provide an operational semantics for Reo which nicely reflects its compositional channel-based approach. The model of constraint automata is equipped with operators which mimic the operations provided for Reo. This yields a compositional approach to construct the corresponding constraint automata to a given Reo circuit is provided.

\textsuperscript{1} The authors are supported by the EU-project Credo and the DFG-and-NWO-project SYANCO

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Notions of bisimulation equivalence for constraint automata yield a definition for the equivalence of constraint automata. This is e.g. important to replace a given Reo connector component by a functional equivalent, but simpler (or cheaper) one. Furthermore bisimulation equivalence can be used as specification formalism: Having a certain coordination mechanism in mind, it is often quite easy to depict an automaton that describes its permissible behavior. In this sense this automaton serves as a specification for the Reo circuit we want to design. Correctness can then be understood w.r.t. bisimulation equivalence. Algorithms to compute bisimulation equivalence classes provide a sound way to check for equivalence of two given constraint automata or Reo circuits.

In this paper we report an implementation of a bisimulation checker using a symbolic approach working on switching functions. One could also try to model the behavior of Reo circuits using well-known bisimulation model checking tools like [3] or [4]. But representing the compositional approach of Reo using standard tools would require to provide a semantics for Reo based on the process algebra operators and therefore leads to huge and rather artificial looking system descriptions. For us the two main challenges were (1) to find an efficient way to deal with the rich labeled transitions occurring in constraint automata and (2) to symbolically compute logical equivalence classes as this is needed for implementation of the partitioning splitter algorithm. We report on how to accomplish the composition of two symbolic representations of constraint automata in a very efficient way.

Then we introduce a pattern equivalence operator which allows to efficiently compute logical equivalence classes and show how it enables a symbolic implementation of the partitioning splitter algorithm. This leads to a tool which enables us to automatically and efficiently treat constraint automata with lots of states but rather few bisimulation equivalence classes. In order to measure the efficiency of our approach we report on some benchmark results.

This paper starts with a summary of the main concepts of Reo and constraint automata in section 2. Section 3 explains how the compositional approach to generate a constraint automaton from a given Reo circuit can work symbolically. Then section 4 explains the main features of our implementation and section 5 reports on our experimental results.

2 Reo and Constraint Automata

We briefly summarize the main concepts of Reo and constraint automata. For further information see [1] and [2]. In Reo every connector is constructed out of channels using the provided operators for channel composition. The simplest channel is a synchronous channel shortly called sync channel. It accepts data written to its source end when the same data can leave the channel at the same moment through its sink end. There are two basic operations used for composition called join and hiding. Join plugs to channel-ends together creating a node at the point of connection. To this node one can connect more channels via join afterwards. If more than one accepting channel-ends are connected to a node every incoming message is simultaneously written to all outgoing channels whenever all outgoing channels at the node are ready to accept data. Whenever more than one channel-end offers
data at a node a non-deterministic choice decides which data is taken and written to all outgoing channels. The hiding operation hides away one node which means that the data-flow occurring at this node cannot be observed from outside and no new channel-end can be connected to this node.

Constraint automata as introduced by [2] can serve as an operational semantics for Reo. Similar to other finite automata, constraint automata consist of states and a transition relation between those states. All transitions in constraint automata are labelled to describe the node where data enters a channel or is taken from a channel. The labeling also determines what data item is transferred. Constraint automata use a finite set $N$ of names. Roughly speaking the elements in $N$ are the nodes in the corresponding Reo circuit. Transitions in constraint automata are labeled with pairs consisting of a non-empty subset $N \subseteq N$ and a data constraint $g$. Data constraints are propositional formulas describing the data items that enable a transition in the constraint automata. These formulas are built out of atoms $d_A$ where $d \in \text{Data}$ a finite data domain and $A \in N$. To simplify the notations we assume that $\text{Data} = \{0, 1\}$. The atom $d_A$ stands for the interpretation ‘$d$ is written to or read from port/node $A$.’ Data constraints are given by the grammar $g ::= \text{true} \mid d_A \mid g_1 \lor g_2 \mid \neg g$ where $A \in N$ is a name and $d \in \text{Data}$. The boolean operators $\land$ (conjunction), $\oplus$ (exclusive or), $\to$ (implication), $\leftrightarrow$ (equivalence) can be derived as usual. We often use derived data constraints such as $d_A = d_B$ which means that on port A and port B the same element out of $\text{Data}$ is transferred. In the sequel we write $DC(N, \text{Data})$ for $N \subseteq \mathcal{N}$ and $N \neq \emptyset$ to denote the set of data constraints that use only atoms $d_A = d$ for $A \in N$. For all data constraints in the whole automaton we write $DC$ as an abbreviation for $DC(N, \text{Data})$.

**Definition 2.1 (Constraint Automata)** A constraint automaton is a tuple $A = (Q, N, \rightarrow, Q_0)$ where $Q$ is the set of states, $N$ is the set of nodes, $\rightarrow$ is a subset of $Q \times 2^N \times DC \times Q$ called the transition relation of $A$ and $Q_0 \subseteq Q$ is the set of initial states. We write $q \overset{N, g}{\rightarrow} p$ instead of $(q, N, g, p) \in \rightarrow$. $N$ is called the name-set and $g$ the guard of the transition. We require that $N \neq \emptyset$ and $g \in DC(N, \text{Data})$. $A$ is called finite iff $Q$ and $\rightarrow$ are finite.\

For an intuitive interpretation one can see the states as representations of the connector configurations and the transitions as the possible single-step behavior.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{constraint_automata.png}
\caption{Two Reo connectors and a constraint automata}
\end{figure}

\footnote{We demand $\text{Data}$ and $Q$ to be finite so here $\rightarrow$ is finite.}
In Figure 1 we present the dining philosophers example as shown in [5]. The left part of Figure 1 shows a Reo circuit which specifies the dining philosophers protocol for two philosophers and two chopsticks. Solid arrows stand for sync channels. To gain a constraint automaton that describes the behavior of this circuit we have to specify what happens inside of a PHIL and a CHOP component. The constraint automaton in the middle of the picture models the behavior of one PHIL component. The philosophers starts in state think and if this is possible writes \( T_0 \) which intuitively means picking one of the chopsticks next to him. After writing the message he enters state wait. If writing the message \( T_1 \) is possible this message is written and the component enters state eat. On the way back to state think the component then has to write two messages which release the chopsticks. What remains is to specify the behavior of the chopsticks. The right part of Figure 1 shows a Reo circuit for the chopsticks. We use a blocking FIFO1 channel (marked by a box in the middle of the channel) and join its sink end with one of the source ends of a synchronous drain (SYNCDRAIN) channel. The blocking FIFO1 channel behaves in a natural way. If its buffer is empty it accepts one data item. Afterwards it does not accept any write operations to its source end and waits for a read operation happening at the sink end. The SYNCDRAIN channel has two source ends and no sink end. It accepts any data item on one of its ends iff at the same moment another data item is written to the other end. As we are not interested in the communication taking place at the inner node we can hide it which is shown by the enclosing box. The semantics of the two atomic channels we used must again be specified using constraint automata. What remains is to show how we can mimic the join and the hiding operation on the automata level.

Due to the fact that in constraint automata there is no way to distinguish between incoming and outgoing ports we have to use two different operations to mimic the merge semantics of Reo nodes. On the automata level we first provide the product automata operator for joining a source node with another node (no matter what type). Then we use a merger automaton to simulate the merger behavior of multiple joined sink channel ends. This merger automaton can be regarded as a new primitive connector in the Reo circuit plugged before the old node which just performs a non-deterministic choice on its source ports and forwards the chosen data to the old node. Its semantics can be represented by a constraint automaton with only one state and self-loop transitions for every incoming channel-end.

Assume two Reo circuits with node-sets \( N_1 \) and \( N_2 \) and their corresponding automata are given. We want to perform a join operation for node-pairs \( (B_i, \bar{B}_i) \in N_1 \times N_2 \), \( i = 1, \ldots, k \) where, for any \( i \), at least one of the nodes \( B_i \) or \( \bar{B}_i \) is a source node. We assume that the nodes are renamed in a way that \( B_1 = \bar{B}_1, \ldots, B_k = \bar{B}_k \) holds and the two automata do not contain other common nodes. Thus we join all common nodes \( B \in N_1 \cap N_2 \). On the automata level the join operator (denoted \( \bowtie \)) is performed as shown in Definition 2.2.

**Definition 2.2 (Product automaton)** The product automaton for the two constraint automata \( A_1 = (Q_1, N_1, \rightarrow_1, Q_{0,1}) \) and \( A_2 = (Q_2, N_2, \rightarrow_2, Q_{0,2}) \) is \( A_1 \bowtie A_2 = (Q_1 \times Q_2, N_1 \cup N_2, \rightarrow, Q_{0,1} \times Q_{0,2}) \) where \( \rightarrow \) is defined by the following rules (let

---

\( ^3 \) One cannot take from the constraint automata whether this is a write or take operation but in the Reo circuit message directions are shown.
\[ q_1, p_1 \in Q_1 \text{ and } q_2, p_2 \in Q_2 \]:

\[
\begin{align*}
q_1 & \xrightarrow{N_1, g_1} p_1, q_2 \xrightarrow{N_2, g_2} p_2, N_1 \cap N_2 = N_2 \cap N_1 \\
\langle q_1, q_2 \rangle & \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2 \rangle \\
q_1 & \xrightarrow{N, g} p_1, N \cap N_2 = \emptyset \\
\langle q_1, q_2 \rangle & \xrightarrow{N, g} \langle p_1, q_2 \rangle
\end{align*}
\]

(1)

And a third rule symmetric to the second one.

The definition of product automaton reflects the possible synchronous and asynchronous behavior of two joined constraint automata. The first rule (1) describes the synchronous behavior. After joining both automata only a synchronous step is possible for transitions involving common nodes. The data transferred in this step has to match \( g_1 \wedge g_2 \). The second rule (2) and the symmetric one define asynchronous actions. If \( N \cap N_2 = \emptyset \) the first automaton can make a step and the second one remains in the current state.

The second operation on Reo circuits also has a counterpart on the constraint automata level. Hiding a node in a constraint automaton produces a new automaton in which data at the hidden node is no longer observable from the outside. For a transition \( q_1 \xrightarrow{\{A\}, d_A = d} q_2 \) which only relies on the hidden port \( A \) this means that the transition can always be taken in state \( q_1 \).

**Definition 2.3 (Hiding on constraint automata)** Let \( A = (Q, N, \rightarrow, Q_0) \) be a constraint automaton and \( C \in N \). After hiding \( C \) the resulting constraint automaton \( \exists C[A] = (Q, N \setminus \{C\}, \rightarrow_C, Q_{0,C}) \) is defined as follows. Let \( \rightsquigarrow^* \) be the transition relation such that \( q \rightsquigarrow^* p \) iff there exists a finite path

\[
q \xrightarrow{\{C\}, g_1} q_1 \xrightarrow{\{C\}, g_2} q_2 \xrightarrow{\{C\}, g_3} \ldots \xrightarrow{\{C\}, g_n} p
\]

where \( g_1, \ldots, g_n \) only depend on \( C \) and are satisfiable. The set \( Q_{0,C} \) of initial states is \( Q_{0,C} = Q_0 \cup \{p \in Q : q_0 \rightsquigarrow^* p \text{ for some } q_0 \in Q_0 \} \).

The transition relation \( \rightarrow_C \) is given by

\[
q \rightsquigarrow^* p, p \xrightarrow{N, g} e, \bar{N} = N \setminus \{C\} \neq \emptyset, \bar{g} = \exists C[g] \\
q \xrightarrow{\bar{N}, \bar{g}, C} e
\]

where \( \exists C[g] = \bigvee_{d \in \text{Data}} g[d_C / d] \). We write \( g[d_C / d] \) to denote the data constraint obtained by syntactically replacing all occurrences of \( d_C \) in \( g \) with \( d \). To be more precise we replace all atoms \( d_C = \bar{d} \) with \( \text{true} \) if \( d = \bar{d} \) and with \( \text{false} \) if \( d \neq \bar{d} \).

When constructing larger Reo circuits the question whether two circuits show the same observable behavior arise naturally. Also often it is easy to give a constraint automaton describing the desired behavior. Then comparing its behavior with the behavior of the constructed Reo circuit shows if the Reo circuit is a correct implementation. This is sometimes called a homogeneous approach to verification (model checking). There is a strong connection between this problem and the bisimilarity of constraint automata. An algorithm computing the bisimulation relation

\[ FACS'07 \text{ Pre-Proceedings} \]
therefore can be of great help when treating larger Reo circuits. We start with some notations which are needed to introduce bisimulation for constraint automata.

For a constraint automaton $A = (Q, N, \rightarrow, Q_0)$, $q \in Q$, $N \subseteq N$ and $P \subseteq Q$ we define $dc_A(q, N, P) = \bigvee \{ q : q \xrightarrow{N \eta} p \text{ for some } p \in P \}$

If the automaton is clear from the context we leave out the subscript and write $dc(q, N, P)$. Then $dc(N, P) = \bigvee_{q \in Q} dc(q, N, P)$ stands for the weakest transition using only $N$ ports from $q$ to an arbitrary chosen state in $P$. If no such transition exists $dc(q, N, P) = false$.

With this notations we can now define bisimulation for constraint automata.

Definition 2.4 (Bisimulation) Let $A = (Q, N, \rightarrow, Q_0)$ be a constraint automaton. An equivalence relation $R$ on $Q$ is called bisimulation for $A$ if for all pairs $(q_1, q_2) \in R$, all $R$-equivalence classes $P \subseteq Q/R$ and every $N \subseteq N$, $dc(q_1, N, P) \equiv dc(q_2, N, P)$ holds.

Two states are called bisimilar (or bisimulation-equivalent) iff there exists a bisimulation $R$ with $(q_1, q_2) \in R$. Intuitively bisimilar states are equal powerful with respect to their outgoing transitions.

We write $q_1 \sim q_2$ iff $q_1$ and $q_2$ are bisimilar and $q_1 \not\sim q_2$ iff they are not.

Two constraint automata $A_1$ and $A_2$ with the same set of names are bisimilar $(A_1 \sim A_2)$ if in the bisimulation equivalence classes for the disjoint union automaton $B = A_1 \cup A_2$ for every initial state of $A_1$ there exists a bisimilar initial state in $A_2$ and vice versa.

3 Symbolic Constraint Automata

The basis of our implementation is a definition of a symbolic representation based on switching functions \[7\] that supports an efficient treatment of join and product. For our implementation we then use ordered binary decision diagrams \[6][9][10][11\] to represent and manipulate those switching functions. Let $Z = \{z_1, \ldots, z_n\}$ be a finite set of boolean variables. An evaluation of $Z$ is a function $\eta : Z \rightarrow \{0, 1\}$ that assigns a value $\eta(z) \in \{0, 1\}$ to each variable $z \in Z$. $Eval(Z)$ identifies the set of all evaluations of $Z$. Let $\overline{\eta} = (a_1, \ldots, a_n) \in \{0, 1\}^n$ and $\overline{\eta} = (z_1, \ldots, z_n) \in \{0, 1\}^n$ such that $z_i, \ldots, z_j$ are distinct. Then $[\overline{\eta} = \overline{\eta}]$ denotes the evaluation $\eta \in Eval(Z)$ with $\eta(z_i) = a_j$, $j = 1, \ldots, n$. If $\overline{b} = (b_1, \ldots, b_n) \in \{0, 1\}^n$ and $\overline{\eta} = (z_1, \ldots, z_n) \in \{0, 1\}^n$ with pairwise different variables $z_{i_j}$ the evaluation $\eta [\overline{\eta} = \overline{b}] \in Eval(Z)$ agrees with $\eta$ on all variables and assigns $b_j$ to $z$ for $z \in \{z_{i_1}, \ldots, z_{i_n}\}$. A switching function is a function $f : Eval(Z) \rightarrow \{0, 1\}$. The set of all switching functions will be called $B(Z)$ or $B(z_1, \ldots, z_n)$. In the following we will use the common notions to denote operations on switching functions. Logical connectors like $\land$, $\neg$ and the derived operators for propositional logic have analog meanings for switching functions. To keep notations simple we use set operations like $\cap$ and $\cup$ with analog meaning for variable tuples like $\overline{b}$. Let $\overline{z}$ and $\overline{b}$ be as before and $f \in B(Z)$. The cofactor of $f$ related to $\overline{z}$ is defined by $f|_{\overline{z}=\overline{b}} \in B(Z)$ where $f|_{\overline{z}=\overline{b}}(\eta) = f(\eta [\overline{z} = \overline{b}])$. Let $f \in B(Z)$ and $z \in Z$ then $\exists z[f] \in B(Z)$ is given by $\exists z[f] = f|_{z=0} \lor f|_{z=1}$. To keep notations simple we write $\exists \overline{z}[f]$ as short form for $\exists z_1 \ldots \exists z_n[f]$ with $\overline{z} = (z_1, \ldots, z_n)$. Further
we use $f(z' \leftarrow z)$ to denote a new function which is derived from $f$ by replacing all $z$ variables by $z'$ variables.

For every constraint automaton $A = (Q, N, \rightarrow, Q_0)$ we can now define the corresponding symbolic representation $(\delta(\bar{x}, \bar{y}, \bar{N}, \bar{d}), \chi_{\text{init}}(\bar{x}))$.

After renaming we can assume a state set $Q = \{q_0, \ldots, q_n\}, n \geq 2$ and we identify element $q_i$ with its number $i$. Using $k = \lceil \log(n + 1) \rceil$ boolean variables $\bar{x} = (x_1, \ldots, x_k)$ we define $\chi_q(\bar{x}) = x_{b_1}^i \land \ldots \land x_{b_k}^i \in (B^k \rightarrow \mathbb{B})$ as symbolic representation for state $q_i$ where $b_j \in \{0, 1\}$, $x_j^0 = \neg x_j$, $x_j^1 = x_j, j = 1, \ldots, k$ and $i = \sum_{j=1}^k b_j \cdot 2^j$.

As we want to reason about transitions from one state to another we have to distinguish between start and end states. We introduce copies of the $x$-variables (that serve to encode the start state of transitions) and deal with variables $\bar{y} = (y_1, \ldots, y_k)$ to represent the end states of transitions.

With $\bar{N} = (A_1, \ldots, A_m)$ we use the names as boolean variables for our ports. Further we define boolean variables $\bar{d} = (d_{A_1}, \ldots, d_{A_m})$ to represent the data items observed at the corresponding ports.

For every name set $N \subseteq \mathcal{N}$ we define

$$\chi_N(\bar{N}) = \bigwedge_{A_i \in N} A_i \land \bigwedge_{A_i \in \mathcal{N} \setminus N} \neg A_i$$

As we restricted our data domain to $\{0, 1\}$ data constraints can be viewed as switching functions over $\bar{d}$.

**Definition 3.1 (Symbolic Constraint Automata)** A symbolic constraint automaton for a constraint automaton $A = (Q, N, \rightarrow, Q_0)$ is a tuple $A_{sym} = (\delta(\bar{x}, \bar{y}, \bar{N}, \bar{d}), \chi_{\text{init}}(\bar{x}))$ where

$$\delta(\bar{x}, \bar{y}, \bar{N}, \bar{d}) = \bigvee_{(q, N, g, p) \in \rightarrow} \chi_q(\bar{x}) \land \chi_N(\bar{N}) \land \chi_g(\bar{d}) \land \chi_p(\bar{y})$$

and $\chi_{\text{init}}(\bar{x}) = \bigvee_{q \in Q_0} \chi_q(\bar{x})$.

**Example 3.2** Figure 2 shows the constraint automata that was introduced in Figure 1 after renaming the state set. The corresponding switching function is given by:

$$\begin{align*}
\delta(\bar{x}, \bar{y}, \bar{N}, \bar{d}) &= (\neg x_1 \land \neg x_0 \land \neg y_1 \land \neg y_0 \land T_0) \lor (\neg x_1 \land x_0 \land y_1 \land \neg y_0 \land T_1) \\
&\lor (x_1 \land \neg x_0 \land y_1 \land y_0 \land F_1) \lor (x_1 \land x_0 \land \neg y_1 \land \neg y_0 \land F_0)
\end{align*}$$

![Fig. 2. Constraint automata for one PHIL component](image-url)
As in [2], the join operator is assumed to be preceded by an appropriate node-renaming such that all ports to be joined are named the same in both automata and therefore the variables for those ports and their constraints agree. Note that if A is a common node then A belongs the variable set of the symbolic representations of both automata.

**Definition 3.3 (Symbolic Product Automaton)** Given two symbolic constraint automata \( A_{sym} = (\delta_A(x_A, \overline{x}_A, N_A, \overline{N}_A), \chi_{init_A}(\overline{x}_A)) \) and \( B_{sym} = (\delta_B(x_B, \overline{y}_B, N_B, \overline{d}_B), \chi_{init_B}(\overline{y}_B)) \) with \( \overline{x}_A \cap \overline{x}_B = \emptyset \) and \( \overline{y}_A \cap \overline{y}_B = \emptyset \) the product automaton \( C_{sym} = A_{sym} \bowtie B_{sym} \) is defined as \( C_{sym} = (\delta_C(x_C, \overline{y}_C, N_C, \overline{d}_C), \chi_{init_C}(\overline{x}_C)) \) with \( x_C = x_A \cup \overline{x}_B, \overline{y}_C = y_A \cup \overline{y}_B, \delta_C(x_C, \overline{y}_C, N_C, \overline{d}_C) = \delta_{sync} \lor \delta_{async} \) and \( \chi_{init_C}(\overline{x}_C) = \chi_{init_A}(\overline{x}_A) \land \chi_{init_B}(\overline{y}_B) \). Where \( \delta_{async} = \delta_A \land \delta_B \) and

\[
\delta_{async} = \left( \delta_A \land \bigwedge_{A \in N_B} \neg A \land (\overline{x}_B \leftrightarrow \overline{y}_B) \right) \lor \left( \delta_B \land \bigwedge_{A \in N_A} \neg A \land (\overline{x}_A \leftrightarrow \overline{y}_A) \right)
\]

Where \( \overline{x} \leftrightarrow \overline{y} \) stands for \( \bigwedge_{i \in \{1,...,k\}} (x_i \leftrightarrow y_i) \).

This operation on switching functions is the exact counterpart of our product automata operation defined in Definition 2.2. Every fulfilling evaluation for \( \delta_{sync} \) is a fulfilling evaluation for \( \delta_A \) and \( \delta_B \). Thus the evaluation for \( N \) is such that common ports of both automata are equally active or passive. The conjunctive combination of the constraints \( g \) belonging to the transitions is given by the conjunctive combination of \( \delta_A \) and \( \delta_B \).

We obtain the asynchronous transitions by evaluating the \( \delta_{async} \) function. The intuitive interpretation for the formula is \( A_{sym} \) makes a step which does not depend on ports in \( B_{sym} \). The formulas \( (\overline{x}_B \leftrightarrow \overline{y}_B) \) ensure that \( B_{sym} \) cannot make a step. This has to be done again with \( A_{sym} \) and \( B_{sym} \) switching their positions. By \( \delta_{sync} \lor \delta_{async} \) we then compute the union of the synchronous and asynchronous transitions. Thus we obtain a function representing all transitions which exist in the product automaton.

Like the product operation hiding can also be done on the symbolic representation. Because hiding relies on at least two steps we need another set of state variables \( \bar{z} \). Remember the definition of hiding on constraint automata. In the symbolic case we have to perform a fixpoint iteration in order to collapse multiple transitions that rely only on the port we want to hide. A pseudo code version of this algorithm is shown in Algorithm 1. Our implementation also provides another variant of hiding using a repeated squaring approach. It depends on the example which one of them is slightly faster. Let \( C \) be the port we want to hide. First we divide our system in two parts. \( \delta_P \) contains all transitions which are labeled only with \( \{C\} \). Analogously \( \delta_R \) represents all transition which are not in \( \delta_P \). The first loop then computes our new set of initial states by performing a reachability analysis from all initial states using transitions in \( \delta_P \) only. The second loop takes every two step transition \( q \xrightarrow{\{C\}} p \xrightarrow{N,g} s \) and adds a new transition \( q \xrightarrow{N,g} s \).

---

4 Hiding some nodes in our examples showed that you cannot expect the BDD representation of an automaton to be smaller after hiding.
there is a sequence of consecutive transitions like \( l \xrightarrow{\{C\}} q \xrightarrow{\{C\}} p \xrightarrow{N,g} s \) we first add \( q \xrightarrow{N,g} s \). The next iteration then adds \( l \xrightarrow{N,g} s \). The loop is continued until no new transition can be added. The last operation \( \delta = \delta_R \land \neg \delta_P \) deletes all transitions labeled with \( \{C\} \).

### Algorithm 1: Hiding for symbolic constraint automata

\[
\begin{align*}
\delta_P &= \delta \land C \land \bigwedge_{A \in \mathcal{N}\backslash\{C\}} \neg A ; \quad \text{\textit{// all transitions labeled } \{C\}} \\
\delta_R &= \exists C \exists d_C [\delta \land \neg \delta_P] ; \quad \text{\textit{// all other transitions}} \\
\delta_P &= \exists N \exists d [\delta_P] ; \quad \text{\textit{// remove labels}} \\
\text{repeat} & \quad \text{\textit{// compute new initial states}} \\
& \quad \chi_{\text{old}} := \chi_{\text{init}} ; \\
& \quad \chi_{\text{init}} := \chi_{\text{init}} \lor \exists x (\chi_{\text{init}} \land \delta_P) \{ x \leftarrow y \} ; \\
\text{until} & \quad \chi_{\text{init}} = \chi_{\text{old}} ; \\
\text{repeat} & \quad \text{\textit{// for } q \xrightarrow{\{C\}} p \xrightarrow{N,g} s \text{ add new transitions } q \xrightarrow{N,g} s} \\
& \quad \delta_O := \delta_R ; \\
& \quad \delta_R := \delta_P \land \delta_R \{ z \leftarrow y, y \leftarrow x \} ; \\
& \quad \delta_R := \exists y [\delta_R \{ y \leftarrow z \} ; \\
& \quad \delta_R := \delta_R \lor \delta_O ; \\
\text{until} & \quad \delta_O = \delta_R ; \\
\delta &= \delta_R \land \neg \delta_P ; \quad \text{\textit{// delete all } \{C\}-\text{transitions}} \\
\text{return} & \quad (\delta(x,y,N \backslash \{C\}, d \backslash \{d_C\}), \chi_{\text{init}}(x))
\end{align*}
\]

When reasoning about equivalence using bisimulation we need to regard two disjoint automata \( A \) and \( B \) as one big automaton. We call the resulting automaton \( C \) a disjoint union of two automata denoted by \( C = A \cup B \). For explicit representations with disjoint node sets the node set of the new automaton is the union of both node sets. The new transition relation is the union of both transition relations. The set of initial states consists of all states that are in the union of the two sets of initial states of the disjoint automata. The same operation on symbolic representations can be done but we have to pay attention to non-essential variables when putting the two automata together.

Symbolic representation using switching functions to encode transition relations and binary decision diagrams to represent those switching functions is a powerful method to handle large transition systems [7].

To obtain compact BDD-representations of constraint automata the variable ordering must be chosen carefully. A detailed description of our heuristics to determine a good variable ordering goes beyond the scope of this paper. We just mention that our heuristics attempt to put variables representing constraints close to variables representing states having transitions relying on those constraints.

### 4 Bisimulation

Checking whether two automata are bisimilar \( (A_1 \sim A_2) \) is known to be coNP-hard [2]. Therefore we cannot expect good performance for every instance when
computing bisimulation quotients. A well known way to compute bisimulation quotient for ordinary labeled transition systems is the algorithm by Kan- nelakis/Smolka [8]. A sketch of how to adapt this algorithm for constraint automata is presented in [2]. We start with some notation needed for this algorithm.

**Partition:** For a constraint automaton \( A = (Q, \mathcal{N}, \rightarrow, Q_0) \) a partition for \( Q \) stands for a set \( \Pi = \{P_1, \ldots, P_n\} \) of pairwise disjoint, non-empty subsets of \( Q \) such that \( Q = P_1 \cup \ldots \cup P_n \). For a partition \( \Pi = \{P_1, \ldots, P_n\} \) of \( Q \) we call the elements \( P_i \) blocks. A super-block then denotes a non-empty union of blocks in \( \Pi \). A splitter denotes a pair \((N, P)\) consisting of a non-empty subset \( N \) of \( \mathcal{N} \) and a super-block \( P \) for \( \Pi \).

**Refine:** Let \( \Pi \) be a partition for \( Q \), \((N, P)\) a splitter for \( \Pi \) and \( B \) a block for \( Q \). For two states \( q_1, q_2 \) we then define the equivalence \( \equiv_{(N,P)} \) such that

\[
q_1 \equiv_{(N,P)} q_2 \text{ iff } dc(q_1, N, P) = dc(q_2, N, P)
\]

Then we define \( \text{Refine}(B, N, C) = B/\equiv_{(N,P)} \). A block \( B \) is called stable with respect to \((N, P)\) if \( \text{Refine}(B, N, P) = \{B\} \). For a refinement of the whole partition we write \( \text{Refine}(\Pi, N, P) = \bigcup_{B \in \Pi} \text{Refine}(B, N, P) \)

The idea of the partitioning splitter algorithm is to construct a sequence \( \Pi_0, \Pi_1, \ldots, \Pi_k \) of partitions with \( \Pi_{i+1} \) is strictly finer than \( \Pi_i \) but coarser than the bisimulation quotient \( Q/\sim \). For a finite set \( Q \) we get \( \Pi_k = Q/\sim \) for some \( k \leq |Q| \). A brief sketch of this algorithm is shown in Algorithm 2. The most critical part is the calculation of the logical equivalence classes of the data constraints and the refinement of the actual block according to those classes. This operation is performed on our symbolic representation using the following ideas. First we choose an arbitrary state \( q \) out of \( B \). Then we select its transitions to the current splitter. The function describing the labeling of this transition is the pattern function \( p \) for the pattern equivalence operator. With \( Z' = \overline{N} \cup d \) we then obtain a function representing all states that are in the same refined block as \( q \). We remove these states from the current block and start with a new arbitrarily chosen state if one is left.

We implemented a symbolic version of Algorithm 2 based on ordered binary decision diagrams (OBDDs) in order to estimate its efficiency for constraint automata. Note that our approach is not restricted to using OBDDs. For this partitioning splitter algorithm we have to compute logical equivalence classes with respect to the data constraints. This cannot be done by the standard operators provided by an OBDD library. The main idea of our approach is to arbitrarily choose a state and regard its outgoing transitions as representative for one equivalence class. What remains is to efficiently compute the class belonging to this representative. For this purpose we define a special pattern equivalence operator on switching functions in order to find logical equivalence classes. First we start with a function \( \text{getAssignment}(\tilde{f}) \) which chooses an arbitrary fulfilling evaluation for \( \tilde{f} \) and returns a switching function that is true for this evaluation and false for all other evaluations.\(^5\) This function can be used to select one state and its transitions. Using an existential quantification we then compute the representative for the logical equivalence class. It remains to explain how to compute the switching function describing the set of all states in

\(^5\) At the OBDD level this means a representative for a path to the 1-sink
Algorithm 2: Partitioning splitter algorithm

\[ \Pi := Q; \]
\[ \text{Splitters} := \{(N, Q) : N \subseteq \mathcal{N}, \bigvee_{q \in Q} \text{dc}(q, N, Q) \neq \text{false}\}; \]

\[ \textbf{while } \text{Splitters} \neq \emptyset \textbf{ do} \]
\[ \text{choose}(N, P) \in \text{Splitters} \text{ and remove } (N, P) \text{ from } \text{Splitters}; \]
\[ \text{forall } B \in \Pi \text{ do} \]
\[ \text{find equivalence classes } D_1, \ldots, D_r \text{ of } \text{dc}(q, N, P), \ q \in B; \]
\[ /\!\!/ \text{ If } r = 1 \text{ then } B \text{ is stable w.r.t. } (N, P) \]
\[ /\!\!/ \text{ in other words } \text{Refine}(B, N, P) := \{B\} \]
\[ \text{if } r \geq 2 \text{ then} \]
\[ B_i = \{q \in Q : \text{dc}(q, N, P) \in D_i\}; \]
\[ /\!\!/ \text{ B is not stable} \]
\[ \Pi := (\Pi \setminus \{B\}) \cup \{B_1, \ldots, B_r\}; \]
\[ \text{forall } (\tilde{N}, B_i) \text{ with } \emptyset \neq \tilde{N} \subseteq N, \text{dc}(\tilde{N}, B_i) \neq \text{false} \textbf{ do} \]
\[ \text{add } (\tilde{N}, B_i) \text{ to } \text{Splitters}; \]

\[ \text{return } \Pi; \]

this logical equivalence class. This can be done by applying our pattern equivalence operator on the state space representation with function \( p \) being our representative and therefore \( Z' \) consisting of all variables representing labels of transitions.

Definition 4.1 Let \( Z = \{z_1, \ldots, z_n\} \) be a finite set of boolean variables and \( Z' = \{z'_1, \ldots, z'_k\} \subseteq Z \) be a subset of \( Z \). Let \( f \in \mathcal{B}(Z) \) a function and \( p \in \mathcal{B}(Z') \) a pattern function. The pattern equivalence of \( f \) with respect to \( p \) (written \( f /\equiv_p \)) is defined by: \( f /\equiv_p : \text{Eval}(Z \setminus Z') \to \mathcal{B} \) where

\[
f /\equiv_p(\eta) = \begin{cases} 
1 & \text{if } p = f|_\eta \text{ where } f|_\eta \text{ is the cofactor of } f \text{ with respect to } \eta \\
0 & \text{otherwise}
\end{cases}
\]

with \( \eta \in \text{Eval}(Z \setminus Z') \).

Example 4.2 Let \( Z = \{x, y, z\} \) and \( Z' = \{y\} \). The function \( f(x, y, z) = (x \land y) \lor z \) and the pattern function \( p(y) = y \). The results for \( f /\equiv_p \) are shown in the following tables.

<table>
<thead>
<tr>
<th>( x, z )</th>
<th>( f(x, z) )</th>
<th>( x, z )</th>
<th>( f /\equiv_p(x, z) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>( y )</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

The pattern equivalence operator allows to efficiently handle symbolic representations of the whole automata in contrast to [12] that treads small state spaces.
explicitly and only uses symbolic methods for the labeling of transitions. In contrast to [13] our approach is capable of handling the rich labeled transitions of constraint automata directly. As our algorithm works on a symbolic representation a complexity analysis is leads to distorted results. One can count the number of symbolic operations but the OBDDs this operations work on can have exponential size in the number of variables.

5 Benchmarks

After describing how the partition splitter algorithm can be implemented on symbolic constraint automata we now present some benchmarks for the implementation of this algorithm.

In Reo a FIFO(N)-component should behave like N FIFO(1)-components after joining them together and hiding away the inner nodes. We used another approach and tried to specify the constraint automaton for a FIFO(N) component directly. Using the partitioning splitter algorithm we found very subtle differences in these two implementations. When building the FIFO(N) directly we considered two possible operations. One can put one data element into the FIFO if the FIFO is not full and one can take one out of it when it is not empty. In contrast using the Reo construction leads to a component which can perform both operations at the same time. Our constraint automata does not specify the behavior of a correct Reo FIFO(N)-component. So the two approaches lead to non-bisimilar constraint automata. To gain bisimilar automata we can eliminate the concurrent operations by adding an AsyncDrain channel which prevents the incoming and outgoing ports from performing operations at the same time. Benchmark results for showing bisimulation equivalence of this components are shown in Tab. 1. Note that the automaton resulting from direct construction has much fewer states than the automaton built from the Reo circuit gained through join and hide. All states of the first automaton build their own equivalence class. While the states of the second automaton are divided in exactly the same number of equivalent classes. This example shows how join and hide can lead to redundancy in the resulting constraint automaton.

<table>
<thead>
<tr>
<th>n</th>
<th>Memory</th>
<th>states₁</th>
<th>trans₁</th>
<th>states₂</th>
<th>trans₂</th>
<th>splitters</th>
<th>classes</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>69</td>
<td>127</td>
<td>252</td>
<td>567</td>
<td>7140</td>
<td>376</td>
<td>127</td>
<td>1.87</td>
</tr>
<tr>
<td>7</td>
<td>73</td>
<td>255</td>
<td>508</td>
<td>1701</td>
<td>35566</td>
<td>760</td>
<td>255</td>
<td>4.22</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
<td>511</td>
<td>1020</td>
<td>5103</td>
<td>179508</td>
<td>1528</td>
<td>511</td>
<td>17.57</td>
</tr>
<tr>
<td>9</td>
<td>95</td>
<td>1023</td>
<td>2044</td>
<td>15309</td>
<td>916574</td>
<td>3064</td>
<td>1023</td>
<td>75.94</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>time out</td>
</tr>
</tbody>
</table>

Table 1. Bisimulation and FIFO(N) versus n-FIFO(1)
Another example used for our benchmarks are the two different Reo circuits specifying a mutual exclusion protocol shown above. This protocol specifies the interaction of $n$ processes that want to enter a critical section of their execution while only $k$ processes are allowed to be in the critical section at the same time. Using the algorithm presented in this paper we could prove that the two versions do not lead to the same observable behavior. This is because the left one allows one process to enter its critical section while another one leaves its critical section at the same time. One can add an asynchronous drain channel (which accepts all data at its ends but not at both at the same time) connecting nodes $\text{Req}$ and $\text{Rel}$ to render this impossible. Then the constraint automata for the two circuits become bisimilar. Results for this example are shown in Tab. 2.

### Table 2
Bisimulation and the mutual exclusion protocol (PROC2)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$k$</th>
<th>$\text{Memory}$</th>
<th>$\text{states}$</th>
<th>$\text{transitions}$</th>
<th>$\text{splitters}$</th>
<th>$\text{classes}$</th>
<th>$\text{Time}_{\text{bisim}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>5</td>
<td>81</td>
<td>$1.37 \cdot 10^{11}$</td>
<td>$1.62 \cdot 10^{15}$</td>
<td>20</td>
<td>6</td>
<td>0.52</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>103</td>
<td>$3.76 \cdot 10^{11}$</td>
<td>$7.98 \cdot 10^{10}$</td>
<td>80</td>
<td>21</td>
<td>2.93</td>
</tr>
<tr>
<td>200</td>
<td>30</td>
<td>120</td>
<td>$3.84 \cdot 10^{13}$</td>
<td>$5.07 \cdot 10^{10}$</td>
<td>120</td>
<td>31</td>
<td>6.95</td>
</tr>
<tr>
<td>200</td>
<td>60</td>
<td>217</td>
<td>$5.84 \cdot 10^{15}$</td>
<td>$2.50 \cdot 10^{13}$</td>
<td>240</td>
<td>61</td>
<td>43.64</td>
</tr>
<tr>
<td>400</td>
<td>5</td>
<td>105</td>
<td>$1.37 \cdot 10^{12}$</td>
<td>$5.17 \cdot 10^{11}$</td>
<td>20</td>
<td>6</td>
<td>1.48</td>
</tr>
<tr>
<td>400</td>
<td>20</td>
<td>145</td>
<td>$1.05 \cdot 10^{14}$</td>
<td>$5.63 \cdot 10^{10}$</td>
<td>80</td>
<td>21</td>
<td>6.95</td>
</tr>
<tr>
<td>400</td>
<td>30</td>
<td>172</td>
<td>$1.50 \cdot 10^{16}$</td>
<td>$4.23 \cdot 10^{12}$</td>
<td>120</td>
<td>31</td>
<td>12.39</td>
</tr>
<tr>
<td>400</td>
<td>60</td>
<td>271</td>
<td>$1.25 \cdot 10^{18}$</td>
<td>$8.82 \cdot 10^{15}$</td>
<td>240</td>
<td>61</td>
<td>71.75</td>
</tr>
<tr>
<td>800</td>
<td>5</td>
<td>174</td>
<td>$5.50 \cdot 10^{12}$</td>
<td>$1.65 \cdot 10^{14}$</td>
<td>20</td>
<td>6</td>
<td>4.57</td>
</tr>
<tr>
<td>800</td>
<td>20</td>
<td>251</td>
<td>$9.98 \cdot 10^{18}$</td>
<td>$4.84 \cdot 10^{16}$</td>
<td>80</td>
<td>21</td>
<td>19.81</td>
</tr>
<tr>
<td>800</td>
<td>30</td>
<td>304</td>
<td>$1.11 \cdot 10^{18}$</td>
<td>$1.40 \cdot 10^{18}$</td>
<td>120</td>
<td>31</td>
<td>32.80</td>
</tr>
<tr>
<td>800</td>
<td>60</td>
<td>388</td>
<td>$1.74 \cdot 10^{18}$</td>
<td>$1.07 \cdot 10^{19}$</td>
<td>240</td>
<td>61</td>
<td>148.92</td>
</tr>
</tbody>
</table>

6 Conclusion and Perspective

Using Reo to model the behavior of coordination protocols raises the question whether two connectors have the same observable behavior. This shows the need...
for a tool to answer this question in an automated way. This work shows how techniques known to work for labeled transition systems can be adapted to constraint automata. Introducing a symbolic representation for constraint automata and explaining how the product and hide operator can be performed in a symbolic way. Large Reo networks with many components can be handled automatically. In order to compare two constraint automata we show how a partition splitter algorithm for computing the bisimulation equivalence classes of constraint automata can be adapted to work on our symbolic representation. By introducing the pattern equivalence operator for switching functions we gain a powerful tool for symbolic treatment of the rich labeled transitions that occur in constraint automata. The benchmarks section gives an impression of how fast constraint automata for huge Reo circuits can be constructed and compared using our ideas. Future work will address several improvements of our implementation. Especially the set of splitters may be implemented in a more efficient way. At the moment we are working on combining our tool with the BTSL model checker for constraint automata presented in [14] and integrating both with the graphical design tool for Reo circuit currently being developed at CWI Amsterdam. Furthermore bisimulation as a homogenous approach to model checking constraint automata should be extended to richer versions of constraint automata like timed or probabilistic ones. Also one should extend the ideas presented here to make them applicable for constraint automata with priorities.

7 Acknowledgment

We like to thank Jörn Ossowski for his help implementing the pattern equivalence operator for his OBDD library.

References


