Festschrift

LNCS 8415

Saddek Bensalem Yassine Lakhneck
Axel Legay (Eds.)

From Programs to Systems

The Systems Perspective in Computing

**ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis
Grenoble, France, April 6, 2014
Proceedings**

Springer

Lecture Notes in Computer Science 8415

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Saddek Bensalem Yassine Lakhneck
Axel Legay (Eds.)

From Programs to Systems

The Systems Perspective in Computing

ETAPS Workshop, FPS 2014
in Honor of Joseph Sifakis
Grenoble, France, April 6, 2014
Proceedings

Springer

Volume Editors

Saddek Bensalem
University Joseph Fourier, Verimag Laboratory
Verimag Centre Équation
2, avenue de Vignate, 38610 Gières, France
E-mail: saddek.bensalem@imag.fr

Yassine Lakhneck
University Joseph Fourier, Verimag Laboratory
Verimag Centre Équation
2, avenue de Vignate, 38610 Gières, France
E-mail: yassine.lakhneck@imag.fr

Axel Legay
Inria, Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
E-mail: axel.legay@inria.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-54847-5 e-ISBN 978-3-642-54848-2
DOI 10.1007/978-3-642-54848-2
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014933961

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Joseph Sifakis

Preface

This group of papers represents the proceedings of the "From Programs to Systems – The Systems Perspective in Computing" workshop (FPS 2014). The workshop was held in honor of Professor Joseph Sifakis in the framework of the 16th European Joint Conferences on Theory and Practice of Software in Grenoble, April 6th, 2014.

The workshop provided a forum for researchers and practitioners from academia and industry to share their work, exchange ideas, and discuss the future directions on a systems perspective in computing. Over the past decades, the focus of computing has been continuously shifting from programs to systems. Programs can be represented as relations independent from the physical resources needed for their execution. Their behavior is often terminating, deterministic and platform-independent. On the contrary, systems are interactive. They continuously interact with an external environment. Their behavior is driven by stimuli from the environment, which, in turn, is affected by their outputs.

Systems are inherently complex and hard to design owing to unpredictable and subtle interactions with their environment, emergent behaviors, and occasional catastrophic cascading failures, rather than to complex data and algorithms. Compared to function software, their complexity is exacerbated by additional factors such as concurrent execution, uncertainty resulting from interaction with unpredictable environments, heterogeneity of interaction between hardware and software, and nonrobustness (small variations in a certain part of the system can have large effects on overall system behavior).

Theory of computation is, by its very nature, of little help for studying systems. Even if we perfectly understand the properties of a program and the properties of a hardware target platform, we have no theory to predict the behavior of the program running on the platform.

FPS addresses the two following issues:
Extending programing theory to systems

(i)To what extent can formal techniques for software development be adapted/ extended to system development?

- Program correctness vs. system correctness;
- Adapting SW engineering techniques to systems engineering;
- Software modeling vs. system modeling;
- How software verification techniques can be adapted to deal with quantitative properties?

(ii) Foundations for system design

- Missing results (theory, methods and tools) enabling rigorous system design;
- Building faithful system models;

- Adaptive resources management – Mixed criticality systems;
- Design space exploration;
- Automated implementation techniques for distributed or many-core platforms.

Joseph Sifakis is a professor and the director of the Rigorous System Design Laboratory at EPFL. His work is characterized by an unusual recurrent pattern: the problem is first studied from an abstract, foundational point of view, which leads to methods and techniques for its solution, which, in turn, leads to an effective implementation that is successfully used in multiple industrial applications.

Joseph Sifakis studied Electrical Engineering at the National Technical University in Greece. As a student he was inclined to be more concerned with theory than with practice. He came to Grenoble in 1970 for graduate studies in Physics. An encounter was decisive for his career: he met Professor Jean Kuntzmann, who was the Director of the Institute of Informatics and Applied Mathematics (IMAG). Joseph Sifakis interest in Computing grew and he decided to quit his studies in Physics and start undergraduate studies at IMAG. He did his Engineering thesis under supervision of Professor Jean Kuntzmann on Modelling the timed behavior of circuits. After his Engineering thesis, he became interested in the theory of concurrency.

From 1974 to 1977 Joseph Sifakis studied Petri nets and other models for concurrent systems. He obtained original and fundamental results on the structural properties of Petri nets as well as on the performance evaluation of timed Petri nets. These results are extensively used today for scheduling data-flow systems.

From 1977 to 1982 he switched his attention to program semantics and verification. Dijkstras papers and books had a deep influence on his work as well as discussions with Michel Sintzoff who was working at that time on program verification. They drew him the idea of fixpoint characterization for temporal modalities, and once again his work yielded original results on the algorithmic verification of concurrent systems based on a fixpoint characterization of the modalities of a branching time temporal logic. These results laid down the foundations of model checking. His student Jean-Pierre Queille developed the first model checker in 1982. Joseph Sifakis met Ed Clarke and Allen Emerson at CMU in November 1982 and they realized that they had been working independently on the same problem.

In the autumn of 1983, Joseph Sifakis met Amir Pnueli at a workshop on the Analysis of Concurrent Systems, organized in Cambridge. This was the beginning of a continuous interaction and collaboration for more than 25 years. Joseph Sifakis and Amir Pnueli setup several European projects in collaboration with Willem-Paul de Roever, on system modeling and verification. They jointly organized with Ed Clarke, the Workshop on the Verification of Finite State Systems in Grenoble in 1989. This workshop is considered as the first edition of the CAV Conference. Amir Pnueli frequently visited Verimag for over ten years and Verimag researchers greatly benefited from his wisdom and support.

In the period 1988-2000 Joseph Sifakis extended his work to deal with modeling and verification of real-time systems and hybrid systems. This included: the

study of hybrid systems and their verification techniques; the development and implementation of the KRONOS model checker, in collaboration with T. Henzinger, the first symbolic model checker for timed automata; the development and implementation of an efficient symbolic synthesis algorithm for timed systems, in collaboration with O. Maler and A. Pnueli; the study of compositional modeling techniques for real-time scheduling by using priorities. In January 1993, Joseph Sifakis founded the Verimag laboratory, a joint-venture between IMAG and Verilog SA. This has been an exciting and fruitful experience. Verimag has transferred the Lustre language designed by Paul Caspi and Nicolas Halbwachs, to the SCADE synchronous programing environment. SCADE is being used by Airbus to develop safety critical systems and has become a de facto standard for aeronautics. SCADE has been qualified as a development tool by the FAA, EASA, and Transport Canada under DO-178B up to Level A. It is currently been commercialized by Esterel Technologies. Verimag has also transferred functional testing and verification techniques to the ObjectGeode tool for modeling real-time distributed applications. This tool has been commercialized by Telelogic purchased by IBM in 2008.

Since 1997, Verimag has been a public research laboratory, associated with CNRS and the University of Grenoble. It plays a prominent role in embedded systems by producing cutting-edge research and leading research initiatives and projects in Europe. As the director of Verimag, Joseph Sifakis has sought a balance between basic and applied research. He has used resources from industrial contracts and collaborative projects to develop new research activities and strengthen the potential in basic research. For him, participation in industrial projects has been a source of inspiration. It allows the definition of new research directions that are scientifically challenging and technically relevant. The virtuous cycle of interaction between research and applications has been the key to Verimag success.

In the late 90s, Joseph Sifakis research interests progressively shifted from verification and formal methods to system design. He was convinced that formal verification was hitting a wall and only incremental improvements in the state-of-the-art could be expected. He stepped down from the Steering Committee of CAV and started a new research program on embedded systems design. Interactions with colleagues such as Hermann Kopetz, Lothar Thiele, Thomas Henzinger, Alberto Sangiovanni Vincentelli and Edward Lee, contributed to elaborating a system perspective for Computing. He worked actively for setting up the Emsoft Conference and for organizing the Embedded Systems community in Europe through the Artist Coordination Measure followed by the Artist2 and ArtistDesign European Networks of Excellence.

During this later period Joseph Sifakis has also played a leading role in the development and implementation of the BIP component framework for rigorous system design. The implementation consists of a language and a set of tools including source-to-source transformers, a compiler and the D-Finder tool for compositional verification. BIP is unique for its expressiveness. It can describe mixed hardware/software systems. It uses a small and powerful set of primitives

encompassing a general concept of system architecture. BIP was successfully used in several industrial projects, in particular for the componentization of legacy software and the automatic generation of implementations for many-core platforms.

Joseph Sifakis is an active and visionary researcher in the area of system design. He believes that endowing design with scientific foundations is at least of equal importance as the quest for scientific truth in natural sciences. As one of his close collaborators, I have constantly benefited from his advice and guidance. I wish Joseph a long and productive career as a researcher and intellectual.

DISTINCTIONS AND HONORS

Turing Award 2007
Silver Medal of CNRS, 2001
Leonardo da Vinci Medal 2012
Grand Officer of the National Order of Merit, France, 2008
Commander of the Legion of Honnour, France, 2011
Award of the Greek Parliament for Commonwealth and Democracy, 2010
Commander of the Order of the Phoenix, Greece 2013
Award of the Town of Grenoble, 2008
Member of the French Academy of Sciences, 2010
Member of Academia Europea, 2008
Member of the French Academy of Engineering, 2008
Doctor Honoris Causa: EPFL, University of Athens,
International Hellenic University
Honorary Professor: University of Patras

April 2014

Saddek Bensalem
Yassine Lakhnech
Axel Legay

Table of Contents

Model-Driven Information Flow Security for Component-Based Systems*

Najah Ben Said[1], Takoua Abdellatif[2], Saddek Bensalem[1], and Marius Bozga[1]

[1] UJF-Grenoble 1/CNRS, VERIMAG UMR 5104, Grenoble, 38041, France
[2] Sousse University, ESSTHS, Hammam Sousse, Tunisia

Abstract. This paper proposes a formal framework for studying information flow security in component-based systems. The security policy is defined and verified from the early steps of the system design. Two kinds of non-interference properties are formally introduced and for both of them, sufficient conditions that ensures and simplifies the automated verification are proposed. The verification is compositional, first locally, by checking the behavior of every atomic component and then globally, by checking the inter-components communication and coordination. The potential benefits are illustrated on a concrete case study about constructing secure heterogeneous distributed systems.

Keywords: component-based systems, information flow security, non-interference, unwinding conditions, automated verification.

1 Introduction

The amount and complexity of nowadays conceived systems and software knows a continuous increase. Information protection and secure information flow between these systems is paramount and represent a great design challenge. Model driven security (MDS) [BDL06] is an innovative approach that tend to solve system-level security issues by providing an advanced modeling process representing security requirements at a high level of abstraction. Indeed, MDS guarantees separation of concerns between functional and security requirements, from early phases of the system development till final implementation.

Information flow security can be ensured using various mechanisms. Amongst the first approaches considered, ones find access control policies [SSM98,Kuh98], that allow protecting data confidentiality by limiting access to data to be read or modified only by authorized users. Unfortunately, these mechanisms have been proven incomplete and limited since only by preventing the direct access to data, indirect (implicit) information flows are still possible given rise to the so called covert channels [SQSL05]. As an alternative, non-interference has been studied as a global property to characterize and to develop techniques ensuring

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement ICT-318772 (D-MILS).

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 1–20, 2014.

information flow security. Initially defined by Goguen and Meseguer [GM82], non-interference ensures that the system's secret information does not affect its public behavior.

In this work, we adapt the MDS approach to develop a component-based framework, named *secBIP*, that guarantees automated verification and implementation of secure information flow systems with respect to specific definition of non-interference. In general, component-based frameworks allow the construction of complex systems by composition of atomic components with communication and coordination operators. That is, systems are obtained from unitary atomic components that can be independently deployed and composed with other components. Component-based frameworks are usually well adopted for managing key issues for functional design including heterogeneity of components, distribution aspects, performance issues, etc. Nonetheless, the use of component-based frameworks is also beneficial for establishing information flow security. Particularly, the explicit system architecture allows tracking easily intra and inter-components information flow.

The *secBIP* framework is built as an extension of the *BIP* [BBS06,BBB+11] framework encompassing information flow security. *secBIP* allows to create systems that are secure by construction if certain local conditions hold for composed components. The *secBIP* extension includes specific annotations for classification of both data and interactions. Thanks to the explicit use of composition operators in *BIP*, the information flow is easily tracked within models and security requirements can be established in a compositional manner, first locally, by checking the behavior of atomic components and then globally, by checking the communication and coordination inter-components.

Information flow security has been traditionally studied separately for language-based models [SS01,SV98] (see also the survey [SM03]) and trace-based models [McC88,McL94,ZL97,Man00]. While the former mostly focus on verification of data-flow security properties in programming languages, the latter is treating security in event-based systems. In *secBIP*, we achieve a useful combination between both aspects, data-flow and event-flow security, in a single semantics model. We introduce and distinguish two types of non-interference, respectively *event non-interference* and *data non-interference*. For events, non-interference states that the observation of public events should not allow to deduce any information about the occurrence of secret events. For data, it states that there is no leakage of secret data into public ones.

The paper is structured as follows. Section 2 recalls the main concepts of the component-based framework adopted in this work. In section 3, we formally introduce the security extension and we provide the two associated definitions of non-interference, respectively for data flows and event flows. Next, in section 4 we formally establish non-interference based on unwinding relations and we provide sufficient conditions that facilitate its automatic verification. In section 5, we provide a use-case as illustrative example. Section 6 discusses the related work and section 7 concludes and presents some lines for future work. All the proofs of technical results are given in the appendix.

2 Component-Based Design

The *secBIP* framework is built as an extension of the *BIP* framework introduced in [BBS06]. *BIP* stands for *Behavior*, *Interaction* and *Priority*, that is, the three layers used for the definition of components and their composition in this framework. *BIP* allows the construction of complex, hierarchically structured models from atomic components characterized by their behavior and their interfaces. Such components are transition systems enriched with data. Transitions are used to move from a source to a destination location. Each time a transition is taken, component data (variables) may be assigned new values, computed by user-defined functions (in C). Atomic components are composed by layered application of interactions and priorities. Interactions express synchronization constraints and do the transfer of data between the interacting components. Priorities are used to filter amongst possible interactions and to steer system evolution so as to meet performance requirements e.g., to express scheduling policies.

In this section, we briefly recall the key concepts of *BIP* which are further relevant for dealing with information flow security. In particular, we give a formal definition of atomic components and their composition through multiparty interactions. Priorities are not considered in this work.

2.1 Atomic Components

Definition 1 (atomic component). *An atomic component B is a tuple (L, X, P, T) where L is a set of locations, X is a set of variables, P is a set of ports and $T \subseteq L \times P \times L$ is a set of port labelled transitions. For every port $p \in P$, we denote by X_p the subset of variables exported and available for interaction through p. For every transition $\tau \in T$, we denote by g_τ its guard, that is, a boolean expression defined on X and by f_τ its update function, that is, a parallel assignment $\{x := e^x_\tau\}_{x \in X}$ to variables of X.*

Figure 1 provides an example of an atomic component. It contains two control locations l_1 and l_2 and two ports p_1 and p_2. The transition labeled with p_1 can take place only if the guard $(0 < x)$ is true. When the transition takes place, the variable y is recalculated as some function of x.

Let $\mathcal{D}$ be the data domain of variables. Given a set of variables Y, we call valuation on Y any function $\mathbf{y} : Y \to \mathcal{D}$ mapping variables to data. We denote by $\mathbf{Y}$ the set of all valuations defined on Y.

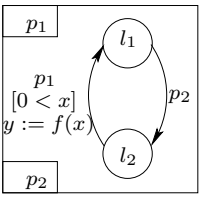


Fig. 1. Atomic Component in *BIP*

Definition 2 (atomic component semantics). *The semantics of an atomic component $B = (L, X, P, T)$ is defined as the labelled transition system* $\text{LTS}(B) =$

$(Q_B, \Sigma_B, \underset{B}{\longrightarrow})$ *where the set of states* $Q_B = L \times \mathbf{X}$*, the set of labels is* $\Sigma_B = P \times \mathbf{X}$ *and the set of labelled transitions* $\underset{B}{\longrightarrow}$ *is defined by the rule:*

$$\textsc{Atom}\ \frac{\tau = \ell \xrightarrow{p} \ell' \in T \qquad \mathbf{x}''_p \in \mathbf{X}_p \qquad g_\tau(\mathbf{x}) \qquad \mathbf{x}' = f_\tau(\mathbf{x}[X_p \leftarrow \mathbf{x}''_p])}{(\ell, \mathbf{x}) \xrightarrow[B]{p(\mathbf{x}''_p)} (\ell', \mathbf{x}')}$$

That is, $(\ell', \mathbf{x}')$ is a successor of $(\ell, \mathbf{x})$ labelled by $p(\mathbf{x}''_p)$ iff (1) $\tau = \ell \xrightarrow{p} \ell'$ is a transition of T, (2) the guard g_τ holds on the current valuation $\mathbf{x}$, (3) $\mathbf{x}''_p$ is a valuation of exported variables X_p and (4) $\mathbf{x}' = f_\tau(\mathbf{x}[X_p \leftarrow \mathbf{x}''_p])$ meaning that, the new valuation $\mathbf{x}'$ is obtained by applying f_τ on $\mathbf{x}$ previously modified according to $\mathbf{x}''_p$. Whenever a p-labelled successor exist in a state, we say that p is *enabled* in that state.

2.2 Composite Components

Composite components are obtained by composing an existing set of atomic components $\{B_i = (L_i, X_i, P_i, T_i)\}_{i=1,n}$ trough specific composition operators. We consider that atomic components have pairwise disjoint sets of states, ports, and variables i.e., for any two $i \neq j$ from $\{1..n\}$, we have $L_i \cap L_j = \emptyset$, $P_i \cap P_j = \emptyset$, and $X_i \cap X_j = \emptyset$. We denote $P = \bigcup_{i=1}^{n} P_i$ the set of all the ports, $L = \bigcup_{i=1}^{n} L_i$ the set of all locations, and $X = \bigcup_{i=1}^{n} X_i$ the set of all variables.

Definition 3 (interaction). *An* interaction *a between atomic components is a triple* (P_a, G_a, F_a)*, where* $P_a \subseteq P$ *is a set of ports,* G_a *is a guard, and* F_a *is an update function. By definition,* P_a *uses at most one port of every component, that is,* $|P_i \cap P_a| \leq 1$ *for all* $i \in \{1..n\}$*. Therefore, we simply denote* $P_a = \{p_i\}_{i \in I}$*, where* $I \subseteq \{1..n\}$ *contains the indices of the components involved in a and for all* $i \in I, p_i \in P_i$*.* G_a *and* F_a *are both defined on the variables exported by the ports in* P_a *(i.e.,* $\bigcup_{p \in P_a} X_p$*).*

Definition 4 (composite component). *A composite component* $C = \gamma(B_1, \ldots, B_n)$ *is obtained by applying a set of interactions* γ *to a set of atomic components* $B_1, \ldots B_n$*.*

Figure 2 presents a classical *Producer-Buffer-Consumer* example modeled in *BIP*. It consists of three atomic components, namely *Producer*, *Buffer* and *Consumer*. The *Buffer* is a shared memory placeholder, which is accessible by both the *Producer* and the *Consumer*. It holds into the local variable x the number of items available. The *Buffer* interacts with the *Producer* (res. *Consumer*) on the *put* (resp. *get*) interaction. On the *put* interaction, an item is added to the *Buffer* and x is incremented. On the *get* interaction, the *Consumer* removes an item from the *Buffer*, if at least one exists (the guard $[x \geq 1]$), and x is decremented. Finally, the transitions labeled *produce* and *consume* do not require synchronization - they are executed alone (on singleton port interactions) by the respective components.

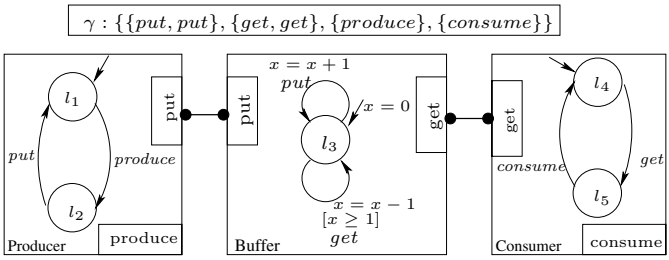


Fig. 2. *BIP* model of the *Producer-Buffer-Consumer* example

Definition 5 (composite component semantics). *Let* $C = \gamma(B_1, \ldots, B_n)$ *be a composite component. Let* $B_i = (L_i, X_i, P_i, T_i)$ *and* $\text{LTS}(B_i) = (Q_i, \Sigma_i, \xrightarrow[B_i]{})$ *their semantics, for all* $i = 1, n$. *The semantics of* C *is the labelled transition system* $\text{LTS}(C) = (Q_C, \Sigma_C, \xrightarrow[C]{})$ *where the set of states* $Q_C = \otimes_{i=1}^{n} Q_i$, *the set of labels* $\Sigma_C = \gamma$ *and the set of labelled transitions* $\xrightarrow[C]{}$ *is defined by the rule:*

$$\text{COMP}\ \frac{\begin{array}{c} a = (\{p_i\}_{i\in I}, G_a, F_a) \in \gamma \qquad G_a(\{\mathbf{x}_{p_i}\}_{i\in I}) \qquad \{\mathbf{x}''_{p_i}\}_{i\in I} = F_a(\{\mathbf{x}_{p_i}\}_{i\in I}) \\ \forall i \in I.\ (\ell_i, \mathbf{x}_i) \xrightarrow[B_i]{p_i(\mathbf{x}''_{p_i})} (\ell'_i, \mathbf{x}'_i) \qquad \forall i \notin I.\ (\ell_i, \mathbf{x}_i) = (\ell'_i, \mathbf{x}'_i) \end{array}}{((\ell_1, \mathbf{x}_1), \ldots, (\ell_n, \mathbf{x}_n)) \xrightarrow[C]{a} ((\ell'_1, \mathbf{x}'_1), \ldots, (\ell'_n, \mathbf{x}'_n))}$$

For each $i \in I$, $\mathbf{x}_{p_i}$ *above denotes the valuation* $\mathbf{x}_i$ *restricted to variables of* X_{p_i}.

The rule expresses that a composite component $C = \gamma(B_1, \ldots, B_n)$ can execute an interaction $a \in \gamma$ *enabled* in state $((\ell_1, \mathbf{x}_1), \ldots, (\ell_n, \mathbf{x}_n))$, iff (1) for each $p_i \in P_a$, the corresponding atomic component B_i can execute a transition labelled by p_i, and (2) the guard G_a of the interaction holds on the current valuation of variables exported on ports participating in a. Execution of interaction a triggers first the update function F_a which modifies variables exported by ports $p_i \in P_a$. The new values obtained, encoded in the valuation $\mathbf{x}''_{p_i}$, are then used by the components' transitions. The states of components that do not participate in the interaction remain unchanged.

Any finite sequences of interactions $w = a_1 ... a_k \in \gamma^*$ executable by the composite component starting at some given initial state q_0 is named a trace. The set of all traces w from state q_0 is denoted by $\text{TRACES}(C, q_0)$.

3 Information Flow Security

We explore information flow policies [DD77,BLP76,GM82] with focus on the noninterference property. In order to track information we adopt the classification technique and we define a classification policy where we annotate the information by assigning security levels to different parts of *secBIP* model (data variables,

ports and interactions). The policy describes how information can flow from one classification with respect to the other.

As an example, we can classify public information as a *Low* (L) security level and secret (confidential) information as *High* (H) security level. Intuitively *High* security level is more restrictive than *Low* security level and we denote it by $L \subseteq H$. In general, security levels are elements of a security domain, defined as follows:

Definition 6 (security domain). *A security domain is a lattice of the form* $\langle S, \subseteq, \cup, \cap \rangle$ *where:*

- S *is a finite set of security levels.*
- $\subseteq$ *is a partial order "can flow to" on* S *that indicates that information can flow from one security level to an equal or a more restrictive one.*
- $\cup$ *is a "join" operator for any two levels in* S *and that represents the upper bound of them.*
- $\cap$ *is a "meet" operator for any two levels in* S *and that represents the lower bound of them.*

As an example, consider the set $S = \{L, M_1, M_2, H\}$ of security levels that are governed by the "can flow to" relation $L \subseteq M_1$, $L \subseteq M_2$, $M_1 \subseteq H$ and $M_2 \subseteq H$. M_1 and M_2 are incomparable and we note $M_1 \nsubseteq M_2$ and $M_1 \not\supseteq M_2$. This security domain is graphically illustrated in Figure 3.

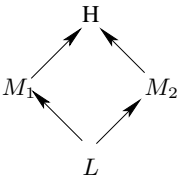


Fig. 3. Example of security domain

Let $C = \gamma(B_1, \ldots B_n)$ be a composite component, fixed. Let X (resp. P) be the set of all variables (resp. ports) defined in all atomic components $(B_i)_{i=1,n}$.

Let $\langle S, \subseteq, \cup, \cap \rangle$ be a security domain, fixed.

Definition 7 (security assignment). *A security assignment for component* C *is a mapping* $\sigma : X \cup P \cup \gamma \to S$ *that associates security levels to variables, ports and interactions such that, moreover, the security levels of ports matches the security levels of interactions, that is, for all* $a \in \gamma$ *and for all* $p \in P$ *it holds* $\sigma(p) = \sigma(a)$.

In atomic components, the security levels considered for ports and variables allow to track intra-component information flows and control the intermediate computation steps. Moreover, inter-components communication, that is, interactions with data exchange, are tracked by the security levels assigned to interactions.

In order to formally introduce the two notions of non-interference for *secBIP* models we need few additional notations, as follows. Let σ be a security assignment for C, fixed.

For a security level $s \in S$, we define $\gamma\downarrow_s^\sigma$ the restriction of γ to interactions with security level at most s that is formally, $\gamma\downarrow_s^\sigma = \{a \in \gamma \mid \sigma(a) \subseteq s\}$.

For a security level $s \in S$, we define $w|_s^\sigma$ the projection of a trace $w \in \gamma^*$ to interactions with security level lower or equal to s. Formally, the projection is recursively defined on traces as $\epsilon|_s^\sigma = \epsilon$, $(aw)|_s^\sigma = a(w|_s^\sigma)$ if $\sigma(a) \subseteq s$ and $(aw)|_s^\sigma = w|_s^\sigma$ if $\sigma(a) \not\subseteq s$. The projection operator $|_s^\sigma$ is naturally lifted to sets of traces W by taking $W|_s^\sigma = \{w|_s^\sigma \mid w \in W\}$.

For a security level $s \in S$, we define the equivalence $\approx_s^\sigma$ on states of C. Two states q_1, q_2 are equivalent, denoted by $q_1 \approx_s^\sigma q_2$ iff (1) they coincide on variables having security levels at most s and (2) they coincide on control locations having outgoing transitions labeled with ports with security level at most s.

We are now ready to define the two notions of non-interference.

Definition 8 (event non-interference). *The security assignment σ ensures event non-interference of $\gamma(B_1, \ldots, B_n)$ at security level s iff,*

$$\forall q_0 \in Q_C^0 : \textsc{traces}(\gamma(B_1, \ldots, B_n), q_0)|_s^\sigma = \textsc{traces}((\gamma\downarrow_s^\sigma)(B_1, \ldots, B_n), q_0)$$

Event non-interference ensures isolation/security at interaction level. The definition excludes the possibility to gain any relevant information about the occurrences of interactions (events) with strictly greater (or incomparable) levels than s, from the exclusive observation of occurrences of interactions with levels lower or equal to s. That is, an external observer is not able to distinguish between the case where such higher interactions are not observable on execution traces and the case these interactions have been actually statically removed from the composition. This definition is very close to Rushby's [Rus92] definition for transitive non-interference. But, let us remark that event non-interference is not concerned about the protection of data.

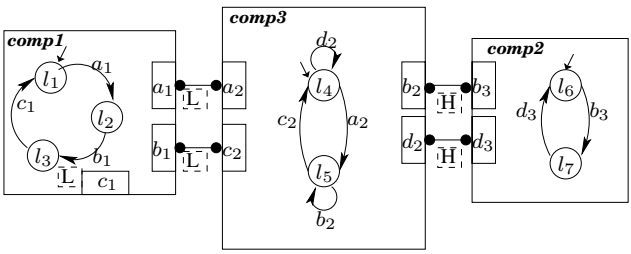


Fig. 4. Example for event non-interference

Example 1. Figure 4 presents a simple illustrative example for event non-interference. The model consists of three atomic components $comp_{i,i=1,2,3}$. Different security levels have been assigned to ports and interactions: $comp_1$ is a low security component, $comp_2$ is a high security component, and $comp_3$ is mixed security component. The security levels are represented by dashed squares related to interactions, internal ports and variables. As a convention, we apply high (H)

level for secret data and interactions and low(L) level for public ones. The set of traces is represented by the automaton in Figure 5 (a). The set of projected execution traces at security level L is represented by the automaton depicted in Figure 5 (b). This set is equal to the set of traces obtained by restricted composition, that is, using interaction with security level at most L and depicted in Figure 5 (c). Therefore, this example satisfies the event non-interference condition at level L.

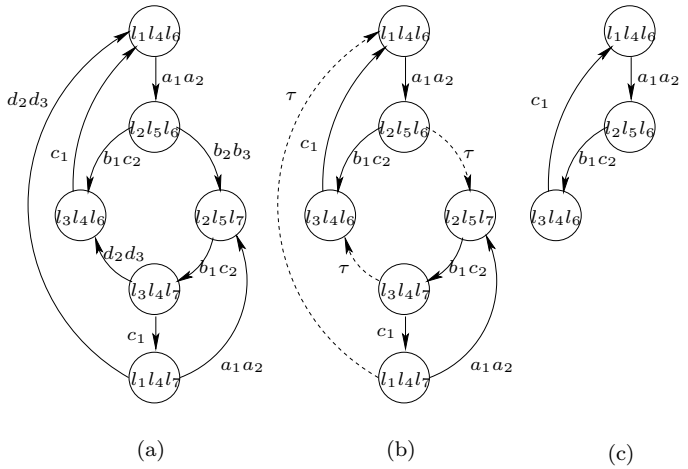


Fig. 5. Sets of traces represented as automata

Definition 9 (data non-interference). *The security assignment σ ensures data non-interference of $C = \gamma(B_1, \ldots, B_n)$ at security level s iff,*

$$\forall q_1, q_2 \in Q_C^0 : \; q_1 \approx_s^\sigma q_2 \Rightarrow$$
$$\forall w_1 \in \text{TRACES}(C, q_1), w_2 \in \text{TRACES}(C, q_2) : \; w_1|_s^\sigma = w_2|_s^\sigma \Rightarrow$$
$$\forall q_1', q_2' \in Q_C : \; q_1 \xrightarrow[C]{w_1} q_1' \wedge q_2 \xrightarrow[C]{w_2} q_2' \Rightarrow q_1' \approx_s^\sigma q_2'$$

Data non-interference provides isolation/security at data level. The definition ensures that, all states reached from initially indistinguishable states at security level s, by execution of arbitrary but identical traces whenever projected at level s, are also indistinguishable at level s. That means that observation of all variables and interactions with level s or lower excludes any gain of relevant information about variables at higher (or incomparable) level than s. Compared to event non-interference, data non-interference is a stronger property that considers the system's global states (local states and valuation of variables) and focus on their equivalence along identical execution traces (at some security level).

Example 2. Figure 6 presents an extension with data variables of the previous example from Figure 4. We consider the following two traces $w_1 = \langle a_1a_2, b_2b_3, c_2b_1, d_2d_3, c_1, a_2a_1\rangle$ and $w_2 = \langle a_1a_2, b_2b_3, c_2b_1, c_1, a_2a_1\rangle$ that start from the initial

state $((l_1, u = 0, v = 0), (l_4, x = 0, y = 0), (l_6, z = 0, w = 0))$. Although the projected traces at level L are equal, that is, $w_1|^{\sigma}_{L} = w_2|^{\sigma}_{L} = \langle a_1 a_2, c_2 b_1, c_1, a_1 a_2 \rangle$, the reached states by w_1 and w_2 are different, respectively $((l_2, u = 4, v = 2), (l_5, x = 3, y = 2), (l_6, z = 1, w = 1))$ and $((l_2, u = 4, v = 2), (l_5, x = 2, y = 2), (l_7, z = 1, w = 0))$ and moreover non-equivalent at low level L. Hence, this example is not data non-interferent at level L.

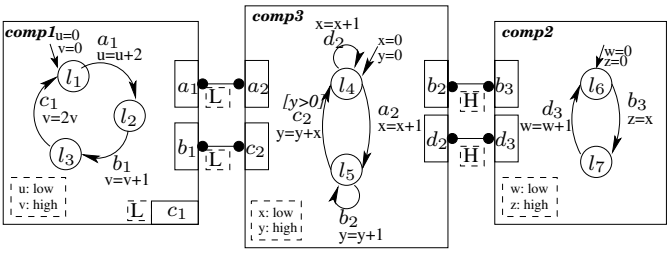


Fig. 6. Example for data non-interference

Definition 10 (secure component). *A security assignment σ is secure for a component $\gamma(B_1, \ldots, B_n)$ iff it ensures both event and data non-interference, at all security levels $s \in S$.*

4 Automated Verification of Non-interference

We propose hereafter an automated verification technique of non-interference for *secBIP* models based on the so-called unwinding conditions. These conditions were first introduced by Goguen and Meseguer for the verification of transitive non-interference for deterministic systems [GM82]. In general, the unwinding approach reduces the verification of information flow security to the existence of certain unwinding relation. This relation is usually an equivalence relation on system states that respects some additional properties on atomic execution steps, which are shown sufficient to imply non-interference. In the case of *secBIP*, the additional properties are formulated in terms of individual interactions/events and therefore easier to handle.

Let $C = \gamma(B_1, \ldots, B_n)$ be a composite component and let σ be a security assignment for C.

Definition 11 (unwinding relation). *An equivalence $\sim_s$ on states of C is called an unwinding relation for σ at security level s iff the two following conditions hold:*

1. *local consistency*
   $\forall q, q' \in Q_C : \ \forall a \in \gamma : \ q \xrightarrow[C]{a} q' \Rightarrow \sigma(a) \subseteq s \vee q \sim_s q'$

2. *output and step consistency*
$\forall q_1, q_2, q_1' \in Q_C :\ \forall a \in \gamma :$
$q_1 \sim_s q_2 \wedge q_1 \xrightarrow[C]{a} q_1' \wedge \sigma(a) \subseteq s \Rightarrow$
$\exists q_2' \in Q_C :\ q_2 \xrightarrow[C]{a} q_2' \wedge$
$\forall q_2' \in Q_C :\ q_2 \xrightarrow[C]{a} q_2' \Rightarrow q_1' \sim_s q_2'$

The existence of unwinding relations is tightly related to non-interference. The following two theorems formalize this relation for the two types of non-interference defined. Let C be a composite component and σ a security assignment.

Theorem 1 (event non-interference). *If an unwinding relation $\sim_s$ exists for the security assignment σ at security level s, then σ ensures event non-interference of C at level s.*

Theorem 2 (data non-interference). *If the equivalence relation $\approx_s^\sigma$ is also an unwinding relation for the security assignment σ at security level s, then σ ensures data non-interference of C at level s.*

The two theorems above are used to derive a practical verification method of non-interference using unwinding. We provide hereafter sufficient syntactic conditions ensuring that indeed the unwinding relations $\sim_s$ and $\approx_s$ exist on the system states. These conditions aim to effectively reduce the verification of non-interference to the checking on local constraints on both transitions (intra-component conditions) and interactions (inter-component conditions). Especially, they give an direct way to automate the verification.

Definition 12 (security conditions). *Let $C = \gamma(B_1, \ldots, B_n)$ be a composite component and let σ be a security assignment. We say that C satisfies the security conditions for security assignment σ iff:*

(i) the security assignment of ports, in every atomic component B_i is locally consistent, that is:

- *for every pair of causal transitions:*

$$\forall \tau_1, \tau_2 \in T_i :\ \tau_1 = \ell_1 \xrightarrow{p_1} \ell_2,\ \tau_2 = \ell_2 \xrightarrow{p_2} \ell_3 \ \Rightarrow \ \ell_1 \neq \ell_2 \ \Rightarrow\ \sigma(p_1) \subseteq \sigma(p_2)$$

- *for every pair of conflicting transitions:*

$$\forall \tau_1, \tau_2 \in T_i :\ \tau_1 = \ell_1 \xrightarrow{p_1} \ell_2,\ \tau_2 = \ell_1 \xrightarrow{p_2} \ell_3 \ \Rightarrow \ \ell_1 \neq \ell_2 \ \Rightarrow\ \sigma(p_1) \subseteq \sigma(p_2)$$

(ii) all assignments $x := e$ occurring in transitions within atomic components and interactions are sequential consistent, in the classical sense:

$$\forall y \in use(e) :\ \sigma(y) \subseteq \sigma(x)$$

(iii) variables are consistently used and assigned in transitions and interactions, that is,

$$\begin{array}{l}\forall \tau \in \cup_{i=1}^{n} T_i \;\; \forall x, y \in X \;\; : x \in def(f_\tau), y \in use(g_\tau) \;\; \Rightarrow \\ \qquad \sigma(y) \subseteq \sigma(p_\tau) \subseteq \sigma(x) \\ \forall a \in \gamma \;\; \forall x, y \in X \;\; : x \in def(F_a), y \in use(G_a) \;\; \Rightarrow \\ \qquad \sigma(y) \subseteq \sigma(a) \subseteq \sigma(x)\end{array}$$

(iv) all atomic components B_i are port deterministic:

$$\begin{array}{l}\forall \tau_1, \tau_2 \in T_i : \;\; \tau_1 = \ell_1 \xrightarrow{p} \ell_2, \tau_2 = \ell_1 \xrightarrow{p} \ell_3 \;\; \Rightarrow \\ \quad (g_{\tau_1} \wedge g_{\tau_2}) \text{ is unsatisfiable}\end{array}$$

The first family of conditions (i) is similar to Accorsi's conditions [AL12] for excluding causal and conflicting places for Petri net transitions having different security levels. Similar conditions have been considered in [FG01,FGF09] and lead to more specific definitions of non-interferences and bisimulations on annotated Petri nets. The second condition (*ii*) represents the classical condition needed to avoid information leakage in sequential assignments. The third condition (*iii*) tackles covert channels issues. Indeed, (*iii*) enforces the security levels of the data flows which have to be consistent with security levels of the ports or interactions (e.g., no low level data has to be updated on a high level port or interaction). Such that, observations of public data would not reveal any secret information. Finally, conditions (*iv*) enforces deterministic behavior on atomic components.

The relation between the syntactic security conditions and the unwinding relations is precisely captured by the following theorem.

Theorem 3 (unwinding theorem). *Whenever the security conditions hold, the equivalence relation $\approx_s^\sigma$ is an unwinding relation for the security assignment σ, at all security level s.*

The following result is the immediate consequence of theorems 1, 2 and 3.

Corollary 1. *Whenever the security conditions hold, the security assignment σ is secure for the component C.*

5 Case Study: Web Service Reservation System

We illustrate the *secBIP* framework to handle information flow security issues for a classical example, the web service reservation system proposed in [HV06]. A businessman, living in France, plans to go to Berlin for a private and secret mission. To organize his travel, he uses an intelligent web service who contacts two travel agencies: The first agency, *AgencyA*, arranges flights in Europe and the second agency, *AgencyB*, arranges flights exclusively to Germany. The reservation service obtains in return specific flight information and their corresponding prices and chooses the flight that is more convenient for him.

In this example, there are two types of interference that can occur, (1) data-interference since learning the flight price may reveal the flight destination and (2) event interference, since observing the interaction with *AgencyB* can reveal the destination as well. Thus, to keep the mission private, the flight prices and interactions with *AgencyB* have to be kept confidential.

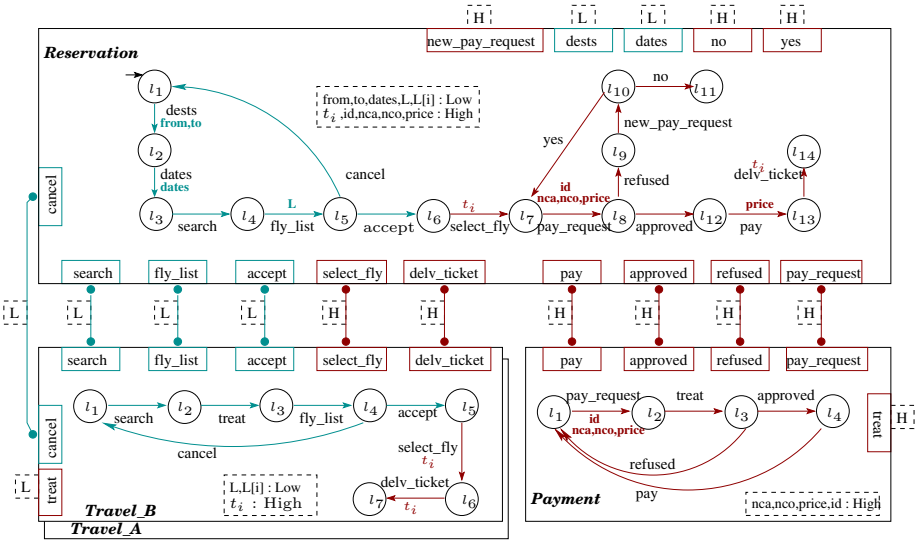


Fig. 7. Model of Reservation Web Service in *secBIP*

The modeling of the system using *secBIP* involves two main distinct steps: first, functional requirements modeling reflecting the system behavior, and second, security annotations enforcing the desired security policy. The model of the system has four components denoted: *Travel_A* and *Travel_B* who are instances from the same component and correspond respectively to AgencyA and AgencyB, and components *Reservation* and *Payment*. To avoid Figure 7 cluttering, we did not represent the interactions with *Travel_A* component. Search parameters are supplied by a user through the *Reservation* component ports *dests* and *dates* to which we associate respectively variables ($from$, to) and $dates$. Next, through search interaction, *Reservation* component contacts *Travel_B* component to search for available flights and obtains in return a list L of specific flights with their corresponding prices. Thereafter, *Reservation* component selects a ticket t_i from the list L and requests the *Payment* component to perform the payment.

All the search parameters $from$, to, $dates$, as well as the flights list L are set to low since users are not identified while sending these queries. Other sensitive data like the selected flight t_i, the price variable p and the payment parameters (identity id, credit card variable cna and code number cno) are set to high. Internal ports *dests* and *dates* as well as *search*, *fly_list*, *accept* interactions are set to low since these interactions (events) do not reveal any information about the

client private trip. However, the $select\_fly$ interaction must be set to high since the observation of the selection event from *AgencyB* allow to deduce deduce the client destination. In the case of a selected flight from *AgencyA*, the $select\_fly$ interaction could be set to low since, in this case, the destination could not be deduced just from the event occurrence.

We recall that any system can be proven non-interferent iff it satisfies the syntactic security conditions from Definition 12. Indeed, these conditions hold for the system model depicted in Figure 7. In particular, it can be easily checked that all assignments occurring in transitions within atomic component as well as within interactions are sequential consistent. For example, at the $select\_fly$ interaction we assign a low level security item from the flight list L to a high security level variable ti, formally $t_i = L[i]$. Besides, the security levels assignments to ports exclude inconsistencies due to causal and conflicting transitions, in all atomic components.

6 Related Work

Non-interference properties have been already studied using different model-based approaches. Recently, [SS12] adapted an MDS method for handling information flow security using UML sequence diagrams. Additionally, Petri-nets have been extensively used for system modeling and information flow security verifications tools such as InDico [AWD11] have been developed. A component-based model has been proposed in [ASRL11] and used to study implementation issues of secure information flows. Our presented work on *secBIP* is however different and original in several respects.

First, *secBIP* is a formal framework. Unlike UML, system's runtime behavior is always meaningfully defined and can be formally analyzed. Moreover, *secBIP* provides a system construction methodology for complex systems. Indeed, big systems are functionally decomposed into multiple sub-components communicating through well-defined interactions. Such a structural decomposition of the system is usually not available on Petri-nets models.

Second, *secBIP* handles both event and data-flow non-interference, in a single semantic model. To the best of our knowledge, these properties have never been jointly considered for component-based models. Nevertheless, the need to consider together event and data flow non-interference has been recently identified in the existing literature. The bottom line is that preserving the safety of data flow in a system does not necessarily preserve safe observability on system's public behavior (i.e., secret/private executions may have an observable impact on system public events). The issue has been recently considered in [AL12], for data leaks and information leaks in business processes based on system's data-flows and work-flows. Also, [BBMP08] showed that formal verification of the system's event behavior is not sufficient to guarantee specific data properties. Furthermore, [FRS05] attempted to fill the gap between respectively language-based and process calculus-based information security and make an explicit distinction between preventing the data leakage through the execution of programs and preventing secret events from being revealed in inter-process communications.

Third, compared to security-typed programming languages [jif,ZZNM02] and operating systems [KYB+07,ZBWM08,EKV+05] enforcing information flow control, *secBIP* is a component-based modeling approach where non-interference is established at a more abstract level. Thus, *secBIP* can be apriori implemented using different programming languages and is independent from a specific operating system and execution platform.

Finally, it is worth mentioning that a lot of classical approaches fall short to handle information flow security [Zda04] for real systems. For *secBIP* we privilege a very pragmatic approach and provide simple (syntactic) sufficient conditions allowing to automate the verification of non-interference. These conditions allow to eliminate a significant amount of security leakages, especially covert channels, independently from system language or the execution platform. However, these conditions can be very restrictive in some cases and a system designer may be interested to relax the non-interference properties.

7 Conclusion and Future Work

We present a MDS framework to secure component-based systems. We formally define two types of non-interference, respectively event and data non-interference. We provide a set of sufficient syntactic conditions which simplify verification of non-interference. These conditions are extensions of security typed language rules applied to our model. The use of our framework has been demonstrated to secure a web service application.

This work is currently being extended in two directions. First, we are investigating additional security conditions allowing to relax the non-interference property and control where downgrading can occur. Second, we are working towards the implementation of a complete design flow for secure systems based on *secBIP*. As a first step, we shall implement the verification method presented for annotated *secBIP* models. Then, use these models for generation of secure implementations, that is, executable code where the security properties are enforced by construction, at the generation time.

References

AL12. Accorsi, R., Lehmann, A.: Automatic information flow analysis of business process models. In: Barros, A., Gal, A., Kindler, E. (eds.) BPM 2012. LNCS, vol. 7481, pp. 172–187. Springer, Heidelberg (2012)

ASRL11. Abdellatif, T., Sfaxi, L., Robbana, R., Lakhnech, Y.: Automating information flow control in component-based distributed systems. In: 14th International ACM Sigsoft Symposium on Component Based Software Engineering (CBSE 2011), pp. 73–82. ACM (2011)

AWD11. Accorsi, R., Wonnemann, C., Dochow, S.: Swat: A security workflow analysis toolkit for reliably secure process-aware information systems. In: Sixth International Conference on Availability, Reliability and Security, ARES 2011, pp. 692–697. IEEE (2011)

BBB+11. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J.: Rigorous component-based design using the BIP framework. IEEE Software, Special Edition – Software Components beyond Programming – from Routines to Services 28(3), 41–48 (2011)

BBMP08. Bartolini, C., Bertolino, A., Marchetti, E., Parissis, I.: Data Flow-Based Validation of Web Services Compositions: Perspectives and Examples. In: de Lemos, R., Di Giandomenico, F., Gacek, C., Muccini, H., Vieira, M. (eds.) Architecting Dependable Systems V. LNCS, vol. 5135, pp. 298–325. Springer, Heidelberg (2008)

BBS06. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Systems in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), pp. 3–12. IEEE Computer Society Press (2006)

BDL06. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: from uml models to access control infrastructures. ACM Transactions on Software Engineering and Methodology 15, 39–91 (2006)

BLP76. Bell, E.D., La Padula, J.L.: Secure computer system: Unified exposition and Multics interpretation (1976)

DD77. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM, 504–513 (1977)

EKV+05. Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., Morris, R.: Labels and Event Processes in the Asbestos Operating System. SIGOPS Operating Systems Review 39(5), 17–30 (2005)

FG01. Focardi, R., Gorrieri, R.: Classification of Security Properties (Part I: Information Flow). In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 331–396. Springer, Heidelberg (2001)

FGF09. Frau, S., Gorrieri, R., Ferigato, C.: Petri net security checker: Structural non-interference at work. In: Degano, P., Guttman, J., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 210–225. Springer, Heidelberg (2009)

FRS05. Focardi, R., Rossi, S., Sabelfeld, A.: Bridging language-based and process calculi security. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 299–315. Springer, Heidelberg (2005)

GM82. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, pp. 11–20 (1982)

HV06. Hutter, D., Volkamer, M.: Information flow control to secure dynamic web service composition. In: Clark, J.A., Paige, R.F., Polack, F.A.C., Brooke, P.J. (eds.) SPC 2006. LNCS, vol. 3934, pp. 196–210. Springer, Heidelberg (2006)

jif. `http://www.cs.cornell.edu/jif/`

Kuh98. Richard Kuhn, D.: Role Based Access Control on MLS Systems without Kernel Changes. In: ACM Workshop on Role Based Access Control (RBAC 1998), pp. 25–32. ACM (1998)

KYB+07. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Frans Kaashoek, M., Kohler, E., Morris, R.: Information Flow Control for Standard OS Abstractions. SIGOPS Operating Systems Review 41(6), 321–334 (2007)

Man00. Mantel, H.: Possibilistic Definitions of Security - An Assembly Kit. In: 13th IEEE Workshop on Computer Security Foundations (CSFW 2000), p. 185. IEEE Computer Society (2000)

McC88. McCullough, D.: Noninterference and the composability of security properties. In: Security and Privacy (SP 1988), pp. 177–186. IEEE Computer Society (1988)
McL94. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: Security and Privacy (SP 1994), p. 79. IEEE Computer Society (1994)
Rus92. Rushby, J.: Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-2, SRI International (1992)
SM03. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1) (2003)
SQSL05. Shen, J., Qing, S., Shen, Q., Li, L.: Covert channel identification founded on information flow analysis. In: Hao, Y., Liu, J., Wang, Y.-P., Cheung, Y.-M., Yin, H., Jiao, L., Ma, J., Jiao, Y.-C. (eds.) CIS 2005. LNCS (LNAI), vol. 3802, pp. 381–387. Springer, Heidelberg (2005)
SS01. Sabelfeld, A., Sands, D.: A per model of secure information flow in sequential programs. Higher Order Symbolic Computation 14(1), 59–91 (2001)
SS12. Seehusen, F., Stølen, K.: A Method for Model-driven Information Flow Security. In: Dependability and Computer Engineering: Concepts for Software-Intensive Systems, pp. 199–229. IGI Global (2012)
SSM98. Sandhu, R., Ravi, S., Munawer, Q.: How to do discretionary access control using roles. In: ACM Workshop on Role-Based Access Control (RBAC 1998), pp. 47–54. ACM (1998)
SV98. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Symposium on Principles of Programming Languages (POPL 1998), pp. 355–364. ACM (1998)
ZBWM08. Zeldovich, N., Boyd-Wickizer, S., Mazières, D.: Securing distributed systems with information flow control. In: 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2008), pp. 293–308. USENIX Association (2008)
Zda04. Zdancewic, S.: Challenges for information-flow security. In: Programming Language Interference and Dependence, PLID 2004 (2004)
ZL97. Zakinthinos, A., Lee, E.S.: A general theory of security properties. In: Security and Privacy (SP 1997), pp. 94–102. IEEE Computer Society (1997)
ZZNM02. Zdancewic, S., Zheng, L., Nystrom, N., Myers, A.C.: Secure program partitioning. ACM Transactions on Computer Systems 20(3), 283–328 (2002)

Appendix

Proof of Theorem 1

Proof. We shall prove $\text{TRACES}(\gamma(B_1, \ldots, B_n), q_0)|_s^\sigma = \text{TRACES}((\gamma \downarrow_s^\sigma)(B_1, \ldots, B_n), q_0)$ by double inclusion. "$\supseteq$" inclusion: Independently of the unwinding relation, by using elementary set properties it holds that $\text{TRACES}((\gamma \downarrow_s^\sigma)(B_1, \ldots, B_n), q_0) = \text{TRACES}((\gamma \downarrow_s^\sigma)(B_1, \ldots, B_n), q_0)|_s^\sigma \subseteq \text{TRACES}(\gamma(B_1, \ldots, B_n), q_0)|_s^\sigma$. "$\subseteq$" inclusion: This direction is an immediate consequence of the following Lemma 1. It states that for every trace w in $\text{TRACES}(\gamma(B_1, \ldots, B_n), q_0)$ its projection $w|_s^\sigma$ is also a valid trace in $\text{TRACES}(\gamma(B_1, \ldots, B_n), q_0)$. But, this also means that $w|_s^\sigma$ is a valid trace in $\text{TRACES}((\gamma \downarrow_s^\sigma)(B_1, \ldots, B_n), q_0)$ which proves the result.

Lemma 1. *In the conditions of Theorem 1, for every trace w in* TRACES$(\gamma(B_1, \ldots, B_n), q_0)$*, for every state q such that $q_0 \xrightarrow[C]{w} q$, the projected trace $w|_s^\sigma$ is also a valid trace in* TRACES$(\gamma(B_1, \ldots, B_n), q_0)$ *and moreover, for every state q' such that $q_0 \xrightarrow[C]{w|_s^\sigma} q'$ it holds $q \sim_s q'$.*

Proof. The lemma is proved by induction on the length of the trace w. For the empty trace $w = \epsilon$ verification is trivial: $\sim_s$ holds for the initial state $q_0 \sim_s q_0$ and $\epsilon = \epsilon|_s^\sigma$. By induction hypothesis, let assume the property holds for traces of length n. We shall prove the property for traces of length $n+1$. Let $w' = wa$ be an arbitrary trace of length $n+1$, let w be its prefix (trace) of length n and let a be the last interaction. Consider states q, q_1 such that $q_0 \xrightarrow[C]{w} q \xrightarrow[C]{a} q_1$. By the induction hypothesis we know that $w|_s^\sigma$ is a valid trace and for all states q' such that $q_0 \xrightarrow[C]{w|_s^\sigma} q'$ it holds $q \sim_s q'$. We distinguish two cases, depending on the security level of a:

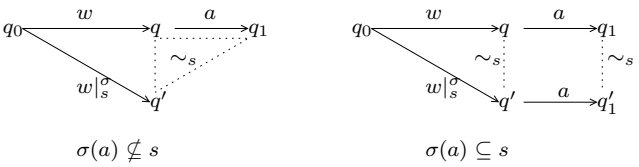


Fig. 8. Proof illustration for Lemma 1

- $\sigma(a) \not\subseteq s$: In this case, $w'|_s^\sigma = w|_s^\sigma$ hence, $w'|_s^\sigma$ is a valid trace as well, reaching the same states q'. Moreover, since a is invisible for s, the unwinding condition (1) ensures that $q \sim_s q_1$. By transitivity, this implies that $q_1 \sim_s q'$, which proves the result.
- $\sigma(a) \subseteq s$: In this case, $w'|_s^\sigma = w|_s^\sigma a$. From the unwinding condition (2), since $q \sim_s q'$ and a is visible and enabled in q then, a must also be enabled in q'. Therefore, $w|_s^\sigma$ can be extended with a from state q' to some q_1' hence, $w'|_s^\sigma$ is indeed a valid trace. Moreover, since $q \sim_s q'$ the unwinding condition (2) ensures also that $q_1 \sim_s q_1'$, which proves the result.

Proof of Theorem 2

Proof. Let us consider two equivalent states $q_1 \approx_s^\sigma q_2$.
The *first condition* for data non-interference requires that, for any trace w_1 from q_1 there exists a trace w_2 from q_2 having the same projection at level s, that is, $w_1|_s^\sigma = w_2|_s^\sigma$.

We shall prove a slightly stronger property, namely, the trace w_2 can be chosen such that, the successors q_1' and q_2' of respectively q_1 by w_1 and q_2 by w_2 are moreover equivalent, that is, $q_1' \approx_s^\sigma q_2'$. The proof is by induction on the length of

the trace w_1. *The base case*: for the empty trace $w_1 = \epsilon$ we take equally $w_2 = \epsilon$ we immediately have $q_1' = q_1 \approx_s^\sigma q_2 = q_2'$. *The induction step:* we assume, by induction hypothesis that the property holds for all traces w_1 such that $|w_1| \leq n$ and we shall prove it for all traces w_1' such that $|w_1'| = n + 1$. Let a be the last interaction executed in w_1', that is, $w_1' = w_1 a$ with $|w_1| = n$. Let q_1'' be the state reached from q_1 by w_1. From the induction hypothesis, there exists a trace w_2 that leads q_2 into q_2'' such that $w_1|_s^\sigma = w_2|_s^\sigma$ and moreover $q_1'' \approx_s^\sigma q_2''$. We distinguish two cases, depending on the security level of a:

- $\sigma(a) \not\subseteq s$: since $\approx_s^\sigma$ is unwinding and $q_1'' \xrightarrow[C]{a} q_1'$ it follows that $q_1'' \approx_s^\sigma q_1'$. In this case, we take $w_2' = w_2$ and $q_2' = q_2''$ which ensures both $w_1'|_s^\sigma = w_1|_s^\sigma = w_2|_s^\sigma = w_2'|_s^\sigma$ and $q_1' \approx_s^\sigma q_1'' \approx q_2'' = q_2'$.
- $\sigma(a) \subseteq s$: since $\approx_s^\sigma$ is unwinding and $q_1'' \approx_s^\sigma q_2''$ and $q_1'' \xrightarrow[C]{a} q_1'$ there must exists q_2' such that $q_2'' \xrightarrow[C]{a} q_2'$ and moreover, for any such choice $q_1' \approx_s^\sigma q_2'$. Hence, in this case, the trace $w_2' = w_2 a$ executed from q_2 and leading to q_2' satisfies our property, namely $w_1'|_s^\sigma = w_1|_s^\sigma a = w_2|_s^\sigma a = w_2'|_s^\sigma$ and $q_1' \approx_s^\sigma q_2'$.

The *second condition* for data non-interference requires that, for any traces w_1 and w_2 with equal projection on security level s, that is $w_1|_s^\sigma = w_2|_s^\sigma$, any successor states q_1' and q_2' of respectively q_1 by w_1 and q_2 by w_2 are also equivalent at level s. This property is proved also by induction on $|w_1| + |w_2|$, that is, on the sum of the lengths of traces w_1, w_2. *The base case*: for empty traces $w_1 = w_2 = \epsilon$ we have that $q_1' = q_1$ and $q_2' = q_2$ and hence trivially $q_1' \approx_s^\sigma q_2'$. *The induction step*: we assume, by induction hypothesis that the property holds for any traces w_1, w_2 such that $|w_1| + |w_2| \leq n$ and we shall prove it for all traces w_1', w_2' such that $|w_1'| + |w_2'| = n + 1$. We distinguish two cases, depending on the security levels of the last interactions occurring in w_1' and w_2'.

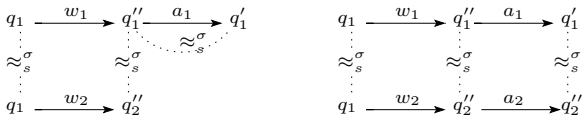


Fig. 9. Proof illustration for Theorem 2

- at least one of the last interactions in w_1' or w_2' has a security level not lower or equal to s. W.l.o.g, consider that indeed $w_1' = w_1 a_1$ and $\sigma(a_1) \not\subseteq s$. This situation is depicted in Figure 9, (left).

  Let q_1'' be the state reached from q_1 after w_1. Since $w_1'|_s^\sigma = w_2'|_s^\sigma$ and $\sigma(a_1) \not\subseteq s$ it follows that $w_1|_s^\sigma = w_1'|_s^\sigma = w_2'|_s^\sigma$. The induction hypothesis holds then for w_1 and w_2' because $|w_1| + |w_2'| = n - 1$ and hence we have that $q_1'' \approx_s^\sigma q_2'$. Moreover, q_1' is a successor of q_1'' by interaction a_1. Since the security level of a_1 is not lower or equal to s, and $\approx_s^\sigma$ is an unwinding relation at level s, it follows from the local consistency condition that $q_1'' \approx_s^\sigma q_1$. Then, by transitivity of $\approx_s^\sigma$ we obtain that $q_1' \approx_s^\sigma q_2'$.

– the last interactions of both traces w_1' and w_2' have security level lower or equal to s. That is, consider $w_1' = w_1 a_1$ and $w_2' = w_2 a_2$ with $\sigma(a_1) \subseteq s$, $\sigma(a_2) \subseteq s$. This situation is depicted in Figure 9, (right).

Let q_1'' and q_2'' be the states reached respectively from q_1 by w_1 and from q_2 by w_2. Since $\sigma(a_1) \subseteq s$,$\sigma(a_2) \subseteq s$ we have $w_1'|_s^\sigma = w_1|_s^\sigma a_1$, $w_2'|_s^\sigma = w_2|_s^\sigma a_2$. From the hypothesis, $w_1'|_s^\sigma = w_2'|_s^\sigma$, it follows that both $a_1 = a_2$ and $w_1|_s^\sigma = w_2|_s^\sigma$. Therefore, the induction hypothesis can be applied for traces w_1, w_2 because $|w_1| + |w_2| = n - 2$ and hence, we obtain $q_1'' \approx_s^\sigma q_2''$. But now, q_1' and q_2' are immediate successors of two equivalent states q_1'' and q_2'' by executing some interaction $a = a_1 = a_2$, having security level lower or equal to s. Since, $\approx_s^\sigma$ is an unwinding relation at level s, it follows from the step consistency condition that successors states q_1' and q_2' are also equivalent at level s, hence, $q_1' \approx_s^\sigma q_2'$.

Proof of Theorem 3

Proof. Let s be an arbitrary fixed security level. We shall prove that $\approx_s^\sigma$ satisfies the local, output and step consistency, as required by Definition 11.

local consistency: Let $q, q' \in Q_C$ be two states such that $q \xrightarrow[C]{a} q'$. We must show that if $\sigma(a) \not\subseteq s$ then $q \approx_s^\sigma q'$.

All variables x modified by a itself and by the transitions participating in a are such that $\sigma(a) \subseteq \sigma(x)$ (security conditions, (*iii*)). Then, since $\sigma(a) \not\subseteq s$, it also follows that all variables modified have security level greater or incomparable to s. Conversely, it follows that all variables with security levels lower or equal to s are not modified by a, hence they have the same values in q in q'.

Regarding control locations, we proceed by contradiction. Let consider that some component B_i is respectively at ℓ_i in q and at ℓ_i' in q' and moreover, either at ℓ_i or ℓ_i' there exists transitions with ports having security levels lower or equal s. Since the location of B_i has changed, it means that it has participated in the interaction a using some transition $\tau_i = \ell_i \xrightarrow{p_i} \ell_i'$. Let consider the two situations:

– there exists transitions with security level lower or equal to s at ℓ_i'. Let $\tau_i' = \ell_i' \xrightarrow{p_i'} \ell_i''$ such a transition. This situation contradicts the security conditions (*i*), as τ_i' is causally dependent on τ_i and has a different, yet not increased security level i.e., $\sigma(p_i) = \sigma(a) \not\subseteq s$ and $\sigma(p_i') \subseteq s$.
– there exists transitions with security level lower or equal to s at ℓ_i. Let $\tau_i' = \ell_i \xrightarrow{p_i'} \ell_i''$ such a transition. This situation contradicts again the security conditions (*i*): as τ_i and τ_i' are now conflicting it must be $\sigma(p_i) \subseteq \sigma(p_i')$ which contradicts that $\sigma(p_i) = \sigma(a) \not\subseteq s$ and $\sigma(p_i') \subseteq s$.

Henceforth, as the two situations lead to contradiction we conclude that, either $\ell_i = \ell_i'$, or otherwise, neither in ℓ_i or ℓ_i' there exists transitions with ports having security level lower or equal to s. This conclude the proof of $q \approx_s^\sigma q'$

output and step consistency: Let q_1, q_2 be two equivalent states $q_1 \approx_s^\sigma q_2$.

Let a be an interaction with security level lower or equal to s enabled in q_1. We show that the same interaction is enabled in q_2. All components participating in

a use transitions with ports with the same level as a, hence at most s. Therefore, these components are at control locations where there are outgoing transitions with level at most s. Then, these components are precisely in the same locations in q_2 since $q_1 \approx_s^\sigma q_2$.

Moreover, all the guards of the interacting transitions as well as the guard of the interaction use variables with security level lower or equal to $\sigma(a)$ and consequently, lower or equal to s (security conditions, (*iii*)). But again, $q_1 \approx_s^\sigma q_2$ implies that all variables with levels lower or equal to s have equal values in q_1 and q_2. Hence, the guards used in a have the same evaluation in q_1 or q_2. Together with equality on control locations, established earlier, this implies that a is enabled in q_2.

Let now consider two arbitrary states q_1', q_2' reached by a from respectively q_1 and q_2. We must show that $q_1' \approx_s^\sigma q_2'$. First, as $\sigma(a) \subseteq s$, it follows that, as explained before, enabledness of a depends exclusively on identical parts of q_1 and q_2. Moreover, due to security conditions (*iv*) it follows also that the execution of a synchronizes *exactly* the same set of transitions when executed either from q_1 or from q_2. Hence, in the successor states q_1' and q_2' all interacting atomic components have moved towards the same locations. The equality condition on the control locations is therefore satisfied. Furthermore, using security conditions (*ii*) it holds that all variables modified by transitions involved in a, if they have security values lower or equal to s, they will be assigned the same values. That is, the assigned expression use only variables with a lower security level, and hence identical on q_1 and q_2. This ensures equality of variables with security level lower or equal to s in q_1' and q_2', which conclude the proof.

Context-Bounded Analysis of TSO Systems

Mohamed Faouzi Atig[1], Ahmed Bouajjani[2], and Gennaro Parlato[3]

[1] Uppsala University, Sweden
[2] LIAFA, Université Paris Diderot & Institut Universitaire de France, France
[3] School of Electronics and Computer Science, University of Southampton, UK

Abstract. We address the state reachability problem in concurrent programs running over the TSO weak memory model. This problem has been shown to be decidable with non-primitive recursive complexity in the case of finite-state threads. For recursive threads this problem is undecidable. The aim of this paper is to provide under-approximate analyses for TSO systems that are decidable and have better (elementary) complexity. We propose three bounding concepts for TSO behaviors that are inspired from the concept of bounding the number of context switches introduced by Qadeer and Rehof for the sequentially consistent (SC) model. We investigate the decidability and the complexity of the state reachability problems under these three bounding concepts for TSO, and provide reduction of these problems to known reachability problems of concurrent systems under the SC semantics.

1 Introduction

Sequential consistency is the standard interleaving model for shared memory concurrent programs, where computations of a concurrent programs are interleaved sequences of actions of the different threads, performed in the same order as they appear in the program. However, for performance reasons, modern multi-processors do not preserve in general the program order, that is, they may actually reorder actions executed by a same thread. This leads to so-called weak or relaxed memory models. One of such models is TSO (Total Store Order), which is adopted for instance in x86 machines [36]. In TSO, write operations can be delayed and overtaken by read operations. This corresponds to the use of FIFO store buffers, one per processor, where write operations wait until they are committed in the main memory. Writes are therefore not visible immediately, which may lead to undesirable behaviors since older values than expected may be read along program computations.

Actually, for data-race free programs it can be shown that weak memory models such as TSO induce the same semantics as SC, that is, all possible computations under TSO are also possible under SC [35,4,5,9,18,31,21]. However, data-race-freedom cannot be ensured in all situations. This is for instance the case for low level lock-free programs used in many concurrency libraries and other performance-critical system services. The design of such algorithms, which must be aware of the underlying memory model, is in general extremely difficult

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 21–38, 2014.

due to the unintuitive and hard to predict effects of the weak memory models. Therefore, it is important to develop automatic verification techniques for programs running on such memory models.

In this paper, we focus on the TSO model and we address the state reachability problem, i.e., whether a state of the program (composed by the control locations of the threads and the memory state) is reachable from an initial state. This problem is of course relevant for checking (violations of) safety properties. To reason about programs running over TSO, we adopt an operational model based on parallel automata with unbounded FIFO queues representing the store buffers. The automata model the threads running on each of the processors. These automata are finite-state when programs do not have recursive procedure calls. For the case of recursive programs, threads are modeled using pushdown automata (automata with unbounded stacks). Note that our models have unbounded stacks and unbounded queues. In fact, although these structures are necessarily finite in actual machines, we may not assume any fixed bound on their size, so a finite-state model would not be sufficient to reason about the correctness of a general algorithm for all possible values of these bounds.

Even for finite-state processor threads, the decidability of the state reachability problem under TSO is not trivial due to the unboundedness of the queues. However, it has been shown that this problem is actually decidable, but unfortunately with very high complexity [11]. Indeed, the complexity of state reachability jumps from PSPACE for SC to non-primitive recursive for TSO. As for the case of recursive programs, it is easy to prove that the problem is undecidable as for SC. Therefore, it is important to investigate conditions under which the complexity of this problem becomes elementary, and for which decidability can be obtained even in the case of recursive programs. The approach we adopt in this paper for this purpose is based on the idea of bounding the number of context switches that has been used for the analysis of shared memory concurrent programs under SC [34].

An important issue is to define a suitable notion of context in the case of TSO systems that offers a good trade-off between coverage, decidability and complexity. The direct transposition of the definition for SC to this case consists in considering that a context is a computation segment where only one processor thread is active. This *processor-centric* definition does not restrict the behavior of the memory manager which can execute at any time write operations taken from any store buffer. A *memory-centric* definition, that is the dual of the previous one, considers that in a context only one store buffer is used for memory updates, without restricting the behaviors of the processor threads. Finally, a combination of the two previous definitions leads to a notion of context where only one processor thread is active, and only its store buffer can be used for memory updates. Notice that the three definitions above coincide with the one for SC when all write operations are immediately executed (i.e., the store buffers are of size 0).

We study the decidability and complexity of the analyses corresponding to these three definitions, named pc-CBA, mc-CBA, and pmc-CBA, for processor,

memory, processor-memory centric context-bounded analysis, respectively. In terms of behavior coverage, pc-CBA and mc-CBA are incomparable, and both of them subsume clearly pmc-CBA.

Actually, pmc-CBA coincides with the analysis that we have introduced and studied in [13]. Interestingly, this analysis can be reduced linearly to the context-bounded analysis for SC, and therefore both analysis have the same decidability and complexity characteristics. In addition to the fact that this analysis is decidable and has an elementary complexity (as opposed to the general TSO reachability analysis which is non-primitive recursive as mentioned above), a nice feature of this reduction is that the resulting analysis does not need explicit representation for the contents of the queues. It is possible to show that the content of the queue can be simulated in this case by adding a linear number of additional copies of the global variables. Also, our result allows to use for analyzing programs under TSO all the techniques and tools developed for SC context-bounded analysis, especially those based on code to code translations to sequential programs [28,26].

Then, the main contributions of this paper concern the decidability and complexity of the other two more powerful analyses pc-CBA and mc-CBA. First, we prove that in the case of finite-state processor threads, the pc-CBA is decidable with an elementary complexity. The complexity upper bound we have is polynomial in the size of the state space of the program (product of the thread automata and the memory state) and doubly exponential in the number of contexts. The proof is based on a reduction to the reachability problem of bounded-reverse-phase multiply pushdown automata (brp-MPDA). These models are multi-stack automata where all computations have a bounded number computation segments called reverse-phases, and within each of these segments only one stack can be used in a non-restricted way, while all the others can only be used for pop operations [32]. The name of reverse-phase is by opposition to the name of phase, used in a preceding work introducing bounded-phase multiply pushdown automata (bp-MPDA) [24], where again only one stack is unrestricted while the others can only be used for push operations. The decidability of the reachability problem in bp-MPDA and brp-MPDA has been established in [24] and [32], respectively.

The reduction from TSO systems to brp-MPDA is far from being trivial. The difficulty is, for each context, in order to simulate with a stack the FIFO queue representing the store buffer of the active threads. A naive way to do it would use an unbounded number of reverse-phases (for stack rotations). We show, and this is the tricky part of the proof, that this is actually possible with only one stack rotation for each context, due to the particular semantics of the store buffers. For the case of recursive threads, we prove that however, the pc-CBA is surprisingly undecidable. Furthermore, we prove that the mc-CBA has the same decidability and complexity characteristics as the pc-CBA. The decidability is in this case obtained by a reduction to the bp-MPDA mentioned above, and the undecidability is established following the same lines as in the previous case.

Related work: Context-bounded analysis has been introduced in [34] as an under-approximate analysis for bug detection in multithreaded programs. It has been subsequently widely studied and extended in several works, e.g., in [28,25,26,14,16]. All these works consider the SC semantics. Our work extends this kind of analysis to programs running over weak memory models.

The decidability and the complexity of the state reachability problem for TSO (without restriction on the behaviors) and for other weak memory models (such as PSO) have been established in [11,12]. We are not aware of other work investigating the decidability and complexity results of the state reachability problem for weak memory models.

Testing and bounded model checking algorithms have been proposed for TSO in [19,20,7]. These methods cannot cover sets of behaviors for arbitrary sizes of the store buffers. Algorithmic methods based on abstractions or on bounding the size of store buffers are proposed in [23,6,2,1,3]. In [30], a regular model checking-based approach, using finite-state automata for representing sets of store buffer contents is proposed. The analysis delivers the precise set of reachable configurations when it terminates, but termination is not guaranteed in general.

Checking (trace-)robustness against TSO, i.e., whether all traces of a given program running over TSO are also traces of computations over SC, has been addressed in [33,8,17,15]. This problem has been shown to be decidable in [17] and to be polynomially reducible to state reachability for SC in [15]. Trace-robustness and (safety-)correctness for SC imply correctness for TSO, but the converse in not true.

2 Concurrent Pushdown Systems

In this section we define concurrent pushdown systems (CPDS) with two semantics: *Sequential Consistency* (SC) and *Total-Store-Order* (TSO). Moreover, we define a behaviour-language reachability problem for them.

2.1 Memory Model

A (shared) *memory model* is a tuple $M = (Var, D, \eta^0, T)$, where Var is a finite set of variable names, D is a finite domain of all variables in Var, $\eta^0 : Var \to D$ is an initial valuation, and T is a finite set of thread names. The set of *memory operations* M_{op} is defined as the smallest set containing the following: nop (*no-operation*), $\mathsf{r}(x, d)$ (*read*), $\mathsf{w}(x, d)$ (*write*), $\mathsf{arw}(x, d, d')$ (*atomic read-write*), for every $x \in Var$ and $d, d' \in D$.

We define the *action function* $act_M : M_{op} \to \{nop, read, write, atomicRW\}$ that maps each memory operation in its type. The *size* of a memory model M, denoted $|M|$, is $|M_{op}| + |D| + |Var|$.

Below, we give the SC and TSO semantics for a memory model.

Sequential Consistency (SC): An SC-*configuration* of a memory model M consists of a valuation map $\eta : Var \to D$. A configuration η is *initial* if $\eta = \eta^0$. Given two SC-configurations η and η' of M, there is an SC-*transition* from η to η' on an operation $op \in M_{op}$ performed by thread t, denoted $\eta \xrightarrow[\text{SC},M,t]{op} \eta'$, if one of the following holds:

[nop] $op = \mathsf{nop}$, and $\eta = \eta'$;
[read] $op = \mathsf{r}(x,d)$, $\eta(x) = d$, and $\eta = \eta'$;
[write] $op = \mathsf{w}(x,d)$, $\eta'(x) = d$, and $\eta'(y) = \eta(y)$ for every $y \in (Var \setminus \{x\})$;
[atomic-read-write] $op = \mathsf{arw}(x,d,d')$, and $\eta \xrightarrow[\text{SC},M,t]{\mathsf{r}(x,d)} \eta \xrightarrow[\text{SC},M,t]{\mathsf{w}(x,d')} \eta'$.

Total Store Order (TSO): In TSO, each thread $t \in T$ is equipped with a FIFO queue σ_t to store write operations performed by t. When t writes value d into variable x, the pair (x,d) is enqueued into σ_t. Write operations stored in queues will affect the content of the shared variables only later in time: a pair (x,d) is non-deterministically dequeued from one of the queues and only at that time d is written into x, hence visible to all the other threads. Conversely, when t reads from x, the value that t recovers is the last value that t has written into x, provided that this operation is still pending in σ_t; otherwise, the returned value for x is that stored in the memory.

Formally, a TSO-*configuration* of M is a tuple $C_M = \langle \eta, \{\sigma_t\}_{t \in T} \rangle$, where $\eta : Var \to D$ is a valuation map, and $\sigma_t \in (Var \times D)^*$ for every $t \in T$. C_M is *initial* if $\eta = \eta^0$ and $\sigma_t = \epsilon$ for every $t \in T$ (where ϵ denotes the empty word).

Let $C = \langle \eta, \{\sigma_t\}_{t \in T} \rangle$ and $C' = \langle \eta', \{\sigma'_t\}_{t \in T} \rangle$ be two TSO-configurations of M. There is a TSO-transition from C to C' on $op \in (M_{op} \cup \{mem\})$ performed by thread t, denoted $C \xrightarrow[\text{TSO},M,t]{op} C'$, if one of the following holds:

[nop] $op = \mathsf{nop}$, $\eta' = \eta$, and $\sigma'_h = \sigma_h$ for every $h \in T$;
[read] $op = \mathsf{r}(x,d)$, $C' = C$, and either $\sigma_t = \pi_1.(x,d).\pi_2$ for some $\pi_1 \in (\Sigma \setminus (\{x\} \times D))^*$, or $\sigma_t \in (\Sigma \setminus (\{x\} \times D))^*$ and $\eta(x) = d$;
[write] $op = \mathsf{w}(x,d)$, $\eta' = \eta$, $\sigma'_t = (x,d).\sigma_t$, and $\sigma'_h = \sigma_h$ for every $h \in (T \setminus \{t\})$;
[atomic-read-write] $op = \mathsf{arw}(x,d,d')$, $\sigma'_t = \sigma_t = \epsilon$, $\eta(x) = d$, $\eta'(x) = d'$, $\eta'(y) = \eta(y)$ for every $y \in (Var \setminus \{x\})$, and $\sigma'_h = \sigma_h$ for every $h \in T$;
[memory] $op = mem$, $\sigma_t = \sigma'_t.(x,d)$, $\eta'(x) = d$, $\eta'(y) = \eta(y)$ for every $y \in (Var \setminus \{x\})$, and $\sigma'_h = \sigma_h$ for every $h \in (T \setminus \{t\})$.

2.2 Concurrent Pushdown Systems

We start with pushdown systems which are meant to model a recursive thread.

Pushdown Systems: A *pushdown system* (PDS) is a tuple $A = (Q, q^0, \Gamma, \Delta)$ where Q is a finite set of control states, $q^0 \in Q$ is the initial state, Γ is a finite stack alphabet, and $\Delta = \Delta_{int} \cup \Delta_{push} \cup \Delta_{pop}$ is the set of A moves, with $\Delta_{int} \subseteq Q \times Q$, $\Delta_{push} \subseteq Q \times Q \times \Gamma$, and $\Delta_{pop} \subseteq Q \times \Gamma \times Q$.

A *configuration* of a PDS A is a pair in $Q \times \Gamma^*$. A configuration $\langle q, \gamma \rangle$ is *initial* if $q = q^0$ and $\gamma = \epsilon$. There is a transition from $\langle q, \gamma \rangle$ to a configuration $\langle q', \gamma' \rangle$ on $\delta \in \Delta$, denoted $\langle q, \gamma \rangle \xrightarrow[A]{\delta} \langle q', \gamma' \rangle$, if one of the following holds:

[internal move] $\delta = (q, q') \in \Delta_{int}$ and $\gamma' = \gamma$;
[push move] $\delta = (q, q', a) \in \Delta_{push}$ and $\gamma' = a.\gamma$;
[pop move] $\delta = (q, a, q') \in \Delta_{pop}$ and $\gamma = a.\gamma'$.

We define an action map $act_A : \Delta \to \{int, push, pop\}$ where $act_A(\delta) = a$ iff $\delta \in \Delta_a$. The *size* of a PDS $A = (Q, q^0, \Gamma, \Delta)$, denoted $|A|$, is $|Q| + |\Delta|$.

A PDS A is a *finite state system* (FSS) if $act_A(\delta) = int$, for every $\delta \in \Delta$.

Concurrent Pushdown Systems: A *concurrent pushdown system* (CPDS) is composed by a finite number of PDS–one per thread–which communicate through a memory model M according to the SC or the TSO semantics.

Syntax. A CPDS over a finite set of thread names T and memory model $M = (Var, D, \eta^0, T)$ is a set of tuples $A = \{(Q_t, q_t^0, \Gamma_t, \Delta_t^M)\}_{t \in T}$, where $A_t = (Q_t, q_t^0, \Gamma_t, \Delta_t)$ is a PDS (called the *thread* t of A), and $\Delta_t^M \subseteq (\Delta_t \times M_{op})$.

The *size* of a CPDS A with memory M is $|M| \cdot \prod_{t \in T} |A_t|$.

A CPDS A over T is a *concurrent finite state system* (CFSS) if for every $t \in T$, thread A_t of A is a FSS.

Semantics. For MEM $\in$ {SC,TSO}, a MEM-*configuration* of A is a pair $C = \langle \{C_t\}_{t \in T}, C_M \rangle$, where C_t is an A_t configuration and C_M is a MEM-configuration of M. Further, C is *initial* if for every $t \in T$, C_t is the initial configuration of A_t, and C_M is the initial MEM-configuration of M.

Define $Act_T = \{int, push, pop\}$ and $Act_M = \{nop, read, write, atomicRW\}$. Let $Act = (Act_T \times Act_M \times T) \cup (\{(nop, mem)\} \times T)$. There is a MEM-*transition* from $C = \langle \{C_t\}_{t \in T}, C_M \rangle$ to $C' = \langle \{C'_t\}_{t \in T}, C'_M \rangle$ on an action $(a, b, t) \in Act$, denoted $C \xrightarrow[\text{MEM}, A]{(a,b,t)} C'$, if $C_h = C'_h$ for every $h \in (T \setminus \{t\})$, $C_M \xrightarrow[\text{MEM}, M, t]{op} C'_M$, and one of the following holds:

[thread & memory transition] $(\delta, op) \in \Delta_t^M$ with $a = act_{A_t}(\delta)$ and $b = act_M(op)$, and $C_t \xrightarrow[A_t]{\delta} C'_t$;
[memory transition only] $a = nop$, $b = op = mem$, and $C_t = C'_t$.

2.3 Reachability Problem

A MEM-*run* of A is a sequence $\pi = C_0 \xrightarrow[\text{MEM}, A]{(a_1,b_1,t_1)} C_1 \xrightarrow[\text{MEM}, A]{(a_2,b_2,t_2)} \dots \xrightarrow[\text{MEM}, A]{(a_n,b_n,t_n)} C_n$ for some $n \in \mathbb{N}$, where C_0 is the initial MEM-configuration of A. We define the *behaviour* of π as the sequence $beh(\pi) = (a_1, b_1, t_1)(a_2, b_2, t_2) \dots (a_n, b_n, t_n)$. For a behaviour language $B \subseteq Act^*$, a MEM-configuration C of A is B-*reachable* if there exists a MEM-run π of A such that $C = C_n$ and $beh(\pi) \in B$. We say that C is *reachable* in A if C is (Act^*)-reachable in A.

Reachability problems for CPDS. Given a CPDS A, a MEM-configuration C of A with MEM $\in \{\text{SC},\text{TSO}\}$, and a behaviour language $B \subseteq Act^*$, the *reachability problem* asks whether C is B-reachable in A.

It is well know that the reachability problem is undecidable for SC-configurations with behaviour language Act^*, as 2 stacks suffice to simulate Turing machines. Furthermore, since CPDS with TSO semantics can simulate CPDS with SC semantics, the reachability problem is also undecidable for TSO-configurations (and behaviour language Act^*). However, if we restrict to CFSS, the reachability problem is non-primitive recursive [11].

In the rest of the paper we consider several behaviour languages B in which we study the decidability and complexity of the reachability problem.

3 Processor-Centric Context-Bounded Analysis

In this section we consider *processor-centric context-bounded analysis* (pc-CBA) for CPDS with TSO semantics. A *pc-context* of a CPDS A is a contiguous part of an A run where only transitions from one thread and the memory are allowed. We study both the decidability and the complexity of the reachability problem for CPDS and CFSS under the TSO semantics up to a given number of pc-contexts. We show that the problem is undecidable for CPDS, and decidable with elementary complexity for CFSS.

Formally, let A be a CPDS over a set of thread names T and shared-memory M, and let k be a positive integer. For $t \in T$ we define L_t as the pc-context behaviour language $((Act_T \times Act_M \times \{t\}) \cup (\{(nop, mem\}) \times T))^*$ for thread t. A k *pc-context* behaviour language over T, denoted L_T^k, is the set of all words $w \in Act^*$ which can be factorized as $w_1 w_2 \dots w_k$, where for every $i \in [k]$, $w_i \in L_{t_i}$, for some thread $t_i \in T$. Given a TSO-configuration C of A, the k *pc-context reachability problem* is the problem of deciding whether C is L_T^k-reachable in A.

In the rest of the section we prove the following 2 theorems.

Theorem 1. *For any $k \in \mathbb{N}$ with $k \geq 5$, the k pc-context reachability problem for* CPDS *under* TSO *is undecidable.*

Theorem 2. *For any $k \in \mathbb{N}$, the k pc-context reachability problem for a* CFSS *A under* TSO *is solvable in double exponential time in the size of A and k.*

3.1 Proof of Theorem 1

The undecidability results is given by a reduction from the emptiness problem of the intersection of two context-free languages [22]: for any two PDA A_1 and A_2, we define a CPDS A that can reaches under TSO a special control state within 5 pc-contexts iff there is a word accepted by both A_1 and A_2.

A *pushdown automaton* (PDA) over a finite alphabet Σ is a tuple $D = (Q, q^0, \Gamma, \Delta_\Sigma, F)$, where $\Delta_\Sigma \subseteq \Delta \times \Sigma$, $E = (Q, q^0, \Gamma, \Delta)$ is a PDS, and $F \subseteq Q$. A word $w = a_1 a_2 \dots a_n \in \Sigma^*$ is *accepted* by B iff there is a sequence $C_0 \xrightarrow[E]{\delta_1} C_1 \xrightarrow[E]{\delta_2} \dots C_{n-1} \xrightarrow[E]{\delta_n} C_n$ such that C_0 is the initial configuration of E,

$(\delta_i, a_i) \in \Delta_\Sigma$ for every $i \in [n]$, and $C_n = \langle q_f, \gamma \rangle$ for some $q_f \in F$ and $\gamma \in \Gamma^*$. Define $L(B)$ to be the set of all words in Σ^* accepted by B.

Let A_1 and A_2 be two PDA over Σ. For simplicity's sake, we assume that $\epsilon \notin L(A_1) \cup L(A_2)$ and that in any word $w \in L(A_1) \cup L(A_2)$ there are no two consecutive identical symbols. We define the CPDS A with memory model M and four threads $T = \{t_1, t_2, t_3, t_4\}$ having the property that a configuration in which all threads are in the special control state, say @, is reachable iff there is a word $w \in L(A_1) \cap L(A_2)$; $M = (Var, \Sigma \cup \{\$\}, \eta^0, T)$ with $Var = \{x_1, x_2, x_3, x_4\}$, and $\eta^0(x_i) = \$$ for every $i \in [4]$.

Below we give a concise description of each thread t_i. We assume that all threads (1) never read or write \$ into a variable, and (2) never read consecutively the same symbol from the same variable.

- The description of t_1 is split in two stages. In the first stage, t_1 non deterministically generates a word $w_1 = a_1 a_2 \dots a_n \in \Sigma^+$, one symbol at a time. Each symbol is also pushed into t_1's stack and simultaneously written into x_1. After the first stage, t_1 has w_1^R stored in its own stack.
- Thread t_2, reading symbols from x_1, simulates the PDA A_1. Every symbol read from x_1 is also written into variable x_2. Nondeterministically, t_1 stops the simulation whenever A_1 reaches a final state and enters the special control state @. Let w_2 be the word composed by the sequence of symbols read by t_2 from x_1. Note that, w_2 is a sub-word of w_1 ($w_2 \subseteq w_1$).
- Thread t_3 acts the same as t_2 except that it simulates A_2 and reads from variable x_2 and writes into x_3. Let w_3 be the word read by t_3 from x_2. It is easy to see that $w_3 \subseteq w_2$.
- Thread t_4 reads a word w_4 from x_3 and rewrites w_4^R into x_4 using its stack, and finally enters the control state @. Again, $w_4 \subseteq w_3$.
- In the second stage, t_1 checks whether it can read w_1^R from x_4, where w_1^R is the content of its stack. If this is the case, t_1 enters the control state @.

From above, it is easy to see that when all threads are in the state @ the following property holds: $w_4 \subseteq w_3 \subseteq w_2 \subseteq w_1$ and $w_1 = w_4$; which is true iff $w_1 = w_2 = w_3 = w_4$. Furthermore, $w_2 = w_3$ is also accepted by both A_1 and A_2. Thus, $L(A_1) \cap L(A_2) \neq \emptyset$ iff A reaches in 5 pc-contexts a configuration where all threads are in the control state @, and this concludes the proof.

3.2 Proof of Theorem 2

The proof is given by a reduction to the reachability problem for CPDS under SC semantics constrained to the *bounded-reverse-phase* behaviour language. A bounded-reverse-phase language is defined as follows. For a thread $t \in T$, define $L_t = ((Act_T \times Act_M \times \{t\}) \cup (Act_T \setminus \{push\} \times Act_M \times T))^*$. A word in L_t describes CPDS sub-runs in which only thread t is allowed to take all its transitions, while the other threads are forbidden to use push transitions. For $h \in \mathbb{N}$, a *h-reverse-phase* word w is such that $w \in Act^*$ and can be factorized as $w_1 w_2 \dots w_h$, where for every $i \in [h]$, $w_i \in L_{t_i}$ for some $t_i \in T$. A h-reverse-phase behaviour language

is the set of all *k-reverse-phase* words. For any given $h \in \mathbb{N}$, the k-reverse-phase reachability problem for SC is decidable in double exponential time as shown below.

Theorem 3. *For any $k \in \mathbb{N}$, the k-reverse-phase reachability problem for a* CPDS *A under SC is solvable in double-exponential time in k and $|A|$, where $|A|$ is the size of A.*

Proof. The upper-bound can be shown by a straightforward reduction to the emptiness problem of k-reverse-phase multi-pushdown automata (introduced in [32]) where there is a shared control-state between all the stacks. The latter problem is known to be solvable in double-exponential time in k and exponential time in the size of the model [32,27]. Then there is a trivial reduction from the k-reverse-phase reachability problem for a CPDS A under SC to the emptiness problem of a k-reverse-phase multi-pushdown automaton B by converting A into an automaton without variables and process states (this can be done by encoding the variable valuation and process states in the shared state of B). This will result in an exponential blow-up and so the k-reverse-phase reachability problem for A can be solved in double-exponential-time in k and $|A|$ (since the size of B is exponential in A). □

The reduction is as follows. Let T be the set of thread names of A. We define a CPDS D that non-deterministically *simulates* A along any bounded pc-context using the SC semantics. More specifically, D simulates consecutively each pc-context of A using 2-reverse-phases. Below we only describe the simulation of a single pc-context.

Invariant. At the beginning and the end of the simulation of each pc-context of A, D encodes the configuration of A as follows. D has all threads of A, where for every thread $t \in T$, t encodes the configuration of the thread with the same name in A along with its FIFO queue. More specifically, the control state of t in A is stored in the control state of t in D, and since t does not use its stack at all–as A is a CFSS–the stack of t in D is used to store the FIFO queue σ_t in A with the head pair on the top of the stack. Moreover, the valuation of the shared variables in A is encoded in the shared variables of D. The shared variables of D also include an auxiliary variable used to keep track on whether the automaton is in a pc-context simulation phase. During the simulation of a pc-context an auxiliary thread $s \notin T$ is used. We guarantee that the stack of s is empty whenever D is not in a simulation phase.

Below we describe the 3 steps for the simulation of a pc-context. During the description we also convey a correctness showing that the invariant above holds after the simulation of a pc-context, provided it holds at the very beginning of that pc-context simulation.

Pre-simulation. D non-deterministically selects a thread $t \in T$ that is allowed to progress in the pc-context under simulation. Then, it reverses the content of the stack of t into the stack of s. Note that, the last pair written in the queue of t (in A) is now on the top of the stack of s.

As D copies the stack content, it also computes two pieces of information that are stored in the control state of s.

The first piece of information consists in collecting for each shared variable $x \in Var$ the value corresponding to the last write pair for x, if any, that still resides in the queue. We compute this information to avoid inspecting the stack of s to simulate read operations from t.

The second piece of information η is used to simulate memory operation concerning thread t again to avoid accessing the stack of s. It consists in a sequence of write pairs whose length is bounded by the number of variables of Var. This sequence is defined by the map *lastseq*. For $\sigma = (x_1, d_1) \dots (x_n, d_n) \in (Var \times D)^*$, $lastseq(\sigma)$ is the subsequence of σ in which we remove all pairs (x_j, d_j) such that $x_j = x_i$ and $j < i$. For example, for $\sigma = (y,5)(z,2)(y,4)(x,2)(z,3)(x,1)$, $lastseq(\sigma) = (y,4)(z,3)(x,1)$. η is defined as follows. The queue content γ^R of t in A, where γ is the stack content of t in D at the beginning of the simulation, can be split in two subsequences $\gamma_1\gamma_2$, where γ_2 is the portion of the queue that is dequeued by means of memory operations of t by the end of the simulation of the current pc-context. This partition is not known at the beginning of the simulation, and is non-deterministically guessed by D. We define η as the sequence $lastseq(\gamma_2)$. Again, s uses η to simulate the memory operation from t without using the stack of s. The idea is that only the elements in η are relevant for the simulation as the remaining write pairs will be overwritten by pairs in η by the end of the simulation hence non visible to the other threads.

Since we do not remove elements from the queue (stack of s) during the simulation, we eliminate them only at the end of the simulation when we copy the queue content from the stack of s to that of t. Thus, when the content of the stack of t is reversed into the stack of s at the beginning of the simulation, D non-deterministically guesses the intermediate point between γ_1 and γ_2 and inserts in the stack a separation symbol \$ to remember which part must be discarded. As a remark, it may happen that all pairs in the queue may be used to update the memory in the current pc-round and thus no \$ is inserted in this phase. If this is the case, we need to update the sequence η to keep it consistent as we simulate write operations.

Simulation. After the pre-simulation step, D non-deterministically simulates a sequence of A transitions that may include moves from t and memory transitions of all threads.

A write operation performed by t in A is simulated by pushing the corresponding write pair (x, d) onto the stack of s. Simultaniously, s updates its control to keep track of the last written value for x. Finally, if \$ has not been pushed in the stack yet this pair is also used to update the sequence η by concatenating (x, d) to η and then removing any other existing pair in η for x. After than, s may non-deterministically decide to push \$ onto the stack of s.

A read transition performed by t in A, say on variable x, is simulated by using the last written value for x stored in the control of s, if any, otherwise the value of x in the shared-memory is used. Note that, when we read a value for which we keep track of its value in the control state of s, it may be the case that such

a write pair has already been used to update the shared-memory and we should use this value instead. However, if this is the case these two values coincide as the shared memory cannot be overwritten by any other thread as they are idle in the current pc-context.

A memory transition from the queue of t', for $t' \in T \setminus \{t\}$, is simulated by popping the pair from the stack of t', which contains the head pair of the queue of t', and then by updating the shared-memory accordingly.

To simulate memory transitions from t's queue, we use the sequence η, stored in the control state of s. We remove the leftmost pair from η and update the shared-memory according to it. It is easy to see, that some write pairs are not simulated at all, in particular we do not simulate all write operations that are not captured by η. However, in terms of correctness this is not an issue as all these values will be overwritten in the shared memory by the end of the simulation of the current pc-context by some pair in η.

Restoring the encoding of the reached A configuration. The simulation of the pc-context can non-deterministically end, provided that the sequence η is empty. We restore the configuration of t by copying back the control state of s into t and the content of the stack of s into the stack of t up to the symbol \$ (while discarding the remaining stack content of s).

2 Phase for each pc-context. From the above description it is easy to see that the number of reverse-phases needed to simulate one pc-context are 2: one is required in the first macro step to copy the stack content from t to s, in the second macro step we only pushes on s's stack and hence do not need any extra reverse-phase, and the last step consumes another reverse-phase for the copy of s's stack into the one of t.

4 Memory-Centric Context-Bounded Analysis

In this section, we consider memory-centric context-bounded analysis (mc-CBA) for CPDS and CFSS under TSO semantics. A *mc-context* of a CPDS A is a contiguous part of an A run where only memory transitions concerning the queue of one thread can be performed and no restriction are posed on the actions of all the threads. We study both the decidability and the complexity of the reachability problem for CPDS and CFSS under the TSO semantics up to a given number of mc-contexts. We show that the problem is undecidable for CPDS, and decidable with elementary complexity for CFSS.

Formally, let A be a CPDS over a set of thread names T and shared-memory M, and let k be a positive integer. For $t \in T$, we define L_t as the mc-context behaviour language $((Act_T \times Act_M \times \{T\}) \cup (\{(nop, mem\}) \times \{t\}))^*$ for thread t. A k *mc-context* behaviour language over T, denoted L_T^k, is the set of all words $w \in Act^*$ which can be factorized as $w_1 w_2 \dots w_k$, where for every $i \in [k]$, $w_i \in L_{t_i}$, for some thread $t_i \in T$. Given a TSO-configuration C of A, the k *mc-context reachability problem* is the problem of deciding whether C is L_T^k-reachable in A.

In the rest of the section we prove the following 2 theorems.

Theorem 4. *For any $k \in \mathbb{N}$ with $k \geq 5$, the k mc-context reachability problem for* Cpds *under* Tso *is undecidable.*

Theorem 5. *For any $k \in \mathbb{N}$, the k mc-context reachability problem for a* Cfss *A under* Tso *is solvable in double exponential time in the size of A and k.*

4.1 Proof of Theorem 4

We exploit the construction given in the proof of Theorem 1 to prove the undecidability of the problem.

We show that the Cpds constructed to decide the intersection of the languages accepted by the pushdown automata A_1 and A_2 has also a 5 mc-context run to witness the existence of a common word accepted by both A_1 and A_2, if any. Thread t_1 runs first, until it finishes its first context. Then, synchronously, we interleave the memory transitions on t_1's queue with the transitions of thread t_2 so that it reads the entire words w from x_1 and writes it into its queue on the variable x_2. The same is done for thread t_2 and t_3, and then for t_3 and t_4. Finally actions by t_1 are synchronised with the memory transitions from t_4's queue. It is direct to see that such a schedule leads to a 5 mc-context run, and this concludes the proof.

4.2 Proof of Theorem 5

The proof is given by a reduction to the reachability problem for Cpds under SC semantics constrained to the *bounded-phase* behaviour language. A *phase* captures the dual notion of a reverse-phase, as it represents a contiguous segment of any run in which only one thread can use its stack with no restrictions, instead all the other threads can only push in their own stack. Formally, a bounded-phase language is defined as follows: For a thread $t \in T$, define $L'_t = ((Act_T \times Act_M \times \{t\}) \cup (Act_T \setminus \{pop\} \times Act_M \times T))^*$. A word in L'_t describes Cpds sub-runs in which only thread t is allowed to take all its transitions, while the other threads are forbidden to use pop transitions. For $h \in \mathbb{N}$, a *h-phase* word w is such that $w \in Act^*$ and can be factorized as $w_1 w_2 \dots w_h$, where for every $i \in [h]$, $w_i \in L'_{t_i}$ for some $t_i \in T$. A h-phase behaviour language is the set of all *k-phase* words. For any given $h \in \mathbb{N}$, the k-phase reachability problem for Sc is decidable in double exponential time as for the case of k-reverse-phase reachability problem for Sc (see Theorem 3).

Theorem 6. *For any $k \in \mathbb{N}$, the k-phase reachability problem for a* Cpds *A under SC is solvable in time double-exponential time in k and $|A|$, where $|A|$ is the size of A.*

Proof. The upper-bound can be shown by a straightforward reduction to the emptiness problem of k-phase multi-pushdown automata (introduced in [24]) where there is a shared control-state between all the stacks. The latter problem

is known to be solvable in double-exponential time in k and exponential time in the size of the model [24,32,10]. Then there is a trivial reduction from the k-phase reachability problem for a CPDS A under SC to the emptiness problem of a k-phase multi-pushdown automaton B by converting A into an automaton without variables and process states (this can be done by encoding the variable valuation and process states in the shared state of B). This will result in an exponential blow-up and so the k-phase reachability problem for A can be solved in double-exponential-time in k and $|A|$ (since the size of B is exponential in A).

Before giving the reduction to the bounded-phase reachability problem for CPDS under SC semantics, we show that any mc-phase can be rewritten such that: (1) In the first part of the run, only one thread $t \in T$ is allowed to perform actions and no memory transitions are not allowed for all the threads, (2) and in the second part, only memory transitions concerning the queue of t are allowed and no restrictions are posed on the actions of all threads except the thread t (which is not allowed to perform any action). Formally, for $t \in T$ we define B_t as a restricted mc-context behaviour language $\big((Act_T \times Act_M \times \{t\})^* \cdot ((Act_T \times Act_M \times (T \setminus \{t\})) \cup (\{(nop, mem\}) \times \{t\}))^*\big)$ for thread t. A *k restricted mc-context* behaviour language over T, denoted B_T^k, is the set of all words $w \in Act^*$ which can be factorized as $w_1 w_2 \dots w_k$, where for every $i \in [k]$, $w_i \in L_{t_i}$, for some thread $t_i \in T$. Given a TSO-configuration C of A, the *k restricted mc-context reachability problem* is the problem of deciding whether C is B_T^k-reachable in A.

Let us assume that in a mc-context, we are only performing memory transitions concerning the queue of one process $t \in T$. Then it is easy to see that the execution of t can never be affected by anyone else (since they don't update the memory). Other threads might effect t's execution if they were able to change the configuration of the shared-memory. However, this is not the case as only memory transitions can occur from t's queue. Instead, memory transitions from t do change the state of the shared-memory, but as we now argue, it cannot deviate the course of t's execution. Recall that the behaviour of t depends on: (1) the value of a variable in the memory if there is no pending write for this variable in the queue of t, and (2) the last write operations that are still reside in the queue of t. This means that performing (or not) memory transitions from the queue of t will not affect the behaviour of t. This implies that any mc-context can be reordered such that in we execute first the sequence of actions of the process t and then we execute the sequence of memory transitions and actions of all the other threads. This leads to the fact that the k mc-context reachability problem for a Tso-configuration C of A can be reduced to the k restricted mc-context reachability problem for C (which stated by the following lemma):

Lemma 1. *Given a* TSO*-configuration C of A, C is B_T^k-reachable in A iff C is L_T^k-reachable in A.*

Next, we show that it is possible to reduce the k restricted mc-context reachability problem for a CFSS under TSO to the k-phase reachability problem for a CPDS under SC. The reduction we propose is similar in spirit to the one for the processor-centric case (see Proof of Theorem 2), and here we only sketch the

differences. We define a CPDS D that simulates every restricted mc-context of A with 3 phases. The set of thread names of B is $T \cup \{s\}$, where $s \notin T$ is an auxiliary thread which is employed for the simulation. The invariant we maintain is the following: when the simulation starts and ends thread s is in an *idle state* meaning that it is in a special control state, say @, and its stack is empty. Furthermore, every other B threads $t \in T$ encodes in its configuration the one of t in A: in its control state it is encoded the control state of t in A and the finite sequence $last(\sigma_t)$ where σ_t is the content of t's queue in A, and in stores in its stack σ_t with the head write pair placed on top of the stack.

The simulation goes as follows. Initially s guesses the thread t from which memory transitions can be executed. The content of t's stack is transferred into s's stack, where now the tail of t's queue is stored on the top of s's stack. Also the control state of t as well as $last(\sigma_t)$ is copied into the control state of s.

Now, we first simulate the moves of t and only after the moves of the remaining threads along with the memory transitions concerning t's queue (in order to respect the definition of restricted mc-context). The simulation of t is as follows. For write operations we update $last(\sigma_t)$ as described in Section 3.2, and push the produced pair on the stack of s. Read operations, instead, will consult $last(\sigma_t)$ to get the value of the read variable, if any, otherwise it recovers the value from the memory.

In the second stage of the simulation we restore back into t's stack the content of s's stack as well as the control state. The sequence *last* will not be copied as it will change after memory operations will be performed. Such sequence is reconstructed at the end of the simulation.

We now simulate all the other threads and memory updates in arbitrary order. Memory transitions are simulated as expected by popping pairs from t's stack and updating the memory accordingly. Transitions of other threads, say $\hat{t}$ are simulated straightforwardly by using $last(\sigma_{\hat{t}})$ and the shared memory in a similar as we have done for t.

Non deterministically the simulation ends and the invariant is reestablished by computing $last(\sigma_t)$. For such a purpose we need to inspect entirely t's stack. Thus we copy it back and forth to the s's stack by paying one more phase. Finally s enters into the special control state @ and the simulation ends.

By using the same argument as in Section 3.2 we can show that the above construction of B allows to reduce in polytime the k restricted mc-context reachability problem for CFSS under TSO to the 3k-phase reachability problem for CPDSunder SC. Thus, from Lemma 1 and Theorem 6 we can state the main result of the section.

Theorem 7. *For any positive integer k, the k mc-context reachability problem for a* CFSS A *under* TSO *is solvable in double exponential time in $|A|$ and k.*

5 Process-Memory Centric Context-Bounded Analysis

In this section, we consider process-memory centric context-bounded analysis (pmc-CBA) for CPDS with TSO semantics. A context, in this case called *pmc-context*, of a CPDS A is a contiguous part of an A computation where only one

processor thread is active, and only its store buffer can be used for memory updates. We consider here the reachability problem for CPDS up to a bounded number of pmc-contexts. We recall that the pmc-CBA for CPDS (resp.TSO-CFSS) with TSO semantics is reducible to the standard context-bounded analysis for CPDS (resp. CFSS) with SC semantics which is known to be decidable [34].

Next, we formally define the bounded pmc-reachability problem for TSO-CPDSs. Let A be a CPDS over thread names T and shared-memory model M, and let k be a positive integer. The *pmc-context language* L_t of a thread $t \in T$ is the set $\big(((Act_T \times Act_M) \cup \{(nop, mem)\}) \times \{t\}\big)^*$. A k pmc-context behavior language over T, denoted L_T^k, is the set of all words $w \in Act_A^*$ which can be factorized as $w_1 w_2 \cdots w_k$, where for every $i \in [k]$, $w_i \in L_{t_i}$, for some thread $t_i \in T$.

Given a MEM $\in$ {SC,TSO} and MEM-configuration C of A, the *k pmc-context reachability problem* is the problem of deciding whether C is L_T^k-reachable in A.

Theorem 8 ([13]). *For any $k \in \mathbb{N}$, the k pmc-context reachability problem for* CPDS *(resp.* CFSS*) under* TSO *is reducible to the k- pmc-context reachability problem for* CPDS *(resp.* CFSS*) under* SC *semantics.*

Moreover, we have:

Theorem 9. *For any $k \in \mathbb{N}$, the k- pmc-context reachability problem for* CPDS *(resp.* CFSS*) under* SC *semantics is solvable in nondeterministic exponential time in k and $|A|$, where $|A|$ is the size of A.*

Proof. The upper-bound can be shown by a straightforward reduction to the reachability problem of k-context multi-pushdown systems (introduced in [34]) where there is a shared control-state between all the stacks. The latter problem is known to be solvable in non-deterministic polynomial time in k and the size of the system [29]. It is easy to see that there is a trivial reduction from the k-pmc-context reachability problem for a CPDS A under SC to k-context multi-pushdown system B by encoding all the process states and the valuation of the memory into one single state. This will result in an exponential blow-up and so the k-pmc-context reachability problem for A can be solved in nondeterministic exponential-time in k and $|A|$ (since the size of B is exponential in A).

As an immediate corollary of Theorem 8 and Theorem 9, we obtain:

Theorem 10. *For any $k \in \mathbb{N}$, the k pmc-context reachability problem for* CPDS *(resp.* CFSS*) A under* TSO *is decidable and can be solved in nondeterministic exponential-time in k and $|A|$.*

6 Conclusion

We have considered three different notions of context-bounded analysis for TSO computations, depending on whether a processor, or a memory, or a processor and memory centric view is adopted. We have shown that each of these three

notions allows to cut-off drastically the complexity of checking state reachability w.r.t. the unrestricted case, although of course the analysis is under-approximate. The work we present in this paper allows to improve our understanding of the trade-offs between expressiveness, decidability, and complexity of checking state reachability under TSO semantics.

While pmc-CBA was already introduced in our previous work [13], this work introduces two other natural and more general concepts of pc-CBA and pm-CBA for which the complexity of the TSO state reachability problem is still elementary. In terms of coverage, pc-CBA and mc-CBA are incomparable while both of them are strictly more general than pmc-CBA. Indeed, these two analyses allow to capture with a given bound on the pc/mc context switches sets of behaviors that would need an unbounded number of pmc context switches. However, this increase in power comes with a price. First, while pcm-CBA is decidable even for recursive programs (pushdown threads), both pc-CBA and mc-CBA are undecidable in this case. For programs without recursive procedures, pmc-CBA is in NEXPTIME while both pc-CBA and mc-CBA are in 2EXPTIME.

An interesting question left for future work is whether the analyses presented here for TSO can be extended to other weak memory models.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Automatic fence insertion in integer programs via predicate abstraction. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 164–180. Springer, Heidelberg (2012)
2. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 204–219. Springer, Heidelberg (2012)
3. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Memorax, a precise and sound tool for automatic fence insertion under TSO. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 530–536. Springer, Heidelberg (2013)
4. Adve, S.V., Hill, M.D.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. 4(6), 613–624 (1993)
5. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: Definitions, implementation, and programming. Distributed Computing 9(1), 37–49 (1995)
6. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013)
7. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013)
8. Alglave, J., Maranget, L.: Stability in weak memory models. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 50–66. Springer, Heidelberg (2011)
9. Aspinall, D., Ševčík, J.: Formalising java's data race free guarantee. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 22–37. Springer, Heidelberg (2007)

10. Atig, M.F., Bollig, B., Habermehl, P.: Emptiness of multi-pushdown automata is 2ETIME-complete. In: Ito, M., Toyama, M. (eds.) DLT 2008. LNCS, vol. 5257, pp. 121–133. Springer, Heidelberg (2008)
11. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL, pp. 7–18. ACM (2010)
12. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: What's decidable about weak memory models? In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 26–46. Springer, Heidelberg (2012)
13. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)
14. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 107–123. Springer, Heidelberg (2009)
15. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013)
16. Bouajjani, A., Emmi, M., Parlato, G.: On sequentializing concurrent programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 129–145. Springer, Heidelberg (2011)
17. Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 428–440. Springer, Heidelberg (2011)
18. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 392–403. ACM (2009)
19. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
20. Burnim, J., Sen, K., Stergiou, C.: Testing concurrent programs on relaxed memory models. In: ISSTA, pp. 122–132. ACM (2011)
21. Friedman, R.: Consistency Conditions for Distributed Shared Memories. Phd. thesis, Technion: Israel Institute of Technology (1994)
22. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation - international edition, 2nd edn. Addison-Wesley (2003)
23. Kuperstein, M., Vechev, M.T., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: PLDI, pp. 187–198. ACM (2011)
24. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: LICS, pp. 161–170. IEEE Computer Society (2007)
25. La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI, pp. 211–222. ACM (2009)
26. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
27. La Torre, S., Napoli, M., Parlato, G.: On the complement of multi-stack visibly pushdown languages. Technical report (2014)
28. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)

29. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)
30. Linden, A., Wolper, P.: An automata-based symbolic approach for verifying programs on relaxed memory models. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 212–226. Springer, Heidelberg (2010)
31. Luchango, V.: Memory Consistency Models for High Performance Distributed Computing. Phd. thesis, Massachusetts Institute of Technology (2001)
32. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: Ball, T., Sagiv, M. (eds.) POPL, pp. 283–294. ACM (2011)
33. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010)
34. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
35. Saraswat, V.A., Jagadeesan, R., Michael, M.M., von Praun, C.: A theory of memory models. In: Yelick, K.A., Mellor-Crummey, J.M. (eds.) PPOPP, pp. 161–172. ACM (2007)
36. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7), 89–97 (2010)

A Model of Dynamic Systems

Manfred Broy

Institut für Informatik, Technische Universität München
80290 München Germany
broy@in.tum.de
http://wwwbroy.informatik.tu-muenchen.de

Abstract. We introduce a model describing discrete dynamic distributed systems. These are systems where their set of connections to the systems in their context captured by their syntactic interfaces as well as the set of their subsystems, and their set of internal connections in their architectures between their subsystems change dynamically over time. To provide such a model we generalize the static system model of Focus (cf. [8]) in terms of their system interfaces and their interface behavior, their system architectures, and their system models in terms of state machines to model dynamic systems. We deal with concepts of causality, composition, abstraction, and system specification for dynamic systems. We analyze properties of dynamic systems and discuss how well the model captures general notions of system dynamics. Finally, we introduce the concept of system classes and their instantiation, which introduces an additional concept of dynamicity.

Keywords: Dynamic Systems, Mobility, Instantiation

1 Introduction: Dynamic Systems

FOCUS (see [8]) is an approach to system modeling where the interfaces of systems and their behaviors are described in terms of streams of interactions exchanged via their input and output channels. Architectures of systems are modeled by sets of subsystems with their interface behavior and their mutual connections by channels. Implementations of systems are described in terms of state machines. FOCUS follows the idea of distributed concurrent data flow.

Systems operate in a timeframe described by a sequence of time intervals called time slots. A time slot behavior of a system is given by a syntactic interface consisting of a set of input channels I and a set of output channels O and a mapping that maps valuations of the input channels to sets of valuations of the output channels. A syntactic interface is defined by a set of input channels and a set of output channels together with their types. In FOCUS this syntactic interface does not change over time. This notion of a syntactic interface is static in FOCUS.

Behaviors of systems then are mappings that associate with every time interval, where time intervals are represented by the natural numbers, a time slot behavior. In a behavior we deal with a set of channels. This is the set of all channels that occur in

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 39–53, 2014.

one time slot behavior. These sets of channels can be infinite, in principle. We assume, however, that in dynamic systems in each time slot the set of channels is finite. For simplicity we do not allow channels occurring both as input and output channels for one component.

An input channel is called *static* for a system if the channel occurs in that system as an input channel in every time slot. In analogy, an output channel is called *static,* if it occurs as an output channel in every time slot.

Composition is defined in FOCUS by composition of the interface behaviors in the time slots where we require that whenever in a composition there is a feedback loop, then the output channel in that time slot is strongly causal. Otherwise composition with feedback could not be properly defined. A simple way to achieve that is to require that output channels are always strongly causal or that feedback does only occur for strongly causal output channels.

In FOCUS, system interfaces are static in the sense that the sets of input and output channels are invariant over the lifetime of a system. Therefore the syntactic interface in FOCUS is static.

In FOCUS besides the notion of an interface, state machine, and architectures are worked out as elements of system models. Both refer to interfaces and both are static in the sense that they support static interfaces. As a consequence, interfaces of architectures are static and thus the architectures are static in the sense that they only support static architectures where the connections between their subsystems do not change over the lifetime of the systems.

In the following we introduce a new, simple, but powerful model for dynamic systems, which are systems where their number of channels (the syntactic interface), and their connections between the subsystems are not necessarily static but may change over time. Basically, we do that by assigning a possibly modified syntactic interface to systems for every time slot of the system.

Causality and strong causality of behaviors is defined as usual.

In the following we introduce such a model for dynamic systems. We first introduce the basics for modeling interactive systems such as streams and histories. Then we introduce the concept of a dynamic interface behavior and finally the idea of dynamics for specifications, state machines, and architectures as well as classes of systems introducing the concept of instantiation.

2 The Dynamic System Model

In this section we introduce the dynamic system model.

2.1 System Propaedeutic

Our approach uses a specific notion of discrete system with the following characteristics and principles.

- A discrete *system* has a well-defined boundary (its “scope”) that determines its *interface*.

  - Everything outside the system boundary is called the system's *environment*. Those parts of the environment that are relevant for the system are called the system's *context*. Actors in the context that interact with the system such as users, neighbored systems, or sensor and actuators connected to the physical environment are called its *operational context*.
  - By a system's interface it is indicated by which steps the system interacts with its operational context. The syntactic interface defines the set of actions that can be performed in interaction with a system over its boundary. In our case, syntactic interfaces are defined by the set of input and output channels together with their types. The input channels and the types determine the input actions for a system while the output channels and their types determine the output actions for a system.
  - We distinguish between *syntactic interface*, also called *static interface*, which describes the set of input and output actions, which can take place over the system boundary, and the *interface behavior* (also called *dynamical interface*), which describes the system's *functionality* in terms of the input and output actions; the interface behavior is captured by the causal relationship between streams of actions captured in input and output *histories*. This way we define a logical behavior as well as a probabilistic behavior for systems.
  - The logical interface behavior of systems is described by logical expressions, called *interface assertions*, by *state machines,* or it can be further decomposed into *architectures*.

- A system has an *internal structure* and some internal *behavior* ("glass box view"). This structure is described by
  - its state space with state transitions and/or
  - its decomposition in sub-systems forming its architecture in case the system is decomposed into a number of subsystems, which interact and also provide the interaction with the system's operational context. The state machine and the architecture associated with a system are called its state view and its structural or architectural view, respectively.
- Complementary, the behaviors of systems can be described by sets of *traces*, which are sets of scenarios of input and output behavior of systems. We distinguish between finite and infinite scenarios.
- Moreover, systems operate in real time. In our case we use discrete time, which seems, in particular, adequate for discrete systems.
- Systems interact with their operational context and sub-systems operate concurrently within the system's architectures.

This gives a highly abstract and at the same time quite comprehensive model of systems. This model is formalized in the following by one specific modeling theory.

This system propaedeutic leads to following notions of a dynamic system

- *dynamic interface behavior*: the syntactic interface changes over time
- *dynamic architecture*: the syntactic interface changes over time
- *dynamic state space* and state transition

In the following we introduce a very compact model that addresses all three aspects of dynamicity.

2.2 Interface Behavior Model

The key concepts for modeling system behavior are streams and communication channels to represent interaction and connections.

Channels and Histories. For the alternative model for dynamic systems we use the following sets

M	universe of all data elements such as messages or values of state attributes,
TYPE	the set of (data) types (each type T in TYPE is a subset of M),
C	the set of typed channels.

The type of the channels in the channel set C is determined by a channel type assignment:

$$type: C \to TYPE$$

By [type(c)] we denote the set of elements associated with type(c). We assume that the types of channels are static, which means they do not change over the lifetime of a channel.

An *interaction pattern* for the set C of typed channels is given by a partial function

$$p: C \to M^*$$

where $Dom(p) \subseteq C$ denotes the set of channels c for which p(c) is defined; furthermore we assume $p(c) \in [type(c)^*]$. An interaction pattern represents the sequences of messages exchanged over the channel set Dom(C) within a time slot.

The set of interaction patterns is denoted by $C^{\rightarrow}$ which is defined by the equation

$$C^{\rightarrow} = \{p: C \to M^*: p \text{ partial } \}$$

We say that the channel c is *active* in pattern p if $c \in Dom(p)$.

Note the subtle difference between the case where $p(c) = \langle\rangle$, which represents the situation where channel c is active but no messages are transmitted, and $c \notin Dom(p)$, which represents the case where channel c is not active and thus there is no specified communication via channel c in that time slot. Then c is actually at that time slot considered not being present as a channel.

A *dynamic channel valuation* (also called a *dynamic history*) for a set C of channels is given by a function:

$$x: \mathbb{N}\backslash\{0\} \to C^{\rightarrow}$$

It specifies for each time slot an interaction pattern. We denote the set of all dynamic valuations of the channels in C by

$$C^{\Rightarrow}$$

We say that channel c is *present* (or *active*) in the history $x \subseteq C^{\Rightarrow}$ at time t if $c \in \mathrm{Dom}(x.t)$. For $x \in C^{\Rightarrow}$ we write x.t.c with time $t \in \mathbb{N}\backslash\{0\}$ and channel $c \in C$ to denote the sequence (x(t))(c) transmitted on channel c in history x in the time slot t provided $c \in \mathrm{Dom}(x.t)$; furthermore we write x(c) for the partial mapping $\mathbb{N}\backslash\{0\} \to$ [type(c)*] where (x(c))(t) is equal to (x(t))(c).

Given the dynamic history $x \in C^{\Rightarrow}$ we denote for given time slot $t \in \mathbb{N}$ by $x{\downarrow}t$ the restriction of the history x to the time slots {1, …, t}:

$$x{\downarrow}t : \{1, \ldots, t\} \to C^{\rightarrow}$$

where

$$(x{\downarrow}t).(k) = x(k) \Leftarrow 1 \leq k \leq t$$

$x{\downarrow}t$ is also called the finite "partial" history of history x till time slot t. A history $x \in C^{\Rightarrow}$ is called *static* if for all times t, t' $\in \mathbb{N}\backslash\{0\}$ Dom(x(t)) = Dom(x(t')). A history $x \in C^{\Rightarrow}$ is called *dynamically finite* if Dom(x(t)) is finite for all $t \in \mathbb{N}\backslash\{0\}$.

Merging Histories. The parallel composition of dynamic channel histories is specified as follows. Given two sets X, Y of channels with consistent types and two dynamic histories $x \in X^{\Rightarrow}$, $y \in Y^{\Rightarrow}$ we define the set of dynamic histories $x \oplus y \subseteq (X \cup Y)^{\Rightarrow}$ composed from histories x and y by the equation

$$\begin{aligned} x \oplus y = \{z \in (X \cup Y)^{\Rightarrow} : \forall\, t \in \mathbb{N}\backslash\{0\} &: \mathrm{Dom}(z.t) = \mathrm{Dom}(x.t) \cup \mathrm{Dom}(y.t) \\ \wedge\ \forall\, c \in (X \cup Y) &: c \in \mathrm{Dom}(z.t) \Rightarrow \\ & (z.t.c = x.t.c \wedge c \notin \mathrm{Dom}(y.t)) \\ \vee\ & (z.t.c = y.t.c \wedge c \notin \mathrm{Dom}(x.t)) \\ \vee\ & (z.t.c \in \mathrm{merge}(x.t.c, y.t.c) \wedge \\ & \qquad c \in \mathrm{Dom}(x.t) \cap \mathrm{Dom}(y.t)))\} \end{aligned}$$

The set-valued function merge yields the set of all interleavings of two finite sequences. It is easily specified as follows:

$$\mathrm{merge}(s_1, s_2) = \{s \in M^* : \exists\, e \in \{1, 2\}^* : \mathrm{proj}(s, e, 1) = s_1 \wedge \mathrm{proj}(s, e, 2) = s_2\}$$

where

$$\mathrm{proj} : M^* \times \mathbb{N}^* \times \mathbb{N} \to M^*$$

is specified by the following equations:

$$\begin{aligned} &\mathrm{proj}(\langle m \rangle \hat{}\, s, \langle k \rangle \hat{}\, e, k) = \langle m \rangle \hat{}\, \mathrm{proj}(s, e, k), \\ &\mathrm{proj}(\langle m \rangle \hat{}\, s, \langle k \rangle \hat{}\, e, k') = \mathrm{proj}(s, e, k) \Leftarrow k \neq k' \\ &\mathrm{proj}(\langle\rangle, s, k) = \mathrm{proj}(s, \langle\rangle, k) = \langle\rangle \end{aligned}$$

As a result $x \oplus y$ denotes the history in which or each channel $c \in (X \cup Y)$ the sequence $x \oplus y.t.c$ is the result of merging the sequences x.t.c and y.t.c provided both are defined, otherwise it is equal to x.t.c or to y.t.c depending on which of these is defined and undefined if both are undefined. For $C' \subseteq C$ and $x \in C^{\Rightarrow}$ we denote by $x|C'$ the

restriction of history x to channels in C' which is the history $z \in C^{\Rightarrow}$ where z.t.c = x.t.c for $c \in C'$ and $t \in \mathbb{N}\backslash\{0\}$ holds.

Interface Behavior. Given two sets I and O of typed channels, we denote the syntactic interface of a dynamic system by $(I \blacktriangleright O)$. A nondeterministic (or under-specified) component behavior (let I and O be sets of typed channels) is represented by the function

$$F: I^{\Rightarrow} \to \wp(O^{\Rightarrow})$$

This function models the behavior of a dynamic component. At every time $t' \in \mathbb{N}\backslash\{0\}$ for given input history x the set Dom(x(t)) denotes the set of channels active at time t as input channels and for $y \in F(x)$ Dom(y(t)) denotes the channels active at time t as output channels.

Given input history $x \in I^{\Rightarrow}$ in each time slot t a sub-interface $(I' \blacktriangleright O')$ with $I' \subseteq I$ and $O' \subseteq O$ of channels is active. It is specified by the sets

$$I' = \mathrm{Dom}(x.t)$$
$$O' = \mathrm{Dom}((y) \text{ where } y \in F(x).(t)$$

$(I' \blacktriangleright O')$ is called the *active syntactic interface at time slot* t for input x and output y. If for all t and all input histories that are static the syntactic interfaces are identical we call the system *static*.

Definition. Causal Interface Behavior
For a mapping

$$F: I^{\Rightarrow} \to \wp(O^{\Rightarrow})$$

we define the set

$$\mathrm{dom}(F) = \{x: F(x) \neq \emptyset\}$$

called the *domain* of F. F is called *total*, if $\mathrm{dom}(F) = I^{\Rightarrow}$, otherwise F is called *partial*.

The mapping F is called *causal*, if (for all $t \in \mathbb{N}$ and all input histories $x, z \in I^{\Rightarrow}$):

$$x, z \in \mathrm{dom}(F) \wedge x{\downarrow}t = z{\downarrow}t \Rightarrow \{y{\downarrow}t: y \in F(x)\} = \{y{\downarrow}t: y \in F(z)\}$$

F is called *strongly causal*, if (for all $t \in \mathbb{N}$ and all input histories $x, z \in I^{\Rightarrow}$):

$$x, z \in \mathrm{dom}(F) \wedge x{\downarrow}t = z{\downarrow}t \Rightarrow \{y{\downarrow}t+1: y \in F(x)\} = \{y{\downarrow}t+1: y \in F(z)\} \quad \square$$

Causality (for an extended discussion see [8]) indicates a consistent time flow between input and output histories in the following sense: in a causal mapping input messages received at time t may influence future output only after time t; this output is given by messages communicated via output channels at times $\geq$ t (in the case of strong causality at times > t, which indicates that there is a delay of at least one time step before input has any effect on output).

Definition. I/O-Behavior
A causal mapping $F: I^{\Rightarrow} \to \wp(O^{\Rightarrow})$ is called a *dynamic I/O-behavior*. By $\mathbb{DIF}[I \blacktriangleright O]$ we denote the set of all (total and partial) dynamic interface behaviors called I/O-behaviors with syntactic interface $(I \blacktriangleright O)$ and by $\mathbb{DIF}$ the set of all I/O-behaviors. $\square$

Interface behaviors model system functionality. For systems we assume that their interface behavior is total. Behaviors F may be deterministic (in this case, the set F(x) of output histories has at most one element for each input history x) or nondeterministic (which allows us to model under-specification). For simplicity we assume throughout the paper that the sets of input and output channels of components are disjoint.

Hiding Channels. Given $F \in [I \blacktriangleright O]$ we hide a channel $c \in I \cup O$ leading to a system with Interface behavior $F \backslash c \in [I' \blacktriangleright O']$ where $I' = I \backslash \{c\}$ and $O' = O \backslash \{c\}$ where for $x' \in I'^{\vec{\ }}$ we define the behavior of $F \backslash c$ by

$(F \backslash c)(x') = F(x) | O'$ where $x \in I^{\vec{\ }}$ is defined as follows

$x.t.c = \langle\rangle$ for all $t \in \mathbb{N}$ and $x(c') = x'(c')$ for $c' \in I'$

Hiding means that we close the channel c to the outside world. F' cannot receive input on c nor produce output on c.

2.3 Composition of Dynamic Systems

In this section we describe the composition of systems in terms of their interface behavior. We show how to calculate the interface behavior of a composed system from the interface behaviors of its components.

The composition of two systems

$$F_1: I_1^{\vec{\ }} \to \wp(O_1^{\vec{\ }}) \text{ and } F_2: I_2^{\vec{\ }} \to \wp(O_2^{\vec{\ }})$$

yields an I/O-behavior $(F_1 \otimes F_2)$ with syntactic interface $(I \blacktriangleright O)$ where $O = O_1 \cup O_2$ and $I = (I_1 \cup I_2) \backslash (O_1 \cup O_2)$ that are strongly causal is specified by the following formula:

$$(F_1 \otimes F_2).x = \{y | (O_1 \cup O_2) : \exists\, y_1, y_2 : y = x \oplus y_1 \oplus y_2 \wedge y_1 \in F_1(y | I_1) \wedge y_2 \in F_2(y | I_2)\}$$

where $x \in ((I_1 \cup I_2) \backslash (O_1 \cup O_2))^{\vec{\ }}$, $y \in (O_1 \cup O_2 \cup I_1 \cup I_2)^{\vec{\ }}$. $(F_1 \otimes F_2)$ is strongly causal again.

The composition of systems is commutative:

$$F_1 \otimes F_2 = F_2 \otimes F_1$$

as well as associative:

$$(F_1 \otimes F_2) \otimes F_3 = F_1 \otimes (F_2 \otimes F_3)$$

The proof of these equations is straightforward.

This way we get a model for dynamic systems, which is more general than the static FOCUS model. It is concise and surprisingly better adapted to describing the dynamics of systems. The model is a direct extension of the FOCUS model of static systems.

2.4 Dynamic State Machines by State Transition Functions

A state space over a given space set V of typed attributes is a set of mappings

$$\sigma: V \to D$$

where D is the universe of all data and for all attributes $v \in V$ the value $\sigma(v)$ is of the type associated with attribute v. A dynamic state space over a set V of typed attributes is the set of partial mappings

$$\sigma : V \to D$$

where for all attributes in $Dom(\sigma)$ the value $\sigma(v)$ is of the type associated with attribute v.

Given V we denote by $\Sigma(V)$ the dynamic state space over V.

State machines with input and output describe system implementations in terms of states and state transitions. A state machine is defined by a state space and a state transition function.

Definition. Dynamic State Machine with Syntactic Interface ($I \blacktriangleright O$)
Given a set V of typed attributes, a state machine (Δ, Λ) with input and output according to the syntactic interface ($I \blacktriangleright O$) consists of a set $\Lambda \subseteq \Sigma(V)$ of initial states as well as of a nondeterministic state transition function

$$\Delta: (\Sigma(V) \times I^{\rightarrow}) \to \wp(\Sigma(V) \times O^{\rightarrow}) \qquad \square$$

For each state $\sigma \in \Sigma(V)$ and each valuation $a \in I^{\rightarrow}$ of the input channels in I by sequences of input messages every pair $(\sigma', b) \in \Delta(\sigma, a)$ defines a successor state σ' and a valuation $b \in O^{\rightarrow}$ of the output channels consisting of the sequences produced by the state transition in one time slot. In every step of the dynamic state machine the structure of the state space and the sets of active input and output channels may change – more precisely the set of active attributes, the set of active input, and the set of active output channels may change.

(Δ, Λ) is a *Mealy machine* with possibly infinite state space. If in every transition $(\sigma', b) \in \Delta(\sigma, a)$ the output b depends on the state σ only but never on the current input a, we speak of a *Moore machine*.

2.5 Dynamic Architectures

In this section, we describe how to form dynamic architectures by composing dynamic sub-systems, called the dynamic components of the architecture. Architectures are concepts to structure systems. Architectures contain precise descriptions for systems in terms of their sub-systems and how the composition of their sub-systems takes place. In other words, architectures are described by the sets of systems in their set of components together with mappings from input channels to output channels that describe internal communication. Architectures form a data flow network.

In the following we assume that each system used in an architecture as a component is identified by a unique identifier k. Let K be the set of identifiers for the components of an architecture.

Definition. Interpreted Architecture
An interpreted architecture (K, ξ, ψ) for set K of component names associates a syntactic dynamic interface $\xi(k) = (I_k \blacktriangleright O_k)$ with every component identifier $k \in K$ and dynamic interface behavior $\psi(k) \in \mathbb{IF}[I_k \blacktriangleright O_k]$, with every component identifier $k \in K$.

$\square$

An architecture can be specified by a syntactic architecture and an interface specification for each of its components.

The interface behavior of an architecture A = (K, ξ, ψ) is given by a mapping

$$F_A : \vec{I} \rightarrow \wp(\vec{O})$$

where (here all channels in I and O can be used both as internal channels and as external channels)

$$I = \cup \{I_k : k \in K\} \qquad O = \cup \{O_k : k \in K\}$$

and the interface behavior F_A of A can be calculated from the interface behaviors of the components:

$$F_A = \otimes \psi(k)$$

In this construction we do not hide "internal" channels, which are channels that lead inside the architecture from one sub-system to another one. This means that we can observe in behavior $F_A(x)$ the communication on internal channels at the system interface since internal channels are also output channels. Of course, we may also introduce more abstract concepts of composition hiding internal channels.

Given an input history x for an architecture we get a static architecture in every time slot. It is defined by a directed graph with the set K of components as its nodes. A channel c defines a connection at time slot t from component k to component k' if

$$c \in Dom((y|O_k).t)$$

and

$$c \in Dom((\psi(k)(y|I_{k'})).t)$$

where

$$y = F_A(x)$$

This way we characterize temporary connections. Given input $x \in \vec{I}$ and output $y \in \vec{O}$, where t is the set of all channels, let $Out_t(k)$ be the set of all channels that are active and lead from component k at time t and $In_t(k)$ be the set of all channels that are active and lead to component k at time t.

By construction a channel is, in general, a multi-connector. At each time slot it connects a set of components that have this channel as input channel with a set of components that have this channel as output channel. As defined components cannot have a channel both as input and output channel.

As a result a channel has in each step a number of active participants

- senders, that issue output to the channels but do not read input from the channel
- receivers, that consume messages as input from the channel

There are several ways the concept of a channel can be used. In a restricted application of the concept of a channel, a channel connects at each time exactly two subsystems, one sender and one receiver.

Given an input history $x \in I^{\rightarrow}$ for the architecture and some output we denote for $t \in \mathbb{N}$ by $Out_t(k)$ the set of active channels in $[(x \oplus y)|I_K](t)$; these are the channels active in the architecture at time t. A component $k \in K$ is said to *have a past* at time t (otherwise it is called *unborn* at time t) if

$$(\exists\, t' \in \mathbb{N} : t' \leq t \wedge \exists\, c: c \in Out_t(k) \cup In_t(k))$$

A component $k \in K$ is said to *have a future* at time t (otherwise it is called *dead* at time t) if

$$(\exists\, t' \in \mathbb{N} : t \leq t' \wedge \exists\, c: c \in Out_t(k) \cup In_{t'}(k))$$

A component $k \in K$ is called *present* at time t if it has a past and a future (otherwise it is called *inactive* at time t), i.e. if

$$(\exists\, t' \in \mathbb{N} : t' \leq t \wedge \exists\, c: c \in Out_t(k) \cup In_{t'}(k))$$
$$\wedge\, (\exists\, t' \in \mathbb{N} : t \leq t' \wedge \exists\, c: c \in Out_t(k) \cup In_t(k))$$

In other words, component k is involved in communications before and after time slot t.

In each time slot an architecture forms a directed graph, which we call its data flow graph. It consists of all components that are present and all their temporarily active channels. Note that in the graph we may find components that are not connected to any channel. They still may compute (in terms of internal state transitions) and may only later get connected to other components. This graph may change over time. This way a dynamic architecture is modeled. If all components are static, the architecture is static.

2.6 System Interface Behavior: Specification by Interface Assertions

The interface behavior of a system can be specified in a descriptive logical style using interface assertions.

Definition. Interface Assertion
Given a syntactic interface (I▶O) with a set I of typed input channels and a set O of typed output channels, an interface assertion is a logical formula with channel identifiers in I and O as free logical variables denoting streams of the respective types. ❑

We specify the behavior F_S for a system with name S with syntactic interface (I▶O) and an interface assertion P by a specification scheme:

S

in	I
out	O
P	

P is called interface assertion.

The scheme specifies the set of all strongly causal interface behaviors F_S of the system with name S which fulfill the formula

$$\forall x \in I^{\Rightarrow}, y \in O^{\Rightarrow}: y \in F_S(x) \Rightarrow P(x, y)$$

where P(x, y) results from P by replacing all channels c occurring in assertion P by streams x(c) or y(c), respectively.

It is more convenient to replace x(c) or y(c) simply by the channel identifier c, where we use the convention that for channels c that are both input and output channels we use c' for y(c) to distinguish x(c) from y(c).

Note that causality properties are implicitly assumed for each specification this way. If an interface behavior F_s that fulfills the interface assertion does not exist then the specification is called inconsistent.

In FOCUS interface assertions are formulas in predicate logic where channels denote streams. Since both channels and sub-systems may be inactive at certain time slots, we need special notations expressing that fact in interface assertions.

The interface assertions for dynamic systems may become more sophisticated due to the fact, that channels may be active or not. We specify a proposition

$$c@t$$

that yields true if channel c is active at time slot t. If we write c.t = … with a defined expression at the right hand side, this allows us to conclude c@t.

In addition, we allow communicating channels as messages (as in π–calculus, see [28]).

A typical example of a simple specification with type C = {c1, c2, c3} reads as follows (by c#s we donate the number of occurrences of c in sequence s)

S
in cha: C, x: Nat
out c1: Nat, c2: Nat, c3: Nat
$\forall t \in \mathbb{N}$:
$\forall c \in C$:
$even(c\#(cha{\downarrow}t)) \Leftrightarrow c@(t+1)$
$c@t+1 \Rightarrow c.t+1 = x.t$

This is an example of a component with three dynamic output channels that forward the input from channel x provided they are active. They get activated by sending their channel id on channel cha and deactivated by sending it once more.

3 Discussion

What we obtain by the introduced concepts is a model that models dynamic systems, however, for the price that syntactic interfaces are no longer static. In each time slot a dynamic system may feature a different syntactic interface.

In FOCUS generally composition of two specifications with interface assertions Q_1 and Q_2 is simply given by the formula

$$Q_1 \wedge Q_2$$

due to the requirement that in Focus their sets of output channels are disjoint. As long as this holds in every time slot, the same formula can be applied for dynamic systems.

Otherwise – and in our case, where we share output channels, a more sophisticated formula for composition is needed, where we have to distinguish between the output of Q_1 and Q_2 on shared channels and their merge. Let c be such a shared channel. We get the interface assertion in the case c is the only shared channel by the assertion (assume that channels c_1 and c_2 are not free in Q_1 or Q_2)

$$\exists\, c_1, c_2: (\exists\, c: Q_1 \wedge c_1 = c) \wedge (\exists\, c: Q_2 \wedge c_2 = c) \wedge c = c_1 \oplus c_2$$

which is equivalent to

$$\exists\, c_1, c_2: Q_1[c_1/c] \wedge Q_2[c_2/c] \wedge c = c_1 \oplus c_2$$

The generalization to a set of shared output channels is straightforward.

4 Parameterized Interfaces and Systems

Following the idea of [8] to consider indexed sets of channels and systems, which are called sheaves. By then we generalize the concept of channel and system identifier to families ("sheaves") by introducing indexes. This is an idea following concepts of object orientation where we replace object identifiers by index values and objects by system interface behavior.

Let K be an arbitrary set of index values (a simple choice would be $K \subseteq IN$); then an indexed channel is a channel name c of the type

$$c: [K]\ Data$$

where Data is the channel type and K is a set of indices.

Actually then the indexed channel c describes a set of channels $\{c[k]: k \in K\}$. Such an indexed channel may be part of a signature. The same way we introduce indexed system names

$$F: [K]\ [I \blacktriangleright O]$$

Then by F we get a set of systems

$$\{F[k]: k \in K\}$$

with interfaces $[I[k] \blacktriangleright O[k]]$ for F[k] where each channel in I or O carries the same index k we may see F[K] as a family of systems with a family of input channels.

By this we can describe large networks with a huge number of components and channels and with many instances of the some behavior.

Based on the concept of indexed components we can go a further step into "object orientation" in the sense that we allow for component creation and instantiation. To do so we introduce a creator component for a parameterized set F: [k] $[I \blacktriangleright O]$ of components (where we assume that K is an infinite set of component identifiers). The instantiation component has two channels, an input channel on which requests are received and an output channel on which identifiers are send which then also are attributed to the created instance identifiers. If it receives a message to create a new component it creates one and returns its identifier on its output channel.

A way to avoid the problem of making sure that individual identifiers are received consistently in return to creation messages is to assume universal output channels for every component for creating new instances and individual input channels for receiving the identifiers of created instances. This leads to networks where every subsystem has an identifier and a number of standard channels indexed by that identifier.

5 Related Work and Alternative Models

Of course, the mathematical models we have introduced are not the only way to construct models and theories for dynamic mobile systems. Much work has been carried out towards the investigation of such models and theories. Only some of that is to be mentioned briefly in the following and related to our model.

Pioneering work goes back to Robin Milner in his work on the π-calculus (see [27], [28]) introducing a process algebra for the dynamics of channel connections. π-calculus captures the dynamics of systems by operational semantics in terms of rules that manipulate "process terms" representing systems. In the π-calculus channels can be passed as messages (which can be done in our model, too) and then used as channels by the respective receiver. This idea is captured by the rules of the structured operational semantics of π-calculus that are quite intuitive. However, the rules of π-calculus do not provide a denotational model but only operational models for dynamic systems. A denotational understanding, however, is essential for the constructions of techniques for the specification of dynamic systems.

Besides theoretical approaches to models of mobility and dynamics these ideas are widely used in practice in object-oriented programming languages, however, their time, distribution, communication are not explicitly addressed. In our model we easily capture object orientation by considering each object as a component (see [6] and [18, 19]). There are a number of approaches to give more specific treatments of the dynamics of object orientation (see [1], [21], [30]). Besides this, there is a lot of practical work in the area of object-oriented concepts to program dynamic systems (see also design patterns).

Another topic aims at concepts to formalize mobility and the idea of scopes and residences. This is modeled by scopes of bindings and information access in the ambient calculus (see [11]). The ambient calculus covers a special aspect of dynamic systems that is most relevant for distributed data basis and information systems. Such ideas could be added to our model by channel hiding.

Let us shortly discuss ambient calculus. What do we model by the ambient calculus? The ambient calculus introduces the following conceptual categories and terms to denote them:

- process,
- capability,
- ambient: an ambient n [P] is a named process P with name n.

A process can be composed to form composed processing units describing systems called *ambients*. Ambients include the following notions

- *scope*: location or space in which an identifier is valid. This is achieved by restriction (like Δ x.P),
- *parallel composition*: composition of ambients,
- *replication*: simple form of iteration or recursion on processes,
- *location*: named location or scope in which a process (composed or not) executes.

What can a process do when executing capabilities? It may carry out activities such as:

- *entering* ambient scopes, leaving ambient scopes and opening scopes. Note: given an ambient

  n[P]

  if in P a sub-process executes the capability "**in** m" (by which P becomes P') then n[P'] moves into the scope of ambient m[Q] (which results in an ambient inside which processes Q and n[P'] are running in parallel),
- *exiting*: exiting is inverse to entering,
- *opening*: when in the process P | m:[Q] a sub-process executes in P the capability open m by which P becomes P' then the name m and its scope disappears and the original process becomes P' | Q.

Finally an ambient can communicate. This is done by a synchronous input and output action. This is expressed by the rewrite rule:

$$(x).P \mid \langle M \rangle \rightarrow P[M/x]$$

Note that both names and capabilities may occur as input and output. Note, moreover, that the naming of processes is essential in ambient calculus. The semantic of ambient calculus is given in an operational style.

The ambient calculus is one, in fact, very sophisticated model of dynamic systems. It captures certain aspects of dynamics quite explicitly and leaves others implicit. It is a more operational model of dynamics than our approach since its semantics is captured by rules of structured operational semantics. It considers additional aspects such as scopes, locality (and thus also the idea of mobility) and restricted access to local data that is not considered in our presented work.

6 Summary and Outlook

We have introduced a denotational model for dynamic systems. In contrast to π-calculus and ambient calculus, which are based on operational models, our emphasis is on a denotational modular model and the notion of interface that can be used as a basis for modeling, specification, and verification.

Future work for the presented approach is needed in the following directions

- Introduction of more pragmatic description and modeling concepts in terms of diagrams and tables
- Application of the concepts to representative application examples

- Generalization to models and concepts that are typical for dynamic systems such as interoperability and connectivity

Dynamic systems are typical for a large class of application systems we find today. Therefore their adequate modeling becomes more and more important. Denotational models provide here a major contribution.

Acknowledgements. I am grateful for stimulating discussions with Ingolf Krüger.

References

1. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: Towards a Theory of Actor Computation. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 565–579. Springer, Heidelberg (1992)
2. Broy, M.: Towards a Mathematical Concept of a Component and its Use. First Components' User Conference, Munich (1996); Revised version in: Software - Concepts and Tools 18, 137–148 (1997)
3. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement. Springer (2001)
4. Cardelli, L.: A Language with Distributed Scope. ACM Trans. Comput. Syst. 8(1), 27–59 (January); ALso appeared in POPL 1995
5. Grosu, R., Stølen, K.: A Model for Mobile Point-to-Point Data Flow Networks without Channel Sharing. In: Wirsing, M., Nivat, M. (eds.) AMAST 1996. LNCS, vol. 1101, pp. 505–519. Springer, Heidelberg (1996)
6. Grosu, R., Stølen, K.: A Denotational Model for Mobile Many-to-Many Data Flow Networks. Technical Report TUM-I9622, Technische Universität München (1996)
7. Haridi, S., van Roy, P., Smolka, G.: An Overview of the Design of Distributed Oz. In: The 2nd International Symposium on Parallel Symbolic Computation (PASCO 1997). ACM, New York (1997)
8. Milner, R.: The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh (1991)
9. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. Part i + ii. Information and Computation 100(1), 1–40, 41–77 (1992)
10. van Roy, P., Haridi, S., Brand, P., Smolka, G., Mehl, M., Scheidhauer, R.: Mobile Objects in Distributed Oz. ACM Toplas 19(5), 805–852 (1997)

From Hierarchical BIP to Petri Calculus*

Roberto Bruni[1], Hernán Melgratti[2], and Ugo Montanari[1]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Departamento de Computación, FCEyN,
Universidad de Buenos Aires - CONICET, Argentina

Abstract. We focus on Hierarchical BIP, an extension of Joseph Sifakis et al's BIP component framework, to provide a semantics-preserving, compositional encoding in the Petri calculus, a recently proposed algebra of stateless connectors and one-position buffers.

1 Introduction

In recent years Joseph Sifakis has successfully pursued a research strand focussed on a component framework called BIP [2], that has also been implemented in a language and a tool-set. BIP is a component framework for constructing systems by superposing three layers of modelling:

1) Behaviour, the lowest level, representing the sequential computation of individual components as automata whose arcs are labelled by sets of ports. The sets of ports of any two different components are disjoint, i.e., each port is uniquely assigned to a component.

2) Interaction, the second level, defining the allowed interactions between components. An interaction is just a set of ports typically of different components.

3) Priority, the top layer, assigning priorities to interactions to enforce scheduling policies, typically with the aim of reducing the size of the state space.

In the absence of priorities, the interaction layer of BIP admits the algebraic presentation given in [3] and comparisons with other models have been shown in [7,1,11], see [8] for an overview. In particular, an equivalent version of BIP systems is presented in [7] in terms of a compositional encoding in Petri nets with boundaries [11].

Here we investigate hierarchically structured BIP systems and show that previous correspondence results on ordinay BIP can be extended to deal with Hierarchical BIP (HBIP) as defined in [10]. HBIP systems are possibly formed by the combination of other HBIP systems, each seen as an ordinary component whose ports are its interactions. We exploit the Petri calculus, a calculus of stateful connectors introduced in [11], to encode in a compositional way HBIP systems while flattening them. Notably the encoding of components and of interactions can be given separately in the Petri calculus and then assembled by ordinary Petri calculus composition.

* Research supported by European FET-IST-257414 Integrated Project ASCENS, Progetto MIUR PRIN CINA Prot. 2010LHT4KM, ANPCyT Project BID-PICT-2008-00319.

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 54–68, 2014.

Structure of the paper. In § 2 we recall the main background on BIP and on the Petri calculus to keep the paper self-contained. In § 3 we define Hierarchical BIP and in § 4 we present the main result of the paper, namely the compositional encoding from HBI(P) to the Petri calculus. In § 5 we give some concluding remarks and discuss alternative approaches to HBIP. A toy running example of a client-server system is used to illustrate the main notions and constructions.

2 Background

2.1 The BIP Component Framework

This section reports on the formal definition of BIP as presented in [4]. Since we disregard priorities, we call BI(P) the framework presented here.

Definition 1 (Interaction). *Given a set of ports P, an* interaction *over P is a non-empty subset $a \subseteq P$.*

We write an interaction $\{p_1, p_2, \ldots, p_n\}$ as $p_1 p_2 \ldots p_n$ and $a \downarrow_{P_i}$ for the projection of an interaction $a \subseteq P$ over the set of ports $P_i \subseteq P$, i.e., $a \downarrow_{P_i} = a \cap P_i$. Projection extends to sets of interactions in the following way $\gamma \downarrow_P = \{a \downarrow_P \mid a \in \gamma \ \wedge \ a \downarrow_P \neq \varnothing\}$.

Definition 2 (Component). *A* component *$B = (Q, P, \rightarrow)$ is a transition system where Q is a set of states, P is a set of ports, and $\rightarrow \subseteq Q \times 2^P \times Q$ is the set of labelled transitions.*

As usual, we write $q \xrightarrow{a} q'$ to denote the transition $(q, a, q') \in \rightarrow$. An interaction a is enabled in q, denoted $q \xrightarrow{a}$, iff there exists q' s.t. $q \xrightarrow{a} q'$. By abusing the notation, we will also write $q \xrightarrow{\varnothing} q$ for any q.

Definition 3 (BI(P) system). *A* BI(P) system *$B = \gamma(B_1, \ldots, B_n)$ is the composition of a finite set $\{B_i\}_{i=1}^n$ of components $B_i = (Q_i, P_i, \rightarrow_i)$ such that their sets of ports are pairwise disjoint, i.e., $P_i \cap P_j = \varnothing$ for $i \neq j$ parameterized by a set $\gamma \subseteq 2^P$ of interactions over the set of ports $P = \biguplus_{i=1}^n P_i$.*

The semantics of a BIP system $\gamma(B_1, \ldots, B_n)$ is given by the transition system $(Q, P, \rightarrow_\gamma)$, with $Q = \Pi_i Q_i$, $P = \biguplus_{i=1}^n P_i$ and $\rightarrow_\gamma \subseteq Q \times 2^P \times Q$ is the least set of transitions satisfying the following inference rule

$$\frac{a \in \gamma \qquad \forall i \in 1..n : q_i \xrightarrow{a \downarrow_{P_i}} q_i'}{(q_1, \ldots, q_n) \xrightarrow{a}_\gamma (q_1', \ldots, q_n')}$$

Example 1. Consider the BI(P) system shown in Fig. 1, which contains a component `Server` that sequentially interacts with a component `Client`. The `Server` accepts a request from the `Client` thanks to the interaction $\{\texttt{acpt}, \texttt{req}\}$. Then, the server can successfully answer the request by returning a value to the client (interaction $\{\texttt{ret}, \texttt{resp}\}$) or can fail the request by rising an error signal, which is handled by the client (interaction $\{\texttt{err}, \texttt{hdle}\}$). ♦

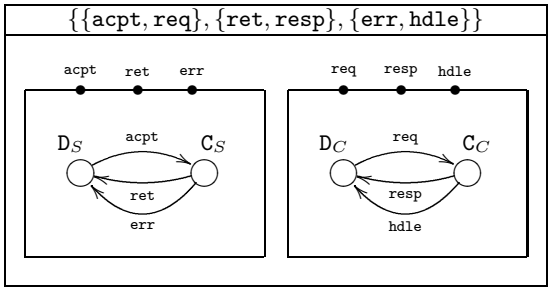


Fig. 1. A simple client/server BI(P) system

$$R ::= \bigcirc \mid \odot \mid \mathsf{I} \mid \mathsf{X} \mid \Delta \mid \nabla \mid \bot \mid \top \mid \wedge \mid \vee \mid \downarrow \mid \uparrow \mid R \otimes R \mid R; R$$

Fig. 2. Petri calculus grammar

2.2 Petri Calculus

The *Petri calculus* [11] enriches the algebra of stateless connectors from [5] with one-place buffers along [1,11,6].

Terms of the Petri Calculus are defined by the grammar in Fig. 2. It consists of the following constants plus parallel and sequential composition: the empty place $\bigcirc$, the full place $\odot$, the identity wire I, the twist (also swap, or symmetry) X, the duplicator (also sync) Δ and its dual ∇, the mutex (also choice) $\wedge$ and its dual $\vee$, the hiding (also bang) $\bot$ and its dual $\top$, the inaction $\downarrow$ and its dual $\uparrow$. The diagrammatical representation of terms is shown in Fig. 3. For $n \in \mathbb{N}$, we write $\underline{n}$ to denote the finite ordinal $\underline{n} \stackrel{\text{def}}{=} \{0, 1, \ldots, n-1\}$.

Any term has a unique associated *sort* (also called *type*) (k, l) with $k, l \in \mathbb{N}$, that fixes the size k of the left (input) interface and the size l of the right (output) interface of P. The type of constants are as follows: $\bigcirc$, $\odot$, and I have type $(1, 1)$, $\mathsf{X} : (2, 2)$, Δ and $\wedge$ have type $(1, 2)$ and their duals ∇ and $\vee$ have type $(2, 1)$, $\bot$ and $\downarrow$ have type $(1, 0)$ and their duals $\top$ and $\uparrow$ have type $(0, 1)$. The sort inference rules for composed processes are in Fig. 4.

The operational semantics is defined by the rules in Fig. 5, where $x, y \in \{0, 1\}$. The labels $\alpha, \beta, \rho, \sigma$ of transitions are binary strings, all transitions are sort-preserving, and if $R \xrightarrow[\beta]{\alpha} R'$ with $R, R' : (n, m)$, then $\#\alpha = n$ and $\#\beta = m$. Notably, bisimilarity induced by such a transition system is a congruence.

Due to space limitation we omit details here and refer the interested reader to [9].

Compound Terms. For the translation presented in § 4, we shall need additional families of compound terms, indexed by $n \in \mathbb{N}_+$ and $k \in \mathbb{N}$ (duals are omitted):

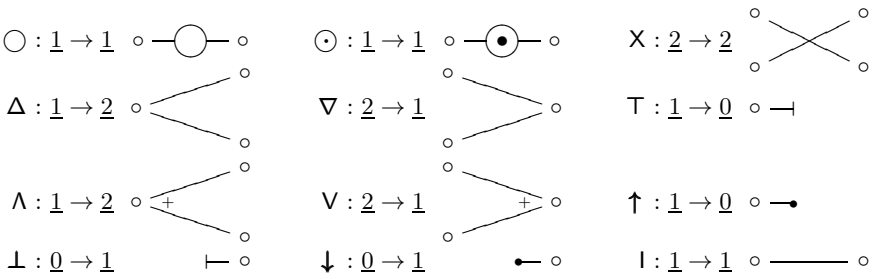


Fig. 3. Graphical representation of terms

$$\frac{R : (k,l) \quad R' : (m,n)}{R \otimes R' : (k+m, l+n)} \qquad \frac{R : (k,n) \quad R' : (n,l)}{R; R' : (k,l)}$$

Fig. 4. Sort inference rules

$$\mathsf{I}_n : (n,\, n) \qquad \mathsf{T}_n : (0,\, n) \qquad \uparrow_n : (0,\, n) \qquad \mathsf{X}_n : (n+1,\, n+1)$$
$$\Delta_n : (n,\, 2n) \quad \Lambda_n : (n,\, 2n) \quad \Delta_n^k : (n,\, k*n) \quad \Lambda_n^k : (n,\, k*n) \qquad d_n : (0,\, 2n)$$

Intuitively, I_n, T_n and $\uparrow_n$ correspond to n parallel copies of I, T and $\uparrow$, respectively. Connector Δ_n (and its dual ∇_n) is similar to Δ but duplicates n wires in the other interface, while Δ_n^k (and its dual ∇_n) replicates k-times the n wires of the other interface. Connector d_n (and its dual e_n) stands for the synchronisation of n pairs of wires. We now give the definitions:

$$\begin{array}{rlll}
\mathsf{I}_n = \bigotimes_n \mathsf{I} & \mathsf{X}_1 = \mathsf{X} & \mathsf{X}_{n+1} = (\mathsf{X}_n \otimes \mathsf{I}); (\mathsf{I}_n \otimes \mathsf{X}) \\
\uparrow_n = \bigotimes_n \uparrow & \Delta_1 = \Delta & \Delta_{n+1} = (\Delta \otimes \Delta_n); (\mathsf{I} \otimes \mathsf{X}_n \otimes \mathsf{I}_n) \\
\mathsf{T}_n = \bigotimes_n \mathsf{T} & \Lambda_1 = \Lambda & \Lambda_{n+1} = (\Lambda \otimes \Lambda_n); (\mathsf{I} \otimes \mathsf{X}_n \otimes \mathsf{I}_n) \\
d_n = \mathsf{T}_n; \Delta_n & \Delta_n^0 = \mathsf{T}_n & \Delta_n^{k+1} = \Delta_n; (\Delta_n^k \otimes \mathsf{I}_n) \\
 & \Lambda_n^0 = \uparrow_n & \Lambda_n^{k+1} = \Lambda_n; (\Lambda_n^k \otimes \mathsf{I}_n)
\end{array}$$

The behaviour of compound terms is characterised by the next proposition.

Proposition 1 (from [6]). *For $n > 0$,*

1. $\mathsf{X}_n \xrightarrow[\beta]{\alpha} t$ *iff* $t = \mathsf{X}_n$, $\alpha = h_0 \dots h_n$ *and* $\beta = h_1 \dots h_n h_0$.
2. $\Delta_n \xrightarrow[\beta]{\alpha} t$ *iff* $t = \Delta_n$, $\#\alpha = n$, $\#\beta = 2n$ *and* $\alpha_i = \beta_i = \beta_{n+i}$ *for all* $i < n$.
3. $\nabla_n \xrightarrow[\beta]{\alpha} t$ *iff* $t = \nabla_n$, $\#\alpha = 2n$, $\#\beta = n$ *and* $\alpha_i = \alpha_{n+i} = \beta_i$ *for all* $i < n$.
4. $\Lambda_n \xrightarrow[\beta]{\alpha} t$ *iff* $t = \Lambda_n$, $\#\alpha = n$, $\#\beta = 2n$, $\alpha_i = \beta_i + \beta_{n+i}$ *for all* $i < n$.
5. $\mathsf{V}_n \xrightarrow[\beta]{\alpha} t$ *iff* $t = \mathsf{V}_n$, $\#\alpha = 2n$, $\#\beta = n$, $\beta_i = \alpha_i + \alpha_{n+i}$ *for all* $i < n$.

$$\frac{}{\bigcirc \xrightarrow{\frac{1}{0}} \circledcirc}\ (\text{TkI}) \qquad \frac{}{\circledcirc \xrightarrow{\frac{0}{1}} \bigcirc}\ (\text{TkO1}) \qquad \frac{}{\mathsf{I} \xrightarrow{\frac{1}{1}} \mathsf{I}}\ (\text{Id}) \qquad \frac{}{\mathsf{X} \xrightarrow{\frac{ab}{ba}} \mathsf{X}}\ (\text{Tw})$$

$$\frac{}{\bot \xrightarrow{1} \bot}\ (\bot) \qquad \frac{}{\top \xrightarrow{}_{1} \top}\ (\top) \qquad \frac{}{\Delta \xrightarrow{\frac{1}{11}} \Delta}\ (\Delta) \qquad \frac{}{\nabla \xrightarrow{\frac{11}{1}} \nabla}\ (\nabla)$$

$$\frac{}{\Lambda \xrightarrow[(1-a)a]{1} \Lambda}\ (\Lambda a) \qquad \frac{}{\mathsf{V} \xrightarrow[1]{(1-a)a} \mathsf{V}}\ (\mathsf{V} a) \qquad \frac{\mathsf{C} : (k, l) \text{ a basic connector}}{\mathsf{C} \xrightarrow[0^l]{0^k} \mathsf{C}}\ (\text{Refl})$$

$$\frac{P \xrightarrow[\gamma]{\alpha} Q \quad R \xrightarrow[\beta]{\gamma} S}{P\,;R \xrightarrow[\beta]{\alpha} Q\,;S}\ (\text{Cut}) \qquad \frac{P \xrightarrow[\beta]{\alpha} Q \quad R \xrightarrow[\Delta]{\gamma} S}{P \otimes R \xrightarrow[\beta\Delta]{\alpha\gamma} Q \otimes S}\ (\text{Ten})$$

Fig. 5. Operational semantics for the Petri Calculus

6. $\Lambda_l^n \xrightarrow[\beta]{\alpha} t$ *iff* $t = \Lambda_l^n$, $\#\alpha = l$, $\#\beta = nl$ *and* $\alpha_i = \Sigma_{j<n}\beta_{jl+i}$ *for all* $i < l$.
7. $\mathsf{V}_l^n \xrightarrow[\beta]{\alpha} t$ *iff* $t = \mathsf{V}_l^n$, $\#\alpha = nl$, $\#\beta = l$ *and* $\beta_i = \Sigma_{j<n}\alpha_{jl+i}$ *for all* $i < l$.
8. $\Delta_l^n \xrightarrow[\beta]{\alpha} t$ *iff* $t = \Delta_l^n$, $\#\alpha = l$, $\#\beta = nl$ *and* $\beta_{lj+i} = \alpha_i$ *for all* $i < l$,$j < n$.
9. $\nabla_l^n \xrightarrow[\beta]{\alpha} t$ *iff* $t = \nabla_l^n$, $\#\alpha = nl$, $\#\beta = l$ *and* $\beta_i = \alpha_{lj+i}$ *for all* $i < l$ *and* $j < n$.
10. $d_n \xrightarrow[\beta]{\alpha} t$ *iff* $t = d_n$, $\#\alpha = 0$, $\#\beta = 2n$ *and* $\beta_i = \beta_{n+i}$ *for all* $i < n$ *and* $j < n$.

Relational Forms The encoding proposed in § 4 uses two classes of terms of the Petri calculus, called the left and right *relational forms*, that represent functions as Petri calculus terms [11].

For any $h \in \mathbb{N}$, there is a bijection $\ulcorner \_ \urcorner : 2^{\underline{h}} \to \{0,1\}^h$ with

$$\ulcorner U \urcorner_i \stackrel{\text{def}}{=} \begin{cases} 1 \text{ if } i \in U \\ 0 \text{ otherwise} \end{cases}$$

For Θ a set of Petri calculus terms, let T_Θ denote the set of terms generated by the following grammar:

$$T_\Theta \ ::= \ \theta \in \Theta \ \mid \ \mathsf{I} \ \mid \ T_\Theta \otimes T_\Theta \ \mid \ T_\Theta\,;T_\Theta.$$

We shall use t_Θ to range over terms of T_Θ.

Definition 4. *A term* $t : (k, l)$ *is in* right relational form *when it is in*

$$T_{\{\bot\}}\,;T_{\{\Delta\}}\,;T_{\{\mathsf{X}\}}\,;T_{\{\mathsf{V}\}}\,;T_{\{\uparrow\}}.$$

Dually, t *is said to be in* left relational form *when it is in*

$$T_{\{\downarrow\}}\,;T_{\{\Lambda\}}\,;T_{\{\mathsf{X}\}}\,;T_{\{\nabla\}}\,;T_{\{\top\}}.$$

The following result spells out the significance of the relational forms.

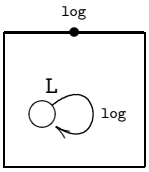


Fig. 6. Component `Logging`

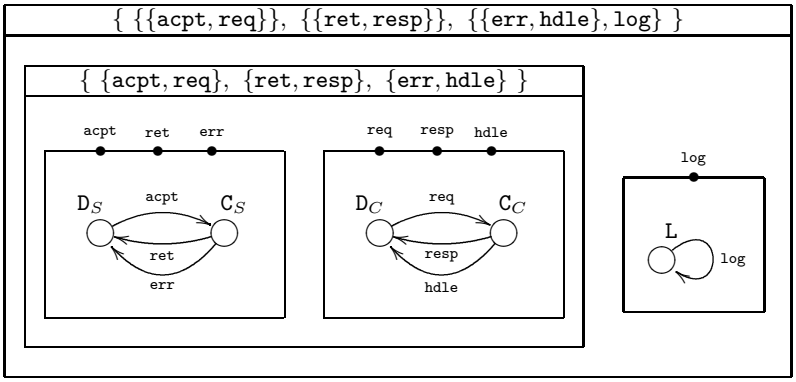


Fig. 7. A simple BI(P) system

Lemma 1 (From [11]). *For each function* $f\colon \underline{k} \to 2^{\underline{l}}$ *there exists a term* $\rho_f : (k, l)$ *in right relational form, the dynamics of which are characterised by the following:*

$$\rho_f \xrightarrow[\beta]{\alpha} \rho_f \;\Leftrightarrow\; \exists U \subseteq \underline{k} \text{ s.t. } \forall u, v \in U.\, u \neq v \Rightarrow f(u) \cap f(v) = \varnothing,\ \alpha = \ulcorner U \urcorner \text{ and } \beta = \ulcorner f(U) \urcorner$$

The symmetric result holds for functions $f\colon \underline{k} \to 2^{\underline{l}}$ *and terms* $t : (l, k)$ *in left relational form. That is, there exists* $\lambda_f : (l, k)$ *in left relational form with semantics*

$$\lambda_f \xrightarrow[\beta]{\alpha} \lambda_f \;\Leftrightarrow\; \exists U \subseteq \underline{k} \text{ s.t. } \forall u, v \in U.\, u \neq v \Rightarrow f(u) \cap f(v) = \varnothing,\ \beta = \ulcorner U \urcorner \text{ and } \alpha = \ulcorner f(U) \urcorner$$

3 Hierarchical BIP Systems

In this section we address the hierarchical composition of BI(P) systems, i.e., BI(P) systems can be taken as components of larger systems.

Example 2. Consider the scenario introduced in Example 1, which should be extended with a logging functionality in order to record all error responses sent by the component `Server`. Assume we already have the simple component for logging depicted in Fig. 6. In this case, we would like to consider the system in

Fig. 1 as a single component to define a new BI(P) system, as the one shown in Fig. 7. We remark that the interface (i.e., the set of ports) exposed by the client/server subsystem is just its set of interactions. This ensures that the composed system does not change the behaviour of the underlying subsystems. ♦

Next definition formally introduces the notion of hierarchical composition of systems

Definition 5 (HBI(P) system). *A Hierarchical BI(P) system (HBI(P)) is either*

- *a BI(P) component $B = (Q, P, \rightarrow)$ with interface $\iota(B) = P$; or*
- *a composite system $B = \gamma(B_1, \ldots, B_n)$ with interface $\iota(B) = \gamma$ where $\{B_1, \ldots, B_n\}$ is a set of hierarchical BI(P) systems with pairwise disjoint interfaces, i.e., $\iota(B_i) \cap \iota(B_j) = \varnothing$ for $i \neq j$, and γ is a set of interactions over $\uplus_{i=1}^{n} \iota(B_i)$.*

The semantics of HBI(P) systems is defined analogously to that of BI(P) systems as the synchronous execution of the transitions of its constituent components matching one defined interaction. We start by defining the state space $\mathbb{Q}_B$ of a HBI(P) system B as follows:

- $\mathbb{Q}_B = Q$ if $B = (Q, P, \rightarrow)$
- $\mathbb{Q}_B = \mathbb{Q}_{B_1} \times \ldots \times \mathbb{Q}_{B_n}$ if $B = \gamma(B_1, \ldots, B_n)$.

Then, the semantics of a composite HBI(P) system B is given by the transition system $(\mathbb{Q}_B, \iota(B), \rightarrow)$ where $\rightarrow \subseteq \mathbb{Q}_B \times \iota(B) \times \mathbb{Q}_B$ is the least set of transitions satisfying the following inference rules

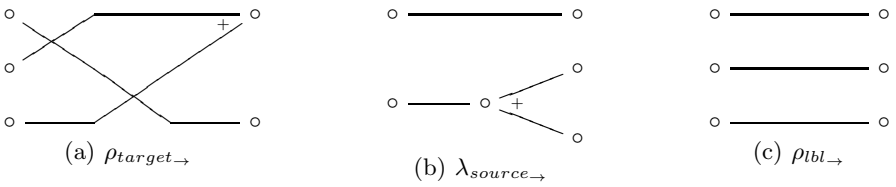
$$\frac{a \in \iota(B) \qquad \forall i \in 1..n : q_i \xrightarrow{a\downarrow_{\iota(B_i)}} q_i'}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

4 HBI(P) Systems as Petri Calculus Terms

This section gives an encoding of HBI(P) systems into the Petri calculus. Note that, differently from BI(P), the Petri calculus uses consecutive natural numbers to designate ports over interfaces. In order to establish a correspondence between HBI(P) systems and Petri calculus terms, we will map names into natural numbers, i.e., given a finite set S with $k = \#S$, we use w_S to denote an injective function $w_S : S \rightarrow \underline{k}$ that orders the elements of S. By abusing the notation, we write w_S to also denote its expected extension $w_S : 2^S \rightarrow 2^{\underline{k}}$.

4.1 Encoding of Basic Components

We first address the encoding of basic components (Definition 2) as Petri calculus terms. The encoding of components follows along the lines of the encoding of



(a) $\rho_{target_\rightarrow}$ (b) $\lambda_{source_\rightarrow}$ (c) $\rho_{lbl_\rightarrow}$

Fig. 8. Petri calculus terms for relational forms

Petri nets proposed in [9], although it is simpler due to the fact that components are just sequential systems.

Given a component $B = (Q, P, \rightarrow)$ with s transitions (i.e., $\# \rightarrow = s$), we rely on the functions w_Q, w_P and $w_\rightarrow$ that respectively order the elements in Q, P and $\rightarrow$. In addition, we will use the following three functions $source_\rightarrow$, $target_\rightarrow$, $lbl_\rightarrow$, which map a labelled transition belonging to $\rightarrow$ into its source, target and labels, when considering the sets of names just as ordinals.

$$\begin{array}{lll} source_\rightarrow : \underline{s} \rightarrow \underline{\#Q} & s.t. & source_\rightarrow(w_\rightarrow(q \xrightarrow{a} q')) = w_Q(q) \\ target_\rightarrow : \underline{s} \rightarrow \underline{\#Q} & s.t. & target_\rightarrow(w_\rightarrow(q \xrightarrow{a} q')) = w_Q(q') \\ lbl_\rightarrow : \underline{s} \rightarrow \underline{2^{\#P}} & s.t. & lbl_\rightarrow(w_\rightarrow(q \xrightarrow{a} q')) = w_P(a) \end{array}$$

Thanks to Lemma 1, we know that the relational forms $\lambda_{source_\rightarrow}$, $\rho_{target_\rightarrow}$ and $\rho_{lbl_\rightarrow}$ exist and their behaviours are in tight correspondence with the associated functions.

Example 3. Consider the component `Server` in Fig. 1 and assume the following ordering functions

$$\begin{array}{lll} w_Q(\mathtt{D}_S) = 0 & w_P(\mathtt{acpt}) = 0 & w_\rightarrow(\mathtt{D}_S \xrightarrow{\mathtt{acpt}} \mathtt{C}_S) = 0 \\ w_Q(\mathtt{C}_S) = 1 & w_P(\mathtt{ret}) = 1 & w_\rightarrow(\mathtt{C}_S \xrightarrow{\mathtt{ret}} \mathtt{D}_S) = 1 \\ & w_P(\mathtt{err}) = 2 & w_\rightarrow(\mathtt{C}_S \xrightarrow{\mathtt{err}} \mathtt{D}_S) = 2 \end{array}$$

Then, the relational forms associated to the above functions are defined as follows and depicted in Fig. 8.

$\rho_{target_\rightarrow} = (\mathsf{X} \otimes \mathsf{I})\ ;\ (\mathsf{I} \otimes \mathsf{X})\ ;\ (\mathsf{V} \otimes \mathsf{I}) \qquad \lambda_{source_\rightarrow} = (\mathsf{I} \otimes \Lambda) \qquad \rho_{lbl_\rightarrow} = \mathsf{I}_3$ ♦

Definition 6. *Let $B = (Q, P, \rightarrow)$ be a component. The Petri Calculus term corresponding to the behaviour of B in state $q \in Q$ is $[\![B]\!] : \underline{0} \rightarrow \underline{\#P}$, which is defined as follows*

$$T_{B_q} \stackrel{\text{def}}{=} d_s\ ;\ (\mathsf{I}_s \otimes (\rho_{target_\rightarrow}\ ;\ Q_{\{q\}}\ ;\ \lambda_{source_\rightarrow}))\ ;\ \nabla_s ; \rho_{lbl_\rightarrow}$$

where

$$Q_{Q'} \stackrel{\text{def}}{=} \bigotimes_{i<\#Q} q_i \quad \textit{where} \quad q_i \stackrel{\text{def}}{=} \begin{cases} \odot & \textit{if } i \in w_Q(Q') \\ \bigcirc & \textit{otherwise} \end{cases}$$

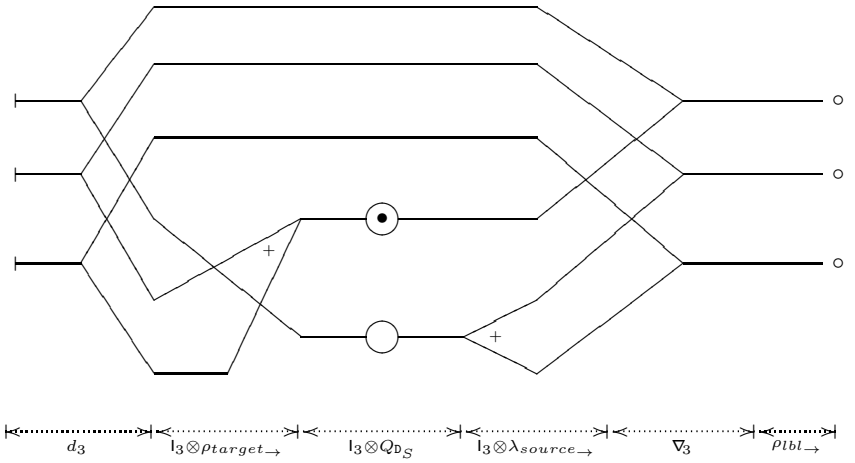


Fig. 9. Encoding of the BI(P) component `Server`

Example 4. Consider the component `Server` introduced in Example 1. Figure 9 shows the term $T_{B_{\mathtt{D}_S}}$ corresponding to the encoding of the component `Server` for the initial state $\mathtt{D}_S$ and the ordering functions given in Example 3. ♦

The following results formalise the relation between the behaviour of components and their encodings. The first lemma is auxiliary and characterises the behavior of terms of the form $Q_{Q'}$

Lemma 2. *$Q_{Q'} \xrightarrow[\ulcorner W \urcorner]{\ulcorner Z \urcorner} R$ iff $R = Q_{Q''}$, $W \subseteq Q'$, $Z \cap Q' = \varnothing$ and $Q'' = (Q' \backslash W) \cup Z$.*

Proof. Examination of either rules ($\bot_1$) and (T_1), together with the rule (Cut) (when $p = 0$) or rules (TkI) and (TkO), together with the rule (Ten) (when $p > 0$).□

Next result ensures that the transitions of a component are in one-to-one correspondence with the moves of the corresponding Petri calculus term.

Theorem 1. *Let $B = (Q, P, \rightarrow)$ a basic component. Then,*

(i) if $q \xrightarrow{a} q'$ then $T_{B_q} \xrightarrow[\ulcorner w_P(a) \urcorner]{} T_{B_{q'}}$.

(ii) if $T_{B_q} \underset{\beta}{\rightarrow} R$ then there exists q' s.t. $q \xrightarrow{a} q'$, $R = T_{B_{q'}}$ and $\beta = \ulcorner w_P(a) \urcorner$.

Proof. 1) By Lemma 2, $Q_q \xrightarrow[\ulcorner w_Q(q) \urcorner]{\ulcorner w_Q(q') \urcorner} Q_{q'}$. Then, by Lemma 1 and rule (Cut) after noting that $source_\rightarrow(w_\rightarrow(q \xrightarrow{a} q')) = w_Q(q)$ and $target_\rightarrow(w_\rightarrow(q \xrightarrow{a} q')) = w_Q(q')$, we have

$$(\rho_{target_\rightarrow} \,;\, Q_{\{q\}} \,;\, \lambda_{source_\rightarrow})) \xrightarrow[\ulcorner w_\rightarrow(q \xrightarrow{a} q') \urcorner]{\ulcorner w_\rightarrow(q \xrightarrow{a} q') \urcorner} (\rho_{target_\rightarrow} \,;\, Q_{\{q\}} \,;\, \lambda_{source_\rightarrow}))$$

and subsequently by Proposition 1 and rules (Cut) and (Ten)

$$d_s\ ;\ (\mathsf{I}_s \otimes (\rho_{target_\rightarrow}\ ;\ Q_{\{q\}}\ ;\ \lambda_{source_\rightarrow}))\ ;\ \nabla_s \xrightarrow{\ulcorner w_\rightarrow(q \xrightarrow{a} q')\urcorner}$$

$$d_s\ ;\ (\mathsf{I}_s \otimes (\rho_{target_\rightarrow}\ ;\ Q_{\{q\}}\ ;\ \lambda_{source_\rightarrow}))\ ;\ \nabla_s$$

The proof is completed by noting that $lbl_\rightarrow(w_\rightarrow(q \xrightarrow{a} q')) = w_P(a)$ and using Lemma 1 and rule (Cut) to conclude

$$T_{B_q} \xrightarrow[\ulcorner w_P(a)\urcorner]{} T_{B_{q'}}$$

2) If $T_{B_q} \underset{\beta}{\rightarrow} R$, then by rule (Cut) $\rho_{lbl_\rightarrow} \underset{\beta}{\overset{\alpha}{\rightarrow}} \rho_{lbl_\rightarrow}, \nabla_s; \rho_{lbl_\rightarrow} \underset{\beta}{\overset{\alpha\alpha}{\rightarrow}} \nabla_s; \rho_{lbl_\rightarrow}$ and

$$(\mathsf{I}_s \otimes (\rho_{target_\rightarrow}\ ;\ Q_{\{q\}}\ ;\ \lambda_{source_\rightarrow})) \underset{\alpha\alpha}{\overset{\alpha\alpha}{\rightarrow}} (\mathsf{I}_s \otimes R)$$

Hence,

$$(\rho_{target_\rightarrow}\ ;\ Q_{\{q\}}\ ;\ \lambda_{source_\rightarrow}) \underset{\alpha}{\overset{\alpha}{\rightarrow}} R$$

The only non trivial transition implies $Q_{\{q\}} \xrightarrow[\ulcorner q\urcorner]{\ulcorner Q'\urcorner} Q'_Q$. By Lemma 1 and $w_Q(q) = source_\rightarrow(w_\rightarrow(q \xrightarrow{a} q'))$, we conclude

$$\lambda_{source_\rightarrow} \xrightarrow[\alpha]{\ulcorner w_Q(q)\urcorner} \lambda_{source_\rightarrow}$$

with $\alpha = \ulcorner w_\rightarrow(q \xrightarrow{a} q')\urcorner$. By reasoning analogously, on $\rho_{target_\rightarrow}$, we have that

$$\rho_{target_\rightarrow} \xrightarrow[\ulcorner w_Q(q')\urcorner]{\alpha} \rho_{target_\rightarrow}$$

Hence, $R = (\rho_{target_\rightarrow}\ ;\ Q_{\{q'\}}\ ;\ \lambda_{source_\rightarrow})$. The proof is completed by using Lemma 1 to conclude that $\rho_{lbl_\rightarrow} \xrightarrow[\ulcorner w_P(a)\urcorner]{\alpha} \rho_{lbl_\rightarrow}$. □

Example 5. It can be easily checked that the term $T_{B_{\mathtt{D}_S}}$ introduced in Example 4 has the following transitions:

$T_{B_{\mathtt{D}_S}} \xrightarrow[000]{} T_{B_{\mathtt{D}_S}}$ $T_{B_{\mathtt{D}_S}} \xrightarrow[100]{} T_{B_{\mathtt{C}_S}}$ $T_{B_{\mathtt{C}_S}} \xrightarrow[010]{} T_{B_{\mathtt{D}_S}}$ $T_{B_{\mathtt{C}_S}} \xrightarrow[001]{} T_{B_{\mathtt{C}_S}}$

that correspond to the transitions

$\mathtt{D}_S \xrightarrow{\varnothing} \mathtt{D}_S$ $\mathtt{D}_S \xrightarrow{\mathtt{acpt}} \mathtt{C}_S$ $\mathtt{C}_S \xrightarrow{\mathtt{ret}} \mathtt{D}_S$ $\mathtt{C}_S \xrightarrow{\mathtt{err}} \mathtt{D}_S$ ♦

4.2 Encoding of Interactions

We now focus on the encoding of an interaction as a stateless connector. For $\alpha \in \{0,1\}^h$ a binary string of length $h > 0$, we let $R^\alpha : (h, 1)$ denote the process inductively defined by:

$$R^0 = \downarrow; \top \qquad R^1 = \mathsf{I} \qquad R^{x\alpha} = (R^x \otimes R^\alpha); \nabla$$

Intuitively, the term R^α synchronizes the ports associated to the positions of α that are set to 1.

Lemma 3. *The process R^α is stateless for any α, i.e., whenever $R^\alpha \xrightarrow[\beta']{\beta} R'$ then $R' = R^\alpha$.*

Proof. The thesis follows simply by noting that R^α is composed out of stateless connectors, i.e., the encoding does not exploit the constants $\bigcirc, \odot$. □

Lemma 4. *$R^\alpha \xrightarrow[\beta']{\beta} R'$ iff*

1. *$\beta = 0^{\#\alpha}$ and $\beta' = 0$; or*
2. *$\beta = \alpha$ and $\beta' = 1$.*

Proof. The thesis follows by induction on the length of α. □

Definition 7. *Let γ be a finite set of interactions over a finite set of ports I, $j = \#I$ and $k = \#\gamma$. After fixing w_I and w_γ, the encoding for the set γ is*

$$[\![\gamma]\!]_I = \Lambda^k_j ; (\!|\gamma|\!)_I ; \Delta_k ; (\mathsf{I}_k \otimes (\mathsf{V}^k_1 ; \bot))$$

where

$$\begin{aligned} (\!|\{a\}|\!)_I &= R^{\ulcorner w_I(a) \urcorner} \\ (\!|\{a\} \cup \gamma'|\!)_I &= R^{\ulcorner w_I(a) \urcorner} \otimes (\!|\gamma'|\!)_I \quad \textit{when } a \textit{ is the minimum in } \{a\} \cup \gamma' \textit{ w.r.t. } w_\gamma \end{aligned}$$

Example 6. Consider the set $\gamma = \{\{\texttt{acpt}, \texttt{req}\}, \{\texttt{ret}, \texttt{resp}\}, \{\texttt{err}, \texttt{hdle}\}\}$ defined over $I = \{\texttt{acpt}, \texttt{ret}, \texttt{err}, \texttt{req}, \texttt{resp}, \texttt{hdle}\}$ with the ordering functions defined as follows:

$$w_\gamma(\{\texttt{acpt}, \texttt{req}\}) = 0 \quad w_\gamma(\{\texttt{ret}, \texttt{resp}\}) = 1 \quad w_\gamma(\{\texttt{err}, \texttt{hdle}\}) = 2$$

$$w_I(\texttt{acpt}) = 0 \quad w_I(\texttt{ret}) = 1 \quad w_I(\texttt{err}) = 2$$

$$w_I(\texttt{req}) = 3 \quad w_I(\texttt{resp}) = 4 \quad w_I(\texttt{hdle}) = 5$$

Figure 6 shows the subterm $\Lambda^3_6 ; (\!|\gamma|\!)_I$ (we use a compact representation in which a chain of several identical connectors like Λ or ∇ are collapsed in a unique node). Note that $(\!|\gamma|\!)_I = R^{\ulcorner w_I(\{\texttt{acpt},\texttt{req}\}) \urcorner} \otimes R^{\ulcorner w_I(\{\texttt{ret},\texttt{resp}\}) \urcorner} \otimes R^{\ulcorner w_I(\{\texttt{err},\texttt{hdle}\}) \urcorner}$ stands for the parallel evaluation of the three interactions in γ. The term Λ^3_6 ensures that conflicting interactions (i.e., the ones sharing a common action) are performed in mutual exclusion. We remark that the term $\Lambda^3_6 ; (\!|\gamma|\!)_I$ still would allow for the concurrent execution of non-conflicting interactions (e.g., involving different sets of components). Since the semantics of BI(P) is purely sequential, we need to forbid the concurrent execution of disjoint interactions. This is ensured in the complete encoding by the subterm $\Delta_3 ; (\mathsf{I}_3 \otimes (\mathsf{V}^3_1 ; \bot))$. ♦

The following results characterise the behaviour of $[\![\gamma]\!]_I$ and are instrumental to the proof of our main result (Theorem 2).

Lemma 5. *Let γ be a synchronization set over I with $\#I = j$ and $\#\gamma = k$, and w_I and w_γ the functions sorting the elements of I and γ. Then*

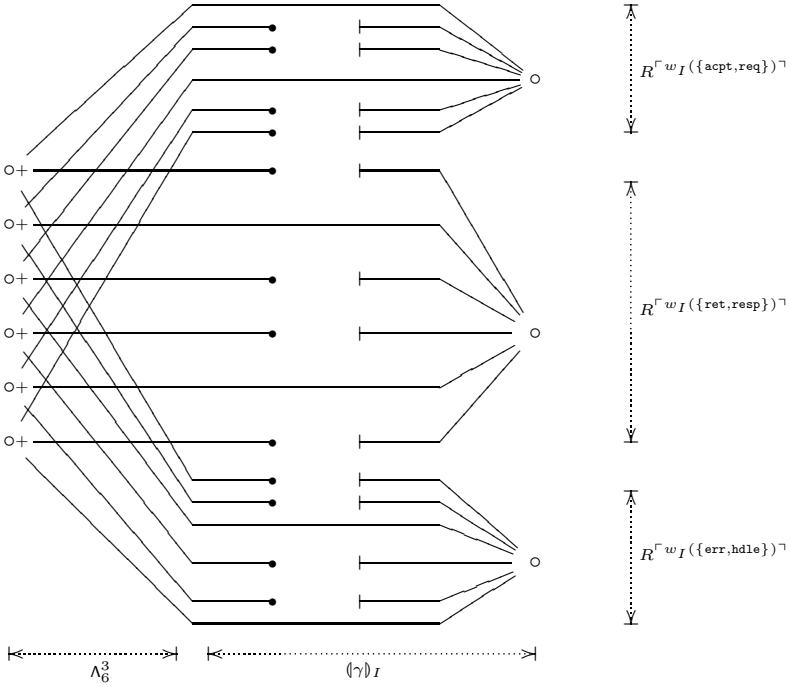


Fig. 10. Graphical representation of $\Lambda^3_6; (\!|\gamma|\!)_I$

1. $(\!|\gamma|\!)_I : (j * k, k)$.
2. $(\!|\gamma|\!)_I$ *is stateless for any* γ*, i.e., whenever* $(\!|\gamma|\!)_I \xrightarrow[\beta]{\alpha} R$ *then* $R = (\!|\gamma|\!)_I$.
3. $(\!|\gamma|\!)_I \xrightarrow[\beta]{\alpha} R$ *iff* $\forall 0 \leq h \leq k-1$ *either*
   - $\beta_h = 0$ *and* $\forall j * h \leq i \leq j * (h+1) - 1 : \alpha_i = 0$, *or*
   - $\beta_h = 1$, *and* $\exists a \in \gamma$ *s.t.* $w_\gamma(a) = h$ *and* $\alpha_{j*h..j*(h+1)-1} = \ulcorner w_I(a) \urcorner$.

Proof. (1) Follows by induction on $\#\gamma$. (2) Follows by noting that $(\!|\gamma|\!)_I$ is composed out of stateless connectors, i.e., the encoding does not exploit the constants $\bigcirc, \odot$. (3) By induction on h and using Lemma 4. □

Lemma 6. *Let* γ *be a synchronization set over* I *with* $\#I = j$ *and* $\#\gamma = k$*, and* w_I *and* w_γ *the functions sorting the elements of* I *and* γ*. Then*

1. $[\![\gamma]\!]_I : (j, k)$.
2. $[\![\gamma]\!]_I$ *is stateless for any* γ*, i.e., whenever* $[\![\gamma]\!]_I \xrightarrow[\beta]{\alpha} R$ *then* $R = [\![\gamma]\!]_I$.
3. $[\![\gamma]\!]_I \xrightarrow[\beta]{\alpha} R$ *iff* $\exists a \subseteq \gamma$ *s.t.* $\#a \leq 1$, $\alpha = \ulcorner w_I(a) \urcorner$ *and* $\beta = \ulcorner w_\gamma(a) \urcorner$.

Proof. (1) Follows from the fact that: $\Lambda^k_{\#I} : (j, j * k)$; $(\!|\gamma|\!)_I : (j * k, k)$ by Lemma 5(1); $\Delta_k : (k, 2k)$; and $(\mathsf{I}_k \otimes (\mathsf{V}^k_1; \bot)) : (2k, k)$. (2) Follows by noting that

$(\!|\gamma|\!)_I$ is composed out of stateless connectors, i.e., the encoding does not exploit the constants $\bigcirc, \odot$. (3) By Proposition 1(5),

$$\mathsf{V}_1^k \xrightarrow[\beta]{\alpha'} \mathsf{V}_1^k$$

with $\#\alpha' = k$, $\#\beta = 1$ and $\beta = \sum_{j<k} \alpha'_j$. Therefore, $\sum_{j<k} \alpha'_j \leq 1$. By the semantics of $\perp$ and ;, we have

$$\mathsf{V}_1^k; \perp \xrightarrow{\alpha'} \mathsf{V}_1^k; \perp$$

By using Proposition 1(2) and the inference rules for ; and $\otimes$

$$\Delta_k; (\mathsf{I}_k \otimes (\mathsf{V}_1^k; \perp)) \xrightarrow[\alpha']{\alpha'} \Delta_k; (\mathsf{I}_k \otimes (\mathsf{V}_1^k; \perp))$$

The proof is completed by using Lemma 6(3), Proposition 1(6) and the semantics of ;. □

4.3 Encoding of HBI(P) Systems

The encoding of HBI(P) systems is defined by a suitable combination of the encoding of basic components and interactions.

Definition 8. *Let B be a HBI(P) system with initial $q \in \mathbb{Q}_B$. The corresponding Petri Calculus term is inductively defined as follows.*

$$[\![B]\!]_q = \begin{cases} T_{B_q} & if B = (Q, P, \rightarrow) \\ ([\![B_1]\!]_{q_1} \otimes \ldots \otimes [\![B_n]\!]_{q_n}); [\![\gamma]\!]_I & if B = \gamma(B_1, \ldots, B_n), q = (q_1, \ldots, q_n), \\ \quad I = \cup_{i=1}^n \iota(B_i), \text{ and } w_I \text{ s.t. } \forall a : w_I(a) = w_{\iota(B_i)}(a) + \sum_{j=1}^{i-1} \#\iota(B_j) & \end{cases}$$

Theorem 2 (Correspondence). *Let B be a HBI(P) system with initial state q. Then $q \xrightarrow{a} q'$ if and only if $[\![B]\!]_q \xrightarrow[\ulcorner w_{\iota(B)(a)} \urcorner]{} [\![B]\!]_{q'}$.*

Proof. By induction on the structure of the system B. Base case ($B = (Q, P, \rightarrow)$) follows by Theorem 1. Inductive step follows by applying inductive hypothesis on $[\![B_1]\!]_{q_1}, \ldots, [\![B_n]\!]_{q_n}$. Then, the proof is completed by using Lemma 6. □

5 Conclusion

This paper studies the hierarchical composition of BI(P) systems and its relation with the Petri calculus. For convenience of presentation we have chosen the particular variant of BI(P) consisting of the basic interaction model and purely sequential execution. Nevertheless, the results presented in this paper can be extended or adapted to handle several variants proposed in the literature. The remaining of this section is devoted to the discussion of some alternative presentations for HBI(P) and their relation with the encodings proposed in this paper.

Concurrent executions. The work in [10] proposes a notion of hierarchical composition of BI(P) systems that allows for the concurrent execution of interactions, i.e., a set of conflict-free interactions (i.e., interactions that are pairwise disjoint) can be fired concurrently if enabled. We could encode such behaviour simply by defining $[\![\gamma]\!]_I$ as follows.

$$[\![\gamma]\!]_I = \Lambda^k_{\#I}; (\!|\gamma|\!)_I$$

This definition simplifies Definition 7 by removing the subterm $\Delta_k; (\mathsf{I}_k \otimes (\mathsf{V}^k_1; \bot))$. As already mentioned in § 4.2, the subterm $(\mathsf{V}^k_1; \bot)$ ensures the execution of a unique interaction at a time. The results presented in the previous sections could also be formulated for this variant with minor adjustments.

For the sake of uniformity, the proposal in [10] also considers concurrent basic components instead of just sequential components as originally proposed in BIP. Concurrent basic components could be modelled as C/E or P/T nets with boundaries, which can be encoded as Petri calculus terms as shown in [9].

Triggers. In order to represent different modes of synchronisation, the BIP model has been extended with a sorting discipline for ports in [3]. Typing associates synchronization types (`trigger` or `synchron`) to ports or connectors. The main difference is that an interaction $\{p_1, \ldots, p_n\}$ actually represents a set of interactions, i.e., all nonempty subset of $\{p_1, \ldots, p_n\}$ that contains some `trigger`; otherwise (if all of the ports are `synchrons`), the only possible interaction is the maximal one. Then, an interaction set γ contains either standard interactions (i.e., without triggers), denoted as before by a, and connectors (i.e., interactions containing at least a trigger), denoted by c. Let c be a connector, we write γ_c for the set of all standard denoted interactions (i.e., all subsets of c that contains a trigger). By assuming a set of interactions, we extend Definition 7 with the rules for encoding connectors

$$(\!|\{\mathsf{c}\}|\!)_I = \Lambda^k_j \,;\, (\!|\gamma_\mathsf{c}|\!)_I \,;\, \mathsf{V}^k_1$$

with $j = \#I$ and $k = \#\gamma_\mathsf{c}$ (rule for $(\!|\{\mathsf{c}\} \cup \gamma'|\!)_I$ is analogous).

By using Lemma 5 and the semantics of Λ^k_j and V^k_j, it is easy to conclude that the only non trivial transitions of $(\!|\{\mathsf{c}\}|\!)_I$ are $(\!|\{\mathsf{c}\}|\!)_I \xrightarrow[1]{\alpha} (\!|\{\mathsf{c}\}|\!)_I$ with $\alpha = \ulcorner w_I(a) \urcorner$ and $a \in \gamma_\mathsf{c}$. This characterization is analogous to the one for standard transitions in Lemma 4. This suffices to show that the correspondence results smoothly extend to the semantics of triggers.

Hiding. Hiding is an usual operator when composing systems hierarchically, because it enables components to compute internally. We can incorporate hiding to the definition of a HBI(P) system by adding the following item to Def. 5:

- a composed system $B = \nu a B_1$ with interface $\iota(B) = \iota(B_1) \setminus \{a\}$ where B_1 is a hierarchical BI(P) system and $a \in \iota(B_1)$.

The semantics of $B = \nu a B_1$ is given by extending the definition of the space state of a HBI(P) system with the equation

– $\mathbb{Q}_B = \{\nu aq | q \in \mathbb{Q}_{B_1}\}$ if $B = \nu a B_1$

and the following two inference rules

$$\frac{q \xrightarrow{a} q'}{\nu aq \xrightarrow{\varnothing} \nu aq'} \qquad \frac{a \neq b \quad q \xrightarrow{b} q'}{\nu aq \xrightarrow{b} \nu aq'}$$

The encoding of HBI(P) system with hiding can be simply handled as follows

$$[\![\nu a B]\!]_{\nu aq} = [\![B]\!]_q \, ; H^{\ulcorner w_{\iota(B)}(\{a\})\urcorner}$$

where H^α for $\alpha \in \{0,1\}^h$ is the the process inductively defined by:

$$H^0 = \mathsf{I} \qquad H^1 = \bot \qquad H^{x\alpha} = (H^x \otimes H^\alpha)$$

We remark that the term H^α replicates on the right interface only the ports of the left interface that are in the positions of α and that are set to 0, while the others are kept hidden. Consequently, $H^{\ulcorner w_{\iota(B)}(\{a\})\urcorner}$ hides the port associated to a. The extension of the correspondence results to HBI(P) systems with hiding is straightforward.

References

1. Arbab, F., Bruni, R., Clarke, D., Lanese, I., Montanari, U.: Tiles for Reo. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 37–55. Springer, Heidelberg (2009)
2. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), pp. 3–12. IEEE Computer Society (2006)
3. Bliudze, S., Sifakis, J.: The algebra of connectors - structuring interaction in BIP. IEEE Trans. Computers 57(10), 1315–1330 (2008)
4. Bliudze, S., Sifakis, J.: Causal semantics for the algebra of connectors. Formal Methods in System Design 36(2), 167–194 (2010)
5. Bruni, R., Lanese, I., Montanari, U.: A basic algebra of stateless connectors. Theor. Comput. Sci. 366(1-2), 98–120 (2006)
6. Bruni, R., Melgratti, H., Montanari, U.: A connector algebra for P/T nets interactions. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 312–326. Springer, Heidelberg (2011)
7. Bruni, R., Melgratti, H., Montanari, U.: Connector algebras, Petri nets, and BIP. In: Clarke, E., Virbitskaite, I., Voronkov, A. (eds.) PSI 2011. LNCS, vol. 7162, pp. 19–38. Springer, Heidelberg (2012)
8. Bruni, R., Melgratti, H., Montanari, U.: A survey on basic connectors and buffers. In: Beckert, B., Damiani, F., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 49–68. Springer, Heidelberg (2012)
9. Bruni, R., Melgratti, H.C., Montanari, U., Sobocinski, P.: Connector algebras for C/E and P/T nets' interactions. Logical Methods in Computer Science 9(3) (2013)
10. Graf, S., Quinton, S.: Contracts for BIP: Hierarchical interaction models for compositional verification. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 1–18. Springer, Heidelberg (2007)
11. Sobociński, P.: Representations of Petri net interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 554–568. Springer, Heidelberg (2010)

Programming and Verifying Component Ensembles ⋆

Rocco De Nicola[1], Alberto Lluch Lafuente[1], Michele Loreti[2], Andrea Morichetta[1], Rosario Pugliese[1], Valerio Senni[1], and Francesco Tiezzi[1]

[1] IMT Institute for Advanced Studies Lucca, Italy
[2] Università degli Studi di Firenze, Italy

Abstract. A simplified version of the kernel language SCEL, that we call SCELight, is introduced as a formalism for programming and verifying properties of so-called cyber-physical systems consisting of software-intensive ensembles of components, featuring complex intercommunications and interactions with humans and other systems. In order to validate the amenability of the language for verification purposes, we provide a translation of SCELight specifications into Promela. We test the feasibility of the approach by formally specifying an application scenario, consisting of a collection of components offering a variety of services meeting different quality levels, and by using SPIN to verify that some desired behaviors are guaranteed.

Keywords: Cyber Physical Systems, Component-based Systems, Formal Methods, Process Calculi, Verification, Model Checking.

1 Introduction

Nowadays much attention is devoted to software-intensive cyber-physical systems. These are systems possibly made of massive numbers of components, featuring complex intercommunications and interactions with humans and other systems and operating in open and unpredictable environments thus needing to dynamically adapt to new requirements, technologies and contextual conditions. Such classes of systems include the so-called *ensembles* [1] and *systems of systems* [2], mainly characterized by the idea of assembling or aggregating groups of autonomous components, which may be independently controlled and managed, and whose interaction may be cooperative or competitive.

The design and the analysis that these classes of systems meet the expectations of their users pose big challenges to language designers and software engineers. The problem for language designers is to provide the right set of programming abstractions together with the formal machinery that permits guaranteeing that the expected behavior is exhibited. To deal with the above mentioned challenges, in [3] we have introduced the kernel language SCEL that permits governing the complexity of such systems by providing flexible abstractions, by

⋆ Research supported by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL, and the Italian PRIN 2010LHT4KM CINA.

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 69–83, 2014.

enabling transparent monitoring of the involved entities and by supporting the implementation of self-* mechanisms such as self-adaptation. The key concepts of the language are those of *Behaviors*, *Knowledge*, *Aggregations* and *Policies* that have proved fruitful in modelling autonomic systems from different application domains such as, e.g., collective robotic systems [3,4], service provision and cloud-computing [5,6,7], and cooperative e-vehicles [8].

One of the distinguishing features of SCEL is the use of flexible, *group-oriented*, communication primitives that allows one to implicitly select the set of components to communicate with, by evaluating a given predicate $\mathcal{P}$ used as the target. When a communication action has predicate $\mathcal{P}$ as a target, it will involve all components that satisfy $\mathcal{P}$. For example, if a system contains elements that export attributes such as *serviceProvided* and *QoS* and one would like to program a component willing to interact with all the components that provide a service s and offer a QoS above q, (s)he can use the predicate $serviceProvided = s \wedge QoS > q$ to select the component's partners.

Contribution. This paper presents a first step towards using SCEL and the SPIN model checker [9] for guaranteeing systems properties. For ease of presentation we introduce a simple variant of SCEL that we call SCELight. We provide a translation of SCELight specifications into Promela, that is the input language of SPIN, and show how to exploit it to verify ensemble-based scenarios with SPIN. We test feasibility of the approach by considering an application scenario, borrowed from [5], consisting of a collection of components offering a variety of services meeting different quality levels.

Structure of the paper. The rest of the paper is organized as follows. In the next section, we introduce our application scenario that will be used also to describe the language constructs. In Section 3 we introduce syntax and informal semantics of SCELight, while in Section 4 we describe our translation and its intricacies demanded by the significantly different nature of SCELight and Promela. In Section 5 we show how SPIN can be used to check and verify properties of SCELight specifications, by relying on the translation into Promela of the SCELight specification of the scenario presented in Section 3. Finally, Section 6 concludes by also touching upon directions for future work.

2 A Service Provision Scenario

We consider an application scenario, borrowed from [5], consisting of a collection of components offering a variety of services. Each component manages and elaborates service requests with different requirements, roughly summarized by the following three service quality levels: *gold*, *silver* and *base*. These requirements are defined via a combination of predicates on the hardware configuration and the runtime state of the provider components. For example, the runtime state can give a measure of the number of service requests currently handled locally. Notice that the hardware measure is static while the load estimate is dynamically updated whenever a component receives or completes a service request.

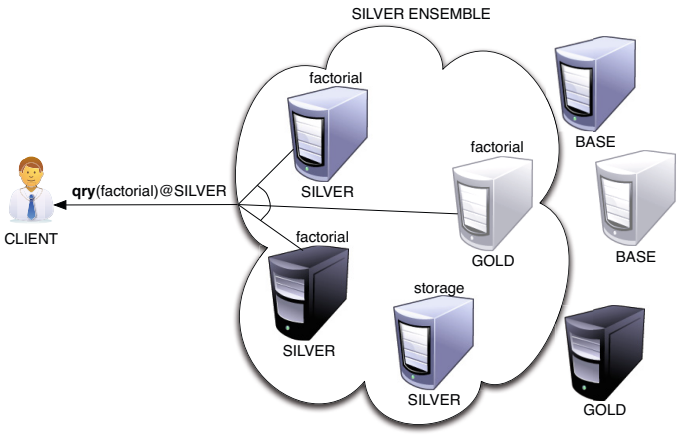


Fig. 1. Group-oriented communication in the service provision scenario

The quality of service, hence, implicitly defines three ensembles, which group together service provider components according to the quality requirements they are able to provide. Clearly, since the quality of service depends on the component state, ensembles are dynamic and components do not need to explicitly migrate from one ensemble to the other: their change of state will implicitly entail their membership to ensembles. The requirements characterizing the three ensembles of service providers are:

- *Gold*: components must have a high level of hardware configuration, i.e. a hardware level greater or equal to 7;
- *Silver*: components must provide a hardware configuration with a level that is at least 4 and, whenever a component provides a hardware level over 7, the computational load must be less than 40%; this latter condition guarantees that gold components can handle requests at silver level only when their computational load is under 40%;
- *Base*: components can have any hardware level, however if they are also gold or silver components then their computational load must be under 20% or 40%, respectively.

We remark that components dynamically and transparently leave or enter an ensemble when their computational load changes. For instance, a *gold* component leaves a *silver* ensemble when its computational load becomes higher than 40%.

Let us now consider a client component willing to submit a request for service *factorial*, which remotely computes the factorial of a natural number. Let us further assume that the client is interested in having the service from a *silver*-quality provider, to ensure the result to be provided within a reasonable amount of time (i.e., on a quite fast, light-loaded server). Before submitting its request, this component interacts with the ensemble of *silver* components searching a provider of the *factorial* service. This search is done by taking advantage of the group-oriented communication (Figure 1), which allows the client to dynamically

identify a component that exposes the service *factorial* at the wanted *silver* service level. If more than one provider component meets these requirements, one of them will be non-deterministically selected. Then, the client posts the actual request to the selected component and waits for the result.

Notice that the application scenario discussed above exploits different forms of communication. First, the invoking client uses group-oriented communication to identify the component that is able to handle specific service request. Then, point-to-point communication is used for client-server interaction.

3 The SCELight Language

SCEL (Software Component Ensemble Language) [3] is a language for programming service computing systems in terms of service components aggregated according to their knowledge and behavioural policies. To enhance flexibility with respect to different application domains, SCEL is parametric with respect to the language for expressing policies, the predicate regulating component interactions, and the notion of knowledge.

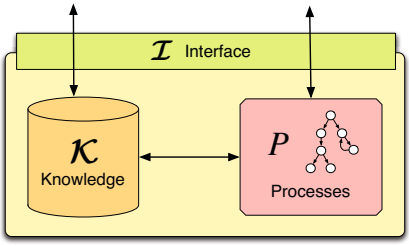


Fig. 2. A SCELight component

For ease of presentation we consider in this work an instantiation of SCEL named SCELight, where no policy language is provided, the interaction predicate interprets the composition of component's processes as a standard interleaving, and knowledge repositories are implemented as multiple distributed tuple-spaces à la Klaim [10]. Moreover, SCELight does not include other sophisticated features of SCEL such as higher-order communication and dynamic creation of new names and components. Last, SCELight includes a specific primitive for atomically updating attribute values, and replaces the non-deterministic choice of SCEL by an ordinary conditional choice. These two standard control flow constructs, that are part of the syntax of Promela, simplify the specification task and can be easily realized in SCEL.

The basic ingredient of SCELight is the notion of (*service*) *component* $\mathcal{I}[\mathcal{K}, P]$, graphically depicted in Figure 2, that consists of:

1. An *interface* $\mathcal{I}$ publishing and making available structural and behavioural information about the component itself in the form of *attributes*, i.e. names acting as references to information stored the component's repository.
2. A *knowledge repository* $\mathcal{K}$ managing both application and awareness data, together with specific handling mechanisms. It stores also the information associated to the interface.
3. A *process* P that can execute local computations, coordinate interaction with the knowledge repository or perform adaptation and reconfiguration.

Table 1. SCELight syntax

DEFINITIONS: $D ::= \emptyset \mid A(\bar{f}) \triangleq P \mid D_1, D_2$		SYSTEMS: $S ::= \mathcal{I}[\mathcal{K}, P] \mid S_1 \parallel S_2$
KNOWLEDGE: $\mathcal{K} ::= \emptyset \mid \langle t \rangle \mid \mathcal{K}_1 \parallel \mathcal{K}_2$	ITEMS: $t ::= e \mid t_1, t_2$	TEMPLATES: $T ::= v \mid x \mid ?\,x \mid T_1, T_2$
PROCESSES: $P ::= \mathbf{nil} \mid a.P \mid \mathbf{if}\ (e)\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \mid P_1 \mid P_2 \mid A(\bar{u})$		TARGETS: $c ::= n \mid \mathcal{P}$
ACTIONS: $a ::= \mathbf{get}(T)@c \mid \mathbf{qry}(T)@c \mid \mathbf{put}(t)@c \mid attr := e$		NAMES: $n ::= i \mid x$

A SCELight SPECIFICATION is a pair $\langle D, S \rangle$ grouping together a set of process DEFINITIONS D and a SYSTEM S. The syntax of definitions and systems is presented in Table 1. A (recursive) process definition has the form $A(\bar{f}) \triangleq P$, with A, $\bar{f}$ and P denoting a process identifier, a list of formal parameters, and a process, respectively. We will use $\bar{u}$ to denote a list of actual parameters. Definitions can be dynamically activated by processes running in system components. We assume that each process identifier has a single definition. Systems aggregate COMPONENTS through the *composition* operator $\_ \parallel \_$.

Knowledge. A KNOWLEDGE repository $\mathcal{K}$ is a tuple-space, i.e. a (possibly empty) multiset of stored tuples $\langle t \rangle$, composed by the operator $\_ \parallel \_$. Tuples are knowledge ITEMS consisting of sequences of values. Such values can result from the evaluation of some given *expression* e. We assume that expressions may contain attribute names $attr$, values v (i.e., component identifiers i, strings and integers), and variables x, together with the corresponding standard operators. To pick a tuple out from a tuple-space by means of a given TEMPLATE T (i.e., a sequence of values and variables), the *pattern-matching* mechanism is used: a tuple matches a template if they have the same number of elements and corresponding elements have matching values or variables; variables match any value of the same type ($?\,x$ is used to bind variables to values) and two values match only if they are identical. If more than one tuple match a given template, one of them is arbitrarily chosen.

Processes and Actions. PROCESSES are the active computational units. Each process is built up from the *inert* process **nil** via *action prefixing* ($a.P$), *conditional choice* ($\mathbf{if}\ (e)\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2$), *parallel composition* ($P_1 \mid P_2$), and *parametrized process invocation* ($A(\bar{u})$). Processes can perform four different kinds of ACTIONS. Actions $\mathbf{get}(T)@c$, $\mathbf{qry}(T)@c$ and $\mathbf{put}(t)@c$ are used to manage shared knowledge repositories by withdrawing/retrieving/adding information items from/to the knowledge repository identified by c. These actions exploit templates T to select knowledge items t from the repositories. They are implemented by invoking

the handling operations provided by the knowledge repository. Action $attr := e$ atomically assigns the value of e to $attr$ and, differently from the other actions, it is not indexed with an address because it always acts locally. Actions **get** and **qry** are blocking and, thus, may cause the process executing them to wait for the wanted element if it is not (yet) available in the knowledge repository. The two actions differ for the fact that **get** removes the retrieved item from the target repository while **qry** leaves the repository unchanged. Actions **put** and := are instead immediately executed.

Different entities may be used as the target c of an action, namely a component name n (in case of *point-to-point* communication) or a *predicate* $\mathcal{P}$ (in case of *group-oriented* communication). In fact, in an action using a predicate $\mathcal{P}$ to indicate the target, the predicate acts as a 'guard' specifying *all* components that may be affected by the execution of the action, i.e. a component must satisfy $\mathcal{P}$ to be the target of the action. Thus, the set of components satisfying a given predicate used as the target of a communication action can be considered as the *ensemble* with which the process performing the action intends to interact. A predicate is a boolean-valued expression obtained by applying standard operators to the results returned by the evaluation of relations between components' attributes and expressions. Notably, an attribute name occurring in a predicate refers to an attribute within the interface of the *object* components (i.e., components that are target of the communication action).

The service provision scenario in SCELight. The application scenario introduced in Section 2 can be formalized in SCELight as the following specification

$$\langle D, \mathcal{I}_{c_1}[\mathcal{K}_{c_1}, P_{c_1}] \parallel \ldots \parallel \mathcal{I}_{c_n}[\mathcal{K}_{c_n}, P_{c_n}] \parallel \mathcal{I}_{p_1}[\mathcal{K}_{p_1}, A_{p_1}] \parallel \ldots \parallel \mathcal{I}_{p_m}[\mathcal{K}_{p_m}, A_{p_m}] \rangle$$

consisting of a composition of n clients $\mathcal{I}_{c_h}[\mathcal{K}_{c_h}, P_{c_h}]$ and m providers $\mathcal{I}_{p_j}[\mathcal{K}_{p_j}, A_{p_j}]$. The latter ones are dynamically organised in ensembles according to requirements expressed in terms of suitable attributes exposed in the components' interfaces. In particular, we assume that attributes named *hw* and *load* are provided by each component. The former can take an integer value from 0 to 10 that gives an indication of the capacity of the hardware configuration of the component, while the latter can take an integer value from 0 to 100 that estimates the actual computational load of the component. The values of such attributes can be dynamically changed through actions $hw := e_1$ and $load := e_2$. Each service component also stores in its knowledge repository a collection of items indicating the available services, together with their component identifier. For example, the provider p_j offering the $factorial$ service stores in its local repository the item $\langle \text{“}service\text{”}, \text{“}factorial\text{”}, i_{p_j} \rangle$. Note that including the identifier in the tuple publishing the service is fundamental as the group-oriented communication primitives are completely anonymous, i.e. the actual objects of a group-oriented communication action are not known to the subject.

The three ensembles of *gold*, *silver* and *base* service providers are characterized by the following predicates:

$$\begin{aligned}
\mathcal{P}_g &\triangleq (\quad hw \geq 7) \\
\mathcal{P}_s &\triangleq (4 \leq hw < 7) \vee (\mathcal{P}_g \wedge load < 40) \\
\mathcal{P}_b &\triangleq (\quad hw < 4) \vee (\mathcal{P}_s \wedge load < 40) \vee (\mathcal{P}_g \wedge load < 20)
\end{aligned}$$

Each client component c_h runs the process P_{c_h}, that takes care of the interaction with the *factorial* service and is of the form

$$\begin{aligned}
&\mathbf{qry}(\text{“}service\text{”}, \text{“}factorial\text{”}, ?x)@\mathcal{P}_k. \\
&\mathbf{put}(\text{“}invoke\text{”}, \text{“}factorial\text{”}, v, i_{c_h})@x. \\
&\mathbf{get}(\text{“}result\text{”}, \text{“}factorial\text{”}, ?y)@i_{c_h}. P'_{c_h}
\end{aligned}$$

for some service level k in $\{b, s, g\}$ and some argument v for the factorial function the client would like the server to execute.

In words, such process first searches, via a **qry** action, among the components belonging to the ensemble identified by predicate $\mathcal{P}_k$, an item matching the template (“*service*”, “*factorial*”, $?x$). In this way, by taking advantage of group-oriented communication, the client is able to dynamically identify a component x that provides the *factorial* service at the desired service level k. Then, via a **put** action, the process invokes the selected service, in a point-to-point fashion, by providing the actual parameter v of the request. After issuing the invocation, the process waits for the result (recall that action **get** is blocking). Whenever the result of the service invocation is made available, the process can withdraw it from the local repository and continue as process P'_{c_h}.

Each server i_{p_j} runs the process A_{p_j} defined in D as:

$$\begin{aligned}
A_{p_j} \triangleq\ & \mathbf{get}(\text{“}invoke\text{”}, \text{“}factorial\text{”}, ?x, ?y)@i_{p_j}. \\
& load := load + 20. \\
& (A_{p_j} \mid Q(x, y))
\end{aligned}$$

The process is triggered by a client request. Whenever this happens, the computational load is updated; we assume that each service instance uses 20% of the sever's capacity. Then, the $factorial$ service becomes again ready to serve other client requests, and the process Q, which actually computes the result of the invoked service for the current request, is executed. We assume that, before its termination, process Q updates the value of attribute *load*, and puts the result of the computation in the repository of the client.

4 Translating SCELight into Promela

In this section we introduce the translation of SCELight specifications into Promela in order to verify ensemble-based scenarios with the model checker SPIN. The translation is formally defined by a family of functions $[\![\cdot]\!]$.

$[\![\langle D, S\rangle]\!] =$
```
/* The type of the interface as a struct of attributes */
typedef interface{
  int attr_1;
  ...
  int attr_w;
}

/* A component-indexed array of interfaces */
interface I[cNum(S)];

/* Component-indexed array of knowledge repositories */
chan K[cNum(S)] = [capacity] of {int, ..., int}
                                 \_____max(S,D)_____/
int initialized = 0;

/* process definitions */
[[D]]^{max(S,D),cNum(S)-1}

/* Component specifications */
[[S]]^{max(S,D),cNum(S)-1}
```

Fig. 3. Translation of SCELight specifications

Specifications. Given a SCELight specification $\langle D, S\rangle$, function $[\![\cdot]\!]$ in Figure 3 returns a Promela specification containing the declaration of the necessary data structures for representing interfaces, knowledge, components and processes. Data structures representing interfaces and knowledge repositories are declared with a global scope; in this way, attributes and knowledge items can be directly accessed by Promela processes.

Interfaces. The translation declares a structured type `interface` as a collection of (integer) variables, one for each attribute; we assume that all components expose the same set of attributes $\{attr_1, \ldots, attr_w\}$. All interfaces are then recorded in the array `I`, whose size is computed by function $cNum(S)$, which returns the number of components in S.

Repositories. All knowledge repositories are grouped together in the array `K`. Each repository is implemented as a channel of tuples of length $max(S, D)$, which corresponds to the maximum length of items used in the definitions D and system S. To simplify message management in Promela, all tuples have the same length and are composed only of integer values. To fulfil this assumption, messages representing shorter items are completed by using dummy values (see Figure 8), while string values are converted into integers in a pre-processing phase. The dimension of repositories is set by means of the parameter *capacity* (its value depends on the application domain).

Initialization and Process -Definitions. The translation also initializes a counter (`initialized`) used to implement a barrier that guarantees that all processes

$$[\![D_1, D_2]\!]^{m,\ell} = [\![D_1]\!]^{m,\ell} \; [\![D_2]\!]^{m,\ell} \qquad [\![S_1 \parallel S_2]\!]^{m,\ell} = [\![S_1]\!]^{m,\ell} \; [\![S_2]\!]^{m,\ell}$$

$$[\![A(\bar{f}) \triangleq P]\!]^{m,\ell} = \texttt{proctype A(f) \{ run A\_0(f) \}} \; [\![P]\!]^{m,\ell,i,\bar{\mathtt{f}} \cup var(P)}_{\mathtt{A\_0}}$$

$$[\![\mathcal{I}_i[\mathcal{K}_i, P_i]]\!]^{m,\ell} =$$

```
active proctype c_i {
  atomic {
    /* Attribute initialization */
    I[i].attr_1 = I_i.attr_1; ... I[i].attr_w = I_i.attr_w;

    /* Knowledge repository initialization */
    ∀ t ∈ K_i : K[i]![[t]];

    /* Increment initialization counter */
    initialized++;
  }
  /* Start when all components are initialized */
  initialized == ℓ + 1 -> run c_i_0(0̄)
}
```

$[\![P_i]\!]^{m,\ell,i,var(P_i)}_{\mathtt{c\_i\_0}}$

Fig. 4. Translation of definitions and system components

start their execution when all initializations of interface attributes and knowledge repositories is terminated. Finally, an auxiliary function $[\![\cdot]\!]^{m,\ell}$ is used to individually translate the process definitions and the system components. This function is parameterized by the maximum length of items m and the highest component index (ranged from 0 to $cNum(S)-1$) necessary to properly translate SCELight processes in D and S.

Process Definitions. The translation of process definitions and system components is reported in Figure 4. A definition $A(\bar{f}) \triangleq P$ is rendered as a declaration of a Promela process (via the `proctype` construct) with the same name and parameters `A(f)`, and followed by the translation of P. As clarified later, the latter is another process declaration that will be activated by the `run` operator within the body of the process declaration `A`.

Components. The translation of a component $\mathcal{I}_i[\mathcal{K}_i, P_i]$ corresponds again to a process declaration, with name `c_`i, that initializes the data structures modelling the component attributes and its knowledge repositories with values in $\mathcal{I}_i$ and $\mathcal{K}_i$. Notably, differently from all other process definitions, component translations are automatically instantiated in the initial system state (by means of the keyword `active`). Since the repository is modelled as a channel `K`$[i]$, the insertion of (the translation of) an item is performed by means of a *send* operation (`!`). When all initializations are completed, the execution of the translation of P_i, defined immediately after `c_`i, is triggered. Such translation is defined as a function $[\![\cdot]\!]^{m,\ell,i,\bar{\mathtt{x}}}_{\mathtt{p}}$ parameterized, besides by m and ℓ, also by the process index i, the set $\bar{\mathtt{x}}$ of variables used in the SCELight process (identified by functions $var(\cdot)$ and

$$[\![\mathbf{nil}]\!]_{\mathtt{p}}^{m,\ell,i,\bar{\mathtt{x}}} = \texttt{proctype p(}\bar{\mathtt{x}}\texttt{) \{ \}}$$

$$[\![a.P]\!]_{\mathtt{p}}^{m,\ell,i,\bar{\mathtt{x}}} = \texttt{proctype p(}\bar{\mathtt{x}}\texttt{) \{ }[\![a]\!]^{m,\ell}\texttt{;run p0 \}}\ [\![P]\!]_{\mathtt{p0}}^{m,\ell,i,\bar{\mathtt{x}}}$$

$$[\![\mathbf{if}\ (e)\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2]\!]_{\mathtt{p}}^{m,\ell,i,\bar{\mathtt{x}}} = \texttt{proctype p(}\bar{\mathtt{x}}\texttt{) \{ if :: atomic\{ }e\texttt{ -> run pt \} :: atomic\{ else -> run pf \} fi \}}\ [\![P_1]\!]_{\mathtt{pt}}^{m,\ell,i,\bar{\mathtt{x}}}\ [\![P_2]\!]_{\mathtt{pf}}^{m,\ell,i,\bar{\mathtt{x}}}$$

$$[\![P_1 \mid P_2]\!]_{\mathtt{p}}^{m,\ell,i,\bar{\mathtt{x}}} = \texttt{proctype p(}\bar{\mathtt{x}}\texttt{) \{ atomic \{ run pl; run pr \} \}}\ [\![P_1]\!]_{\mathtt{pl}}^{m,\ell,i,\bar{\mathtt{x}}}\ [\![P_2]\!]_{\mathtt{pr}}^{m,\ell,i,\bar{\mathtt{x}}}$$

$$[\![A(\bar{u})]\!]_{\mathtt{p}}^{m,\ell,i,\bar{\mathtt{x}}} = \texttt{proctype p(}\bar{\mathtt{x}}\texttt{) \{ run A(}\bar{\mathtt{u}}\texttt{) \}}$$

Fig. 5. Translation of processes

passed as parameters from a process to another) and the name `p` to be used for the process declaration to generate unique process names. To guarantee the uniqueness of process declaration names, for each component i the names of its declarations are prefixed by `c_i_0` and are built by adding a character for each translated construct: `0` for action prefix, `t` or `f` for conditional choice (depending on the branch), `l` or `r` for parallel composition (depending on the side).

Processes. The translation of processes is reported in Figure 5. Each SCELight process is naturally translated into a Promela process declaration. The base cases are the translations of the empty process **nil** and call $A(\bar{u})$, which consist of an empty declaration and a declaration containing only a run statement (see the translation of definitions in Figure 4), respectively. In case an action prefixing $a.P$, the process declaration contains the translation of a, which models the action execution, while the translation of the continuation P is outside the declaration and is activated only after the termination of the action execution. The translation of the other constructs, namely conditional choice and parallel composition, is similar and straightforwardly relies on the Promela constructs for selection (`if ... fi`) and for the parallel execution of processes (via multiple `run` statements). Both cases use an `atomic` block: in case of conditional choice, it just aims at reducing the complexity of the verification model (by restricting the amount of interleaving), while in case of parallel execution this ensures the simultaneous activation of the parallel processes.

Actions. Translation $[\![\cdot]\!]^{m,\ell,i}$ of actions is defined in Figure 6. It is worth noticing that in most cases `atomic` blocks are used to guarantee atomic execution

$[\![\mathbf{get}(T)@n]\!]^{m,\ell,i}$ = `atomic{ K[`n`]???`$[\![T]\!]^m$ `}`

$[\![\mathbf{get}(T)@\mathcal{P}]\!]^{m,\ell,i}$ =
```
if
  :: atomic {P|0 && K[0]??[[T]^m] -> K[0]???[T]^m }
  ...
  :: atomic {P|ℓ && K[ℓ]??[[T]^m] -> K[ℓ]???[T]^m }
fi
```

$[\![\mathbf{qry}(T)@n]\!]^{m,\ell,i}$ = `atomic{ K[`n`]???<`$[\![T]\!]^m$`> }`

$[\![\mathbf{qry}(T)@\mathcal{P}]\!]^{m,\ell,i}$ =
```
if
  :: atomic {P|0 && K[0]??[[T]^m] -> K[0]???<[T]^m> }
  ...
  :: atomic {P|ℓ && K[ℓ]??[[T]^m] -> K[ℓ]???<[T]^m> }
fi
```

$[\![\mathbf{put}(t)@n]\!]^{m,\ell,i}$ = `K[eval(`n`)]!`$[\![t]\!]$`;`

$[\![\mathbf{put}(t)@\mathcal{P}]\!]^{m,\ell,i}$ =
```
atomic{
  int j=0;
  do
    :: j == ℓ -> break
    :: P|j    -> K[j]![[t]]; j++
    :: else   -> j++
  od
}
```

$[\![attr := e]\!]^{m,\ell,i}$ = `I[`i`].attr = e;`

Fig. 6. Translation of actions

of the actions. We also recall that the FIFO receive operations of Promela on asynchronous channels are `q?m` (remove the first message from channel `q` if it matches `m` and update the variables in `m` accordingly); `q?<m>` (test if the first message on channel `q` matches `m` and update the variables in `m` accordingly); and `q?[m]` (test if the first message on channel `q` matches `m` without side-effects on the variables of `m`). In addition, Promela provides three so-called random receive variants of the previous ones (denoted with `??` in place of `?`), which remove/test the oldest tuple matching the pattern instead of the first one.

A point-to-point action $\mathbf{get}(T)@n$ is basically modeled as a (pattern-matching-based) receive operation (`???`) on the channel `K[eval(`n`)]` corresponding to the knowledge repository of the component identified by n. Note that `q???m` is not the primitive Promela operation `q??m` but an abbreviation defined in Figure 7. The receive operation `q??m` does encode the semantics we need since it removes the

```
i = len(q);
do
  :: q??m -> break
  :: i>0  -> q??m; q!m; i--
od
```

Fig. 7. Abbreviation `q???m`

$$[\![T]\!]^m = [\![T]\!], \underbrace{\_, \ldots, \_}_{m-|T|} \qquad [\![v]\!] = \texttt{v} \qquad [\![x]\!] = \texttt{eval(x)} \qquad [\![?\,x]\!] = \texttt{x;} \qquad [\![T_1, T_2]\!] = [\![T_1]\!], [\![T_2]\!]$$

$$[\![e]\!] = \texttt{e} \qquad [\![t_1, t_2]\!] = [\![t_1]\!], [\![t_2]\!]$$

Fig. 8. Translation of templates and items

oldest tuple in the channel among those matching the template `m` and not *any* of them as required by the semantics of **get**. The abbreviation `q???m` ensures a non-deterministic removal by non-deterministically choosing between (i) removing the oldest matched item and (ii) looping after reinserting the oldest matched item in the queue so that it becomes the newest such item. The latter can be attempted as many times as the size of `q` to ensure termination and it guarantees that all possible messages matching `m` will be considered.

A group-oriented action $\mathbf{get}(T)@\mathcal{P}$ is translated as a non-deterministic choice among a set of input actions on each repository. In particular, for each repository i there is a branch guarded by $\mathcal{P}|_i$ `&& K[`i`]??[`$[\![T]\!]^m$`]` which will ensure the transition to fire only if the target predicate holds for component i and i has a matching item in its repository. If that is the case the item is indeed removed using again the non-deterministic input operation in `K[`i`]???`$[\![T]\!]^m$.

Actions $\mathbf{qry}(T)@n$ and $\mathbf{qry}(T)@\mathcal{P}$ are translated in the same way, except for the use of the non-consuming variant (`???<...>`) of the receive operation, while a point-to-point **put** action is simply translated as a send operation (`!`) on the appropriate channel, while the group-oriented one consists of a loop that sends the tuple to the repositories of all components satisfying the target predicate. The selection statement permits ignoring the components that do not satisfy the predicate (in fact, the **put** action is non-blocking).

Action $attr := e$ is straightforwardly translated as an assignment of expression e to the attribute $attr$ exposed in the interface of the proper component (the latter is identified by the parameter i of the translation function).

Templates and Items. Function $[\![\cdot]\!]^m$ (Figure 8) returns a template of length m by concatenating the translation of the template given as argument with a sequence of so-called *hidden* variables (denoted by "`_`"). The translation functions $[\![T]\!]$ and $[\![t]\!]$ are straightforward. Filling the tuple with dummy values is not needed in the translation of items; this is automatically done by SPIN. It is worth to recall as well that function `eval( · )`, instead, is used for evaluating variables and protecting them from assignments in the matching mechanism.

5 Verification

We illustrate in this section some examples of how SPIN can be used to check and verify properties of SCELight specifications, by resorting to the translation of the SCELight specification of the scenario presented in Section 3.

Checking Deadlock Absence. One first property one would like to check is absence of deadlocks. Obviously not every instance of our scenario is deadlock free. Indeed, if the instance contains clients requiring a service that is not offered by any server or that cannot be served at the required quality level, deadlocks may arise since SCEL input operations have blocking semantics. Notably, the system can have valid terminal states as well, since clients gracefully terminate after successfully receiving the results from servers.

Below, we report an example result of invoking SPIN for checking deadlock absence in an instance of our scenario with 3 servers with different hardware configurations and 5 clients invoking the services offered by the servers:

```
State-vector 1828 byte, depth reached 81, errors: 0
  3849511 states, stored
```

The result is positive (no errors) and SPIN explores a few millions of states.

Checking Server Overload. Another typical use of SPIN that is very convenient for our purposes is to look for interesting executions by characterizing them by means of an LTL formula and asking SPIN for a counterexample. For example, in our scenario, to obtain system runs overloading the server s_i we can specify a formula $\Box\, \mathcal{I}_{s_i}.load \leq 100$, which states that server s_i will never be overloaded.

Indeed, if we check the above invariant in an instance of our scenario with one *gold* server and 6 clients requiring a *gold* service, SPIN returns a counterexample

```
pan:1: assertion violated  !( !((I[0].load<=100))) (at depth 145)
pan: wrote client-server-scenario.pml.trail
```

which consists of an execution of the system, i.e. a trail (stored in the file `client-server-scenario.pml.trail`), in which the server accepts and executes the six requests concurrently, which causes its load to be $6 \times 20\% = 120\%$. One may think that if the clients request a *base* service, it would not be possible to overload the server, as the *gold* server will accept to serve only a few *base* requests concurrently. Actually, this is not true. The reason is that even if a *gold* server will belong to the *base* ensemble only if its load is below 20%, it may be identified as a target by several concurrent clients before actually accepting any service request (and hence updating its load). Indeed, SPIN provides a counterexample also for the above property for a configuration with *gold* server and 6 clients sending *base* requests. Of course, the problem raised by this verification result can be easily fixed by changing the servers specification in order to check the *load* value before accepting additonal requests.

Checking Responsiveness. Finally, we show an example of a typical liveness property expressing the fact that clients are guaranteed responsiveness: whenever a client invokes the factorial service, it will eventually get a result. This can be formalized with the usual LTL formulae of the form $\Box(request \rightarrow \Diamond response)$. SPIN provides positive answers for all possible instances of our scenario, since once a client finds an appropriate server for the required service, the server cannot avoid providing the service.

6 Concluding Remarks

We have presented a formal approach to the specification and verification of ensemble-based systems, by providing a translation of SCELight specifications into Promela, the specification language of the SPIN model checker. SCELight is a dialect of the SCEL specification language specifically devised in the EU project Ascens [11] for modelling autonomic, ensemble-based, systems. We have illustrated our approach by verifying a few properties of a service provision scenario. The presented approach enriches the toolset support for SCEL-based engineering of ensemble systems, which currently includes statistical model checking in MiScel [12], the Maude-based SCEL interpreter, and run-time testing with jRESP [13], the Java-based run-time environment for SCEL.

As future work, we plan to continue our programme to verify ensemble-based systems by pursuing different lines of research. The proposed approach will be enhanced by optimizing the generated Promela code to enable a more efficient verification, e.g. by reducing the number of process declarations and invocations, which is actually only required to deal with parallel composition and recursion. Moreover, to foster the practical application of the approach, the SCELight to Promela translation will be implemented in a standard programming language, like Java, by resorting to supporting framework specifically devised for this purpose like Xtext [14]. From a more theoretical perspective, we intend to formally prove that the presented encoding is sound and complete with respect to the operational semantics of SCEL and Promela.

We also plan to extend the work by considering the SCEL constructs not included in SCELight. The main challenge will be to treat the dynamic creation of new names and components for which SPIN does not offer any (efficient) verification support. Some techniques have been proposed to deal with dynamic aspects of software in SPIN (see e.g. [15,16]), but they are not included in the official SPIN distribution. To deal with dynamicity, we plan to investigate the use of other verification tools that provide a better support to these features. We plan also to consider the possibility of using the operational semantics of SCEL as a starting point to generate systems descriptions that can be provided as input to the BIP toolset. The challenge here is understanding if the dynamic part of full SCEL specifications can be "constrained" to provide a full model to be analyzed in BIP and if Dy-BIP [17], the extension of BIP [18] to deal with dynamic architectures, will do a better service.

Finally, another promising line of research that we intend to explore concerns the extension of Promela and BIP, with primitives for group-oriented communication. In fact, on the one hand, the suitability of SCEL to model ensemble-based systems points out the benefits of such form of communication in this application domain. On the other hand, avoiding specification translations would improve efficiency of the verification and, hence, its effectiveness.

References

1. Project InterLink (2007), `http://interlink.ics.forth.gr`

2. Sommerville, I., Cliff, D., Calinescu, R., Keen, J., Kelly, T., Kwiatkowska, M.Z., McDermid, J.A., Paige, R.F.: Large-scale complex IT systems. Commun. ACM 55(7), 71–77 (2012)
3. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: the SCEL Language. ACM Transactions on Autonomous and Adaptive Systems (to appear, 2014), available as Technical Report from http://eprints.imtlucca.it/2117/
4. Cesari, L., De Nicola, R., Pugliese, R., Puviani, M., Tiezzi, F., Zambonelli, F.: Formalising Adaptation Patterns for Autonomic Ensembles. In: Proc. of the 10th International Symposium on Formal Aspects of Component Software (FACS 2013). LNCS, Springer, Heidelberg (2014)
5. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: A Language-Based Approach to Autonomic Computing. In: Beckert, B., Damiani, F., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 25–48. Springer, Heidelberg (2012), http://rap.dsi.unifi.it/scel/
6. Margheri, A., Pugliese, R., Tiezzi, F.: Linguistic Abstractions for Programming and Policing Autonomic Computing Systems. In: Proc. of the 10th IEEE International Conference on Autonomic and Trusted Computing (ATC 2013). IEEE Computer Society (2014)
7. Mayer, P., Klarl, A., Hennicker, R., Puviani, M., Tiezzi, F., Pugliese, R., Keznikl, J., Bures, T.: The Autonomic Cloud: A vision of voluntary, peer-2-peer cloud computing. In: Proc. of the 2013 IEEE Seventh International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2013). IEEE Computer Society (2014)
8. Bures, T., De Nicola, R., Gerostathopoulos, I., Hoch, N., Kit, M., Koch, N., Monreale, G., Montanari, U., Pugliese, R., Serbedzija, N., Wirsing, M., Zambonelli, F.: A Life Cycle for the Development of Autonomic Systems: The e-mobility showcase. In: Proc. of the 2013 IEEE Seventh International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2013). IEEE Computer Society (2014)
9. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. 23(5), 279–295 (1997)
10. De Nicola, R., Ferrari, G., Pugliese, R.: Klaim: A Kernel Language for Agents Interaction and Mobility. IEEE Trans. Software Eng. 24(5), 315–330 (1998)
11. ASCENS: Autonomic service-component ensembles, http://www.ascens-ist.eu/
12. Belzner, L., De Nicola, R., Vandin, A., Wirsing, M.: Reasoning (on) Service Component Ensembles in Rewriting Logic. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi, SAS 2014 (to appear, April 2014)
13. jRESP, http://code.google.com/p/jresp/
14. Xtext, http://www.eclipse.org/Xtext/
15. Demartini, C., Iosif, R., Sisto, R.: dSPIN: A Dynamic Extension of SPIN. In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 261–276. Springer, Heidelberg (1999)
16. Iosif, R.: Symmetry reductions for model checking of concurrent dynamic software. STTT 6(4), 302–319 (2004)
17. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling Dynamic Architectures Using Dy-BIP. In: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (eds.) SC 2012. LNCS, vol. 7306, pp. 1–16. Springer, Heidelberg (2012)
18. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)

Parametric and Quantitative Extensions of Modal Transition Systems

Uli Fahrenberg[1], Kim Guldstrand Larsen[2],
Axel Legay[1], and Louis-Marie Traonouez[1]

[1] Inria/IRISA, Rennes, France
[2] Aalborg University, Aalborg, Denmark

Abstract. Modal transition systems provide a behavioral and compositional specification formalism for reactive systems. We survey two extensions of modal transition systems: parametric modal transition systems for specifications with parameters, and weighted modal transition systems for quantitative specifications.

1 Introduction

Modal transition systems [21,23] provide a behavioral and compositional specification formalism for reactive systems. They grew out of the notion of relativized bisimulation [20], which allows for simple specifications of components by allowing the notion of bisimulation to take into account the restricted use that a given component may have in its context.

A modal transition system is essentially a (labeled) transition system, but with two types of transitions: so-called *may*-transitions which any implementation may (or may not) have, and *must*-transitions which any implementation is required to have. In fact, ordinary labeled transition systems (or implementations) are modal transition systems where the set of may- and must-transitions coincide. Modal transition systems come equipped with a bisimulation-like notion of (modal) refinement, reflecting that the more must-transitions and the fewer may-transitions a modal specification has the more refined and closer to a final implementation it is.

Example 1. Consider the modal transition system shown in Fig. 1 which models the requirements of a simple email system in which emails are first received and then delivered; must- and may-transitions are represented by solid and dashed arrows, respectively. Before delivering the email, the system may check or process the email, *e.g.* for encryption or decryption, filtering of spam emails, or generating automatic answers using an auto-reply feature. Any implementation of this email system specification *must* be able to receive and deliver email, and it *may* also be able to check arriving email before delivering it. No other behavior is allowed. Such a valid implementation is given in Fig. 2.

The theory of modal transition systems (MTS), or *modal specifications* as they were called in the paper [21] in the proceedings of the first CAV conference

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 84–97, 2014.

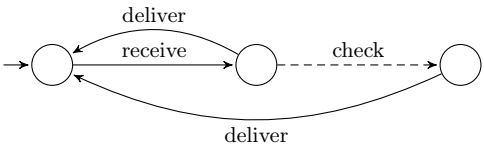


Fig. 1. Modal transition system modeling a simple email system, with an optional behavior: Once an email is received, it may be checked, *e.g.* be scanned for containing viruses, or automatically decrypted, before it is delivered to the receiver

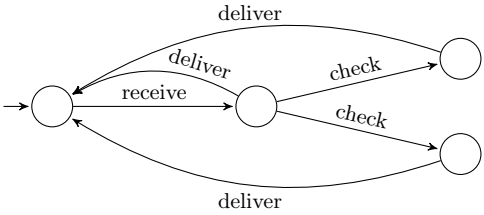


Fig. 2. An implementation of the simple email system in Fig. 1 in which we explicitly model two distinct types of email pre-processing

organized by Joseph Sifakis in Grenoble,[1] was aiming at providing a *behavioral* compositional specification formalism for reactive systems. At the time of the introduction of MTS, there were two predominant approaches to specifications formalisms and verification methods for reactive and concurrent systems: *logical* approaches where a specification is a set of properties of implementations (labeled transition systems), and *graphical* approaches promoted by the various process algebras, where implementations and specifications are systems of the same kind – namely labeled transition systems, and verification amounts to compare such systems with respect to a given behavioral preorder, *e.g.* bisimilarity.

In search for a complete specification theory, the following properties have been considered desirable (the first three were listed in the early paper [6]):

expressiveness: the specification formalism should be powerful enough to express all properties of a given implementation. In other words it should be possible to completely specify any labeled transition system, up to bisimulation.

modularity: implementations are often made out of several components, and it should be possible to infer satisfaction of an overall specification solely on the basis of sub-specification of the sub-components.

refinement: one should have the ability to deal with partial specifications, requiring more and more properties about a system, up to its complete specification.

[1] In fact, the first CAV conference was not called CAV, but had the rather lengthy title "Automatic Verification Methods for Finite State Systems."

logical composition: specification should be composable with respect to usual logical operators such as conjunction and (possibly) disjunction.

quotienting: given an overall specification S of a composite systems as well as a sub-specification T of a sub-component, the existence of a quotient specification $S\backslash T$ will describe the sufficient and necessary condition of the remaining components in order that S is satisfied by the total systems.

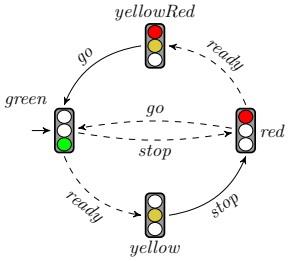
Applying these criteria to the logical and graphical (*i.e.* bisimulation) framework, as was done in [6], we see that the logical and graphical frameworks offer complementary advantages: on the graphical side, expressiveness is trivial since a process i a specification of itself. Modularity is usually guaranteed by the fact that bisimulations are compatible with (most) process constructors. On the logical side, expressiveness is achieved if we allow possibly infinite sets of formulae as logical specifications, or admit recursively specified properties. The point of modularity has proved more difficult with early attempts of Sifakis and Graf [15] and Holmstrøm [17] providing sound and highly usable proof systems for specifications mixing logical and behavioral constructs (as well as fix-point constructs) but lacking accompanying completeness results. Much later the work of Mardare and Policriti [25] provided a first matching completeness result.

In the rest of this paper, we survey two extensions of modal transition systems. The first extension, *parametric* modal transition systems, is concerned with systems whose behaviors depend on parameters [4]. The second extension, *weighted* modal transition systems [1, 2] permits to reason on systems whose behaviors depend on quantities. Another paper in this volume [11] will be concerned with other extensions of modal transition systems which are more closely related to applications.

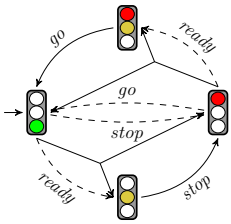
2 Parametric Modal Transition Systems

It is well admitted (see *e.g.* [27]) that MTS and their extensions like disjunctive MTS (DMTS) [24], 1-selecting MTS (1MTS) [13] and transition systems with obligations (OTS) [5] provide strong support for a specification formalism allowing for step-wise refinement process. Moreover, the MTS formalisms have applications in other contexts, which include verification of product lines [16,22], interface theories [27,28] and modal abstractions in program analysis [14,18,26].

Unfortunately, all of these formalisms lack the capability to express some intuitive specification requirements like exclusive, conditional and persistent choices. In [4] the expressive power of MTS and its variants has been extended considerably so it can model model arbitrary Boolean conditions on transitions and also allows to instantiate persistent transitions. The model, called *parametric modal transition systems* (PMTS), is equipped with a finite set of parameters that are fixed prior to the instantiation of the transitions in the specification. The generalized notion of modal refinement is designed to handle the parametric extension and it specializes to the well-studied modal refinements on all the subclasses of our model like MTS, disjunctive MTS and MTS with obligations.

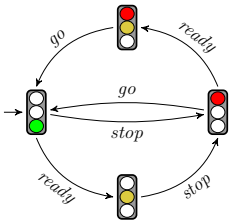


(a) MTS specification S_1

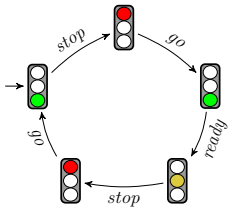


(b) DMTS specification S_2

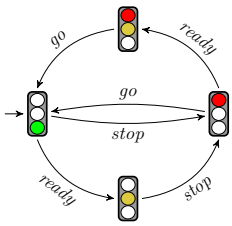
(c) Implementation I_1



(d) Implementation I_2

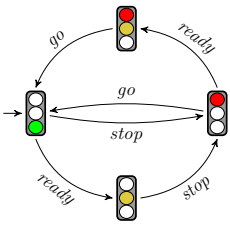


(e) Implementation I_3



Obligation function:
$\Phi(green) = (stop, red) \oplus (ready, yellow)$
$\Phi(red) = (go, green) \oplus (ready, yellowRed)$

(f) Specification S_3



Parameters: $\{reqYfromR, reqYfromG\}$
Obligation function:
$\Phi(green) = ((stop, red) \oplus (ready, yellow))$
$\wedge (reqYfromG \Leftrightarrow (ready, yellow))$
$\Phi(red) = ((go, green) \oplus (ready, yellowRed))$
$\wedge (reqYfromR \Leftrightarrow (ready, yellowRed))$

(g) PMTS specification S_4

Fig. 3. Specifications and implementations of a traffic light controller

2.1 Motivation

We shall now discuss these limitations on an example as a motivation for the introduction of parametric MTS formalism with general Boolean conditions in specification requirements.

Consider a simple specification of a traffic light controller that can be at any moment in one of the four predefined states: *red*, *green*, *yellow* or *yellowRed*. The requirements of the specification are: when *green* is on the traffic light may either change to *red* or *yellow* and if it turned *yellow* it must go to *red* afterward; when *red* is on it may either turn to *green* or *yellowRed*, and if it turns *yellowRed* (as it is the case in some countries) it must go to *green* afterwords.

Fig. 3a shows an obvious MTS specification of the proposed specification. The transitions in the standard MTS formalism are either of type may (optional transitions depicted as dashed lines) or must (required transitions depicted as solid lines). In Fig. 3c, Fig. 3d and Fig. 3e we present three different implementations of the MTS specification where there are no more optional transitions. The implementation I_1 does not implement any may transition as it is a valid possibility to satisfy the specification S_1. Of course, in our concrete example, this means that the light is constantly *green* and it is clearly an undesirable behavior that cannot be, however, easily avoided. The second implementation I_2 on the other hand implements all may transitions, again a legal implementation in the MTS methodology but not a desirable implementation of a traffic light as the next action is not always deterministically given. Finally, the implementation I_3 of S_1 illustrates the third problem with the MTS specifications, namely that the choices made in each turn are not persistent and the implementation alternates between entering *yellow* or not. None of these problems can be avoided when using the MTS formalism.

A more expressive formalism of disjunctive modal transition systems (DMTS) can overcome some of the above mentioned problems. A possible DMTS specification S_2 is depicted in Fig. 3b. Here the *ready* and *stop* transitions, as well as *ready* and *go* ones, are disjunctive, meaning that it is still optional which one is implemented but at least one of them must be present. Now the system I_1 in Fig. 3c is not a valid implementation of S_2 any more. Nevertheless, the undesirable implementations I_2 and I_3 are still possible and the modeling power of DMTS is insufficient to eliminate them.

Inspired by the recent notion of transition systems with obligations [5], we can model the traffic light using specification as a transition system with arbitrary[2] obligation formulae. These formulae are Boolean propositions over the outgoing transitions from each state, whose satisfying assignments yield the allowed combinations of outgoing transitions. A possible specification called S_3 is given in Fig. 3f and it uses the operation of exclusive-or. We will follow an agreement that whenever the obligation function for some node is not listed in the system description then it is implicitly understood as requiring all the available outgoing transitions to be present. Due to the use of exclusive-or in the obligation function, the transition systems I_1 and I_2 are not valid implementation any more. Nevertheless, the implementation I_3 in Fig. 3e cannot be avoided in this formalism either.

Finally, the problem with the alternating implementation I_3 is that we cannot enforce in any of the above mentioned formalisms a uniform (persistent) implementation of the same transitions in all its states. In order to overcome this problem, we propose the so-called parametric MTS where we can, moreover, choose persistently whether the transition to *yellow* is present or not via the use of parameters. The PMTS specification with two parameters *reqYfromR* and *reqYfromG* is shown in Fig. 3g. Fixing a priori the (Boolean) values of the

[2] In the transition systems with obligations only positive Boolean formulae are allowed.

parameters makes the choices permanent in the whole implementation, hence we eliminate also the last problematic implementation I_3.

2.2 Definition

We shall now formally capture the intuition behind parametric MTS introduced above. First, we recall the standard propositional logic.

A Boolean formula over a set X of atomic propositions is given by the following abstract syntax

$$\varphi ::= \mathbf{tt} \mid x \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi$$

where x ranges over X. The set of all Boolean formulae over the set X is denoted by $\mathcal{B}(X)$. Let $\nu \subseteq X$ be a truth assignment, *i.e.* a set of variables with value true, then the satisfaction relation $\nu \models \varphi$ is given by $\nu \models \mathbf{tt}$, $\nu \models x$ iff $x \in \nu$, and the satisfaction of the remaining Boolean connectives is defined in the standard way. We also use the standard derived operators like exclusive-or $\varphi \oplus \psi = (\varphi \wedge \neg\psi) \vee (\neg\varphi \wedge \psi)$, implication $\varphi \Rightarrow \psi = \neg\varphi \vee \psi$ and equivalence $\varphi \Leftrightarrow \psi = (\neg\varphi \vee \psi) \wedge (\varphi \vee \neg\psi)$.

We can now proceed with the definition of parametric MTS.

Definition 1. *A* parametric MTS *(PMTS) over an action alphabet Σ is a tuple (S, T, P, Φ) where S is a set of* states*, $T \subseteq S \times \Sigma \times S$ is a* transition relation*, P is a finite set of* parameters*, and $\Phi : S \to \mathcal{B}((\Sigma \times S) \cup P)$ is an* obligation function *over the atomic propositions containing outgoing transitions and parameters. We implicitly assume that whenever $(a, t) \in \Phi(s)$ then $(s, a, t) \in T$. By $T(s) = \{(a, t) \mid (s, a, t) \in T\}$ we denote the set of all outgoing transitions of s.*

PMTS has been provided a refinement notion that generalizes the well-studied refinement notions on its subclasses including that of MTS. In the definition, the parameters are fixed first (persistence) followed by all valid choices modulo the fixed parameters that now behave as constants.

First we set the following notation. Let (S, T, P, Φ) be a PMTS and $\nu \subseteq P$ be a truth assignment. For $s \in S$, we denote by $\mathrm{Tran}_\nu(s) = \{E \subseteq T(s) \mid E \cup \nu \models \Phi(s)\}$ the set of all admissible sets of transitions from s under the fixed truth values of the parameters.

We can now define the notion of modal refinement between PMTS.

Definition 2. *Let (S_1, T_1, P_1, Φ_1) and (S_2, T_2, P_2, Φ_2) be two PMTS. A binary relation $R \subseteq S_1 \times S_2$ is a* modal refinement *if for each $\mu \subseteq P_1$ there exists $\nu \subseteq P_2$ such that for every $(s, t) \in R$ holds*

$$\forall M \in \mathit{Tran}_\mu(s) : \exists N \in \mathit{Tran}_\nu(t) : \begin{array}{l} \forall (a, s') \in M : \exists (a, t') \in N : (s', t') \in R \ \wedge \\ \forall (a, t') \in N : \exists (a, s') \in M : (s', t') \in R \ . \end{array}$$

We say that s modally refines t, denoted by $s \leq_{\mathsf{m}} t$, if there exists a modal refinement R such that $(s, t) \in R$.

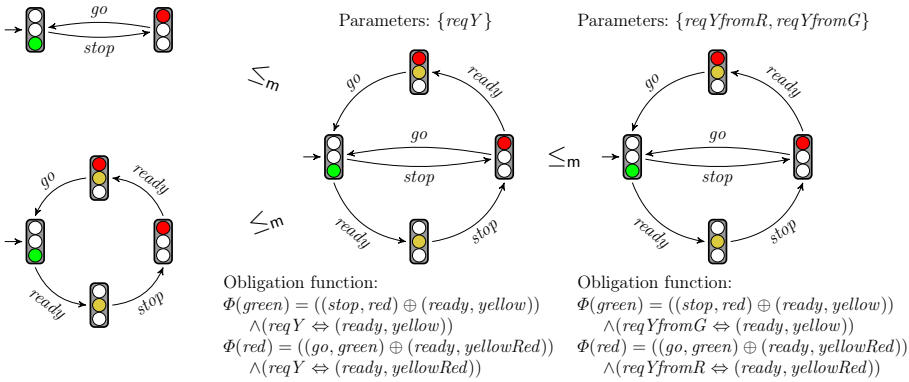


Fig. 4. Example of modal refinement

Example 2. Consider the rightmost PMTS in Fig. 4. It has two parameters *reqYfromG* and *reqYfromR* whose values can be set independently and it can be refined by the system in the middle of the figure having only one parameter $reqY$. This single parameter simply binds the two original parameters to the same value. The PMTS in the middle can be further refined into the implementations where either *yellow* is always used in both cases, or never at all. Notice that there are in principle infinitely many implementations of the system in the middle, however, they are all bisimilar to either of the two implementations depicted in the left of Fig. 4.

[4] provides an extensive study of the complexity of refinement checking between parametric modal transitions with classification depending on the complexity of obligations as well as the presence or absence of parameters. For each combination the complexity class of the polynomial hierarchy for which modal refinement is complete is provided. In short, the complexities ranges from P-complete to Π_4^p-complete (thus in PSPACE).

3 Quantitative Modal Transition Systems

Motivated by applications to embedded, real-time and hybrid systems, the modal transition system framework has been extended in order to reason about *quantitative* aspects [3,19]. With these applications in mind, it is necessary not only to be able to *specify* quantitative aspects of systems, but also to formalize successive *refinement* of quantities. To illustrate this extension, consider again the modal transition system of Fig. 1, but this time with quantities, see Fig. 5: Every transition label is extended by integer intervals modeling upper and lower bounds on time required for performing the corresponding actions. For instance, the reception of a new email (action *receive*) must take between one and three time units, the checking of the email (action *check*) is allowed to take up to five time units.

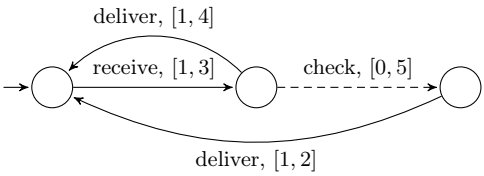


Fig. 5. Specification of a simple email system, similar to Fig. 1, but extended by integer intervals modeling time units for performing the corresponding actions

In this quantitative setting, there is a problem with using a *Boolean* notion of refinement as is done in the preceding section: If one only can decide *whether or not* an implementation refines a specification, then the quantitative aspects get lost in the refinement process. As an example, consider the email system implementations in Fig. 6. Implementation (a) does not refine the specification, as there is an error in the discrete structure of actions: after receiving an email, the system can check it indefinitely without ever delivering it. Also implementations (b) and (c) do not refine the specification: (b) takes too long to receive email, (c) does not deliver email fast enough after checking it. Implementation (d) on the other hand is a perfect refinement of the specification.

Intuitively however, implementations (b) and (c) conform much better to the specification than implementation (a) in Fig. 6: there are no discrepancies in the discrete structure, only the weights are off by 1. Additionally, the quantitative error in implementation (c) occurs later than the one in (b). Hence one may want to say that implementation (d) is in perfect refinement of the specification, (c) is slightly off, (b) is a bit more problematic, whereas implementation (a) is completely unacceptable. A Boolean notion of refinement does not allow to make such distinctions between different negative answers.

To sum up, a Boolean notion of refinement is too *fragile* for quantitative formalisms. Minor and major modifications in the implementation cannot be distinguished, as both of them may reverse the Boolean answer. As observed *e.g.* in [9], this view is obsolete; engineers need quantitative notions on how modified implementations differ. The introduction of such a quantitative notion of refinement, and its consequences for the specification theory, are the subject of this section, which is based on the papers [1, 2].

Depending on the precise application of our quantitative formalism, there are a few choices which one has to make. One such choice is the precise definition of quantitative refinement, as the way quantitative discrepancies between specifications is measured *e.g.* depends on whether differences accumulate over time or the interest more lies in the maximal individual differences. Another choice is how to combine quantities during structural composition: when modeling *e.g.* energy consumption, they should be added; when modeling timing constraints, some form of conjunction should be used.

To facilitate quantitative reasoning on specifications and implementations, we introduce a real-valued *distance* between specifications such that perfect refinement corresponds to distance 0, small quantitative discrepancies give rise to

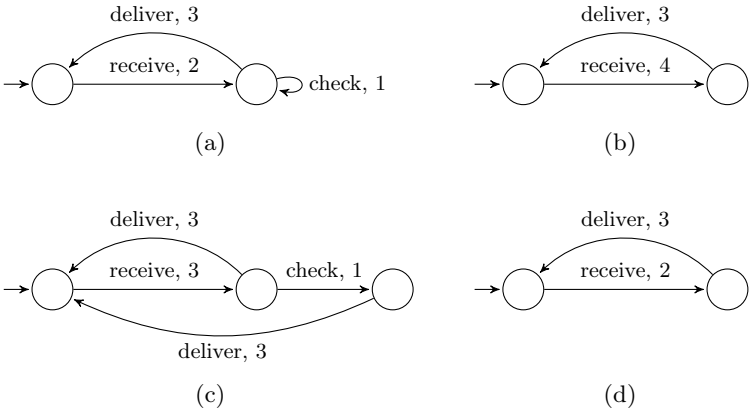


Fig. 6. Four implementations of the simple email system in Fig. 5

small distances, and differences in the discrete control structure correspond to distance ∞. For the examples in Figs. 5 and 6, we will deduce the following chain of decreasing distances:

$$\infty = d(I_1, S) > d(I_2, S) > d(I_3, S) > d(I_4, S) = 0$$

3.1 Weighted Modal Transition Systems

Let Σ be a set of labels with a preorder $\sqsubseteq \subseteq \Sigma \times \Sigma$, and denote by $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ the set of finite and infinite traces over Σ. $\mathsf{len}(\sigma)$, for $\sigma \in \Sigma^\infty$, denotes the length (finite or infinite) of a trace σ. Let $\varepsilon \in \Sigma^\infty$ denote the empty trace, and for $a \in \Sigma$, $\sigma \in \Sigma^\infty$, denote by $a.\sigma$ their concatenation.

A *weighted modal transition system* (WMTS) is a tuple $\mathcal{S} = (S, s^0, \dashrightarrow, \longrightarrow)$ consisting of a set S of states, an initial state $s_0 \in S$, and must- and may-transitions $\longrightarrow, \dashrightarrow \subseteq S \times \Sigma \times S$ for which it holds that for all $s \xrightarrow{a} s'$ there is $s \overset{b}{\dashrightarrow} s'$ with $a \sqsubseteq b$.

Intuitively, a may-transition $s \overset{b}{\dashrightarrow} t$ specifies that an implementation $\mathcal{I}$ of $\mathcal{S}$ is *permitted* to have a corresponding transition $i \xrightarrow{a} j$, for any $a \sqsubseteq b$, whereas a must-transition $s \xrightarrow{b} t$ postulates that $\mathcal{I}$ is *required* to implement at least one corresponding transition $i \xrightarrow{a} j$ for some $a \sqsubseteq b$. We will make this precise below.

An WMTS $\mathcal{S} = (S, s^0, \dashrightarrow, \longrightarrow)$ is an *implementation* if $\longrightarrow = \dashrightarrow$. Hence in an implementation, all optional behavior has been resolved.

Definition 3. *A* modal refinement *of WMTS* $\mathcal{S}_1 = (S_1, s_1^0, \dashrightarrow_1, \longrightarrow_1)$, $\mathcal{S}_2 = (S_2, s_2^0, \dashrightarrow_2, \longrightarrow_2)$ *is a relation* $R \subseteq S_1 \times S_2$ *such that for any* $(s_1, s_2) \in R$,

- *whenever* $s_1 \overset{a_1}{\dashrightarrow}_1 t_1$, *then also* $s_2 \overset{a_2}{\dashrightarrow}_2 t_2$ *for some* $a_1 \sqsubseteq a_2$ *and* $(t_1, t_2) \in R$,
- *whenever* $s_2 \xrightarrow{a_2}_2 t_2$, *then also* $s_1 \xrightarrow{a_1}_1 t_1$ *for some* $a_1 \sqsubseteq a_2$ *and* $(t_1, t_2) \in R$.

Thus any behavior which is permitted in $\mathcal{S}_1$ is also permitted in $\mathcal{S}_2$, and any behavior required in $\mathcal{S}_2$ is also required in $\mathcal{S}_1$. We write $\mathcal{S}_1 \leq_{\mathsf{m}} \mathcal{S}_2$ if there is a modal refinement $R \subseteq S_1 \times S_2$ with $(s_1^0, s_2^0) \in R$.

The *implementation semantics* of a WMTS $\mathcal{S}$ is the set $[\![\mathcal{S}]\!] = \{\mathcal{I} \leq_{\mathsf{m}} \mathcal{S} \mid \mathcal{I} \text{ implementation}\}$, and we write $\mathcal{S}_1 \leq_{\mathsf{t}} \mathcal{S}_1$ if $[\![\mathcal{S}_1]\!] \subseteq [\![\mathcal{S}_2]\!]$, saying that $\mathcal{S}_1$ *thoroughly refines* $\mathcal{S}_2$. It follows by transitivity of $\leq_{\mathsf{m}}$ that $\mathcal{S}_1 \leq_{\mathsf{m}} \mathcal{S}_2$ implies $\mathcal{S}_1 \leq_{\mathsf{t}} \mathcal{S}_2$, hence modal refinement is a *syntactic over-approximation* of thorough refinement.

3.2 Distances

Recall that a *hemimetric* on a set X is a function $d : X \times X \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ which satisfies $d(x,x) = 0$ and $d(x,y) + d(y,z) \geq d(x,z)$ (the *triangle inequality*) for all $x, y, z \in X$. Note that our hemimetrics are *extended* in that they can take the value ∞.

We will need to generalize hemimetrics to codomains other than $\mathbb{R}_{\geq 0} \cup \{\infty\}$. For a partially ordered monoid $(\mathbb{L}, \sqsubseteq, \oplus, \mathbb{0})$, an $\mathbb{L}$-*hemimetric* on X is a function $d : X \times X \to \mathbb{L}$ which satisfies $d(x,x) = \mathbb{0}$ and $d(x,y) \oplus d(y,z) \sqsupseteq d(x,z)$ for all $x, y, z \in X$.

Definition 4. *A* trace distance *is a hemimetric $td : \Sigma^\infty \times \Sigma^\infty \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ for which $td(a,b) = 0$ for all $a, b \in \Sigma$ with $a \sqsubseteq b$ and $td(\sigma, \tau) = \infty$ whenever $\mathsf{len}(\sigma) \neq \mathsf{len}(\tau)$.*

For any set M, let $\mathbb{L}M = (\mathbb{R}_{\geq 0} \cup \{\infty\})^M$ the set of functions from M to the extended non-negative real line. Then $\mathbb{L}M$ is a complete lattice with partial order $\sqsubseteq \subseteq \mathbb{L}M \times \mathbb{L}M$ given by $\alpha \sqsubseteq \beta$ if and only if $\alpha(x) \leq \beta(x)$ for all $x \in M$, and with an addition $\oplus$ given by $(\alpha \oplus \beta)(x) = \alpha(x) + \beta(x)$. The bottom element of $\mathbb{L}M$ is also the zero of $\oplus$ and given by $\bot(x) = 0$, and the top element is $\top(x) = \infty$.

Definition 5. *A* recursive specification *of a trace distance td consists of*

- *a set M with a lattice homomorphism $\mathsf{eval} : \mathbb{L}M \to \mathbb{R}_{\geq 0} \cup \{\infty\}$,*
- *an $\mathbb{L}M$-hemimetric $td^{\mathbb{L}M} : \Sigma^\infty \times \Sigma^\infty \to \mathbb{L}M$ which satisfies $td = \mathsf{eval} \circ td^{\mathbb{L}M}$ and $td^{\mathbb{L}M}(a,b) = \bot$ for all $a, b \in \Sigma$ with $a \sqsubseteq b$, and*
- *a function $F : \Sigma \times \Sigma \times \mathbb{L}M \to \mathbb{L}M$.*

F must be monotone in the third coordinate and satisfy, for all $a, b \in \Sigma$ and $\sigma, \tau \in \Sigma^\infty$, that $td^{\mathbb{L}M}(a.\sigma, b.\tau) = F(a, b, td^{\mathbb{L}M}(\sigma, \tau))$.

Note that the definition implies that for all $a, b \in \Sigma$, $td^{\mathbb{L}M}(a,b) = td^{\mathbb{L}M}(a.\varepsilon, b.\varepsilon) = F(a, b, td^{\mathbb{L}M}(\varepsilon, \varepsilon)) = F(a, b, \bot)$. Hence also $F(a, a, \bot) = td^{\mathbb{L}M}(a,a) = \bot$ for all $a \in \Sigma$.

We have shown in [2, 10, 12] that all commonly used trace distances obey a recursive characterization as above. The point-wise distance from [8], for example, has $\mathbb{L} = \mathbb{R}_{\geq 0} \cup \{\infty\}$, $\mathsf{eval} = \mathsf{id}$ and $d_{\mathsf{m}}^{\mathbb{L}M}(a.\sigma, b.\tau) = \max(d(a,b), d_{\mathsf{m}}^{\mathbb{L}M}(\sigma, \tau))$,

where $d : \Sigma \times \Sigma \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a hemimetric on labels. The limit-average distance used in *e.g.* [7] has $\mathbb{L} = (\mathbb{R}_{\geq 0} \cup \{\infty\})^{\mathbb{N}}$, the complete lattice of functions $\mathbb{N} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$, $\mathsf{eval}(\alpha) = \liminf_{j \in \mathbb{N}} \alpha(j)$ and $d_{\mathsf{m}}^{\mathbb{L}M}(a.\sigma, b.\tau)(j) = \frac{1}{j+1} d(a,b) + \frac{j}{j+1} d_{\mathsf{m}}^{\mathbb{L}M}(\sigma,\tau)$.

For the rest of this section, we fix a recursively specified trace distance. A WMTS $(S, s^0, \dashrightarrow, \longrightarrow)$ is *deterministic* if it holds for all $s \in S$, $s \overset{a_1}{\dashrightarrow} s_1$, $s \overset{a_2}{\dashrightarrow} s_2$ for which there is $a \in \Sigma$ with $td^{\mathbb{L}M}(a, a_1) \neq \top$ and $td^{\mathbb{L}M}(a, a_2) \neq \top$ that $a_1 = a_2$ and $s_1 = s_2$.

Definition 6. *The* lifted modal refinement distance $d_{\mathsf{m}}^{\mathbb{L}M} : S_1 \times S_2 \to \mathbb{L}$ *between the states of WMTS* $\mathcal{S}_1 = (S_1, s_1^0, \dashrightarrow_1, \longrightarrow_1)$, $\mathcal{S}_2 = (S_2, s_2^0, \dashrightarrow_2, \longrightarrow_2)$ *is defined to be the least fixed point to the equations*

$$d_{\mathsf{m}}^{\mathbb{L}M}(s_1, s_2) = \max \begin{cases} \sup\limits_{s_1 \overset{a_1}{\dashrightarrow}_1 t_1} \inf\limits_{s_2 \overset{a_2}{\dashrightarrow}_2 t_2} F(a_1, a_2, d_{\mathsf{m}}^{\mathbb{L}M}(t_1, t_2)), \\ \sup\limits_{s_2 \overset{a_2}{\longrightarrow}_2 t_2} \inf\limits_{s_1 \overset{a_1}{\longrightarrow}_1 t_1} F(a_1, a_2, d_{\mathsf{m}}^{\mathbb{L}M}(t_1, t_2)). \end{cases}$$

We let $d_{\mathsf{m}}^{\mathbb{L}M}(\mathcal{S}_1, \mathcal{S}_2) = d_{\mathsf{m}}^{\mathbb{L}M}(s_1^0, s_2^0)$. The *modal refinement distance* is $d_{\mathsf{m}} = \mathsf{eval} \circ d_{\mathsf{m}}^{\mathbb{L}M}$, and we write $\mathcal{S}_1 \leq_{\mathsf{m}}^{\varepsilon} \mathcal{S}_2$, for $\varepsilon \in \mathbb{R}_{\geq 0} \cup \{\infty\}$, if $d_{\mathsf{m}}^{\mathbb{L}M}(\mathcal{S}_1, \mathcal{S}_2) \leq \varepsilon$.

Proposition 1. *The modal refinement distance is a well-defined hemimetric, and* $\mathcal{S}_1 \leq_{\mathsf{m}} \mathcal{S}_2$ *implies* $\mathcal{S}_1 \leq_{\mathsf{m}}^0 \mathcal{S}_2$.

The *thorough refinement distance* between WMTS $\mathcal{S}_1$, $\mathcal{S}_2$ is

$$d_{\mathsf{t}}(\mathcal{S}_1, \mathcal{S}_2) = \sup_{\mathcal{I}_1 \in [\![\mathcal{S}_1]\!]} \inf_{\mathcal{I}_2 \in [\![\mathcal{S}_2]\!]} d_{\mathsf{m}}(\mathcal{I}_1, \mathcal{I}_2),$$

and we write $\mathcal{S}_1 \leq_{\mathsf{t}}^{\varepsilon} \mathcal{S}_2$, for $\varepsilon \in \mathbb{R}_{\geq 0} \cup \{\infty\}$, if $d_{\mathsf{t}}(\mathcal{S}_1, \mathcal{S}_2) \leq \varepsilon$. As for the modal distance, d_{t} is a hemimetric, and $\mathcal{S}_1 \leq_{\mathsf{t}} \mathcal{S}_2$ implies $\mathcal{S}_1 \leq_{\mathsf{t}}^0 \mathcal{S}_2$.

Theorem 1. *For all WMTS* $\mathcal{S}_1$, $\mathcal{S}_2$, $d_{\mathsf{t}}(\mathcal{S}_1, \mathcal{S}_2) \leq d_{\mathsf{m}}(\mathcal{S}_1, \mathcal{S}_2)$. *If* $\mathcal{S}_2$ *is deterministic, then* $d_{\mathsf{t}}(\mathcal{S}_1, \mathcal{S}_2) = d_{\mathsf{m}}(\mathcal{S}_1, \mathcal{S}_2)$.

3.3 Conjunction

Let $\owedge : \Sigma \times \Sigma \hookrightarrow \Sigma$ be a commutative partial *label conjunction* operator for which it holds, for all $b_1, b_2 \in \Sigma$, that there is $a \in \Sigma$ for which both $td^{\mathbb{L}M}(a, b_1) \neq \top$ and $td^{\mathbb{L}M}(a, b_2) \neq \top$ iff there exists $c \in \Sigma$ for which both $b_1 \owedge c$ and $b_2 \owedge c$ are defined. This is to relate determinism (left-hand side of the above) to a similar property for label conjunction which is needed in the proof of Theorem 2.

Additionally, we assume that $\owedge$ is greatest lower bound on labels, *i.e.*

- for all $a, b \in \Sigma$ with $a \owedge b$ defined, $a \owedge b \sqsubseteq a$ and $a \owedge b \sqsubseteq b$;
- for all $a, b, c \in \Sigma$ with $a \sqsubseteq b$ and $a \sqsubseteq c$, $b \owedge c$ is defined and $a \sqsubseteq b \owedge c$.

In the definition below, we denote by $\rho_B(\mathcal{S})$ the *pruning* of a WMTS $\mathcal{S} = (S, s^0, \dashrightarrow, \longrightarrow)$ with respect to the states in a ("bad") subset $B \subseteq S$, which is obtained as follows: Define a must-predecessor operator $\mathsf{pre} : 2^S \to 2^S$ by $\mathsf{pre}(S') = \{s \in S \mid \exists a \in \Sigma, s' \in S' : s \xrightarrow{a} s'\}$ and let pre^* be the reflexive, transitive closure of pre. Then $\rho_B(\mathcal{S})$ is defined if $s^0 \notin \mathsf{pre}^*(B)$, and in that case, $\rho_B(\mathcal{S}) = (S_\rho, s^0, \dashrightarrow_\rho, \longrightarrow_\rho)$ with $S_\rho = S \setminus \mathsf{pre}^*(B)$, $\dashrightarrow_\rho = \dashrightarrow \cap (S_\rho \times \Sigma \times S_\rho)$, and $\longrightarrow_\rho = \longrightarrow \cap (S_\rho \times \Sigma \times S_\rho)$.

Definition 7. *The* conjunction *of two WMTS* $\mathcal{S}_1 = (S_1, s_1^0, \dashrightarrow_1, \longrightarrow_1)$, $\mathcal{S}_2 = (S_2, s_2^0, \dashrightarrow_2, \longrightarrow_2)$ *is the WMTS* $\mathcal{S}_1 \wedge \mathcal{S}_2 = \rho_B(S_1 \times S_2, (s_1^0, s_2^0), \dashrightarrow, \longrightarrow)$ *given as follows (if it exists):*

$$\frac{s_1 \xrightarrow{a_1}_1 t_1 \quad s_2 \xrightarrow{a_2}_2 t_2 \quad a_1 \oslash a_2 \text{ defined}}{(s_1, s_2) \xrightarrow{a_1 \oslash a_2} (t_1, t_2)} \qquad \frac{s_1 \dashrightarrow_1 t_1 \quad s_2 \xrightarrow{a_2}_2 t_2 \quad a_1 \oslash a_2 \text{ defined}}{(s_1, s_2) \xrightarrow{a_1 \oslash a_2} (t_1, t_2)}$$

$$\frac{s_1 \dashrightarrow_1 t_1 \quad s_2 \dashrightarrow_2 t_2 \quad a_1 \oslash a_2 \text{ defined}}{(s_1, s_2) \dashrightarrow (t_1, t_2)}$$

$$\frac{s_1 \xrightarrow{a_1}_1 t_1 \quad \forall s_2 \dashrightarrow_2 t_2 : a_1 \oslash a_2 \text{ undef.}}{(s_1, s_2) \in B} \qquad \frac{s_2 \xrightarrow{a_2}_2 t_2 \quad \forall s_1 \dashrightarrow_1 t_1 : a_1 \oslash a_2 \text{ undef.}}{(s_1, s_2) \in B}$$

Note that conjunction of WMTS may give inconsistent states which need to be pruned away after. As seen in the last two SOS rules above, this is the case when one WMTS specifies a must-transition which the other WMTS cannot synchronize with. Here, the demand on implementations of the conjunction would be that they simultaneously *must* and *cannot* have a transition, which of course is unsatisfiable.

Theorem 2. *Let* $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$ *be WMTS.*

- *If* $\mathcal{S}_1 \wedge \mathcal{S}_2$ *is defined, then* $\mathcal{S}_1 \wedge \mathcal{S}_2 \leq_{\mathsf{m}} \mathcal{S}_1$ *and* $\mathcal{S}_1 \wedge \mathcal{S}_2 \leq_{\mathsf{m}} \mathcal{S}_2$.
- *If* $\mathcal{S}_1 \leq_{\mathsf{m}} \mathcal{S}_2$, $\mathcal{S}_1 \leq_{\mathsf{m}} \mathcal{S}_3$, *and* $\mathcal{S}_2$ *or* $\mathcal{S}_3$ *is deterministic, then* $\mathcal{S}_2 \wedge \mathcal{S}_3$ *is defined and* $\mathcal{S}_1 \leq_{\mathsf{m}} \mathcal{S}_2 \wedge \mathcal{S}_2$.

3.4 Structural Composition

Let $\oslash : \Sigma \times \Sigma \hookrightarrow \Sigma$ be a commutative partial *label composition* operator which specifies which labels can synchronize. Again we need to relate determinism to an analogous property for label composition, hence we require that it holds, for all $b_1, b_2 \in \Sigma$, that there is $a \in \Sigma$ for which both $d(a, b_1) \neq \top_{\mathbb{L}}$ and $d(a, b_2) \neq \top_{\mathbb{L}}$ iff there exists $c \in \Sigma$ for which both $b_1 \oslash c$ and $b_2 \oslash c$ are defined.

Additionally, we assume that there exists a function $P : \mathbb{L} \times \mathbb{L} \to \mathbb{L}$ which allows us to infer bounds on distances on synchronized labels. We assume that P is monotone in both coordinates, has $P(\bot_{\mathbb{L}}, \bot_{\mathbb{L}}) = \bot_{\mathbb{L}}$, $P(\alpha, \top_{\mathbb{L}}) = P(\top_{\mathbb{L}}, \alpha) = \top_{\mathbb{L}}$ for all $\alpha \in \mathbb{L}$, and that

$$F(a_1 \oslash a_2, b_1 \oslash b_2, P(\alpha_1, \alpha_2)) \sqsubseteq_{\mathbb{L}} P(F(a_1, b_1, \alpha_1), F(a_2, b_2, \alpha_2)) \tag{1}$$

for all $a_1, b_1, a_2, b_2 \in \Sigma$ and $\alpha_1, \alpha_2 \in \mathbb{L}$ for which $a_1 \oslash a_2$ and $b_1 \oslash b_2$ are defined. Hence $d(a_1 \oslash a_2, b_1 \oslash b_2) \sqsubseteq P(d(a_1, b_1), d(a_2, b_2))$ for all such $a_1, b_1, a_2, b_2 \in \Sigma$.

Intuitively, P gives a *uniform bound* on label composition: distances between composed labels can be bounded above using P and the individual labels' distances, and (1) ensures that this bound holds recursively.

Definition 8. *The* structural composition *of two WMTS* $\mathcal{S}_1 = (S_1, s_1^0, \dashrightarrow_1, \longrightarrow_1)$, $\mathcal{S}_2 = (S_2, s_2^0, \dashrightarrow_2, \longrightarrow_2)$ *is the WMTS* $\mathcal{S}_1 \| \mathcal{S}_2 = (S_1 \times S_2, (s_0^1, s_0^2), \dashrightarrow, \longrightarrow)$ *with transitions defined as follows:*

$$\frac{s_1 \overset{a_1}{\dashrightarrow}_1 t_1 \quad s_2 \overset{a_2}{\dashrightarrow}_2 t_2 \quad a_1 \oplus a_2 \text{ def.}}{(s_1, s_2) \overset{a_1 \oplus a_2}{\dashrightarrow} (t_1, t_2)} \qquad \frac{s_1 \overset{a_1}{\longrightarrow}_1 t_1 \quad s_2 \overset{a_2}{\longrightarrow}_2 t_2 \quad a_1 \oplus a_2 \text{ def.}}{(s_1, s_2) \overset{a_1 \oplus a_2}{\longrightarrow} (t_1, t_2)}$$

The next theorem shows that structural composition supports *quantitative independent implementability*: the distance between structural compositions can bounded above using P and the distances between the individual components.

Theorem 3. *For all WMTS* $\mathcal{S}_1$, $\mathcal{T}_1$, $\mathcal{S}_2$, $\mathcal{T}_2$ *with* $d_{\mathsf{m}}(\mathcal{S}_1 \| \mathcal{S}_2, \mathcal{T}_1 \| \mathcal{T}_2) \neq \top_{\mathbb{L}}$, *we have* $d_{\mathsf{m}}(\mathcal{S}_1 \| \mathcal{S}_2, \mathcal{T}_1 \| \mathcal{T}_2) \sqsubseteq_{\mathbb{L}} P(d_{\mathsf{m}}(\mathcal{S}_1, \mathcal{T}_1), d_{\mathsf{m}}(\mathcal{S}_2, \mathcal{T}_2))$.

Acknowledgment. This survey paper presents research which we have conducted with a number of coauthors; in alphabetical order, these are Sebastian S. Bauer, Nikola Beneš, Line Juhl, Jan Křetínský, Mikael H. Møller, Jiří Srba, and Claus Thrane. We acknowledge their cooperation in this work; any errors in this presentation are, however, our own.

References

1. Bauer, S.S., Fahrenberg, U., Juhl, L., Larsen, K.G., Legay, A., Thrane, C.: Quantitative refinement for weighted modal transition systems. In: Murlak, F., Sankowski, P. (eds.) MFCS 2011. LNCS, vol. 6907, pp. 60–71. Springer, Heidelberg (2011)
2. Bauer, S.S., Fahrenberg, U., Legay, A., Thrane, C.: General quantitative specification theories with modalities. In: Hirsch, E.A., Karhumäki, J., Lepistö, A., Prilutskii, M. (eds.) CSR 2012. LNCS, vol. 7353, pp. 18–30. Springer, Heidelberg (2012)
3. Bauer, S.S., Juhl, L., Larsen, K.G., Legay, A., Srba, J.: Extending modal transition systems with structured labels. Mathematical Structures in Computer Science 22(4), 581–617 (2012)
4. Beneš, N., Křetínský, J., Larsen, K.G., Møller, M.H., Srba, J.: Parametric modal transition systems. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 275–289. Springer, Heidelberg (2011)
5. Beneš, N., Křetínský, J.: Process algebra for modal transition systemses. In: MEMICS. OASICS, vol. 16, pp. 9–18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2010)
6. Boudol, G., Larsen, K.G.: Graphical versus logical specifications. In: Arnold, A. (ed.) CAAP 1990. LNCS, vol. 431, pp. 57–71. Springer, Heidelberg (1990)
7. Černý, P., Henzinger, T.A., Radhakrishna, A.: Simulation distances. Theor. Comput. Sci. 413(1), 21–35 (2012)
8. de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: Model checking discounted temporal properties. Theor. Comput. Sci. 345(1), 139–170 (2005)

9. de Alfaro, L., Faella, M., Stoelinga, M.: Linear and branching system metrics. IEEE Trans. Software Eng. 35(2), 258–273 (2009)
10. Fahrenberg, U., Legay, A., Thrane, C.: The quantitative linear-time–branching-time spectrum. In: FSTTCS. LIPIcs, vol. 13, pp. 103–114. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
11. Fahrenberg, U., Legay, A., Traonouez, L.-M.: Specification theories for probabilistic and real-time systems. In: Bensalem, S., Lakhnech, Y., Legay, A. (eds.) FPS 2014 (Sifakis Festschrift). LNCS, vol. 8415, pp. 98–117. Springer, Heidelberg (2014)
12. Fahrenberg, U., Thrane, C.R., Larsen, K.G.: Distances for weighted transition systems: Games and properties. In: QAPL. Electr. Proc. Theor. Comput. Sci., vol. 57, pp. 134–147 (2011)
13. Fecher, H., Schmidt, H.: Comparing disjunctive modal transition systems with an one-selecting variant. J. Logic Alg. Program. 77(1-2), 20–39 (2008)
14. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001)
15. Graf, S., Sifakis, J.: A logic for the description of non-deterministic programs and their properties. Inf. Control 68(1-3), 254–270 (1986)
16. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 113–131. Springer, Heidelberg (2008)
17. Holmström, S.: A refinement calculus for specifications in Hennessy-Milner logic with recursion. Formal Asp. Comput. 1(3), 242–272 (1989)
18. Huth, M., Jagadeesan, R., Schmidt, D.A.: Modal transition systems: A foundation for three-valued program analysis. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 155–169. Springer, Heidelberg (2001)
19. Juhl, L., Larsen, K.G., Srba, J.: Modal transition systems with weight intervals. J. Log. Algebr. Program. 81(4), 408–421 (2012)
20. Larsen, K.G.: A context dependent equivalence between processes. Theor. Comput. Sci. 49, 184–215 (1987)
21. Larsen, K.G.: Modal specifications. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 232–246. Springer, Heidelberg (1990)
22. Larsen, K.G., Nyman, U., Wąsowski, A.: On modal refinement and consistency. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 105–119. Springer, Heidelberg (2007)
23. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS, pp. 203–210. IEEE Computer Society (1988)
24. Larsen, K.G., Xinxin, L.: Equation solving using modal transition systems. In: LICS, pp. 108–117. IEEE Computer Society (1990)
25. Mardare, R., Policriti, A.: A complete axiomatic system for a process-based spatial logic. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 491–502. Springer, Heidelberg (2008)
26. Nanz, S., Nielson, F., Riis Nielson, H.: Modal abstractions of concurrent behaviour. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 159–173. Springer, Heidelberg (2008)
27. Raclet, J.-B., Badouel, E., Benveniste, A., Caillaud, B., Passerone, R.: Why are modalities good for interface theories? In: ACSD, pp. 119–127. IEEE (2009)
28. Uchitel, S., Chechik, M.: Merging partial behavioural models. In: FSE, pp. 43–52. ACM (2004)

Specification Theories for Probabilistic and Real-Time Systems

Uli Fahrenberg, Axel Legay, and Louis-Marie Traonouez

Inria/IRISA, Rennes, France

Abstract. We survey extensions of modal transition systems to specification theories for probabilistic and timed systems.

1 Introduction

Many modern systems are big and complex assemblies of numerous components called implementations. The implementations are often designed by independent teams, working under a common agreement on what the specification of each implementation should be.

Over the past, one has agreed that any good specification theory should be equipped with a satisfaction relation (to decide whether an implementation satisfies a specification), a consistency check (to decide whether the specification admits an implementation), a refinement (to compare specifications in terms of inclusion of sets of implementations), logical composition (to compute the intersection of sets of implementations), and structural composition (to combine specifications).

The design of "good" specification theories has been the subject of intensive study, most of them for the case where implementations are represented by transition systems. In this paper, we survey two seminal works on extending specification theories to both probabilistic and timed systems.

Specification Theory for Probabilistic Systems. We consider implementations represented by *probabilistic automata* (PA). Probabilistic automata constitute a mathematical framework for the description and analysis of nondeterministic probabilistic systems. They have been developed by Segala [30] to model and analyze asynchronous, concurrent systems with discrete probabilistic choice in a formal and precise way. PA are akin to Markov decision processes (MDP). A detailed comparison with models such as MDP, as well as generative and reactive probabilistic transition systems is given in [29]. PA are recognized as an adequate formalism for randomized distributed algorithms and fault tolerant systems. They are used as semantic model for formalisms such as probabilistic process algebra [28] and a probabilistic variant of Harel's statecharts [20]. An input-output version of PA is the basis of PIOA and variants thereof [4, 7]. PA have been enriched with notions such as weak and strong (bi)simulations [30], decision algorithms for these notions [6] and a statistical testing theory [8].

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 98–117, 2014.

In [14], we have introduced *abstract probabilistic automata* (APA) as a specification theory for PA. APA aims at model reduction by collapsing sets of concrete states to abstract states, *e.g.* by partitioning the concrete state space. This paper presents a three-valued abstraction of PA. The main design principle of our model is to abstract sets of distributions by constraint functions. This generalizes earlier work on interval-based abstraction of probabilistic systems [19,21,22]. To abstract from action transitions, we introduce *may*- and *must*-modalities in the spirit of modal transition systems [24, 26]. If all states in a partition p have a must-transition on action a to some state in partition p', the abstraction yields a must-transition between p and p'. If some of the p-states have no such transition while others do, it gives rise to a may-transition between p and p'. In this paper we will summarize main results on APA. We will also show how the model can be used as a specification theory for PA.

Specification Theory for Timed Systems. In [9,10], we represent both specifications and implementations by timed input/output transition systems [23], *i.e.* timed transitions systems whose sets of discrete transitions are split into Input and Output transitions. In contrast to [11] and [23], we distinguish between implementations and specifications by adding conditions on the models. This is done by assuming that the former have fixed timing behavior and they can always advance either by producing an output or delaying. In this paper, we summarize the specification theory for timed systems of [9,10]. We also show how a game-based methodology can be used to decide whether a specification is consistent, *i.e.* whether it has at least one implementation. The latter reduces to deciding existence of a strategy that despite the behavior of the environment will avoid states that cannot possibly satisfy the implementation requirements. Finally, we show that the approach extends to a robust theory for timed systems.

2 Abstract Probabilistic Automata

For any finite set S, $\mathsf{Dist}(S)$ denotes the set of all discrete probability distributions over S (*i.e.* all mappings $\mu : S \to [0,1]$ with $\sum_{s\in S} \mu(s) = 1$). $C(S)$ denotes a set of *probability constraints* together with a mapping $\mathsf{Sat} : C(S) \to 2^{\mathsf{Dist}(S)}$.

2.1 Abstract Probabilistic Automata

A *probabilistic automaton* (PA) [30] is a tuple (S, A, L, AP, V, s^0), where S is a finite set of states with the initial state $s^0 \in S$, A is a finite set of actions, $L : S \times A \times \mathsf{Dist}(S) \to \{\bot, \top\}$ is a transition relation, AP is a finite set of atomic propositions and $V : S \to 2^{AP}$ is a state-labeling function.

PA were introduced in [30] as a model suitable for systems which encompass both non-deterministic and stochastic behavior. Hence they generalize both LTS (non-determinism) and Markov chains (stochasticity). The notation $L : S \times A \times \mathsf{Dist}(S) \to \{\bot, \top\}$ instead of $L \subseteq S \times A \times \mathsf{Dist}(S)$ is traditional and will be convenient below. The left part of Fig. 1 shows an example of a PA.

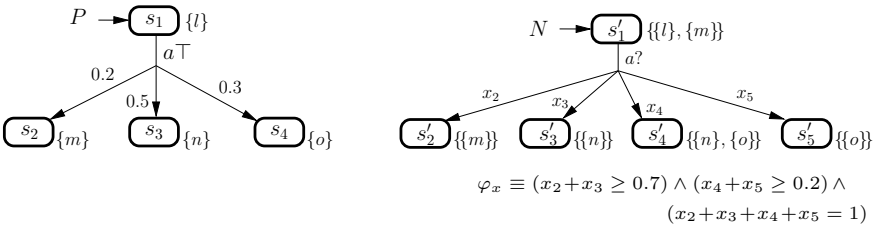


Fig. 1. Example PA P (left) and APA N with $P \models N$ (right)

As specifications of PA we use *abstract probabilistic automata* [14]. These can be seen as a common generalization of modal transition systems [16] and *constraint Markov chains* [3].

Definition 1. *An* abstract probabilistic automaton *(APA) [14] is a tuple* (S, A, L, AP, V, s^0)*, where* S *is a finite set of states,* $s^0 \in S$ *is the initial state,* A *is a finite set of actions, and* AP *is a finite set of atomic propositions.* $L : S \times A \times C(S) \to \{\bot, ?, \top\}$ *is a three-valued distribution-constraint function, and* $V : S \to 2^{2^{AP}}$ *maps each state in* S *to a set of admissible labelings.*

It is natural to think that distribution constraints should be *intervals* on transition probabilities as *e.g.* in *interval Markov chains* [19]. However, we will later see that natural constructions on APA such as conjunction or structural composition make it necessary to allow other, more expressive types of constraints.

The following notation will be convenient later: for $s, t \in S$ and $a \in A$, let $\mathsf{succ}_{s,a}(t) = \{s' \in S \mid V(s') = V(t), \exists \varphi \in C(S), \mu \in \mathsf{Sat}(\varphi) : L(s, a, \varphi) \neq \bot, \mu(s') > 0\}$ be the set of potential a-successors of s that have $V(t)$ as their valuation. Remark that when N is deterministic, we have $|\mathsf{succ}_{s,a}(v)| \leq 1$ for all s, t, a.

An APA is *deterministic* if (1) there is at most one outgoing transition for each action in all states and (2) two states with overlapping atomic propositions can never be reached with the same transition. An APA is in *single valuation normal form* (SVNF) if the valuation function V assigns at most one valuation to all states, *i.e.* $\forall s \in S, |V(s)| \leq 1$. From [14], we know that every APA can be turned into an APA in SVNF with the same set of implementations, and that this construction preserves determinism.

Note that every PA is an APA in SVNF where all constraints represent a single distribution. As a consequence, all the definitions we present for APA in the following can be directly extended to PA.

Let S and S' be non-empty sets and $\mu \in \mathsf{Dist}(S)$, $\mu' \in \mathsf{Dist}(S')$. We say that μ is *simulated* by μ' with respect to a relation $R \subseteq S \times S'$ and a *correspondence function* $\delta : S \to (S' \to [0, 1])$ if

1. for all $s \in S$ with $\mu(s) > 0$, $\delta(s)$ is a distribution on S',
2. for all $s' \in S'$, $\sum_{s \in S} \mu(s) \cdot \delta(s)(s') = \mu'(s')$, and
3. whenever $\delta(s)(s') > 0$, then $(s, s') \in R$.

We write $\mu \Subset_R^\delta \mu'$ if μ is simulated by μ' w.r.t. R and δ, $\mu \Subset_R \mu'$ if there exists δ with $\mu \Subset_R^\delta \mu'$, and $\mu \Subset^\delta \mu'$ for $\mu \Subset_{S\times S'}^\delta \mu'$.

For $\varphi \in C(S)$, $\varphi' \in C(S')$ and $R \subseteq S \times S'$, we write $\varphi \Subset_R \varphi'$ if $\forall \mu \in \mathsf{Sat}(\varphi) : \exists \mu' \in \mathsf{Sat}(\varphi') : \mu \Subset_R \mu'$.

Definition 2. *Let $N_1 = (S_1, A, L_1, AP, V_1, s_0^1)$ and $N_2 = (S_2, A, L_2, AP, V_2, s_0^2)$ be APA. A relation $R \subseteq S_1 \times S_2$ is a (weak)* modal refinement *if, for all $(s_1, s_2) \in R$, we have $V_1(s_1) \subseteq V_2(s_2)$ and*

1. *$\forall a \in A$, $\forall \varphi_2 \in C(S_2)$, if $L_2(s_2, a, \varphi_2) = \top$, then $\exists \varphi_1 \in C(S_1)$ such that $L_1(s_1, a, \varphi_1) = \top$ and $\varphi_1 \Subset_R \varphi_2$,*
2. *$\forall a \in A$, $\forall \varphi_1 \in C(S_1)$, if $L_1(s_1, a, \varphi_1) \neq \bot$, then $\exists \varphi_2 \in C(S_2)$ such that $L_2(s_2, a, \varphi_2) \neq \bot$ and $\varphi_1 \Subset_R \varphi_2$.*

We say that N_1 refines N_2 and write $N_1 \leq_{\mathsf{m}} N_2$, if there is a modal refinement relation $R \subseteq S_1 \times S_2$ with $(s_0^1, s_0^2) \in R$.

2.2 Conjunction

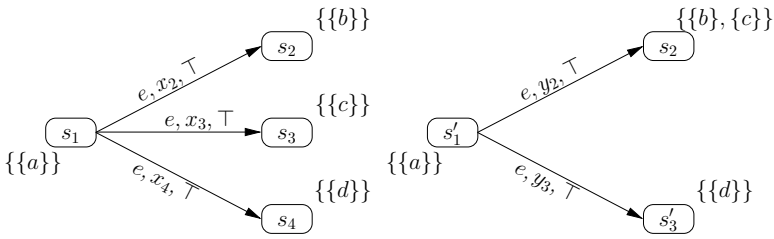
Definition 3. *Let $N = (S, A, L, AP, V, s_0)$, $N' = (S', A, L', AP, V', s_0')$ be deterministic APA which share actions and propositions. The* conjunction *of N and N' is the APA $N \wedge N' = (S \times S', A, \tilde{L}, AP, \tilde{V}, (s_0, s_0'))$, with $\tilde{V}((s, s')) = V(s) \cap V'(s')$ and $\tilde{L}$ defined as follows, for all $a \in A$ and $(s, s') \in S \times S'$:*

- *If there exists $\varphi \in C(S)$ such that $L(s, a, \varphi) = \top$ and for all $\varphi' \in C(S')$, we have $L'(s', a, \varphi') = \bot$, or if there exists $\varphi' \in C(S')$ such that $L'(s', a, \varphi') = \top$ and for all $\varphi \in C(S)$, we have $L(s, a, \varphi) = \bot$, then $\tilde{L}((s, s'), a, \mathit{false}) = \top$.*
- *Else, if either for all $\varphi \in C(S)$, we have $L(s, a, \varphi) = \bot$ or for all $\varphi' \in C(S')$, we have $L'(s', a, \varphi') = \bot$, then for all $\tilde{\varphi} \in C(S \times S')$, $\tilde{L}((s, s'), a, \tilde{\varphi}) = \bot$.*
- *Otherwise, for all $\varphi \in C(S)$ and $\varphi' \in C(S')$ such that $L(s, a, \varphi) \neq \bot$ and $L'(s', a, \varphi') \neq \bot$, define $\tilde{L}((s, s'), a, \tilde{\varphi}) = L(s, a, \varphi) \sqcup L'(s', a, \varphi')$ with $\tilde{\varphi}$ the constraint in $C(S \times S')$ such that $\tilde{\mu} \in \mathsf{Sat}(\tilde{\varphi})$ iff the distribution $t \to \sum_{t' \in S'} \tilde{\mu}((t, t')) \in \mathsf{Sat}(\varphi)$, and the distribution $t' \to \sum_{t \in S} \tilde{\mu}((t, t')) \in \mathsf{Sat}(\varphi')$.*
- *Finally, for all other $\tilde{\varphi}' \in C(S \times S')$, let $\tilde{L}((s, s'), a, \tilde{\varphi}') = \bot$.*

Observe that the conjunction of two deterministic APA is again deterministic. By the following theorem, conjunction is indeed the greatest lower bound.

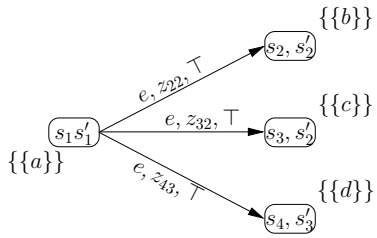
Theorem 1. *Let N_1, N_2, N_3 be deterministic APA. We have $N_1 \wedge N_2 \leq_{\mathsf{m}} N_1$ and $N_1 \wedge N_2 \leq_{\mathsf{m}} N_2$, and if $N_3 \leq_{\mathsf{m}} N_1$ and $N_3 \leq_{\mathsf{m}} N_2$, then also $N_3 \leq_{\mathsf{m}} N_1 \wedge N_2$.*

We finish this section with an example in which the conjunction of two APA with interval constraints is not an APA with interval constraints; hence interval constraints are not closed under conjunction. For the two APA N, N' in Fig. 2, which employ only interval constraints, the conjunction $N \wedge N'$ creates a constraint $0.4 \leq z_{22} + z_{32} \leq 0.8$ which is not an interval.



$\varphi_x : 0 \le x_2 \le \frac{1}{2} \wedge 0.2 \le x_3 \le 0.7 \wedge 0 \le x_4 \le \frac{1}{2}$

$\varphi_y : 0.4 \le y_2 \le 0.8 \wedge 0 \le y_3 \le 1$



$\varphi_z : 0 \le z_{22} \le 0.5 \wedge 0.2 \le z_{32} \le 0.7 \wedge 0 \le z_{43} \le 0.5 \wedge 0.4 \le z_{22} + z_{32} \le 0.8$

Fig. 2. Two APA with interval constraints (top) and their conjunction (bottom)

2.3 Structural Composition

Definition 4. *Let $N = (S, A, L, AP, V, s_0)$, $N' = (S', A, L', AP', V', s'_0)$ be APA with $AP \cap AP' = \emptyset$. The* structural composition *of N and N' is $N \| N' = (S \times S', A, \tilde{L}, AP \cup AP', \tilde{V}, (s_0, s'_0))$, with $\tilde{V}((s, s')) = \{B \cup B' \mid B \in V(s), B' \in V'(s')\}$ and $\tilde{L}$ defined as follows, for all $(s, s') \in S \times S'$ and $a \in A$:*

- *For all $\varphi \in C(S)$, $\varphi' \in C(S')$ for which $L(s, a, \varphi) \neq \bot$ and $L'(s', a, \varphi') \neq \bot$, let $\tilde{\varphi} \in C(S \times S')$ be a constraint for which $\tilde{\mu} \in \mathsf{Sat}(\tilde{\varphi})$ iff $\exists \mu \in \mathsf{Sat}(\varphi), \mu' \in \mathsf{Sat}(\varphi') : \forall t \in S, t' \in S' : \tilde{\mu}(t, t') = \mu(t)\mu(t')$. Now if $L(s, a, \varphi) = L'(s', a, \varphi') = \top$, let $\tilde{L}((s, s'), a, \tilde{\varphi}) = \top$, otherwise let $\tilde{L}((s, s'), a, \tilde{\varphi}) = ?$.*
- *If $L(s, a, \varphi) = \bot$ for all $\varphi \in C(S)$ or $L'(s', a, \varphi') = \bot$ for all $\varphi' \in C(S')$, let $\tilde{L}((s, s'), \alpha, \tilde{\varphi}) = \bot$ for all $\tilde{\varphi} \in C(S \times S')$.*

By the next theorem, structural composition respects refinement (or, in other words, refinement is a pre-congruence with respect to $\|$). This entails *independent implementability*: any composition of implementations of N_1 and N_2 is automatically an implementation of $N_1 \| N_2$.

Theorem 2. *For all APA N_1, N'_1, N_2, N'_2, $N_1 \le_{\mathsf{m}} N'_1$ and $N_2 \le_{\mathsf{m}} N'_2$ imply $N_1 \| N_2 \le_{\mathsf{m}} N'_1 \| N'_2$.*

It can be shown that structural composition of APA with interval constraints may yield APA with *polynomial* constraints, *e.g.* of the form $k_1 \le x_1 x_2 + x_2 x_3 \le$

k_2. APA with polynomial constraints are, however, closed under both structural composition and conjunction. The tool APAC [15] implements most of APA operations, for APA with polynomial constraints, and uses the Z3 solver [12] for algorithms on polynomial constraints.

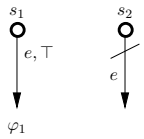
2.4 Over-Approximating Difference

We now turn to computing differences of APA. For APA N_1, N_2, we are interested in computing an APA representation of their implementation difference $[\![N_1]\!] \setminus [\![N_1]\!]$. This is based on work presented in [13].

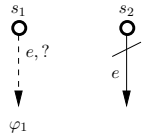
Let $N_1 = (S_1, A, L_1, AP, V_1, \{s_0^1\})$, $N_2 = (S_2, A, L_2, AP, V_2, \{s_0^2\})$ be deterministic APA in SVNF. Because N_1 and N_2 are deterministic, we know that the difference $[\![N_1]\!] \setminus [\![N_2]\!]$ is non-empty iff $N_1 \not\leq_{\mathsf{m}} N_2$. So let us assume that $N_1 \not\leq_{\mathsf{m}} N_2$, and let R be a maximal refinement relation between N_1 and N_2. Since $N_1 \not\leq_{\mathsf{m}} N_2$, we know that $(s_0^1, s_0^2) \notin R$. Given $(s_1, s_2) \in S_1 \times S_2$, we can distinguish between the following cases:

1. $(s_1, s_2) \in R$,
2. $V_1(s_1) \neq V_2(s_2)$, or
3. $(s_1, s_2) \notin R$ and $V_1(s_1) = V_2(s_2)$, and

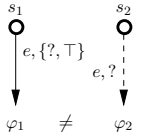
(a) there exists $e \in A$ and $\varphi_1 \in C(S_1)$ such that $L_1(s_1, e, \varphi_1) = \top$ and $\forall \varphi_2 \in C(S_2) : L_2(s_2, e, \varphi_2) = \bot$,



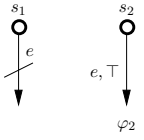
(b) there exists $e \in A$ and $\varphi_1 \in C(S_1)$ such that $L_1(s_1, e, \varphi_1) = ?$ and $\forall \varphi_2 \in C(S_2) : L_2(s_2, e, \varphi_2) = \bot$,



(c) there exists $e \in A$ and $\varphi_1 \in C(S_1)$ such that $L_1(s_1, e, \varphi_1) \geq ?$ and $\exists \varphi_2 \in C(S_2) : L_2(s_2, e, \varphi_2) = ?, \exists \mu \in \mathsf{Sat}(\varphi_1)$ such that $\forall \mu' \in Sat(\varphi_2) : \mu \not\in_R \mu'$,



(d) there exists $e \in A$ and $\varphi_2 \in C(S_2)$ such that $L_2(s_2, e, \varphi_2) = \top$ and $\forall \varphi_1 \in C(S_1) : L_1(s_1, e, \varphi_1) = \bot$,



(e) there exists $e \in A$ and $\varphi_2 \in C(S_2)$ such that $L_2(s_2, e, \varphi_2) = \top$ and $\exists \varphi_1 \in C(S_1) : L_1(s_1, e, \varphi_1) = ?$,

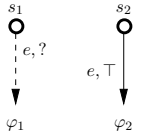
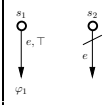
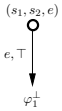
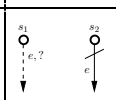
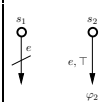
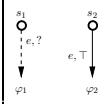
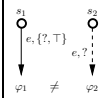
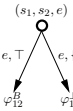
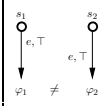
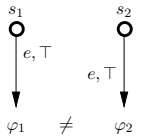


Table 1. Definition of the transition function L in $N_1 \setminus^* N_2$

$e \in$	N_1, N_2	$N_1 \setminus^* N_2$	Definition of L
$B_a(s_1, s_2)$			For all $a \neq e \in A$ and $\varphi \in C(S_1)$ such that $L_1(s_1, a, \varphi) \neq \bot$, let $L((s_1, s_2, e), a, \varphi^\bot) = L_1(s_1, a, \varphi)$. In addition, let $L((s_1, s_2, e), e, \varphi_1^\bot) = \top$. For all other $b \in A$ and $\varphi \in C(S)$, let $L((s_1, s_2, e), b, \varphi) = \bot$.
$B_b(s_1, s_2)$			
$B_d(s_1, s_2)$			For all $a \in A$ and $\varphi \in C(S_1)$ such that $L_1(s_1, a, \varphi) \neq \bot$, let $L((s_1, s_2, e), a, \varphi^\bot) = L_1(s_1, a, \varphi)$. For all other $b \in A$ and $\varphi \in C(S)$, let $L((s_1, s_2, e), b, \varphi) = \bot$.
$B_e(s_1, s_2)$			For all $a \neq e \in A$ and $\varphi \in C(S_1)$ such that $L_1(s_1, a, \varphi) \neq \bot$, let $L((s_1, s_2, e), a, \varphi^\bot) = L_1(s_1, a, \varphi)$. In addition, let $L((s_1, s_2, e), e, \varphi_{12}^B) = ?$. For all other $b \in A$ and $\varphi \in C(S)$, let $L((s_1, s_2, e), b, \varphi) = \bot$.
$B_c(s_1, s_2)$			For all $a \in A$ and $\varphi \in C(S_1)$ such that $L_1(s_1, a, \varphi) \neq \bot$ (including e and φ_1), let $L((s_1, s_2, e), a, \varphi^\bot) = L_1(s_1, a, \varphi)$. In addition, let $L((s_1, s_2, e), e, \varphi_{12}^B) = \top$. For all other $b \in A$ and $\varphi \in C(S)$, let $L((s_1, s_2, e), b, \varphi) = \bot$.
$B_f(s_1, s_2)$			

(f) there exists $e \in A$ and $\varphi_2 \in C(S_2)$ such that $L_2(s_2, e, \varphi_2) = \top$, $\exists \varphi_1 \in C(S_1) : L_1(s_1, e, \varphi_1) = \top$ and $\exists \mu \in \mathsf{Sat}(\varphi_1)$ such that $\forall \mu' \in \mathsf{Sat}(\varphi_2) : \mu \not\leqslant_R \mu'$.



Remark that due to determinism and SVNF, cases 1, 2 and 3 cannot happen at the same time. Moreover, although the cases in 3 can happen simultaneously, they cannot be triggered by the same action. We define the following sets.

Given a pair of states (s_1, s_2), let $B_a(s_1, s_2)$ be the set of actions in A such that case 3.a above holds. If there is no such action, then $B_a(s_1, s_2) = \emptyset$. Similarly, we define $B_b(s_1, s_2), B_c(s_1, s_2), B_d(s_1, s_2), B_e(s_1, s_2)$ and $B_f(s_1, s_2)$ to be the sets of actions such that case 3.b, c, d, e and 3.f holds, respectively. Given a set $X \subseteq \{a, b, c, d, e, f\}$, let $B_X(s_1, s_2) = \cup_{x \in X} B_x(s_1, s_2)$. In addition, let $B(s_1, s_2) = B_{\{a,b,c,d,e,f\}}(s_1, s_2)$.

Definition 5. *Let $N_1 = (S_1, A, L_1, AP, V_1, \{s_0^1\})$, $N_2 = (S_2, A, L_2, AP, V_2, \{s_0^2\})$ be deterministic APA in SVNF. If $N_1 \leq_{\mathsf{m}} N_2$, then $N_1 \setminus^* N_2$ is undefined; if $V_1(s_0^1) \neq V_2(s_0^2)$, we let $N_1 \setminus^* N_2 = N_1$. Otherwise, define $N_1 \setminus^* N_2 =$*

(S, A, L, AP, V, S_0), *where* $S = S_1 \times (S_2 \cup \{\bot\}) \times (A \cup \{\varepsilon\})$, $V(s_1, s_2, a) = V(s_1)$, *and* $S_0 = \{(s_0^1, s_0^2, f) \mid f \in B(s_0^1, s_0^2)\}$. L *is defined as follows:*

- *If* $s_2 = \bot$ *or* $e = \varepsilon$ *or* (s_1, s_2) *in case* 1 *or* 2, *then for all* $a \in A$ *and* $\varphi \in C(S_1)$ *such that* $L_1(s_1, a, \varphi) \neq \bot$, *let* $L((s_1, s_2, e), a, \varphi^\bot) = L_1(s_1, a, \varphi)$, *with* $\varphi^\bot$ *defined below. For all other* $b \in A$ *and* $\varphi \in C(S)$, *let* $L((s_1, s_2, e), b, \varphi) = \bot$.
- *Else, we have* (s_1, s_2) *in case* 3 *and* $B(s_1, s_2) \neq \emptyset$ *by construction. The definition of* L *is given in Table 1, with the constraints* $\varphi^\bot$ *and* φ_{12}^B *defined below.*

For $\varphi \in C(S_1)$, $\varphi^\bot \in C(S)$ *is defined as follows:* $\mu \in \mathsf{Sat}(\varphi^\bot)$ *iff* $\forall s_1 \in S_1$, $\forall s_2 \neq \bot$, $\forall b \neq \varepsilon$, $\mu(s_1, s_2, b) = 0$ *and the distribution* $s_1 \mapsto \mu(s_1, \bot, \varepsilon) \in \mathsf{Sat}(\varphi)$.

For a state $(s_1, s_2, e) \in S$ *with* $s_2 \neq \bot$, $e \neq \varepsilon$ *and two constraints* $\varphi_1 \in C(S_1)$, $\varphi_2 \in C(S_2)$ *such that* $L_1(s_1, e, \varphi_1) \neq \bot$ *and* $L_2(s_2, e, \varphi_2) \neq \bot$, *the constraint* $\varphi_{12}^B \in C(S)$ *is defined as follows:* $\mu \in \mathsf{Sat}(\varphi_{12}^B)$ *iff*

1. *for all* $(s_1', s_2', c) \in S$ *with* $\mu(s_1', s_2', c) > 0$, $c \in B(s_1', s_2') \cup \{\varepsilon\}$ *and either* $\mathsf{succ}_{s_2, e}(s_1') = \emptyset$ *and* $s_2' = \bot$, *or* $s_2' = \mathsf{succ}_{s_2, e}(s_1')$,
2. *the distribution* $s_1' \mapsto \sum_{c \in A \cup \{\varepsilon\}, s_2' \in S_2 \cup \{\bot\}} \mu(s_1', s_2', c) \in \mathsf{Sat}(\varphi_1)$, *and*
3. *either (a) there exists* $(s_1', \bot, c)$ *such that* $\mu(s_1', \bot, c) > 0$, *or (b) the distribution* $s_2' \mapsto \sum_{c \in A \cup \{\varepsilon\}, s_1' \in S_1} \mu(s_1', s_2', c) \notin \mathsf{Sat}(\varphi_2)$, *or (c) there exists* $s_1' \in S_1$, $s_2' \in S_2$ *and* $c \neq \varepsilon$ *such that* $\mu(s_1', s_2', c) > 0$.

Informally, distributions in φ_{12}^B must (1) follow the corresponding execution in N_1 and N_2 if possible, (2) satisfy φ_1 and (3) either (a) reach a state in N_1 that cannot be matched in N_2 or (b) break the constraint φ_2, or (c) report breaking the relation to at least one successor state.

The following theorem shows that the $*$-difference over-approximates the real difference.

Theorem 3. *For all deterministic APA* N_1 *and* N_2 *in SVNF such that* $N_1 \not\leq_{\mathsf{m}} N_2$, *we have* $[\![N_1]\!] \setminus [\![N_2]\!] \subseteq [\![N_1 \setminus^* N_2]\!]$.

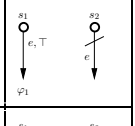
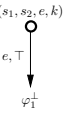
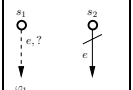
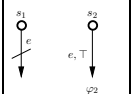
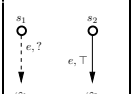
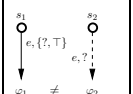
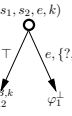
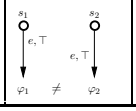
2.5 Under-Approximating Differences

Instead of the over-approximating difference $N_1 \setminus^* N_2$, we can also compute under-approximating differences. Intuitively, this is done by unfolding the APA N_1, N_2 up to some level K and then compute the difference of unfoldings:

Definition 6. *Let* $N_1 = (S_1, A, L_1, AP, V_1, \{s_0^1\})$, $N_2 = (S_2, A, L_2, AP, V_2, \{s_0^2\})$ *be deterministic APA in SVNF and* $K \in \mathbb{N}$. *If* $N_1 \leq_{\mathsf{m}} N_2$, *then* $N_1 \setminus^K N_2$ *is undefined; if* $V_1(s_0^1) \neq V_2(s_0^2)$, *we let* $N_1 \setminus^K N_2 = N_1$. *Otherwise, define* $N_1 \setminus^K N_2 = (S, A, L, AP, V, S_0^K)$, *where* $S = S_1 \times (S_2 \cup \{\bot\}) \times (A \cup \{\varepsilon\}) \times \{1, \ldots, K\}$, $V(s_1, s_2, a, k) = V(s_1)$, *and* $S_0^K = \{(s_0^1, s_0^2, f, K) \mid f \in B(s_0^1, s_0^2)\}$. L *is defined as follows:*

- *If* $s_2 = \bot$ *or* $e = \varepsilon$ *or* (s_1, s_2) *in case* 1 *or* 2, *then for all* $a \in A$ *and* $\varphi \in C(S_1)$ *such that* $L_1(s_1, a, \varphi) \neq \bot$, *let* $L((s_1, s_2, e, k), a, \varphi^\bot) = L_1(s_1, a, \varphi)$, *with* $\varphi^\bot$ *defined below. For all other* $b \in A$ *and* $\varphi \in C(S)$, *let* $L((s_1, s_2, e, k), b, \varphi) = \bot$.

Table 2. Definition of the transition function L in $N_1 \setminus^K N_2$

$e \in$	N_1, N_2	$N_1 \setminus^K N_2$	Definition of L
$B_a(s_1, s_2)$			For all $a \neq e \in A$ and $\varphi \in C(S_1)$ such that $L_1(s_1, a, \varphi) \neq \bot$, let $L((s_1, s_2, e, k), a, \varphi^\bot) = L_1(s_1, a, \varphi)$. In addition, let $L((s_1, s_2, e, k), e, \varphi_1^\bot) = \top$. For all other $b \in A$ and $\varphi \in C(S)$, let $L((s_1, s_2, e, k), b, \varphi) = \bot$.
$B_b(s_1, s_2)$			
$B_d(s_1, s_2)$			For all $a \in A$ and $\varphi \in C(S_1)$ such that $L_1(s_1, a, \varphi) \neq \bot$, let $L((s_1, s_2, e, k), a, \varphi^\bot) = L_1(s_1, a, \varphi)$. For all other $b \in A$ and $\varphi \in C(S)$, let $L((s_1, s_2, e, k), b, \varphi) = \bot$.
$B_e(s_1, s_2)$			For all $a \neq e \in A$ and $\varphi \in C(S_1)$ such that $L_1(s_1, a, \varphi) \neq \bot$, let $L((s_1, s_2, e, k), a, \varphi^\bot) = L_1(s_1, a, \varphi)$. In addition, let $L((s_1, s_2, e, k), e, \varphi_{12}^{B,k}) = ?$. For all other $b \in A$ and $\varphi \in C(S)$, let $L((s_1, s_2, e, k), b, \varphi) = \bot$.
$B_c(s_1, s_2)$			For all $a \in A$ and $\varphi \in C(S_1)$ such that $L_1(s_1, a, \varphi) \neq \bot$ (including e and φ_1), let $L((s_1, s_2, e, k), a, \varphi^\bot) = L_1(s_1, a, \varphi)$. In addition, let $L((s_1, s_2, e, k), e, \varphi_{12}^{B,k}) = \top$. For all other $b \in A$ and $\varphi \in C(S)$, let $L((s_1, s_2, e, k), b, \varphi) = \bot$.
$B_f(s_1, s_2)$			

– *Else we have (s_1, s_2) in case 3 and $B(s_1, s_2) \neq \emptyset$ by construction. The definition of L is given in Table 2, with the constraints $\varphi^\bot$ and $\varphi_{12}^{B,k}$ defined below.*

For $\varphi \in C(S_1)$, $\varphi^\bot \in C(S)$ is defined as follows: $\mu \in \mathsf{Sat}(\varphi^\bot)$ iff $\forall s_1 \in S_1$, $\forall s_2 \neq \bot$, $\forall b \neq \varepsilon$, $\forall k \neq 1, \mu(s_1, s_2, b, k) = 0$ and the distribution $s_1 \mapsto \mu(s_1, \bot, \varepsilon, 1) \in \mathsf{Sat}(\varphi)$.

For a state $(s_1, s_2, e, k) \in S$ with $s_2 \neq \bot$, $e \neq \varepsilon$ and two constraints $\varphi_1 \in C(S_1)$ and $\varphi_2 \in C(S_2)$ such that $L_1(s_1, e, \varphi_1) \neq \bot$ and $L_2(s_2, e, \varphi_2) \neq \bot$, the constraint $\varphi_{12}^{B,k} \in C(S)$ is defined as follows: $\mu \in \mathsf{Sat}(\varphi_{12}^{B,k})$ iff

1. *for all $(s_1', s_2', c, k') \in S$, if $\mu(s_1', s_2', c, k') > 0$, then $c \in B(s_1', s_2') \cup \{\varepsilon\}$ and either $\mathsf{succ}_{s_2,e}(s_1') = \emptyset$, $s_2' = \bot$ and $k' = 1$, or $s_2' = \mathsf{succ}_{s_2,e}(s_1')$,*
2. *the distribution $s_1' \mapsto \sum_{c \in A \cup \{\varepsilon\}, s_2' \in S_2 \cup \{\bot\}, k' \geq 1} \mu(s_1', s_2', c, k') \in \mathsf{Sat}(\varphi_1)$, and*
3. *either (a) there exists $(s_1', \bot, c, 1)$ such that $\mu(s_1', \bot, c, 1) > 0$, or (b) the distribution $s_2' \mapsto \sum_{c \in A \cup \{\varepsilon\}, s_1' \in S_1, k' \geq 1} \mu(s_1', s_2', c, k') \notin \mathsf{Sat}(\varphi_2)$, or (c) $k \neq 1$ and there exists $s_1' \in S_1$, $s_2' \in S_2$, $c \neq \varepsilon$ and $k' < k$ such that $\mu(s_1', s_2', c, k') > 0$.*

Theorem 4. *For all deterministic APA N_1, N_2 in SVNF such that $N_1 \not\leq_{\mathsf{m}} N_2$,*

1. *for all $K \in \mathbb{N}$, we have $N_1 \setminus^K N_2 \leq_{\mathsf{m}} N_1 \setminus^{K+1} N_2$,*
2. *for all $K \in \mathbb{N}$, $[\![N_1 \setminus^K N_2]\!] \subseteq [\![N_1]\!] \setminus [\![N_2]\!]$, and*
3. *for all PA $P \in [\![N_1]\!] \setminus [\![N_2]\!]$, there exists $K \in \mathbb{N}$ such that $P \in [\![N_1 \setminus^K N_2]\!]$.*

Note that item 3 implies that for all PA $P \in [\![N_1]\!] \setminus [\![N_2]\!]$, there is a finite specification capturing $[\![N_1]\!] \setminus [\![N_2]\!]$ "up to" P. Hence $\varinjlim [\![N_1 \setminus^K N_2]\!] = [\![N_1]\!] \setminus [\![N_2]\!]$, the direct limit.

2.6 Distances

In order to better assess how close the differences $N_1 \setminus^* N_2$ and $N_1 \setminus^K N_2$ approximate the real difference $[\![N_1]\!] \setminus [\![N_2]\!]$, we define distances on APA. These distances are based on work in [2,17,18]; see also [16].

Let $\lambda \in \mathbb{R}$ with $0 < \lambda < 1$ be a *discounting factor*.

Definition 7. *The* modal refinement distance *between the states of APA $N_1 = (S_1, A, L_1, AP, V_1, S_0^1)$, $N_2 = (S_2, A, L_2, AP, V_2, S_0^2)$ is defined to be the least fixed point to the equations*

$$d_{\mathsf{m}}(s_1, s_2) = \begin{cases} 1 & \text{if } V_1(s_1) \not\subseteq V_2(s_2), \\ \max \left\{ \begin{array}{ll} \sup\limits_{a,\varphi_1 : L_1(s_1,a,\varphi_1) \neq \bot} & \inf\limits_{\varphi_2 : L_2(s_2,a,\varphi_2) \neq \bot} D(\varphi_1, \varphi_2) \\ \sup\limits_{a,\varphi_2 : L_2(s_2,a,\varphi_2) = \top} & \inf\limits_{\varphi_1 : L_1(s_1,a,\varphi_1) = \top} D(\varphi_1, \varphi_2) \end{array} \right\} & \text{otherwise,} \end{cases}$$

where

$$D(\varphi_1, \varphi_2) = \sup_{\mu_1 \in \mathsf{Sat}(\varphi_1)} \inf_{\mu_2 \in \mathsf{Sat}(\varphi_2)} \inf_{\delta : \mu_1 \Subset^{\delta} \mu_2} \sum_{(s_1,s_2) \in S_1 \times S_2} \lambda\, \mu_1(s_1) \delta(s_1, s_2) d_{\mathsf{m}}(s_1, s_2).$$

We let $d_{\mathsf{m}}(N_1, N_2) = \max_{s_1^0 \in S_1^0} \min_{s_2^0 \in S_2^0} d_{\mathsf{m}}(s_1^0, s_2^0)$.

Note that $\sup \emptyset = 1$. The *through refinement distance* is

$$d_{\mathsf{t}}(N_1, N_2) = \sup_{P_1 \in [\![N_1]\!]} \inf_{P_2 \in [\![N_2]\!]} d_{\mathsf{m}}(P_1, P_2).$$

We need to extend this to general sets of PA; for $\mathcal{S}_1$, $\mathcal{S}_2$ sets of PA, we let $d_{\mathsf{t}}(\mathcal{S}_1, \mathcal{S}_2) = \sup_{P_1 \in \mathcal{S}_1} \inf_{P_2 \in \mathcal{S}_2} d_{\mathsf{m}}(P_1, P_2)$. The next proposition shows that our distances behave as expected, *cf.* [16].

Proposition 1. *For all APA N_1, N_2, $d_{\mathsf{t}}(N_1, N_2) \leq d_{\mathsf{m}}(N_1, N_2)$, and $N_1 \leq_{\mathsf{m}} N_2$ implies $d_{\mathsf{m}}(N_1, N_2) = 0$.*

Theorem 5. *Let N_1, N_2 be deterministic APA in SVNF such that $N_1 \not\leq_{\mathsf{m}} N_2$.*

1. *The sequence $(N_1 \setminus^K N_2)_{K \in \mathbb{N}}$ converges in the distance d_{m}, and $\lim\limits_{K \to \infty} d_{\mathsf{m}}(N_1 \setminus^* N_2, N_1 \setminus^K N_2) = 0$.*

2. *The sequence* $(\llbracket N_1 \setminus^K N_2 \rrbracket)_{K\in\mathbb{N}}$ *converges in the distance* $d_{\mathfrak{t}}$*, and* $\lim_{K\to\infty} d_{\mathfrak{t}}(\llbracket N_1 \rrbracket \setminus \llbracket N_2 \rrbracket, \llbracket N_1 \setminus^K N_2 \rrbracket) = 0$.
3. *The distance* $d_{\mathfrak{t}}(\llbracket N_1 \setminus^* N_2 \rrbracket, \llbracket N_1 \rrbracket \setminus \llbracket N_2 \rrbracket) = 0$.

Note that item 3 follows directly from items 1 and 2. It implies that even though $N_1 \setminus^* N_2$ is an over-approximation of the real difference, the two are infinitesimally close in the distance $d_{\mathfrak{t}}$. Similarly, the under-approximating differences $N_1 \setminus^K N_2$ come arbitrarily close to the real difference for sufficiently large K.

3 Real-Time Specifications

In this section we consider that $\Sigma = \Sigma^i \uplus \Sigma^o$ is a finite set of actions partitioned into inputs Σ^i and outputs Σ^o. We first define basic models for timed systems, namely TIOTS and TIOA.

A *timed I/O transition system* (TIOTS) is a tuple $(S, s^0, \Sigma, \longrightarrow)$, where S is an infinite set of states, $s_0 \in S$ is the initial state, and $\longrightarrow : S \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times S$ is a transition relation. We assume that any TIOTS satisfies the following conditions:

1. Time determinism: $\forall s, s', s'' \in S. \forall d \in \mathbb{R}_{\geq 0}$, if $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$, then $s' = s''$.
2. Time reflexivity: $\forall s \in S.\ s \xrightarrow{0} s$.
3. Time additivity: $\forall s, s'' \in S. \forall d, d' \in \mathbb{R}_{\geq 0}$, $s \xrightarrow{d+d'} s''$ iff $\exists s' \in S.\ s \xrightarrow{d} s'$ and $s' \xrightarrow{d'} s''$.

We now consider a finite set C of real-time *clocks*. A *clock valuation* u over C is a mapping $C \mapsto \mathbb{R}_{\geq 0}$. Let $d \in \mathbb{R}_{\geq 0}$, we denote $u + d$ the valuation such that $\forall x \in C.\ (u+d)(x) = u(x) + d$. Let $\lambda \subseteq C$, we denote $u[\lambda]$ the valuation agreeing with u on clocks in $C \setminus \lambda$, and assigning 0 on clocks in λ. Let $\mathcal{B}(C)$ denote all *clock constraints* φ generated by the grammar $\varphi ::= x \prec k \mid x - y \prec k \mid \varphi \wedge \varphi$, where $k \in \mathbb{Q}$, $x, y \in C$ and $\prec \in \{<, \leq, >, \geq\}$. By $\mathcal{U}(C) \subset \mathcal{B}(C)$, we denote the set of constraints restricted to upper bounds and without clock differences. We write $u \models \varphi$ if u satisfies φ. Let $Z \subseteq \mathbb{R}^C_{\geq 0}$, we write $Z \models \varphi$ if $\forall u \in Z.\, u \models \varphi$ and we denote $\llbracket \varphi \rrbracket = \{u \in \mathbb{R}^C_{\geq 0} \mid u \models \varphi\}$.

A *timed I/O automaton* is a tuple $\mathcal{A} = (L, l^0, C, E, \Sigma, I)$, where L is a finite set of *locations*, $l^0 \in L$ is the *initial location*, C is a finite set of real valued *clocks*, $E \subseteq L \times \Sigma \times \mathcal{B}(C) \times 2^C \times L$ is a set of *edges*, $I : L \mapsto \mathcal{U}(C)$ assigns an *invariant* to each location.

The semantics of a TIOA is a TIOTS $\langle\!\langle \mathcal{A} \rangle\!\rangle = (L \times \mathbb{R}^C_{\geq 0}, (l^0, \mathbf{0}), \Sigma, \longrightarrow)$, where $\mathbf{0}$ is the valuation mapping all clocks to zero, and $\longrightarrow$ is the largest transition relation generated by the following rules:

$$\frac{(l, a, \varphi, \lambda, l') \in E \quad u \models \varphi \quad u' = u[\lambda]}{(l, u) \xrightarrow{a} (l', u')} \qquad \frac{d \in \mathbb{R}_{\geq 0} \quad u + d \models I(l)}{(l, u) \xrightarrow{d} (l, u + d)}$$

Examples of TIOA are shown on Fig. 3. Edges with input actions are drawn with continuous lines, while edges with output actions are drawn with dashed lines.

3.1 Timed Specifications

A timed specification theory is introduced in [9, 10] using TIOA and TIOTS. Specifications and implementations models are defined with TIOA with additional requirements for their TIOTS semantics.

Definition 8. *A* specification $\mathcal{S}$ *is a TIOA whose semantics* $\langle\!\langle \mathcal{S} \rangle\!\rangle$ *satisfies the following conditions:*

1. *Action determinism:* $\forall s, s', s'' \in S. \forall a \in \Sigma \cup \mathbb{R}_{\geq 0}$, *if* $s \xrightarrow{a} s'$ *and* $s \xrightarrow{a} s''$, *then* $s' = s''$.
2. *Input-enabledness:* $\forall s \in S. \forall i? \in \Sigma^i. \exists s' \in S.\, s \xrightarrow{i?} s'$.

An implementation $\mathcal{I}$ *is a specification whose semantics* $\langle\!\langle \mathcal{I} \rangle\!\rangle$ *satisfies the additional conditions:*

3. *Output urgency:* $\forall s, s', s'' \in S$, *if* $\exists o! \in \Sigma^o.\, s \xrightarrow{o!} s'$ *and* $\exists d \in \mathbb{R}_{\geq 0}.\, s \xrightarrow{d} s''$, *then* $d = 0$.
4. *Independent progress:* $\forall s \in S$, *either* $\forall d \in \mathbb{R}_{\geq 0}. \exists s' \in S.\, s \xrightarrow{d} s'$, *or* $\exists d \in \mathbb{R}_{\geq 0}. \exists o! \in \Sigma^o. \exists s', s'' \in S.\, s \xrightarrow{d} s'$ *and* $s' \xrightarrow{o!} s''$.

An *alternating timed simulation* between two TIOTS $P_1 = (S_1, s_1^0, \Sigma, \longrightarrow_1)$ and $P_2 = (S_2, s_2^0, \Sigma, \longrightarrow_2)$ is a relation $R \subseteq S_1 \times S_2$ such that $\forall (s_1, s_2) \in R$,

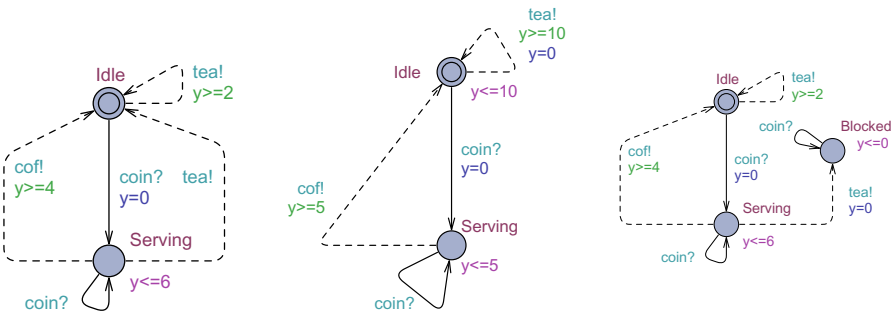
1. if $s_1 \xrightarrow{a}_1 t_1$ for some $a \in \Sigma^o \cup \mathbb{R}_{\geq 0}$, then $s_2 \xrightarrow{a}_2 t_2$ and $(t_1, t_2) \in R$.
2. if $s_2 \xrightarrow{a}_2 t_2$ for some $a \in \Sigma^i$, then $s_1 \xrightarrow{a}_1 t_1$ and $(t_1, t_2) \in R$.

We write $P_1 \leq P_2$ if there exits an alternating simulation $R \subseteq S_1 \times S_2$ with $(s_1^0, s_2^0) \in R$. For two specifications $\mathcal{S}_1$ and $\mathcal{S}_2$, we say that $\mathcal{S}_1$ *refines* $\mathcal{S}_2$, written $\mathcal{S}_1 \leq \mathcal{S}_2$, iff $\langle\!\langle \mathcal{S}_1 \rangle\!\rangle \leq \langle\!\langle \mathcal{S}_2 \rangle\!\rangle$.

An implementation $\mathcal{I}$ *satisfies* a specification $\mathcal{S}$, denoted $\mathcal{I} \models \mathcal{S}$, iff $\langle\!\langle \mathcal{I} \rangle\!\rangle \leq \langle\!\langle \mathcal{S} \rangle\!\rangle$. A specification $\mathcal{S}$ is *consistent* iff there exists an implementation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{S}$. We write $[\![\mathcal{S}]\!] = \{\mathcal{I} \mid \mathcal{I}$ is an implementation and $\mathcal{I} \models \mathcal{S}\}$ the set of all implementations of a specification.

It is shown in [10] that timed specifications also define timed games between two players: an input player that represents the environment and plays with input actions, and an output player that represents the component and plays with output actions. This timed game semantics is used to solve various decision problems, for instance consistency and refinement checking.

Consider the timed specification M of a coffee machine in Fig. 3a, and the implementation MI in Fig. 3b. We can check that this implementation satisfies the specification using a refinement game. The game proceeds as a turn-based game with two players: a spoiler starts by playing delays or output actions from the implementation, or input actions from the specification; then a replicator tries to copy the action on the other model. The spoiler wins whenever the replicator cannot mimic one of its move. Otherwise the replicator wins. For instance a strategy for the spoiler could start by delaying MI by 10 time units. Then the strategy of the replicator is two delay M by 10 time units. On the



(a) Specification M (b) Implementation MI (c) Partially inconsistent specification

Fig. 3. Specification and implementation of a coffee machine with TIOA

second move the spoiler plays action coin? on M and reaches location Serving. The replicator does the same on MI. On the third move the spoiler delays MI by 5 time units. This is allowed by the specification, so the replicator still has a winning strategy. Then the spoiler is forced to play action coff! on MI, due to the invariant in location Serving, and replicator does the same on M. The game has then returned to the initial state.

In this game a winning strategy for the replicator is necessarily infinite, as he will have to play as long as the spoiler is playing actions. However there exists symbolic techniques and algorithms for timed games [5] that restrict the game to memoryless state-based strategies on a finite number of symbolic states.

Similarly, consistency is solved using a safety game. The verifier controls the output actions of the specification, while the spoiler controls the input. The spoiler objectives is to reach an inconsistent state, that does not satisfy the independent progress condition (*i.e.* the verifier has no delay or output actions). Contrary to the refinement game, the game is concurrent: both players choose a couple delay and action at the same time, then the move that is performed is the one with the smaller delay. Consider for instance another specification of a coffee machine shown in Fig. 3c. The location Blocked is inconsistent, but the verifier can still play a strategy to avoid it (for instance by never playing action tea!). Therefore this specification is also consistent, and indeed one can check that the MI also satisfies this specification.

3.2 Robust Timed Specifications

We now introduce some perturbations in the timing constants of the models and check whether "good" properties are still satisfied. This is known as the robustness problem. Let $\varphi \in \mathcal{B}(C)$ be a guard over clocks C and let $\Delta \in \mathbb{Q}_{\geq 0}$. The *enlarged guard* $\lceil \varphi \rceil_\Delta$ is constructed according to the following rules:

- Any term $x \prec k$ of φ with $\prec \in \{<, \leq\}$ is replaced by $x \prec k + \Delta$
- Any term $x \succ k$ of φ with $\succ \in \{>, \geq\}$ is replaced by $x \succ k - \Delta$

Similarly, the *restricted guard* $\lfloor\varphi\rfloor_\Delta$ is constructed with the following rules:

- Any term $x \prec k$ of φ with $\prec \in \{<, \leq\}$ is replaced by $x \prec k - \Delta$
- Any term $x \succ k$ of φ with $\succ \in \{>, \geq\}$ is replaced by $x \succ k + \Delta$.

We lift the perturbation to implementations models. Given a jitter Δ, the perturbation means a Δ-enlargement of invariants and output edge guards, and on contrary a Δ-restriction of input edge guards:

Definition 9. *Let $\mathcal{I} = (L, l^0, C, E, \Sigma, I)$ be an implementation and $\Delta \in \mathbb{Q}_{\geq 0}$, the* Δ-perturbation *of $\mathcal{I}$ is the TIOA $\mathcal{I}_\Delta = (L \cup l^{\mathrm{u}}, l^0, C, E_\Delta, \Sigma, I_\Delta)$, where*

1. *Every edge $(l, o!, \varphi, \lambda, l') \in E$ is replaced by $(l, o!, \lceil\varphi\rceil_\Delta, \lambda, l') \in E_\Delta$.*
2. *Every edge $(l, i?, \varphi, \lambda, l') \in E$ is replaced by $(l, i?, \lfloor\varphi\rfloor_\Delta, \lambda, l') \in E_\Delta$.*
3. *$\forall l \in L.\, I_\Delta(l) = \lceil I(l)\rceil_\Delta$.*
4. *$\forall l \in L. \forall i? \in \Sigma^i$ there exists an edge $(l, i?, \varphi^{\mathrm{u}}, \emptyset, l^{\mathrm{u}}) \in E_\Delta$ with*

$$\varphi^{\mathrm{u}} = \neg\Big(\bigvee_{(l, i?, \varphi, \lambda, l') \in E} \lfloor\varphi\rfloor_\Delta\Big).$$

l^{u} is a universal location *such that, $\forall a \in \Sigma. \exists (l^{\mathrm{u}}, a, \top, \emptyset, l^{\mathrm{u}}) \in E$, where $\top$ is the clock constraints such that $[\![\top]\!] = \mathbb{R}^C_{\geq 0}$.*

An implementation $\mathcal{I}$ *robustly satisfies* a specification $\mathcal{S}$ for a given delay $\Delta \in \mathbb{Q}_{\geq 0}$, denoted $\mathcal{I} \models_\Delta \mathcal{S}$, if $\mathcal{I}_\Delta \leq \mathcal{S}$. A specification $\mathcal{S}$ is Δ-*robust consistent* iff there exists an implementation $\mathcal{I}$ such that $\mathcal{I} \models_\Delta \mathcal{S}$. We write $[\![\mathcal{S}]\!]_\Delta = \{\mathcal{I} \mid \mathcal{I}$ is an implementation and $\mathcal{I} \models_\Delta \mathcal{S}\}$ the set of all Δ-robust implementations a specification.

Refinement game is used to check robust satisfaction. Consider again the specification M and the implementation MI from 3. The Δ-perturbation of MI is presented on Fig.4. For $\Delta = 1$, we can check that $\mathsf{MI}_1 \leq \mathsf{M}$. For $\Delta = 2$ the spoiler has the following winning strategy: he plays coin? on M, then delays by 7 time units on MI_2. This cannot be mimicked by the replicator since he cannot delays more than 6 time units on the specification M. Indeed we can show then $\Delta = 1$ is the maximum value such that MI robustly satisfies M.

To solve robust consistency, the technique from [25] transforms the consistency game into a robust game. Then, the same game algorithms can be applied on this robust game. This transformation is illustrated in Fig.5. On the left, consider the specification of Fig.5a, of which we want to check the robust consistency for $\Delta = 1$. We transform the TIOA by splitting output edges, as shown on the right in Fig. 5b. In this game in location Serving, if the verifier plays its move at time $y = 5$, he must wait 1 time unit in location Serving_a and then reach location Serving_b at $y = 6$. Here the spoiler has a strategy to reach the location Bad and wins. Therefore the winning strategy for the verifier is to move to Serving_a at $y = 4$, then wait 1 time unit, and reach Serving_b at $y = 5$, where the spoiler is forced to return to location Idle. For $\Delta = 2$ this strategy fails, since location Serving_b is only reached after $y \geq 6$. This shows that the specification is 1-robust consistent.

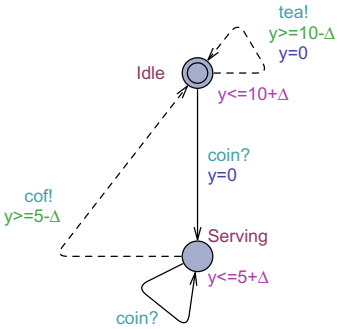
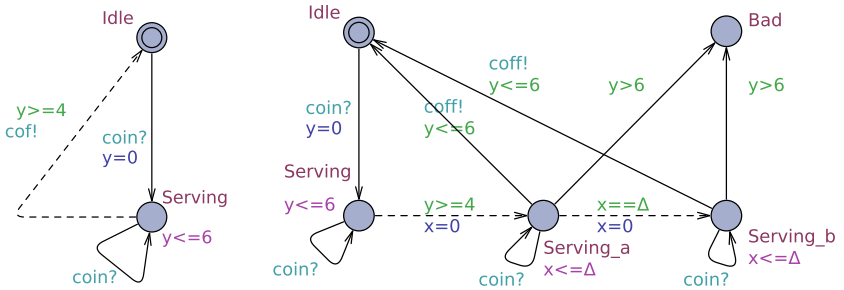


Fig. 4. Δ-perturbation of an implementation



(a) Specification (b) Robust consistency game

Fig. 5. Robust consistency game transformation for a timed specification

3.3 Conjunction

Definition 10. *The conjunction of two timed specifications* $\mathcal{S}_1 = (L_1, l_1^0, C_1, E_1, \Sigma, I_1)$, $\mathcal{S}_2 = (L_2, l_2^0, C_2, E_2, \Sigma, I_2)$ *is the TIOA* $\mathcal{S}_1 \wedge \mathcal{S}_2 = (L, l^0, C, E, \Sigma, I)$ *where* $L = L_1 \times L_2$, $l^0 = (l_1^0, l_2^0)$, $C = C_1 \uplus C_2$, $I((l_1, l_2)) = I_1(l_1) \wedge I_2(l_2)$, *and the set of edges is defined according to the following rule:*

$$((l_1, l_2), a, \varphi_1 \wedge \varphi_2, \lambda_1 \cup \lambda_2, (l_1', l_2')) \in E \text{ iff}$$
$$(l_1, a, \varphi_1, \lambda_1, l_1') \in E_1 \text{ and } (l_2, a, \varphi_2, \lambda_2, l_2') \in E_2$$

Theorem 6. *For any timed specification* $\mathcal{S}_1$, $\mathcal{S}_2$, *and* $\mathcal{T}$ *over the same alphabet:*

1. $\mathcal{S}_1 \wedge \mathcal{S}_2 \leq \mathcal{S}_2$ *and* $\mathcal{S}_1 \wedge \mathcal{S}_2 \leq \mathcal{S}_1$
2. $(\mathcal{T} \leq \mathcal{S}_1)$ *and* $(\mathcal{T} \leq \mathcal{S}_2)$ *implies* $\mathcal{T} \leq (\mathcal{S}_1 \wedge \mathcal{S}_2)$
3. $[\![\mathcal{S}_1 \wedge \mathcal{S}_2]\!] = [\![\mathcal{S}_1]\!] \cap [\![\mathcal{S}_2]\!]$
4. $[\![(\mathcal{S}_1 \wedge \mathcal{S}_2) \wedge \mathcal{T}]\!] = [\![\mathcal{S}_1 \wedge (\mathcal{S}_2 \wedge \mathcal{T})]\!]$

It turns out that this operator is robust, in the sense of precisely characterizing also the intersection of the sets of *robust* implementations. So not only conjunction is the greatest lower bound with respect to implementation semantics, but also with respect to the robust implementation semantics. More precisely:

Theorem 7. *For any timed specifications $\mathcal{S}_1$ and $\mathcal{S}_2$ over the same alphabet and $\Delta \in \mathbb{Q}_{\geq 0}$, $[\![\mathcal{S}_1 \wedge \mathcal{S}_2]\!]^\Delta = [\![\mathcal{S}_1]\!]^\Delta \cap [\![\mathcal{S}_2]\!]^\Delta$.*

3.4 Structural Composition

Two specifications $\mathcal{S}_1$, $\mathcal{S}_2$ can be composed iff $\Sigma_1^o \cap \Sigma_2^o = \emptyset$. Structural composition is obtained in by a product, where the inputs of one specification synchronize with the outputs of the other:

Definition 11. *The* structural composition *of two composable timed specifications $\mathcal{S}_1 = (L_1, l_1^0, C_1, E_1, \Sigma_1, I_1)$, $\mathcal{S}_2 = (L_2, l_2^0, C_2, E_2, \Sigma_2, I_2)$ is the TIOA $\mathcal{S}_1 \parallel \mathcal{S}_2 = (L, l^0, C, E, \Sigma, I)$, where $L = L_1 \times L_2$, $l^0 = (l_1^0, l_2^0)$, $C = C_1 \uplus C_2$, $\Sigma = \Sigma^o \cup \Sigma^i$ with $\Sigma^o = \Sigma_1^o \uplus \Sigma_2^o$ and $\Sigma^i = (\Sigma_1^i \setminus \Sigma_2^o) \cup (\Sigma_2^i \setminus \Sigma_1^o)$, $I((l_1, l_2)) = I_1(l_1) \wedge I_2(l_2)$, and for all $l_1, l_1' \in L_1$, $l_2, l_2' \in L_2$, the set of edges is defined according to the following rules:*

1. $\forall a \in \Sigma_1 \setminus \Sigma_2$, $((l_1, l_2), a, \varphi_1, \lambda_1, (l_1', l_2)) \in E$ *iff* $(l_1, a, \varphi_1, \lambda_1, l_1') \in E_1$.
2. $\forall a \in \Sigma_2 \setminus \Sigma_1$, $((l_1, l_2), a, \varphi_2, \lambda_2, (l_1, l_2')) \in E$ *iff* $(l_2, a, \varphi_2, \lambda_2, l_2') \in E_2$.
3. $\forall a \in \Sigma_1 \cap \Sigma_2$, $((l_1, l_2), a, \varphi_1 \wedge \varphi_2, \lambda_1 \cup \lambda_2, (l_1', l_2')) \in E$ *iff* $(l_1, a, \varphi_1, \lambda_1, l_1') \in E_1$ *and* $(l_2, a, \varphi_2, \lambda_2, l_2') \in E_2$.

Theorem 8. *For all specifications $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{T}$ such that $\mathcal{S}_1 \leq \mathcal{S}_2$ and $\mathcal{S}_1$ is composable with $\mathcal{T}$, we have that $\mathcal{S}_2$ is composable with $\mathcal{T}$ and $\mathcal{S}_1 \parallel \mathcal{T} \leq \mathcal{S}_2 \parallel \mathcal{T}$.*

Theorem 8 allows the independent implementability scenario: for any consistent specification $\mathcal{S}_1$ and $\mathcal{S}_2$, such that $\mathcal{S}_1$ is composable with $\mathcal{S}_2$, $\mathcal{S}_1 \parallel \mathcal{S}_2$ is consistent. Moreover, if $\mathcal{I}_1$ is an implementation that satisfies $\mathcal{S}_1$ and $\mathcal{I}_2$ is an implementation that satisfies $\mathcal{S}_2$, then $\mathcal{I}_1 \parallel \mathcal{I}_2$ $\mathsf{Sat}\mathcal{S}_1 \parallel \mathcal{S}_2$.

Finally, Theorem 9 show that this independent implementability can be extended to robust implementability:

Theorem 9. *For any Δ-robust consistent specification $\mathcal{S}_2$ and $\mathcal{S}_2$ such that $\mathcal{S}_1$ is composable with $\mathcal{S}_2$, let $\mathcal{I}_1$ be a Δ-robust implementation of $\mathcal{S}_1$ and $\mathcal{I}_2$ be a Δ-robust implementation of $\mathcal{S}_2$, then $\mathcal{I}_2 \parallel \mathcal{I}_2$ Sat_Δ $\mathcal{S}_1 \parallel \mathcal{S}_2$.*

3.5 Parametric Robustness Evaluation

Robustness problems, like robust consistency and robust satisfaction, can be solved with traditional timed games algorithms for a given value of the perturbation Δ. When considering Δ as a parameter we want to determine the maximum value of the perturbation such that these problems are satisfied.

Let $(\mathcal{A}^\Delta, W)$ be a parametric timed game, where $\mathcal{A}$ is a TIOA parametrized by Δ and W is a safety objective. We define $\Delta_{max} = \mathsf{Sup}\{\Delta \mid (\mathcal{A}^\Delta, W)$ *has a winning strategy*$\}$. Computing Δ_{max} would in general require to solve a parametric timed game, which is undecidable [1]. Therefore, considering that the problems are monotonic, we have propose in [25] a technique to estimate the maximum value of Δ with a given precision parameter. This procedure is described in Algorithm 1.

Algorithm 1: Evaluation of the maximum robustness

Input: $(\mathcal{A}^{\Delta}, W)$: parametric robust timed game,
Δ_{init}: initial maximum value,
ε: precision
Output: Δ_{good} such that $\Delta_{max} - \Delta_{good} \leq \varepsilon$

```
1 begin
2     Δ_good ← 0
3     Δ_bad ← Δ_init
4     while Δ_bad − Δ_good > ε do
5         (Δ_good, Δ_bad) ← RefineValues((A^Δ, W), Δ_good, Δ_bad)
6     end
7     return Δ_good
8 end
```

The algorithm assumes that the game $(\mathcal{A}^0, Bad)$ is won, and on contrary that $(\mathcal{A}^{\Delta_{init}}, Bad)$ is lost. At the heart of the algorithm the procedure RefineValues solves the game $(\mathcal{A}^{\Delta}, Bad)$ for a value $\Delta \in [\Delta_{good}, \Delta_{bad}]$ and updates the variables Δ_{good} and Δ_{bad} according to the result.

Different algorithms can be used to implement RefineValues. In [25] we have compared a basic binary search approach, with a counter strategy refinement approach. In this latter we analyze the winning strategies for the spoiler in order to determine the maximum value of Δ that invalidates these strategies. In practice, this technique implemented in the tool PyEcdar [27], allows Algorithm 1 to converge faster.

Δ_{good}	Δ_{bad}
0	6
0	3
0	1.5
0.75	1.5
0.75	1.125

We show in the table on the right how to run Algorithm 1 with binary search to check the robust consistency of the specification from Fig. 5a. First, we consider the robust game automaton on Fig. 5b. Δ_{init} is set to 6, which is the maximum constant in the model, and $\varepsilon = 0.5$. In the first iteration the algorithm considers $\Delta = 3$ and solves the game, which is lost. Therefore it updates the value of Δ_{bad} to 3. On the third iteration, for $\Delta = 0.75$ the game is won. In that case Δ_{good} is updated to 0.75. The algorithm stops when $1.125 - 0.75 \leq \varepsilon$.

Finally, in Table 3 we present the results of an experiment performed on an example of timed specifications that model the administration of a university (with the coffee machine specification M, presented in Fig. 3a, an administration specification A, a researcher specification R, and the structural compositions of these specifications). The results compare the performances of Algorithm 1 when checking robust consistency using either a binary search approach (BS) or a counter strategy refinement approach (CS).

Table 3. Comparing methods to check robust consistency of timed specifications

			$\Delta_{init} = 8$		$\Delta_{init} = 6$		$\Delta_{init} = 8$		$\Delta_{init} = 6$	
	Game size		$\varepsilon = 0.1$		$\varepsilon = 0.1$		$\varepsilon = 0.01$		$\varepsilon = 0.01$	
Model	loc.	edges	CR	BS	CR	BS	CR	BS	CR	BS
M	9	21	119ms	314ms	119ms	262ms	119ms	438ms	119ms	437ms
R	11	27	188ms	303ms	188ms	299ms	188ms	419ms	188ms	523ms
A	9	22	133ms	316ms	133ms	287ms	133ms	441ms	133ms	483ms
M ‖ **A**	41	158	10.1s	10.1s	10.1s	9.6s	10.4s	17.5s	10.4s	17.6s
R ‖ **A**	48	201	14.1s	12.1s	12.5s	11s	14.1s	19.6s	12.5s	19.4s
M ‖ **R**	44	152	10s	15.5s	9.81s	15.8s	10.3s	22.9s	9.78s	29.2s
M ‖ **R** ‖ **A**	180	803	54.4s	56.3s	54.6s	112s	55s	58.8s	55.7s	216s

Acknowledgment. This survey paper presents research which we have conducted with a number of coauthors; in alphabetical order, these are Alexandre David, Benoît Delahaye, Joost-Pieter Katoen, Kim G. Larsen, Ulrik Nyman, Mikkel L. Pedersen, Falak Sher, and Andrzej Wąsowski. We acknowledge their cooperation in this work; any errors in this presentation are, however, our own.

References

1. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: STOC, pp. 592–601 (1993)
2. Bauer, S.S., Fahrenberg, U., Legay, A., Thrane, C.: General quantitative specification theories with modalities. In: Hirsch, E.A., Karhumäki, J., Lepistö, A., Prilutskii, M. (eds.) CSR 2012. LNCS, vol. 7353, pp. 18–30. Springer, Heidelberg (2012)
3. Caillaud, B., Delahaye, B., Larsen, K.G., Legay, A., Pedersen, M.L., Wąsowski, A.: Compositional design methodology with constraint Markov chains. In: QEST, pp. 123–132. IEEE Computer Society (2010)
4. Canetti, R., Cheung, L., Kaynar, D.K., Liskov, M., Lynch, N.A., Pereira, O., Segala, R.: Analyzing security protocols using time-bounded task-PIOAs. Discrete Event Dynamic Systems 18(1), 111–159 (2008)
5. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)
6. Cattani, S., Segala, R.: Decision algorithms for probabilistic bisimulation. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 371–385. Springer, Heidelberg (2002)
7. Cheung, L., Lynch, N.A., Segala, R., Vaandrager, F.W.: Switched PIOA: Parallel composition via distributed scheduling. Theor. Comput. Sci. 365(1-2), 83–108 (2006)
8. Cheung, L., Stoelinga, M., Vaandrager, F.W.: A testing scenario for probabilistic processes. J. ACM 54(6) (2007)
9. David, A., Larsen, K.G., Legay, A., Nyman, U., Traonouez, L.-M., Wąsowski, A.: Real-time specifications. Int. J. Softw. Tools Techn. Transfer (2013), `http://dx.doi.org/10.1007/s10009-013-0286-x`

10. David, A., Larsen, K.G., Legay, A., Nyman, U., Wąsowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: HSCC, pp. 91–100. ACM (2010)
11. de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed interfaces. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 108–122. Springer, Heidelberg (2002)
12. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
13. Delahaye, B., Fahrenberg, U., Larsen, K.G., Legay, A.: Refinement and difference for probabilistic automata. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 22–38. Springer, Heidelberg (2013)
14. Delahaye, B., Katoen, J.-P., Larsen, K.G., Legay, A., Pedersen, M.L., Sher, F., Wąsowski, A.: Abstract probabilistic automata. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 324–339. Springer, Heidelberg (2011)
15. Delahaye, B., Larsen, K.G., Legay, A., Pedersen, M.L., Wąsowski, A.: APAC: A tool for reasoning about abstract probabilistic automata. In: QEST, pp. 151–152. IEEE Computer Society (2011)
16. Fahrenberg, U., Larsen, K.G., Legay, A., Traonouez, L.-M.: Parametric and quantitative extensions of modal transition systems. In: Bensalem, S., Lakhnech, Y., Legay, A. (eds.) FPS 2014 (Sifakis Festschrift). LNCS, vol. 8415, pp. 84–97. Springer, Heidelberg (2014)
17. Fahrenberg, U., Legay, A., Thrane, C.: The quantitative linear-time–branching-time spectrum. In: FSTTCS. LIPIcs, vol. 13, pp. 103–114. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
18. Fahrenberg, U., Thrane, C.R., Larsen, K.G.: Distances for weighted transition systems: Games and properties. In: QAPL. Electr. Proc. Theor. Comput. Sci., vol. 57, pp. 134–147 (2011)
19. Fecher, H., Leucker, M., Wolf, V.: *Don't know* in probabilistic systems. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 71–88. Springer, Heidelberg (2006)
20. Jansen, D.N., Hermanns, H., Katoen, J.-P.: A probabilistic extension of UML statecharts. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 355–374. Springer, Heidelberg (2002)
21. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: LICS, pp. 266–277. IEEE (1991)
22. Katoen, J.-P., Klink, D., Leucker, M., Wolf, V.: Three-valued abstraction for continuous-time Markov chains. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 311–324. Springer, Heidelberg (2007)
23. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. In: RTSS, pp. 166–177. Society Press (2003)
24. Larsen, K.G.: Modal specifications. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 232–246. Springer, Heidelberg (1990)
25. Larsen, K.G., Legay, A., Traonouez, L.-M., Wąsowski, A.: Robust synthesis for real-time systems. Theor. Comput. Sci. 515, 96–122 (2014)
26. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS, pp. 203–210. IEEE Computer Society (1988)

27. Legay, A., Traonouez, L.-M.: PyEcdar: Towards open source implementation for timed systems. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 460–463. Springer, Heidelberg (2013)
28. Parma, A., Segala, R.: Axiomatization of trace semantics for stochastic nondeterministic processes. In: QEST, pp. 294–303. IEEE (2004)
29. Segala, R.: Probability and nondeterminism in operational models of concurrency. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 64–78. Springer, Heidelberg (2006)
30. Segala, R., Lynch, N.A.: Probabilistic simulations for probabilistic processes. NJC 2(2), 250–273 (1995)

Compositional Branching-Time Measurements

Radu Grosu[1], Doron Peled[2], C.R. Ramakrishnan[3], Scott A. Smolka[3], Scott D. Stoller[3], and Junxing Yang[3]

[1] Vienna University of Technology
[2] Bar Ilan University
[3] Stony Brook University

Abstract. Formal methods are used to increase the reliability of software and hardware systems. Methods such as model checking, verification and testing are used to search for design and coding errors, integrated in the process of system design. Beyond checking whether a system satisfies a particular specification, we may want to measure some of its quantitative properties. Earlier works on system measurements suggest extending model checking techniques to measure quantitative artifacts, based on weights associated with the transitions of a transition system. Other works allow counting while performing model checking or runtime verification. This paper presents a simple and efficient compositional measuring framework based on quantitative state testers. The framework allows combining multiple measures, such as distance and power consumption, using a variety of functions, such as min, max, and average. This supports calculation of interesting compound measures that quantitatively characterize a system's behavior.

1 Introduction

Model checking techniques [6,12] are successfully integrated into the software and hardware development process. More than 30 years of research has produced multiple techniques. Some of them are quite impressive in the size of systems that they can handle and in verification speed. A recent trend is to look at quantitative properties, for example, providing measures on how robustly a property is satisfied in probabilistic automata [9] or weighted automata [1,5]. Another approach is to ask for, in addition to the qualitative indication of the satisfaction of a property, a measurement, which is usually based on the accumulated time required to satisfy parts of the specification [2,8]. These measures can be used for optimizing various parameters of the system.

We are motivated to provide a framework for measuring branching properties of a system. This has received so far little attention, whereas linear properties have been intensively studied. We aim to develop an efficient compositional measuring framework that extends the idea of testers [10,11] to allow quantitative operations. These extended testers are applied to the states of the measured structure and conceptually communicate with each other through flow of information between adjacent states. We generically refer to nodes in the given graph

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 118–128, 2014.

structure as "states", although the nodes may represent states in a state space, locations in a geographical space, etc.

This approach generalizes CTL specification, and in fact we show how to encode model checking of CTL in this way. This framework inherits its efficiency from the CTL logic. Nevertheless, instead of writing a CTL formula annotated with some measurement parameters, we provide a combination of recursive observers (functions) that work together to provide the desired measurement. The value associated with each state of the structure by the tester is dependent on the values at adjacent, i.e., successor and predecessor, states, henceforth called the *neighbors*. For example, the property EXp holds in a state if either p holds there, or there is a successor state where EXp holds.

In order to allow various measurements of a structure, we permit the observing functions, which combine values to produce outputs, to be over non-Boolean domains. We can describe the computation associated with this framework as a synchronized update of values associated with each state, where the new value is dependent on the previous values associated with that state and its neighbors. This allows a simple implementation, where in each step, each state applies the observing function to the previous values associated with its neighbors. On the one hand, this may not provide the most efficient implementation and should be optimized. On the other hand, it suggests a way of parallelizing the measurements.

This kind of measuring can be viewed as extending the runtime verification idea in [11] from checking whether a finite sequence can be extended to satisfy an LTL formula to measuring an entire structure. Our approach is not limited to CTL-like Boolean properties but can use different domains, including combinations of domains. Section 3 presents an example that uses two measures: the amount of battery power consumed when a robot traverses a path, and the shortest distance from the robot to a towing station. Our framework provides a more flexible measuring tool than most quantitative extensions of logic, which deal with one fixed measure, most often the average, minimum, or maximum accumulated length along paths.

When extending the framework from Boolean to more general domains, it becomes challenging to ensure that the measurement calculation terminates. Indeed, in general, applying functions recursively may not terminate, and termination itself can be undecidable. For this reason, we require that a well founded ordering is used, and successive values of a state in a measured structure must decrease in this ordering.

Our view of measurement is local, from the point of view of a state in the measured structure. To demonstrate the difference from a global view, consider CTL model checking, where a monotonicity argument can be applied on the (increasing or decreasing, respectively) size of the states while calculating the (least or greatest, respectively) fixed-points. Instead, our measurement needs to decrease locally, on the values calculated on the states. As the calculated values may not contain the information needed to show progress toward termination, we add another component: a counter that counts down from the size of the state

space (the number of states) or its diameter (the length of the longest simple path). This is a generalization of the bounds used in bounded model checking [3].

Our formalism presents a combination of observing functions, applied synchronously to the states of the measured structure, and guarded by a local well-foundedness argument to ensure termination. This can be viewed as a denotational approach, defining the measurement, but is also very close to the operational approach, defining the kind of computation needed at each state of the structure, for a finite number of steps. Therefore, we also suggest developing logics that allow more abstract and denotational representations of desired measurements, and translating (compiling) the logics into the testers used in our framework. We illustrate this approach in the example in Section 3, by presenting some CTL-like formulas extended to express various measurements.

The closest work to this paper, as far as we know, is [4]. There, an integer-based measurement of a state space is sought, and calculated using quantitative bound automata, with decision procedures based on dynamic programming. Our approach, based on testers, is more of a specification formalism, allowing us to directly describe the measure of a structure by the combination of functions (whose type can be Boolean, integer, or any other finitely representable type) and flow of information between states and their neighbors. Both approaches realize the need to bound the computation, where this is done in [4] by providing a bound on the number of iterations, and in our framework by requiring values to decrease in a well-founded ordering.

The remainder of the paper is organized as follows. Section 2 presents our framework, including our extended concept of testers, and shows how to express CTL model checking in our framework. Section 3 presents an example based on a mobile robot that illustrates the expressiveness of our framework.

2 Measuring Structures

Our computation model is based on a collection of synchronous processes that communicate via shared variables. Each process has input, output, and state variables, which may take values from arbitrary domains, including Boolean, Integers, Reals, and Cartesian products. In each step of the computation, each process updates its output and state variables based on a transition relation that relates the values of its output and next-state variables to the values of its input and current state variables. Transitions may in general be nondeterministic, although the examples in this paper focus on deterministic processes for the purpose of measurement.

The topology of this process network is specified by mapping output variables of a process to input variables of other processes. Additionally, each process is associated with a function that maps the current value of its state variables to a value from a well-founded domain. This value is required to decrease whenever the tester's state changes, guaranteeing termination.

Testers

An *atomic tester* T is a process having the following components:

- $T\langle i\rangle$ is a finite set of input variables;
- $T\langle o\rangle$ is a finite set of output variables;
- $T\langle state\rangle$ is a finite set of state variables (with specific initial values);
- $T\langle \rho\rangle$ is a transition function, mapping values of $T\langle i\rangle \cup T\langle state\rangle$ to values of $T\langle state\rangle \cup T\langle o\rangle$; and
- $T\langle w\rangle$, a function mapping values of $T\langle state\rangle$ to values in a well-founded order $(W, \ll)$.

Note that we represent testers as structures with named components, and we use the notation $T\langle c\rangle$ to select component c of tester T. A tester is *stateless* if it has no state variables.

The above definition of testers differs from the original notion in [10] in several ways. First, our testers are primarily intended for measuring properties of finite structures, and are not equipped with justice or compassion predicates. Second, our testers have explicit input and output variables, while those in [10] operate over streams of boolean values and have input and output defined implicitly. Third, and perhaps most importantly, the variables in our testers are not restricted to be boolean; in fact, tester variables can range over any domain including structured ones. Finally, each tester is equipped with a function $T\langle w\rangle$ used to ensure termination. Semantically, we require that, in every transition of a tester, either the valuation of its state variables remains unchanged, or the valuation changes from v to v' and $T\langle w\rangle(v') \ll T\langle w\rangle(v)$. This ensures termination.

A *tester circuit*, $\mathcal{C}$, is a collection of interconnected atomic testers. More precisely, $\mathcal{C} = \langle \mathcal{T}, I, O, \mathcal{S}\rangle$, where $\mathcal{T}$ is a set of atomic testers, $I \subseteq \cup_{T\in\mathcal{T}} T\langle i\rangle$ and $O \subseteq \cup_{T\in\mathcal{T}} T\langle o\rangle$ are the sets of inputs and outputs, respectively, of the circuit, and $\mathcal{S} \subseteq ((\bigcup_{T\in\mathcal{T}} T\langle o\rangle) \times (\bigcup_{T\in\mathcal{T}} T\langle i\rangle))$ specifies the connections between testers, by associating input variables with output variables. For convenience, we assume that input and output variables have globally unique names.

The computation in a tester circuit starts with all testers in initial states. The initial output of all testers is a special value "$\perp$". Each tester evolves synchronously, reading its inputs, and evaluating the transition function, thereby computing its next state and outputs.

CTL Model Checking with Testers

We now illustrate the use of testers, treating the model checking of CTL formulas as an instance of measurement of a given Kripke structure. In the following, we assume a standard definition of a Kripke structure K over a finite set of states S and atomic propositions P, given by $\langle S, \rightarrow, \sigma\rangle$, where $\rightarrow \subseteq S \times S$ is a transition relation, and σ is a function mapping states in S to sets of propositions. Given a Kripke structure K, let $n(K)$ denote the number of states in K, and let $d(K)$

denote the *diameter* of K, i.e., the length of the longest cycle-free path in its transition graph. A state t is a *successor* of a states s if $s \to t$.

We consider CTL formulas over a set of propositions P specified using the following syntax:

$$\varphi ::= P \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid EX\varphi \mid E(\varphi U \varphi) \mid A(\varphi U \varphi)$$

Additional operators of CTL can be defined in terms of these, e.g., $true = \neg(p \wedge \neg p)$ for some proposition p, $EF\varphi = E(true\ U\ \varphi)$, $AG\varphi = \neg EF(true\ U\ \neg\varphi)$, and $AX\varphi = \neg EX\neg\varphi$.

We now describe the construction of a tester circuit for checking whether a given Kripke structure is a model for a given CTL formula φ. Recall that the outputs of all testers are initially set to an undefined value $\perp$. We extend the standard Boolean operators in a symmetric way as follows: $true \vee \perp = true$, $true \wedge \perp = \perp$, $false \vee \perp = \perp$, $false \wedge \perp = false$, and $\neg\perp = \perp$.

For each formula φ and state $s \in S$, we construct a circuit $\mathcal{C}_{\varphi,s}$ based on the structure of φ, as defined below. Each circuit $\mathcal{C}_{\varphi,s}$ has a single output variable and is designed so that the final value of the output variable is true iff φ holds at state s. For CTL model checking, the circuits have no inputs, so for brevity, we omit the specification of input variables for circuits in the following construction. We express each transition relation as a set of equations (one for each state variable and output variable) in which unprimed variables represent the values of variables in the current state, and primed variables represent the values of variables in the next state. For brevity, we also omit the mapping to well-founded orders; standard arguments can be used to show termination of this calculation.

case: φ is a proposition p: Let $T_{p,s}$ be a stateless tester such that $T_{p,s}\langle i\rangle = \{\}, T_{p,s}\langle o\rangle = \{o\}$, and $T_{p,s}\langle\rho\rangle = \{o' = (p \in \sigma(s))\}$. Then $\mathcal{C}_{\varphi,s} = \langle\{T_{p,s}\}, \{o\}, \emptyset\rangle$.

case: φ is a negated formula $\neg\varphi_1$: Let $T_{\varphi,s}$ be a stateless tester such that $T_{\varphi,s}\langle i\rangle = \{i\}, T_{\varphi,s}\langle o\rangle = \{o\}$, and $T_{\varphi,s}\langle\rho\rangle = \{o' = \neg i\}$.
Let $\mathcal{C}_{\varphi_1,s} = \langle\mathcal{T}_1, \{o_1\}, \mathcal{S}_1\rangle$. Then $\mathcal{C}_{\varphi,s} = \langle\{T_{\varphi,s}\} \cup \mathcal{T}_1, \{o\}, \mathcal{S} \cup \{(o_1, i)\}\rangle$.

case: φ is a conjunction $\varphi_1 \wedge \varphi_2$: Let $T_{\varphi,s}$ be a stateless tester such that $T_{\varphi,s}\langle i\rangle = \{i_1, i_2\}, T_{\varphi,s}\langle o\rangle = \{o\}$, and $T_{\varphi,s}\langle\rho\rangle = \{o' = i_1 \wedge i_2\}$. Let $\mathcal{C}_{\varphi_1,s} = \langle\mathcal{T}_1, \{o_1\}, \mathcal{S}_1\rangle$ and $\mathcal{C}_{\varphi_2,s} = \langle\mathcal{T}_2, \{o_2\}, \mathcal{S}_2\rangle$. Then $\mathcal{C}_{\varphi,s} = \langle\{T_{\varphi,s}\} \cup \mathcal{T}_1 \cup \mathcal{T}_2, \{o\}, \mathcal{S} \cup \{(o_1, i_1), (o_2, i_2)\}\rangle$.

case: φ is an exists-next formula $EX\varphi_1$: Let state s have n successors. Let $T_{\varphi,s}$ be stateless tester with n inputs, namely, $r_1, \ldots r_n$, and one output o such that $o' = \bigvee_i r_i$. Formally, $T_{\varphi,s}\langle i\rangle = \{r_1, \ldots, r_n\}, T_{\varphi,s}\langle o\rangle = \{o\}$, and $T_{\varphi,s}\langle\rho\rangle = \{o' = \bigvee_i r_i\}$.

Let the successors of s be $t_1, t_2, \ldots, t_n$. Let $\mathcal{C}_{\varphi_1,t_j} = \langle\mathcal{T}_j, o_j, \mathcal{S}_j\rangle$ for each successor t_j.
Then $\mathcal{C}_{\varphi,s} = \langle\{T_{\varphi,s}\} \cup \bigcup_{j=1..n} \mathcal{T}_j, \{o\}, \mathcal{S} \cup \bigcup_{j=1..n}\{(o_j, r_j)\}\rangle$.

case: φ is an exists-until formula $E(\varphi_1 U \varphi_2)$: Let state s have n successors. Let $T_{\varphi,s}$ be a tester with $2 + n$ inputs, namely, $i_1, i_2, r_1, \ldots r_n$, one output o, and one state variable x initialized to $d(K)$, the diameter of the Kripke

structure. The tester is such that at each step after i_1 and i_2 get non-$\bot$ values, x is decremented, and the output o' is computed as $i_2 \vee (i_1 \wedge (\bigvee_{j=1..n} r_j))$. If x reaches 0 and the output is $\bot$, then it is set to *false*. Formally,

$$T_{\varphi,s}\langle i\rangle = \{i_1, i_2, r_1, \ldots, r_n\}$$
$$T_{\varphi,s}\langle o\rangle = o$$
$$T_{\varphi,s}\langle state\rangle = \{x = d(K)\}$$
$$T_{\varphi,s}\langle \rho\rangle = \left\{ \begin{array}{l} x' = \left\{ \begin{array}{l} \text{if } i_1 = \bot \vee i_2 = \bot \text{ then } x \\ \text{else if } x > 0 \text{ then } x - 1 \text{ else } 0 \end{array} \right. \\ o' = \text{ if } x = 0 \wedge v = \bot \text{ then } \mathit{false} \text{ else } v \\ \qquad \text{where } v = i_2 \vee (i_1 \wedge (\bigvee_{j=1..n} r_j)) \end{array} \right\}$$

Let $\mathcal{C}_{\varphi_1,s} = \langle \mathcal{T}_1, \{o_1\}, \mathcal{S}_1\rangle$ and $\mathcal{C}_{\varphi_2,s} = \langle \mathcal{T}_2, \{o_2\}, \mathcal{S}_2\rangle$. Let s have n successors, namely $t_1, t_2, \ldots, t_n$. Let $\mathcal{C}_{\varphi,t_j} = \langle \hat{\mathcal{T}}_j, \{\hat{o}_j\}, \hat{\mathcal{S}}_j\rangle$ for each successor t_j. Let $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \bigcup_{j=1..n} \hat{\mathcal{T}}_j$ and $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \bigcup_{j=1..n} \hat{\mathcal{S}}_j$.
Then $\mathcal{C}_{\varphi,s} = \langle \mathcal{T} \cup \{T_{\varphi,s}\}, \{o\}, \mathcal{S} \cup \{(o_1, i_1), (o_2, i_2)\} \cup \bigcup_{j=1..n} \{(\hat{o}_j, r_j)\}\rangle$.

case: φ is an always-until formula $A(\varphi_1 U \varphi_2)$: This is similar to the exists-until case above, except that it performs a conjunction (instead of a disjunction) over the results r_i from successor states. Let state s have n successors. Let $T_{\varphi,s}$ be a tester with $2 + n$ inputs, namely, $i_1, i_2, r_1, \ldots r_n$, one output o, and one state variable x initialized to $d(K)$, the diameter of the Kripke structure. Let

$$T_{\varphi,s}\langle i\rangle = \{i_1, i_2, r_1, \ldots, r_n\}$$
$$T_{\varphi,s}\langle o\rangle = o$$
$$T_{\varphi,s}\langle state\rangle = \{x = d(K)\}$$
$$T_{\varphi,s}\langle \rho\rangle = \left\{ \begin{array}{l} x' = \left\{ \begin{array}{l} \text{if } i_1 = \bot \vee i_2 = \bot \text{ then } x \\ \text{else if } x > 0 \text{ then } x - 1 \text{ else } 0 \end{array} \right. \\ o' = \text{ if } x = 0 \wedge v = \bot \text{ then } \mathit{false} \text{ else } v \\ \qquad \text{where } v = i_2 \vee (i_1 \wedge (\bigwedge_{j=1..n} r_j)) \end{array} \right\}$$

Let $\mathcal{C}_{\varphi_1,s} = \langle \mathcal{T}_1, \{o_1\}, \mathcal{S}_1\rangle$ and $\mathcal{C}_{\varphi_2,s} = \langle \mathcal{T}_2, \{o_2\}, \mathcal{S}_2\rangle$. Let s have n successors, namely $t_1, t_2, \ldots, t_n$. Let $\mathcal{C}_{\varphi,t_j} = \langle \hat{\mathcal{T}}_j, \hat{o}_j, \hat{\mathcal{S}}_j\rangle$ for each successor t_j. Let $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \bigcup_{j=1..n} \hat{\mathcal{T}}_j$ and $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \bigcup_{j=1..n} \hat{\mathcal{S}}_j$.
Then $\mathcal{C}_{\varphi,s} = \langle \mathcal{T} \cup \{T_{\varphi,s}\}, \{o\}, \mathcal{S} \cup \{(o_1, i_1), (o_2, i_2)\} \cup \bigcup_{j=1..n} \{(\hat{o}_j, r_j)\}\rangle$.

3 Example

We illustrate the use of testers by measuring the weighted branching structure of an autonomous robot with respect to a desired CTL property. The movement of the robot within its environment is expressed with the finite, weighted Kripke structure $K = (S, s_1, \rightarrow, \sigma, c, d)$ shown in Figure 1(a). State s_1 is marked by σ as initial, and states s_5 and s_6 are marked by σ as towing. Each transition $\rightarrow_{ij}$ of K is annotated with two weights. The first weight, c_{ij}, is the energy consumed (as a percentage of a fully charged battery) along $\rightarrow_{ij}$, i.e., when moving from state s_i to state s_j. The second weight, d_{ij}, is the distance traveled (in meters) along $\rightarrow_{ij}$.

We would like to check and measure the following property φ: *Whenever the robot reaches a state where its overall battery consumption bc* > 80*, there is a*

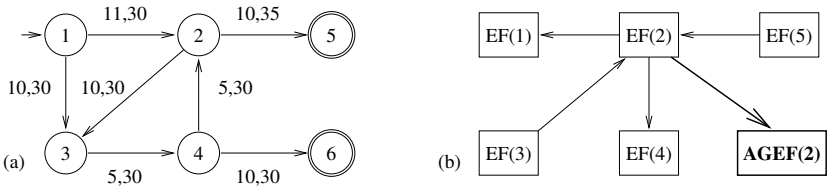


Fig. 1. (a) Weighted Kripke structure associated with the movement of a robot. On each edge, the first weight represents the energy consumed on that edge, and the second weight represents the distance traveled on that edge. (b) Communication structure of the tester $EF(2)$. This tester receives horizontal messages from the testers $EF(5)$ and $EF(3)$, sends horizontal messages to the testers $EF(1)$ and $EF(4)$, and vertical messages to the tester $AGEF(2)$.

towing state within a distance $td < 100$. Let ts be an atomic proposition that is true in towing states. Then, φ can be written in a CTL-like logic as:

$$\varphi \;\doteq\; AG_{bc>80}\; EF_{td<100}\; ts$$

Although φ seems to capture our intuition, we encounter difficulties as soon as we seek to measure it, due to the inherent nondeterminism in CTL formulas. In fact, we are not interested in any towing distance $td<100$ from a state, but in its *minimal* towing distance. Hence, the property we would like to measure can be more precisely stated as: *Whenever the robot reaches a state where its overall battery consumption* $bc>80$*, there is a* minimal *towing distance* $td<100$.

Testers

In order to capture, and more importantly, to properly *measure* such properties in all their generality, we associate with each temporal operator, and each state of the weighted Kripke structure K, a tester. These testers communicate in a data-flow fashion both horizontally, with the testers of the same kind, and vertically, with the testers corresponding to the enclosing operator.

For example, the communication structure of the EF tester associated with state s_2 is shown in Figure 1(b). This tester receives messages from the EF testers associated with its successor states s_3 and s_5 in K, and sends messages to the EF testers associated with its predecessor states s_1 and s_4 in K. It also sends messages to the tester $AGEF$ of its enclosing formula at state s_2.

As the example indicates, the communication structure of a tester is completely defined by the structure K, the formula φ we intend to measure, and the state s with which the tester is associated. A tester accumulates (folds) path information in K starting at (or ending in) s and passes it to the other testers. In order to do this, the tester requires the following information: (1) a procedure for folding information along a single path, (2) a procedure for folding the information from multiple paths, and (3) a termination condition.

A natural way to provide such information is through a complete, idempotent dioid structure $D=(+,\times,0,1)$, where $+$ is an additive, commutative and

idempotent monoid, with neutral element 0, and $\times$ is a multiplicative monoid, with neutral element 1. In this setting, (1) is taken care of by multiplication, (2) is taken care of by the addition, and (3) is taken care of by the completeness of the dioid. In some cases, however, one would like to fold information using other operations, such as averaging. In such situations, an explicit termination condition, which we refer to as a "stopping criterion", has to be provided, as the completeness of the dioid does not suffice. Stopping criteria can also be used to speed up the computation.

For AF and AG formulas, an operator folding the information computed in all their satisfying states must also be provided. To make the above discussion more precise, let us define the testers associated with φ.

The *EF* Tester

Consider first the EF tester associated with the subformula $EF_{td<100}\, ts$ at state s_4. In this state, we have two paths that satisfy the constraint $td<100$: the path $p_1 = s_4 s_6$ with associated towing distance $td_1 = 30$, and the path $p_2 = s_4 s_2 s_5$ with associated towing distance $td_2 = 65$.

The towing distances td_1 and td_2 are computed by folding the weights along the paths p_1 and p_2, respectively, in an additive fashion. Moreover, since we are only interested in the shortest path, we fold the information among different paths by taking their minimum. Hence, the idempotent dioid structure we are interested in is $D = (min, +, \infty, 0)$. The stopping criterion, $\psi \doteq (td \geq 100)$ is not necessary for convergence, but it speeds up the computation, and helps compute the measure for the entire CTL formula φ. The atomic proposition we are passing to the EF tester as a parameter is simply $p \doteq ts$.

Given the above considerations, the general specification of an $EF^D_\psi\, p\,(s, K)$ tester can be given once and for all as below. We assume that the communication structure is automatically compiled in EF from K and φ. For readibility of the tester, we instantiate p, ψ and D in its definition.

$EF^{(min,+,\infty,0)}_{td\geq 100}\, ts\,(s, K)$
{
  init: `td =` $(\sigma(s) = ts)$ `? 0 :` ∞
  stop: `td` ≥ 100
  transition:
    `td =` $min_{t\in(s\to t)}$ `(td, receiveEF(`t`)` $+ d_{st}$`)`
    `sendEF(`t`)`$_{t\in(t\to s)}$ `= td`
    `sendAGEF(`s`) = (td` ≥ 100`) ? (`b`=F,` td`=td) : (`b`=T,` td`=td);`
}

Note that this section uses a more concise and less formal notation for testers, including a send-receive notation for communication. For example, `sendEF(`t`) = td` denotes sending the value of `td` to the tester for the EF formula at state t, and `receiveEF(`t`)` denotes the value received from the EF tester at state t. This notation can be translated straightforwardly into the more formal notation in Section 2. We also use tuples with named fields as communication messages.

The AG Tester

The tester associated with the $AG_{bc>80}$ property can be defined independently of the rest of the subformulas in φ. The purpose of this tester is to compute the battery consumption in a forward fashion, starting from the initial state.

We would like to stop the computation of the battery consumption at a given state s, as soon as we arrive (for the first time) at s with $bc > 80$. Moreover, if we arrive at s along two different paths with values bc_1 and bc_2, we would like to consider only the maximum of bc_1 and bc_2. Hence, the dioid structure we are interested in has the form $D = (max, +, -\infty, 0)$. The termination condition in this case is $\psi \doteq (bc > 80)$.

The intuition is as follows. In structure K of Figure 1(a), there is no simple path ending in a state with $bc > 80$. However, looping sufficiently many times within the cycle $s_2 s_3 s_4 s_2$ increases the battery consumption until we arrive at s_3 with $bc = 81$, and thereafter in s_4, s_2, and s_6 with bc equal to 86, 91 and 96, respectively. Hence, we can define (once and for all) the AG tester as below. Again, for readability, we instantiate the parameters ψ and D.

$AG_{bc>80}^{(max,+,-\infty,0)}(s, K)$
{
  init: `bc =` $(\sigma(s) = init)$ `? 0 :` $-\infty$
  stop: `bc` > 80
  transition:
    if `(bc` ≤ 80`) then`
      `bc =` $max_{t \in (t \to s)}$ `(bc, receiveEF(`t`) +` c_{ts}`))`
    `sendAG(`t`)`$_{t \in (s \to t)}$ `= bc`
    `sendAGEF(`s`) = bc` > 80 `? (`b`=T,` bc`=bc) : (`b`=F,` bc`=bc)`
}

The AGEF Tester

The $AGEF$ tester at state s collects the information from its $AGEF$ peers, and from the AG and the EF testers at state s. Then, in conjunction with its $AGEF$ peers, it checks and measures the top-level formula φ.

For each state where if $bc > 80$ holds then it is also the case that $td < 100$ holds (which is the formal requirement in φ), we would like for our example that $AGEF(s)$ first compute the linear combination $lc = 0.6 \times bc + 0.4 \times td$. We would then like it to back propagate this information so that the initial state will contain the average of all such linear combinations.

The main problem in the back propagation is the cycle $s_2 s_3 s_4 s_2$: simply sending lc to the $AGEF$ testers of all its predecessor states would result in double counting. Assuming the existence of a linear order on the states, with the initial state as minimal, we therefore send the lc information to only a single predecessor, the one which is minimal in the state ordering.

To ensure termination as well as proper property checking and measuring, each $AGEF$ tester sends a tuple $(bb, ns, lcSum, done)$. The value bb is true when the state satisfies $\neg(bc > 80) \vee (td < 100)$, and all of its greater $AGEF$ successors do. The value $lcSum$ is the sum of the lc value of the current $AGEF$ state and the

lc values of its greater successors. The value of ns is the number of states whose lc values are summed in $lcSum$. The value of $done$ is true when all the greater $AGEF$ successors are done; this value is also used as the stopping condition of the $AGEF$ tester. Note that the average of the lc values can always be computed as $lcSum/ns$; for brevity, this calculation is omitted from the pseudocode for the $AGEF$ tester.

The structure $D = (avg, lin(0.6, 0.4), 0)$ contains the commutative monoid for avg, and the linear-combination operator (with its associated weights). One therefore needs in addition a state ordering, and the termination condition $done$. Given all these considerations, the specification of the $AGEF$ tester is as follows:

$AGEF^{(avg,lin(0.6,0.4),0)}_{done}\ (s, K)$
{
  init:
    bb = (receiveAG(s).b ⇒ receiveEF(s).b)
    lcSum = bb ? lin(0.6,0.4)(receiveAG(s).bc, receiveEF(s).td): 0
    ns = 1
    done = F
  stop: done
  transition:
    alreadyDone = done
    done = $\wedge_{t\in(s\to t)\wedge(t>s)}$(receiveAGEF($t$).$done$)
    if (done ∧ ¬alreadyDone) **then**
      sb = $\wedge_{t\in(s\to t)\wedge(t>s)}$(receiveAGEF($t$).$bb$)
      bb = bb ∧ sb
      ns = ns + $sum_{t\in(s\to t)\wedge(t>s)}$(receiveAGEF($t$).$ns$)
      lcSum = lcSum + $sum_{t\in(s\to t)\wedge(t>s)}$(receiveAGEF($t$).$lcSum$)
    predecessor = $min_{t\in(t\to s)\wedge(t<s)}$($t$)
    sendAGEF(predecessor) = (bb = bb, ns = ns, $lcSum$ = lcSum, $done$ = done)
}

To simplify the pseudo-code, we assume that the receives from the AG and EF testers in the **init** section block the $AGEF$ tester until the computations of the AG and EF testers have terminated, because we want the $AGEF$ tester to compute with the final values from those testers. This implicit synchronization can be implemented explicitly using additional variables.

This tester has the property that the formula φ is true, provided that the $AGEF$ tester of state s_1 is done and its Boolean value bb is true. In that case, the desired average is $lcSum/ns$.

Implementation and Results

We implemented the robot example in MATLAB to gain experience with the behavior and performance of these testers. We used centralized data structures for initial-prototyping purposes. As future work, we plan to develop a distributed implementation.

In Table 1, we present the results obtained from the testers. One can easily check that all boolean and real values are correctly computed and propagated to a tester's neighbors. The final result is the tuple $(T, 6, 337.0)$ computed by

the $AGEF$ tester for the initial state s_1. These values convey the fact that the property does hold for the weighted Kripke structure K of Figure 1(a) and yields the desired average as $337.0/6 = 56.17$.

Table 1. Results obtained from the testers for the robot example

	EF Testers		AG Testers		$AGEF$ Testers		
	b	td	b	bc	bb	ns	$lcSum$
s_1	T	65	F	0	T	6	337.0
s_2	T	35	T	91	T	2	117.2
s_3	T	60	T	81	T	3	193.8
s_4	T	30	T	86	T	2	121.2
s_5	T	0	T	81	T	1	48.6
s_6	T	0	T	96	T	1	57.6

References

1. Almagor, S., Boker, U., Kupferman, O.: What's Decidable about Weighted Automata? In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 482–491. Springer, Heidelberg (2011)
2. Alur, R., Etessami, K., La Torre, S., Peled, D.: Parametric temporal logic for "model measuring". ACM Trans. Comput. Log. 2(3), 388–407 (2001)
3. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. Advances in Computers, vol. 58. Academic Press (2003)
4. Chakrabarti, A., Chatterjee, K., Henzinger, T.A., Kupferman, O., Majumdar, R.: Verifying Quantitative Properties Using Bound Functions. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 50–64. Springer, Heidelberg (2005)
5. Chatterjee, K., Doyen, L., Henzinger, T.A.: Expressiveness and Closure Properties for Quantitative Languages. Logical Methods in Computer Science 6(3) (2010)
6. Clarke, E.M., Allen Emerson, E.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
7. Allen Emerson, E., Clarke, E.M.: Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 169–181. Springer, Heidelberg (1980)
8. Faymonville, P., Finkbeiner, B., Peled, D.: Monitoring Parametric Temporal Logic. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 357–375. Springer, Heidelberg (2014)
9. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated Verification Techniques for Probabilistic Systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 53–113. Springer, Heidelberg (2011)
10. Kesten, Y., Pnueli, A., Raviv, L.-O.: Algorithmic Verification of Linear Temporal Logic Specifications. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 1–16. Springer, Heidelberg (1998)
11. Pnueli, A., Zaks, A.: PSL Model Checking and Run-Time Verification Via Testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006)
12. Queille, J.-P., Sifakis, J.: Iterative Methods for the Analysis of Petri Nets. In: Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets. LNCS, vol. 51, pp. 161–167. Springer (1981)

Steps towards Scenario-Based Programming with a Natural Language Interface

Michal Gordon and David Harel

Weizmann Institute of Science, Rehovot Israel

Abstract. Programming, i.e., the act of creating a runnable artifact applicable to multiple inputs/tasks, is an art that requires substantial knowledge of programming languages and development techniques. As the use of software is becoming far more prevalent in all aspects of life, programming has changed and the need to program has become relevant to a much broader community. In the interest of broadening the pool of potential programmers, we believe that a natural language interface to an intuitive programming language may have a major role to play. In this paper, we discuss recent work on carrying out scenario-based programming directly in a controlled natural language, and sketch possible future directions.

1 Introduction

Imagine a future home, with a slew of smart home devices installed, such that the various parts of the controlling software can communicate. Now imagine the owners have a new idea about the desired behavior of the system, something that none of the vendors had considered, and which is therefore not a new *configuration*. They would like the shades to be lowered, whenever the TV is turned on and the light outside is too bright. However, if the kids are home, they do not want to dim the living room. Will they have to contact the professional designers of their smart home, or the vendors? We hope not. In fact, we would like to believe that in the future more people will be able to *program*, or enhance existing programs by adding requirements, on their own. In fact, throughout the paper, when we refer to the "programmer", we mean the person (or team of people) who creates a program, not necessarily a professional programmer in the usual sense of the word, perhaps more like a system's engineer.

The art of programming is already available to a broad community, with the open architecture of mobile phone applications, languages and toolkits that let children program games, robots and more [27,37], and methods that allow programming by demonstration and end-user programming [7]. Here we discuss an approach, whereby, the computer is able to derive an executable program directly from the human's language, or something very close thereto.

Computerized understanding of natural language is an extremely complicated and broad problem, and has been studied widely (see, e.g., [41,38,7]). It is a central facet of intelligence, and its difficulty is thus closely related to tackling the

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 129–144, 2014.

Turing test. We do not attempt to solve this problem, nor is it the focus of our discussion. Rather, we concentrate on *programming in controlled natural language* (PiCNL). Subsets of natural language have been used to allow intuitive yet formal interfaces for various tasks. They are often called *simplified* or *controlled* language [1,7]. We review these in Section 2.1. Our main question here is whether we can use controlled language to 'explain' to the computer what we want our system to do under all possible circumstances (very much like the way we would have explained it to a person), and cause it to generate a fully executable program.

The reader may claim that we can already talk to computers: For example, we can ask our smart-phone, to call a particular friend, schedule dinner and make a two-person reservation in our favorite restaurant. These activities, however, are not programming. Such natural language interactions are often called *command and control* [38,12], and can be carried out using voice or natural text interfaces.

In Section 2 we discuss the quest for programming in natural language, controlled natural language and various interfaces that we distinguish from programming. In Section 3 we review several efforts at programming in natural language. Finally, in Section 4 we discuss in some detail a relatively new scenario-based programming paradigm, called *behavioral programming* [22], and show how we use natural language to program reactive systems, and how it can be enriched with GUI-based play-in [21].

2 The Quest for Programming in Natural Language

The concept of programming in controlled natural language, i.e., PiCNL, has been suggested in the past, however it has not become prevalent as a programming method. In this section we discuss controlled natural language, we define programming in natural language, and distinguish PiCNL from various natural language interfaces. We review how PiCNL has developed over the years, including the obstacles encountered and proposed solutions.

The simplest form of using natural language with a computer is *dictation*. The text can be entered to the computer by typing or by speech, and all the computer does is the parsing into words and sentences. There is no requirement that the text be analyzed or understood in any way. Thus, if typing is used, this task is trivial. If the text is spoken we get into the realm of speech recognition, which we will not discuss here. For our purposes in this paper, entering the text can be done either way, depending on the preference of the user.

2.1 Using Controlled Natural Language

Controlled natural language (CNL) is created by restricting the grammar of a natural language to reduce ambiguity and complexity. It can be made more formal than unrestricted NL, well-structured and better amenable to semantics, since due to its restrictions, relevant constructs can be semantically annotated. CNLs enjoy some of the advantages of natural language; they are easy to use and

understand, have a quick learning curve, and are close to the application domain. They also avoid much of the ambiguity and vagueness of natural language, and thus also enjoy some of the advantages of formal languages. However, their use is mostly in the technical arena. You probably wouldn't want to try to write "controlled poetry"....

Controlled languages can be learned, and then easily written, with the aid of appropriate editors and parsers. Due to their relative simplicity, they support automation of tasks, and are used in automatic translation, formal document writing, reasoning, robot-controllers, specifications, and ontologies [1,11,29].

Simplified English, as controlled English was originally called, was developed for the aerospace industry's maintenance manuals. It was also referred to as *plain language* or in its trademarked version as simplified technical English (STE) [1]. The idea was to restrict the lengths of sentences and paragraphs, avoid slang and jargon, use active voice, etc. The STE dictionary includes approved words that can be used only according to their specific meaning. For example, the verb *close* can be used in the sense of "close a door" but not as the adjective, "stand close to the landing gear". An alternative word is often suggested — here it was *near*, as in "stand near the landing gear".

2.2 Programming vs. Command and Control

A more advanced form of using natural language with a computer involves the variety of operations that employ familiar natural language for controlling or manipulating systems. These activities fall under the general term of *command & control*, or *natural language interactions*; see, e.g., [38].

Natural language is used for search, for personal assistance, and in general as an interface for commanding other applications [25,38]. A user can say to her smart-phone "call Martin Jones". The word "call" will be recognized semantically, the contacts will be searched for the person in question, and if successful, the command will be performed, i.e., the call will be placed. Command & control systems can retrieve answers that may be given in natural language, or sometimes in more appropriate forms. For example, a user can ask a mobile phone "how far am I from the Fairview Green restaurant?", and the answer may be given as "1.7 miles". Smarter applications can take into account the moving speed of the mobile phone and provide a more relevant answer, like "a fifteen minute walk" or "a three minute drive".

Natural language has been used as a conversational user interface to describe and manipulate visualizations of data [39]. Thus, in a specific setting the user can ask (taken verbatim from [39]): "What is the correlation between the depth and water's temperature", and receive a graph plotting the two. He/she can then add a command "please color by pH", and obtain a color code layered atop the graph. This kind of interface is more advanced and permits complex connections between various commands.

Although the interfaces we described so far are quite elaborate, none of them is considered programming. In *programming*, the directions/recipes are to be applicable in the future to many inputs, most often to infinitely many of them.

Thus, one may say that a central characteristic of a program (as opposed to a command & control system) is its being *reusable* when different inputs arrive from the environment.

The difference is similar to that between giving one-time directions for someone to do something, and teaching that person a skill to use when applicable. For example, showing a child how to wash some specific dishes is different from explaining when and how you decide whether to do the dishes, and how to do so in general. In the latter case, the child will know how to handle a sink full of dishes, an empty sink, or a sink with a single dish. Untreated cases, such as a blocked sink, would require additional directions, or an extension of the "program". Similarly, command & control is telling the car radio to turn on and to tune to channel Z-100. A program is when I tell the car radio that *whenever* I turn on the radio and switch to Z-100, if the channel is airing conversation, rather than music for more than one full minute, switch to another channel.

Although programming has to do with multiple runs of the same program, this is not the same as creating recurrent behavior. Anyone can easily set a recurrent event in an electronic calendar, like a family dinner every Saturday night. Indeed, advance interfaces, such as Google Calendar's *Quick Add* feature, permit setting such recurrent scheduling in natural language. Entering "Family dinner every Saturday" will set the recurring event, with a weekly reminder. However, this is not programming, as it does not depend on varying inputs.

In programming there are multiple different inputs to the same general set of commands and instructions, and the program can deal with all of them, even those that have not been considered explicitly by the programmer. Programming is not something that can be achieved merely by using different menus, or by configuring the system. An example is "whenever it rains, and my calendar shows a soccer game the same day, cancel the event and notify all participants". It is not only the addition of the notification action that makes this programming, but the constant monitoring of rain as an external event. This is a lot more than a command. It is a program rule; a kind of program snippet. And this raises the harder question of how to deal with multiple program snippets, which constitute a full program. For example, how to manage several of these program-rules that interact, and perhaps even contradict one another? We discuss this further in Section 4.

2.3 Programming in Natural Language

Natural language has been used in software engineering for tasks that are highly related to programming. For example, some database queries can be specified by natural language and may be considered a type of programming [42]. Viewing and displaying data can be done with natural language, e.g., with the Articulate system [39]. Natural language has also been used in computer aided software engineering (CASE) tools, to help the process of modeling or creating software engineering artifacts. For example, in [13], natural language is used to help create better use cases for system development, and other papers produce other UML documents, etc.

Methods, that use statistical NLP, combine learning techniques with NLP to analyze natural language and automatically create partial code. For example, transforming English specifications of input file format (with additional information from sample input files) to automatically generate C++ code for input parsers [30], or analyzing API documents to infer API library specifications [43].

As to full programming in natural language, Dijkstra claimed in 1978 [10]:

> "In order to make machines significantly easier to use, it has been proposed (to try) to design machines that we could instruct in our native tongues. This would, admittedly, make the machines much more complicated, but, it was argued, by letting the machine carry a larger share of the burden, life would become easier for us. It sounds sensible provided you blame the obligation to use a formal symbolism as the source of your difficulties. But is the argument valid? I doubt. [...] Instead of regarding the obligation to use formal symbols as a burden, we should regard the convenience of using them as a privilege: thanks to them, schoolchildren can learn to do what in earlier days only genius could achieve."

In our opinion, programming with *controlled* natural language, should be also viewed as a privilege. Those willing to express their system requirements in a CNL and disambiguate their input to the computer when it is not clear enough, can gain the benefits of instantly executable programs.

A 2004 study by Liu and Lieberman [31] discussed how natural language descriptions could be used to make human-machine communications more natural. Those authors state that "several developments might now make programming in natural language feasible".

Sloppy programming [7] initially used unstructured text, yet later found that the unstructured approach caused too many false interpretations. Sloppy programming uses a specific grammar, based on an existing set of scripts, to allow the user to enter something simple and natural. The essence of sloppy programming is to interpret and make sense of the 'sloppy' input. Controlled natural language is similar, in that it is simple and natural enough, yet it restricts the grammar, and requires the user to learn the restrictions from examples and by feedback.

Chickenfoot and CoScripter [7] allow users to write web-customization scripts using simplified Java script commands. They use a domain specific vocabulary to allow performing various programming tasks in the web domain. End-user programming and scenario-based programming are similar in that they aim to create a more natural means of authoring behavioral fragments.

In [34], the authors show how some of the more subtle aspects of procedural programming — steps and loops — can be handled effectively, and express their believe that advances in natural language processing can contribute to the task of natural language programming, for descriptive and procedural programming paradigms.

Indeed, one large community that can benefit from PiCNL are children, too young to acquire formal education of programming. Several advances in natural interfaces to programming for children have been made, most notably with languages like Scratch [37], which enable children to formally describe their system

requirements via visual blocks that help them overcome syntax problems. Scratch is available in multiple languages, and allows children the feel of programming naturally. Although, not precisely PiCNL, Scratch allows formal programming with blocks containing natural English text.

In [10], Dijkstra also remarks that

> "there is a sharp decline in people's mastery of their own language [...], and many people are no longer able to use their own native tongues effectively."

He says that this "New Illiteracy"

> "should discourage those believers in natural language programming that lack the technical insight needed to predict its failure."

Despite Dijkstra's gloomy statements, we feel that the time is ripe for major efforts to program in natural language.[1] Dijkstra's pessimism can be overcome by a careful choice of the limited CNL to be used, and the appropriate programming paradigm into which it will be translated. PiCNL will be suitable only for those willing to master CNL, disambiguate problems, and understand how faults may occur. In Section 4, we shall discuss how such a paradigm can reduce some of the technicalities that make natural language programming difficult.

3 Approaches to Programming in Natural Language

Several research efforts have led to languages that can create executable code from a CNL. Each one defines its own CNL style and translates the CNL into a different notation, each with its own merits and areas of applicability. We describe some of these, focusing not on methods that support computing, but rather on those that create executable artifacts.

In [6], use-case templates, written a CNL, are translated into *process algebra* (in the CSP notation). This method was implemented in a Microsoft Word^{TM} plug-in that checks adherence of use-case specifications to a CNL grammar and translates them into process algebra. It then allows carrying out system property verification. This technique supports user-view use-cases, which can be used to specify user operation and expected system responses, and component-view use-cases, with one component that invokes an action and another that provides the service. After automatic translation to the CSP notation, a model checker is used to check refinement between user and component views.

The CNL in [6] is used to write imperative sentences, which describe actor actions, and affirmative sentences, which describe system characteristics. Requirements are written in tabular form, in numbered steps. For the user-view use-case, each step includes a user action, a system state, and the system's response. The CNL reflects the selected domain. Besides automatically generating formal models, the use of the CNL in [6] prevents the introduction of ambiguous formatted sentences in the use-case specification, thus helping to increase document quality.

[1] The second-listed author's work on visual languages has given him an earlier reason to believe that Dijkstra's pessimism need not always be taken too seriously.

Attempto controlled english (ACE) [11] is an example of a CNL that was designed to serve as a knowledge representation language, and its output is fully executable. ACE accepts a sequence of anaphorically interrelated sentences. This means that references to objects mentioned in previous sentences are acceptable, creating a coherent text of linked sentences. These can include coordination, subordination, quantification and negation. One can describe something that is the case — a fact, an event, a state. The interpretation of the sentences is deterministic, and a paraphrase reflects the interpretation to the programmer.

The lexicon can be modified by the programmer for domain specific content. Questions can be written in CNL and are translated into Prolog queries, which are then answered by logical inference. The knowledge can be executed for simulation or prototyping. Execution involves adding statements that would start the simulation, e.g., "customer1 is a customer", "card1 is a card", etc.

ACE is used in a variety of applications: as an NL interface in database query languages and robot controllers, in planning medical reports, for the semantic web (translation to and from web-languages) for protein ontologies, and as a reasoner that performs deductions [29].

Two-level-grammar (TLG) [5], is an object-oriented requirements specification language with a natural language style. It is sufficiently formal to allow automatic transformations into UML class diagrams and into object-oriented code, such as Java. The methods are described in natural language as a sequence of behaviors, allowing services and functions to be referred to and called upon. This formalism allows one to describe object-oriented behavior naturally, and each function definition is composed of logical rules executed in the order they are given.

TLG is natural-language-like in style, but is sufficiently formal to be automatically translated into object-oriented formal specifications.

In *spoken Java* [3], programmers can describe their Java program orally in natural language. The method was developed for programmers who suffer from repetitive strain injuries, and therefore the natural language is very similar to Java and programming knowledge is a prerequisite.

The efforts in references [5,11], are general. However, domain specific applications also exist; e.g., for robot controllers. In [28] *linear temporal logic mission planning toolkit* (LTLMoP) is used for writing specifications in structured English. The language is used to specify safety and liveness properties. The implementation is through a grammar that translates into LTL.

The MOOIDE system [7], based on the Metaphor system tests the idea of describing behavior with stories in the domain of a virtual reality storytelling game. The game itself is a reactive system. The interface in MOOIDE takes the form of a dialog in natural language about a growing set of terms that are added to the world, and it uses common sense semantics. The MOOIDE system, although domain specific, has many elements similar to the scenario-based programming approach that we describe in the next section.

4 Behavioral Programming in Controlled Natural Language

Behavioral programming (BP) is a recently proposed programming paradigm in which system behavior is described in scenarios, similar to the way people naturally specify behavior [22]. This naturalness appears to be a crucial component of the quest for *liberating programming* [19]. We have developed a natural language input interface for BP [14], in which scenarios are described with CNL and are transformed automatically into a BP formalism called *live sequence charts* (LSC) [9]. These, in turn can be executed, using play-out [20], planning algorithms or synthesis [32].

We focus on two of the main concepts underlying behavioral programming, namely, inter-object programming and unification.

In the *inter-object* approach a behavior is usually described as a "story" that considers the operations that occur between objects, rather than focusing on the operations within objects, as is the case in the *intra-object* style of object-oriented programming. Although in both cases each object has unique operations and properties, in intra-object behavior the programming process focuses on the objects, whereas in inter-object programming, the focus is on the interaction *between* the objects. Shifting the focus to the between-objects behavior, allows for a far more natural and "liberated" style of programming. See [19,21].

Here are examples of inter-object specifications: "If the alarm of a watch is set, then whenever the current time reaches the alarm time, the beeper turns on". "Whenever the beeper is on, it beeps every two seconds".

These scenarios may be easy to describe and follow, but they cannot be executed together as a single system, unless the idea of *unification* is introduced. Unification means that events of the same type between the same objects, represent the same event. Since in the specifier's mind the operation of sending a text message (which is also an event) is the same in both scenarios, in order to execute what the programmer meant, these two events should be unified.

4.1 Live Sequence Charts

Live sequence charts, constitute a visual formalism for specifying multi-modal scenarios. An LSC can assert mandatory behavior (termed "hot"), possible behavior (termed "cold"), as well as forbidden behaviors and their combinations. The LSC language [9,21] extends message sequence charts (MSC) [26] (termed sequence diagrams in UML [40]). In an LSC, objects are represented by vertical lines, called lifelines, and messages between objects are represented by horizontal arrows between objects. Time advances along the vertical axis and messages entail an obvious partial ordering.

A cold monitored event (dashed blue arrows) is monitored, and if it occurs the next event in the partial order should be monitored or executed. A hot executed event (solid red arrows) means that the system should perform the event eventually. The LSC language also includes conditions, assertions, loops,

switch cases, time, symbolic instances, and several additional constructs. Figure 1 shows a typical LSC.

A set of LSCs can be executed using the *play-out* mechanism [21], which monitors at all times what must be done, what may be done and what is forbidden, and proceeds accordingly. This results in a full execution of the LSC specification using a naïve strategy, considering the current state and progressing by choosing arbitrarily from all possible next events to be triggered.

Since different fragmented scenarios are combined into a single functional executable system, there is a risk of contradictory requirements that can produce violations during execution.

Contradictions can subtly, arise from multiple scenarios. Finding an execution order that makes it possible to execute without violations requires considering future states when choosing an event. Techniques that use model-checking, planning and synthesis, have been developed, to look ahead and choose an execution order in a smarter fashion [18,32]. Synthesis can often be used to verify that the specification is valid or to exhibit inconsistencies [33].

4.2 Natural Language Play-in

In [14], we describe a natural language interface to the LSC formalism, named *NL-play-in*. The programmer can write in a controlled English, using terms, e.g., nouns, verbs, adjectives, that are relevant to the system being described, and reusing them in further requirements to allow unification during execution. The terms used become part of a growing *system model* that includes the system's objects, and their methods and properties.

The interface consists of a context-free grammar (CFG) bottom-up parser and a dialog system that help the programmer create both a system model and a set of LSC scenarios. The resulting system is fully executable. The controlled natural language accepts declarative requirement sentences. The parser includes semantic information for creating the LSCs, adding loops and conditions, and specifying which events should be monitored and which should be executed. The grammar is general: it analyzes all terms with the help of the WordNet dictionary [35], in order to determine whether a word is a noun, a verb, or an adjective and whether it is meant as an object, a method or a persistent property.

When a sentence is analyzed, terms that are not completely understood by the system are disambiguated using a quick-fix interface to the programmer. The word in question is marked with a squiggly line, and hovering over it with the mouse provides the programmer with additional information, and a list of possible solutions. Disambiguation includes resolving grammar problems and semantic issues.

Grammar problems include incomplete sentences, sentences without a verb, or sentences that are not part of the grammar. Semantic problems include phrases that can be either a target object or a parameter for a method. Semantic problems are more prevalent at early stages. As the requirements accumulate, and the programmer resolves problems, the information becomes part of the model and is used to resolve further ambiguities automatically.

The process of developing a system and its requirements is intermixed. Sometimes the programmer knows what the system should do, and only then considers what the system model will be, while often it is the other way around. The process continues throughout development, adding requirements and extending the model. Our method supports both development directions: creating the model as it becomes necessary when adding requirements, or adding the requirements for an existing model.

In one development direction, when a requirement is parsed, non-existing model parts, e.g., objects, classes, methods, and properties, are verified with the programmer as necessary, and the model is augmented with new model parts. Any addition of model parts is explicit, to verify that new parts are introduced only when they cannot be unified with existing parts. We call this process *model disambiguation*. Only after the model is complete, a new LSC is created that captures the requirement. Viewing the LSC allows the programmer to verify that the requirement was parsed correctly. Finally, the system created can be executed at any stage with the existing model and the LSCs.

In the other development direction, when a system model exists, it is possible for the programmer to specify requirements, and any references to the model parts are immediately understood, and require no additional user interaction. Many times the model is actually a non-behaving graphical user interface (GUI) of the final system. In this case, the model will be created automatically from the GUI objects.

When objects and methods are created automatically, they can be later replaced by graphical entities, or augmented with low-level code. For example, a button object may have a method *click* that is referenced in a scenario. The same *click* can later be implemented, to show and accept clicks from the user.

For a thorough guide we refer the reader to [14], in which an example of a wristwatch is described (also available in `http://www.weizmann.ac.il/mediawiki/playgo/index.php/Wristwatch_Example`).

The following CNL demonstrates the style of programming with NL-play-in:

> When the time value changes, if the time value equals the alarm value and the alarm state is enabled, the beeper turns on.
> When the beeper state changes to on, as long as the beeper state is on and two seconds elapse, the beeper beeps, the display mode may not change.
> When the user clicks any button, the beeper turns to off.

The fully executable diagrams that result automatically from these NL requirements are shown in Figures 1, 3 and 2.

It is possible to extend the model disambiguation to suggest connections between different terms according to word similarity or synonyms. For example, if "opening the radio", and "unlocking the radio", both appear, the parser can suggest to the programmer to make the connection between the methods, causing unification between these terms during execution.

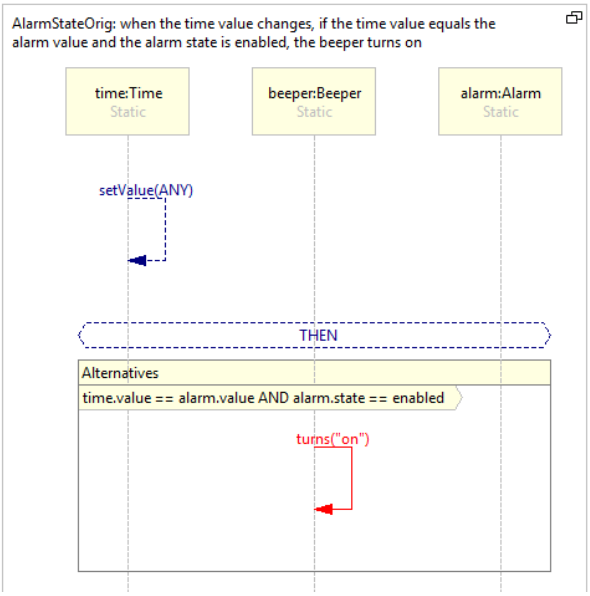


Fig. 1. A simple LSC created for the sentence "when the time value changes, if the time value equals the alarm value and the alarm state is enabled, the beeper turns on"

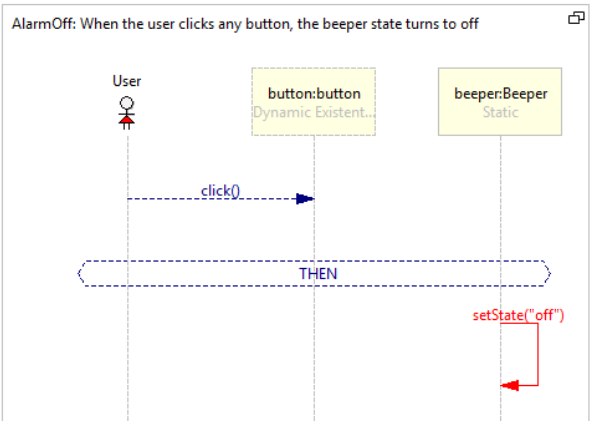


Fig. 2. The LSC created for the requirement "When the user clicks any button, the beeper turns to off"

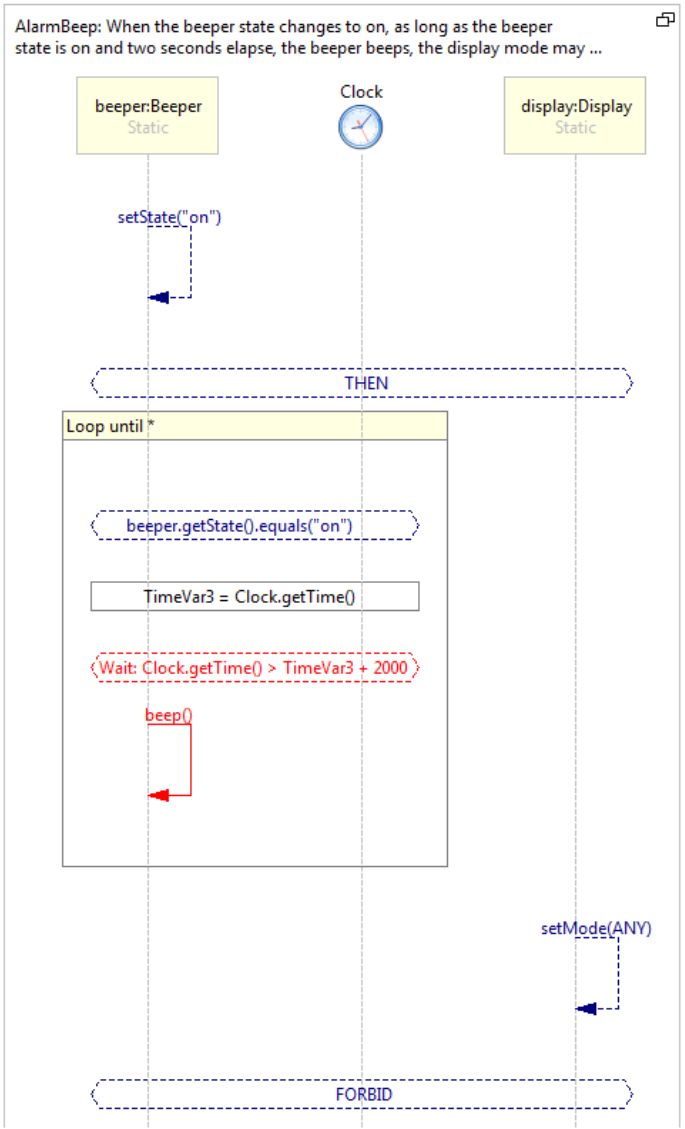


Fig. 3. The LSC created for the requirement "When the beeper state changes to on, as long as the beeper state is on and two seconds elapse, the beeper beeps, the display mode may not change."

4.3 Show and Tell

In [15] we describe an extension of the NL-play-in interface for LSCs, which combines it with the *play-in* method [17], by interweaving CNL and user interaction. In play-in, the programmer specifies scenarios by playing-them-in directly from

a graphical user interface (GUI) of the system being developed, similar to programming by demonstration [8]. *Show & tell* means that the programmer can combine writing in natural language with actual showing. Some parts of a scenario, for example, the `when-then`, or the `if`, are easier to write than to show, while other parts, like the `click` of a button, are easier to show. Show & tell also helps avoid typos and the necessity to specify the names or the operations with their exact terms.

While play-in is similar to programming by demonstration [8], show & tell is similar to the *put-that-there* method [4], and other multi-modal user interfaces, see `http://www.wisdom.weizmann.ac.il/~michalk/Projects/SaT/` for a demo. Experiments we have carried out [16] show that when the interface is a mouse and keyboard, show & tell combinations may not be more convenient for people who type quickly. Our experiments also expose the learnability of the NL-play-in approach.

4.4 Limitations and Future Work

NL-play-in meant for the high-level *programming* of reactive systems — dynamic systems that respond to events, depending on their current state [23]. The method, as part of the BP paradigm, supports incremental development of systems by continuously adding requirements. NL-play-in can help bridge the gap between the requirement engineering process and the development of the final system, and allow a shorter life cycle. NL-play-in is suitable for interweaving fragmented requirements, including negative requirements, and can be combined with other programming styles, e.g., statecharts [2]. The systems we have already created include a wristwatch, a chess game, a baby monitor, and parts of an ATM machine.

The programmer's identity can range from the professional to the end-user, and the vocabulary and level-of-detail can change according to the programmer's needs. When programming the behavior of a robot, pertaining to where it heads, what it sees, or what directions it receives, the terms will be different from the case of programming at the level of the robot moving body parts.

For modifying existing systems, e.g., augmenting existing behavior with additional requirements or forbidden behaviors, the programmer should be familiar with the details of the model. However, even if the model is less known, show & tell can help the programmer refer directly to relevant objects and terms.

The natural language interface can be improved substantially, with, e.g., reference resolutions, verbal shortcuts and the use of synonyms, all of which can make the writing more friendly, as is the case with ACE [11] or MOOIDE [7].

Another challenge in using natural language for programming is its assimilation. For non-programmers this requires developing teaching methods. The BPJ library [22] lets expert programmers use BP concepts in their own programming environment, supporting a gradual transition from procedural programming to BP and later to natural language programming.

When broader groups of people will program, software engineering activities will probably broaden too, and will require better visualization and navigation

methods; some research on these approaches in the content of LSCs has already started [24], and this work can be adopted to the NL interface too.

Another consideration is requirement coverage. The one responsibility of the programmer is to enter the requirements. However, since requirements need to cover multiple possibilities and many system states, he may require help in considering all possibilities. Such support could include supplying many views that will help him understand the system. For example, complex systems may benefit from requirements analysis by other formats than natural language, e.g. tabular visualization, that will help the programmer see the bigger picture and uncover gaps in the specification. Precise documentation in software engineering [36] becomes extremely relevant when programming in natural language because in a way the documentation becomes the final program.

It is not only the understanding of how to program, but also the need to program that is still elusive. What will new programmers want to program? Prior to the introduction of smartphones, few people thought of creating their own applications. However, at present there is an astonishing variety of applications, and their number is growing rapidly. We hypothesize that as technology comes to play a much larger role in people's lives, the need to program or re-program such systems will increase dramatically. This, in fact, constitutes a major part of the motivation for PiCNL.

Acknowledgments. The research was supported in part by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant to DH from the European Research Council (ERC) under the European Community's FP7 Programme.

References

1. AECMA Official Site, `http://www.asd-ste100.org`
2. Barak, D., Harel, D., Marelly, R.: InterPlay: Horizontal Scale-Up and Transition to Design in Scenario-Based Programming. IEEE Trans. Soft. Eng. 32(7), 467–485 (2006)
3. Begel, A., Graham, S.: Spoken programs. In: Proc. IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC 2005), pp. 99–106 (2005)
4. Bolt, R.A.: "Put-that-there": Voice and Gesture at the Graphics Interface. SIGGRAPH Comput. Graph. 14(3), 262–270 (1980)
5. Bryant, B.R., Lee, B.-S.: Two-Level Grammar as an Object-Oriented Requirements Specification Language. In: Proc. 35th Annual Hawaii Int. Conf. on System Sciences, HICSS 2002, pp. 280–289 (2002)
6. Cabral, G., Sampaio, A.: Formal Specification Generation from Requirement Documents. Electron. Notes Theor. Comput. Sci. 195, 171–188 (2008)
7. Cypher, A., Dontcheva, M., Lau, T., Nichols, J.: No Code Required: Giving Users Tools to Transform the Web. Morgan Kaufmann Publishers Inc. (2010)

8. Cypher, A., Halbert, D.C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B.A., Turransky, A. (eds.): Watch What I Do: Programming by Demonstration. MIT Press (1993)
9. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design 19(1), 45–80 (2001)
10. Dijkstra, E.W.: On the Foolishness of "Natural Language Programming". In: Gerhart, S.L., et al. (eds.) Program Construction. LNCS, vol. 69, pp. 51–53. Springer, Heidelberg (1979)
11. Fuchs, N.E., Schwitter, R.: Attempto Controlled English (ACE). In: Proc. 1st Int. Workshop on Controlled Language Applications, pp. 124–136 (1996)
12. Geller, T.: Talking to Machines. Commun. ACM 55(4), 14–16 (2012)
13. Giganto, R.T.: A Three-Level Algorithm for Generating Use Case Specifications. In: Proc. Software Innovation and Engineering New Zealand Workshop, SIENZ 2007 (2007)
14. Gordon, M., Harel, D.: Generating executable scenarios from natural language. In: Gelbukh, A. (ed.) CICLing 2009. LNCS, vol. 5449, pp. 456–467. Springer, Heidelberg (2009)
15. Gordon, M., Harel, D.: Show-and-Tell Play-In: Combining Natural Language with User Interaction for Specifying Behavior. In: Proc. IADIS Interfaces and Human Computer Interaction (IHCI 2011), pp. 360–364 (2011)
16. Gordon, M., Harel, D.: Evaluating a Natural Language Interface for Behavioral Programming. In: Proc. IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC 2012), pp. 17–20 (2012)
17. Harel, D.: From Play-In Scenarios To Code: An Achievable Dream. Computer 34(1), 53–60 (2001)
18. Harel, D.: Playing with Verification, Planning and Aspects: Unusual Methods for Running Scenario-Based Programs. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 3–4. Springer, Heidelberg (2006)
19. Harel, D.: Can Programming be Liberated, Period? Computer 41(1), 28–37 (2008)
20. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-Out of Behavioral Requirements. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 378–398. Springer, Heidelberg (2002)
21. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine. Springer (2003) (See also paper in Software and System Modeling 2(2), 82–107 (2003))
22. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Commun. ACM 55(7), 90–100 (2012)
23. Harel, D., Pnueli, A.: On the Development of Reactive Systems. In: Logics and Models of Concurrent Systems, pp. 477–498. Springer, New York (1985)
24. Harel, D., Segall, I.: Visualizing Inter-Dependencies between Scenarios. In: Proc. 4th ACM Symp. on Software Visualization (SoftVis 2008), pp. 145–153 (2008)
25. Hearst, M.A.: "Natural" Search User Interfaces. Commun. ACM 54(11), 60–67 (2011)
26. ITU: International Telecommunication Union. Recommendation Z.120: Message Sequence Chart (MSC). Technical report (1996)
27. Kim, S.-H., Jeon, J.W.: Programming LEGO Mindstorms NXT with Visual Programming. In: Proc. Int. Conf. on Control, Automation and Systems, ICCAS 2007, pp. 2468–2472 (2007)
28. Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Translating Structured English to Robot Controllers. Advanced Robotics Special Issue on Selected Papers From IROS 2007 22(12), 1343–1359 (2008)

29. Kuhn, T., Fuchs, N.E. (eds.): CNL 2012. LNCS, vol. 7427. Springer, Heidelberg (2012)
30. Lei, T., Long, F., Barzilay, R., Rinard, M.: From Natural Language Specifications to Program Input Parsers. In: Proc. Annual Meeting Assoc. for Computational Linguistics, ACL 2013 (2013)
31. Liu, H., Lieberman, H.: Toward a Programmatic Semantics of Natural Language. In: Proc. IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC 2004), pp. 281–282 (2004)
32. Maoz, S., Harel, D., Kleinbort, A.: A Compiler for Multimodal Scenarios: Transforming LSCs into AspectJ. ACM Trans. Softw. Eng. Methodol. 20(4), 18 (2011)
33. Maoz, S., Sa'ar, Y.: Two-Way Traceability and Conflict Debugging for AspectLTL Programs. In: Leavens, G.T., Chiba, S., Tanter, É. (eds.) Transactions on AOSD X. LNCS, vol. 7800, pp. 39–72. Springer, Heidelberg (2013)
34. Mihalcea, R., Liu, H., Lieberman, H.: NLP (Natural Language Processing) for NLP (Natural Language Programming). In: Gelbukh, A. (ed.) CICLing 2006. LNCS, vol. 3878, pp. 319–330. Springer, Heidelberg (2006)
35. Miller, G.A., Beckwith, R., Fellbaum, C., Gross, D., Miller, K.: Introduction to WordNet: An On-line Lexical Database (1993), `http://wordnet.princeton.edu/`
36. Parnas, D.L.: Precise Documentation: The key to Better Software. In: The Future of Software Engineering, pp. 125–148. Springer (2011)
37. Resnick, M., et al.: Scratch: Programming for All. Comm. of the ACM 52(11), 60–67 (2009)
38. Shneiderman, B.: Designing the User Interface: Strategies for Effective Human-Computer Interaction. Addison-Wesley Longman (1986)
39. Sun, Y., Leigh, J., Johnson, A., Lee, S.: *Articulate*: A Semi-automated Model for Translating Natural Language Queries into Meaningful Visualizations. In: Taylor, R., Boulanger, P., Krüger, A., Olivier, P. (eds.) SG 2010. LNCS, vol. 6133, pp. 184–195. Springer, Heidelberg (2010)
40. UML. Unified Modeling Language Superstructure, v2.1.1. Technical Report formal/2007-02-03, Object Management Group (2007)
41. Winograd, T.: Understanding Natural Language. Cognitive Psychology 3(1), 1–191 (1972)
42. Wong, Y.W., Mooney, R.J.: Learning Synchronous Grammars for Semantic Parsing with Lambda Calculus. In: Proc. 45th Annual Meeting of the Assoc. for Computational Linguistics, ACL 2007 (2007)
43. Zhong, H., Zhang, L., Xie, T., Mei, H.: Inferring Resource Specifications from Natural Language API Documentation. In: Proc. IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2009), pp. 307–318 (2009)

Assembly Theories for Communication-Safe Component Systems*

Rolf Hennicker[1], Alexander Knapp[2], and Martin Wirsing[1]

[1] Ludwig-Maximilians-Universität München
{hennicke,wirsing}@pst.ifi.lmu.de
[2] Universität Augsburg
knapp@informatik.uni-augsburg.de

Dedicated to Joseph Sifakis

Abstract. We propose an abstract notion of an assembly theory that formalizes rudimentary requirements for systems of interacting components. Among these are a composition operator for assemblies, a communication-safety predicate to express the absence of communication errors, a refinement relation for assemblies, and a packing operation to encapsulate assemblies into components thus allowing hierarchical system constructions. We establish laws that must be satisfied by any concrete assembly theory in order to support compositionality of communication-safety, of encapsulation and of refinement. Moreover, refinement must behave well w.r.t. communication-safety and encapsulation. As a concrete instance we investigate a modal assembly theory using modal I/O-interfaces (MIOs) for modeling observable component behaviors and MIOs with possible error states (indicating communication errors) for modeling assembly behaviors. We show that all rules of an assembly theory are satisfied by modal assemblies, in particular the compositionality requirements hold.

1 Introduction

In his recent article [20], Joseph Sifakis advocates "rigorous system design" to build systems of guaranteed quality. The abstract principles of component-based design and correctness-by-construction are two main ingredients of his approach. He writes that "components are essential for enhanced productivity and correctness through reuse and architectures" and aims at "theory and rules for building complex designs ... by composing properties of simpler designs." As a concrete instance of these principles Joseph and his research group at Verimag have developed the BIP component framework (see, e.g., [3,20]). BIP allows the modeling of composite, hierarchically structured systems from atomic components characterized by their behavior and their interface.

Several other approaches have been proposed for specifying structural as well as behavioral aspects of components and their interfaces; an overview of a collection of

* This work has been partially sponsored by the European Union under the FP7-project ASCENS, 257414.

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 145–160, 2014.

component frameworks, that have been applied to a common component modeling example, is given in [19]. The specification of networks of components with arbitrarily (finitely) many members and their interactions is supported by BIP [3,13] as well as by other formalisms for modeling distributed component systems, like CFSMs [7], team automata [8], component-interaction automata [9], or modal assemblies [15].

Abstract principles for the construction of component-based concurrent systems have been investigated and formalized as so-called interface theories by de Alfaro and Henzinger [11,12]. As in Joseph's approach, compositionality requirements are at the heart of these formalisms expressing, e.g., the principles of incremental design and independent implementability at an abstract level. Interfaces for complex components are constructed from simpler ones by interface composition. The result of an interface composition yields again an interface which intuitively describes the visible (black-box) behavior of a composite component. However, in spite of their importance for distributed component systems, there is no abstract formalization of the interaction behavior of component networks with arbitrarily (finitely) many members.

In this work we investigate networks of interacting components in the spirit of Joseph's correctness-by-construction approach and de Alfaro's and Henzinger's interface theories: we develop novel compositional principles for the safe interaction of networks and formalize them by the abstract notion of assembly theory; moreover, we present so-called modal assemblies as a concrete instance of an assembly theory.

An assembly theory formalizes basic requirements for systems of interacting components. In particular, an assembly theory comprises a composition operator for assemblies, a communication-safety predicate for expressing the absence of communication errors, a refinement relation for assemblies, and a packing operation for encapsulating assemblies into components thus allowing hierarchical system constructions. Any assembly theory must satisfy compositionality and compatibility laws for communication-safety, encapsulation and refinement.

We instantiate our framework by a novel concrete assembly theory which uses an extension of MIOs (modal I/O-transition systems; see [17]) to model interface and assembly behaviors. Modal assemblies have already been considered in [15], but the new approach is a significant enhancement. We consider now assembly composition, a new definition of assembly behaviors which explicitly takes into account communication-errors and a new, much more flexible refinement notion for modal assemblies. As a consequence, we get novel results for compositionality of the communication-safety property and for assembly refinement. Moreover, [15] does not define a rigorous, abstract meta-theory for assemblies but provides only some first ideas in that direction.

Outline of the paper. In Sect. 2 we develop the general concepts and laws of an assembly theory. Sect. 3 summarizes the basic notions of modal I/O-transition systems needed in Sect. 4 to build a modal assembly theory as an instance of our abstract framework. Finally, in Sect. 5, we finish with some concluding remarks.

Personal Note. The third author has known Joseph for several years and collaborates with him and his Verimag research group since 2010. Initial ideas for the cooperation started in 2008 at a workshop of the EC Coordinated Action INTERLINK. Joseph gave a keynote speech on Rigorous System Design while MW was coordinating a Working

Group on software-intensive systems and was giving a talk about Ensemble Engineering. Two years later both (together with Ugo Montanari, Rocco De Nicola, and others) teamed up for planning a joint EU project on the systematic construction of autonomous systems. The project proposal was successful: the FP7 Integrated Project ASCENS [2] on "Autonomic Service-Component Ensembles" is coordinated by MW; Joseph and his group are responsible for the work package on "Correctness of Service Components and Service Component Ensembles". Design techniques ensuring correctness-by-construction play a main role in ASCENS; current results comprise an extension of BIP for modeling dynamic architectures [6] and a novel implementation of the D-Finder tool for compositional deadlock detection in concurrent systems [5].

Working with Joseph is an excellent experience; we admire his deep insights and technical precision, and are looking forward to many further inspiring exchanges.

2 Assembly Theories

We develop a general framework that is intended to capture and formalize rudimentary properties that we believe should be satisfied by any concrete framework for distributed component systems to form a reasonable assembly theory. For this purpose we will consider some abstract domains, operators, relations and laws that altogether form a meta-theory for assemblies. Our starting assumption is that an assembly consists of a finite set of components which can interact. Since components are encapsulated units, we represent them by interface specifications (shortly called interfaces). Therefore we consider an assembly as a (non-empty) finite set of interface specifications which fit syntactically together according to some composability criterion.[1] In the following we are interested to collect a number of general properties that must be satisfied by a concrete framework to form an assembly theory.

As a basis we assume given a class $\mathcal{F}$ of *interface specifications* together with a reflexive and transitive *interface refinement* relation $\preccurlyeq \subseteq \mathcal{F} \times \mathcal{F}$. For two interfaces F and G, $F \preccurlyeq G$ means that F is a refinement of the interface specification G. We denote by $\wp_{\mathrm{fin}}(\mathcal{F})$ the class of the finite subsets of $\mathcal{F}$. In general, not all elements of $\wp_{\mathrm{fin}}(\mathcal{F})$ form assemblies. Usually there are some syntactic composability conditions required for the members of an assembly. Hence, any assembly theory must first define a particular class $\mathcal{A} \subseteq \wp_{\mathrm{fin}}(\mathcal{F})$ whose elements form valid interface assemblies. We require that any assembly must have at least one element, that any interface induces a (singleton) assembly and that non-empty subsets of assemblies are assemblies as well. These conditions are stated in the first item of Def. 1. In order to combine assemblies to larger ones we require a partial assembly composition operator $\boxtimes$ which is defined, if and only if, the union of two assemblies is an admissible assembly again. We require that an assembly theory must offer a packing operation $pack : \mathcal{A} \to \mathcal{F}$, which allows us to encapsulate an assembly into a component interface by hiding the internals of the assembly. Thus hierarchical assemblies can be constructed by using packed assemblies as their components. To address behavioral compatibility of the interacting members of an assembly, we introduce a communication-safety predicate $cs \subseteq \mathcal{A}$ on assemblies.

[1] To be as abstract as possible, we deliberately take this simplified view not considering other ingredients like ports, connectors etc.

Similarly to interfaces, an assembly theory must also offer a reflexive and transitive refinement relation for assemblies, denoted by $\sqsubseteq \subseteq \mathcal{A} \times \mathcal{A}$.

Some crucial properties relating composition, encapsulation, communication-safety, and refinement are required for any concrete assembly theory. The properties (A1), (A2) and (A3) specify in their (a) part straightforward rules for singleton assemblies. Their (b) and (c) parts state compositionality requirements: (A1)(c) states that communication-safe assemblies can be packed piecewise if the two components obtained from packing A and B are communication-safe. At the end the still visible boundary of the single packed assemblies must be hidden by applying another pack.

(A2)(b) deals with compositionality of communication-safety. If two communication-safe assemblies A and B are combinable, then for the result to be communication-safe it suffices to check that the two components obtained from packing A and B are communication-safe. Hence, once A and B are locally "fine", it only remains to consider the interactions on the boundary between A and B. This important property supports also efficient communication-safety checking, since in concrete applications it is often possible to consider minimized versions of $pack(A)$ and $pack(B)$. (A3)(b) and (c) formulate a compositionality requirement for refinement of communication-safe assemblies. In part (b) local refinements are given and the other assumptions are the same as for (A1)(c) and (A2)(b). (A4) is straightforward requiring that encapsulation of communication-safe assemblies, which are in refinement relation, leads to interfaces which are also in refinement relation. Another important property is expressed by (A5) guaranteeing that refinements of communication-safe assemblies are communication-safe.

Definition 1 (Assembly theory). *An* assembly theory $(\mathcal{A}, \boxtimes, pack, cs, \sqsubseteq)$ *over* $(\mathcal{F}, \preccurlyeq)$ *is given by*

- *a class* $\mathcal{A} \subseteq \wp_{\mathrm{fin}}(\mathcal{F})$ *of* assemblies*, such that*
  1. $\emptyset \notin \mathcal{A}$*,*
  2. *for all* $F \in \mathcal{F}, \{F\} \in \mathcal{A}$*, and*
  3. $\mathcal{A}$ *is closed under the formation of non-empty subsets, i.e., if* $A \in \mathcal{A}$ *and* $\emptyset \neq B \subseteq A$*, then* $B \in \mathcal{A}$*;*
- *a partial* assembly composition *operator* $\boxtimes : \mathcal{A} \times \mathcal{A} \rightharpoonup \mathcal{A}$ *defined by* $A \boxtimes B = A \cup B$ *if* $A \cup B \in \mathcal{A}$*, undefined otherwise;*[2]
- *an* encapsulation *operation* $pack : \mathcal{A} \to \mathcal{F}$*,*
- *a* communication-safety *predicate* $cs \subseteq \mathcal{A}$ *(we will write* $cs(A)$ *for* $A \in cs$*), and*
- *a reflexive and transitive* assembly refinement *relation* $\sqsubseteq \subseteq \mathcal{A} \times \mathcal{A}$*,*

such that for all $F, G \in \mathcal{F}$ *and* $A, B, A_1, A_2, B_1, B_2 \in \mathcal{A}$ *the following holds:*

A1. Compositionality of encapsulation:
  (a) $pack(\{F\}) = F$*.*
  (b) If $A \boxtimes B$ *is defined, then* $\{pack(A)\} \boxtimes \{pack(B)\}$ *is defined.*

[2] $\boxtimes$ is commutative in the sense that for all $A, B \in \mathcal{A}$, if $A \boxtimes B$ is defined then $B \boxtimes A$ is defined and $A \boxtimes B = B \boxtimes A$. $\boxtimes$ is also associative in the sense that for all $A, B, C \in \mathcal{A}$, if $A \boxtimes B$ and $(A \boxtimes B) \boxtimes C$ are defined, then $B \boxtimes C$ and $A \boxtimes (B \boxtimes C)$ are defined and $(A \boxtimes B) \boxtimes C = A \boxtimes (B \boxtimes C)$. This follows from the subset-closedness condition.

(c) *If* $A \boxtimes B$ *is defined, and if* $cs(A)$, $cs(B)$ *and* $cs(\{pack(A)\} \boxtimes \{pack(B)\})$, *then* $pack(A \boxtimes B) = pack(\{pack(A)\} \boxtimes \{pack(B)\})$.

A2. Compositionality of communication-safety:

(a) $cs(\{F\})$.

(b) *If* $A \boxtimes B$ *is defined, and if* $cs(A)$, $cs(B)$ *and* $cs(\{pack(A)\} \boxtimes \{pack(B)\})$, *then* $cs(A \boxtimes B)$.

A3. Compositionality of refinement:

(a) *If* $F \preccurlyeq G$, *then* $\{F\} \sqsubseteq \{G\}$.

(b) *If* $B_1 \boxtimes B_2$ *is defined, and if* $A_i \sqsubseteq B_i$ *for* $i \in \{1, 2\}$, *then* $A_1 \boxtimes A_2$ *is defined.*

(c) *If* $B_1 \boxtimes B_2$ *is defined, and if* $cs(B_1)$, $cs(B_2)$, $cs(\{pack(B_1)\} \boxtimes \{pack(B_2)\})$, *and if* $A_i \sqsubseteq B_i$ *for* $i \in \{1, 2\}$, *then* $A_1 \boxtimes A_2 \sqsubseteq B_1 \boxtimes B_2$.

A4. Preservation of refinement by encapsulation:
If $A \sqsubseteq B$ *and* $cs(B)$, *then* $pack(A) \preccurlyeq pack(B)$.

A5. Preservation of communication-safety by refinement:
If $A \sqsubseteq B$ *and* $cs(B)$, *then* $cs(A)$.

From the laws of an assembly theory it follows that communication-safe assemblies can be constructed in an incremental manner, i.e. by enlarging the assembly by one interface at a time, each time checking that the packed assembly up to now is communication-safe with the additional interface.

Incremental design: Let $A \in \mathcal{A}$ be an assembly and let $F \in \mathcal{F}$ such that $A \cup \{F\} \in \mathcal{A}$. If $cs(A)$ and $cs(\{pack(A), F\})$, then $cs(A \cup \{F\})$.

Similarly, the following law of independent implementability is also a consequence of the properties of an assembly theory.

Independent implementability: Let $A, B \in \mathcal{A}$ such that $A \sqsubseteq B$ and let $F, G \in \mathcal{F}$ such that $F \preccurlyeq G$ and $B \cup \{G\} \in \mathcal{A}$. If $cs(B)$ and $cs(\{pack(B), G\})$, then $A \cup \{F\} \sqsubseteq B \cup \{G\}$.

The idea to consider assemblies as sets of components, automata, or interfaces is present in many approaches in the literature; see e.g. CFSMs [7], the BIP framework [3,13], team automata [8], component-interaction automata [9], and modal assemblies [15]. So it is an interesting question to what extent the concepts and laws from above appear in the different frameworks. In [7] communication protocols are studied based on collections of communicating finite state machines. Communication is asynchronous via queues and the focus there is particularly on communication properties, like specified reception (and how to check this), which could be used as a communication-safety predicate in our sense. In the BIP framework systems of components are considered together with particular interaction models, which are not (yet) incorporated into our notion of an assembly. BIP provides, as required for an assembly theory, a composition operator, it deals with certain properties of systems, like interaction safety, and focuses on compositionality results much in the spirit of an assembly theory. Compositionality results are also studied in [8] for systems of reactive transition systems (playing the role of interfaces). Our notion of an assembly could be instantiated by the concept of a composable system, and communication-safety by the notion of a compatible system. Different synchronization strategies are applicable and interpreted

via team automata. For the case of the synchronous product, [8] shows, in Cor. 9, a compositionality result, which is very similar to property (A2) required for assembly theories. In [9] systems of composable component-interaction automata are used as assemblies. [9] focuses merely on substitutability of components which is very much related to our principle of independent implementability. Communication-safety is not an issue there. In [15] both communication-safety and refinement are studied using modal I/O-transition systems (MIOs) as interfaces and systems of connectable MIOs as assemblies. We have defined there an ad hoc refinement relation for assemblies requiring that the interfaces of abstract and concrete assemblies must be related by pairwise interface refinements. This refinement relation is, however, far too restrictive and we will propose a more flexible one in Sect. 4.3. We will see that the new assembly refinement relation needs a reconsideration of the behavior model for assemblies indicating communication errors such that the property (A5) for preservation of communication-safety is satisfied.

3 Modal I/O-Transition Systems and Weak Modal Refinement

We give a short introduction to modal I/O-transition systems (MIOs) and their refinement. MIOs will be used hereafter as a basic framework to build a modal assembly theory. Modal transition systems (MTS) have been introduced in [18] and later extended by Input/Output alphabets in [17]. As a verification tool we use the MIO-workbench presented first in [4]. We have chosen MIOs as our basic formalism since they allow us to distinguish between transitions which are optional (*may*) or mandatory (*must*) and thus support very well loose specifications and refinements. Like any labeled transition system also MIOs model actions by labels on the transitions. We distinguish four kinds of actions and hence labels: input labels, output labels, communication labels and the internal action τ. In contrast to Larsen et al. [17], internal actions are not explicitly named here but represented by the invisible action τ and communication labels are added in our approach to model synchronous communication.

Each MIO is based on an *I/O-labeling* $L = (I_L, O_L, T_L)$ consisting of pairwise disjoint sets of *input labels* I_L, *output labels* O_L, and *communication labels* T_L, such that $\tau \notin I_L \cup O_L \cup T_L$. We write $\bigcup L$ for the set $I_L \cup O_L \cup T_L$ of all labels of L. The I/O-labeling of a MIO will be pictorially shown on its frame. For easier readability, input labels will be suffixed with "?" and output labels with "!" on the transitions.

A *modal I/O-transition system* $M = (L_M, S_M, s_{0,M}, \dashrightarrow_M, \longrightarrow_M)$ consists of an I/O-labeling $L_M = (I_M, O_M, T_M)$, a set of *states* S_M, an *initial state* $s_{0,M} \in S_M$, a *may-transition relation* $\dashrightarrow_M \subseteq S_M \times (\bigcup L_M \cup \{\tau\}) \times S_M$, and a *must-transition relation* $\longrightarrow_M \subseteq \dashrightarrow_M$, i.e. any must-transition is also a may-transition. A MIO M is called an *implementation* if all transitions are must-transitions, i.e. $\longrightarrow_M = \dashrightarrow_M$. The set of the *reachable states* from the initial state $s_{0,M}$ of M w.r.t. may-transitions is denoted by $\mathscr{R}(M)$. For $l \in \bigcup L_M \cup \{\tau\}$, we write $s \overset{l}{\dashrightarrow}_M s'$ for $(s, l, s') \in \dashrightarrow_M$ and $s \xrightarrow{l}_M s'$ for $(s, l, s') \in \longrightarrow_M$. Since $\longrightarrow_M \subseteq \dashrightarrow_M$, $s \xrightarrow{l}_M s'$ implies $s \overset{l}{\dashrightarrow}_M s'$.

We consider two operators on MIOs, synchronous composition and hiding of communication labels.

Synchronous composition. Two MIOs M, N with labelings $L_M = (I_M, O_M, T_M)$ and $L_N = (I_N, O_N, T_N)$ resp. are composable, if their labels overlap only on complementary

types, i.e. $\bigcup L_M \cap \bigcup L_N = (I_M \cap O_N) \cup (I_N \cap O_M)$. Hence, whenever a label is shared, then it is either an input label of the first MIO and an output label of the second or conversely. The synchronous composition of two composable MIOs M and N is denoted by $M \otimes^{\text{sy}} N$ and defined as the usual product of automata such that transitions with shared actions are performed (only) simultaneously. After composition the shared labels become communication labels. A synchronization transition in $M \otimes^{\text{sy}} N$ is a must-transition only if both of the single synchronizing transitions are must-transitions.

Composability and synchronous composition are straightforwardly extended to finite sets of MIOs: A non-empty finite set $A = \{M_1, \ldots, M_n\}$ of MIOs is composable, if the single MIOs M_i are pairwise composable. Then labels of each M_i can only be shared with at most one other MIO M_j ($j \neq i$). Since the synchronous composition is commutative and associative (up to a bijection between states) the synchronous composition of A can be inductively defined by $\bigotimes^{\text{sy}} A = M_1 \otimes^{\text{sy}} \ldots \otimes^{\text{sy}} M_n$.

Hiding. Hiding is used to build abstractions of labeled transition systems. Usually, hiding is obtained by considering a specified set of previously visible actions as invisible. In the case of MIOs we use a simple, uniform hiding operator which makes communication (obtained by previous compositions) invisible.

Formally, the *hiding* of communication labels of an I/O-labeling $L = (I_L, O_L, T_L)$ is given by $L\xi = (I_L, O_L, \emptyset)$ and the *hiding* of communications labels of a MIO M, denoted by $M\xi$, is defined by moving all communication labels on the transitions of M to τ.

Weak modal refinement. The basic idea of modal refinement is that required (*must*) transitions of an abstract specification must also occur in the concrete specification. Conversely, allowed (*may*) transitions of the concrete specification must be allowed by the abstract specification, but can be omitted in the concrete one. We will use the weak form of modal refinement introduced by Hüttel and Larsen in [16] which supports observational abstraction, i.e., internal transitions can be dropped and inserted as long as the modalities and the simulation relation are preserved. Their definition assumes distinguished sets of external and internal actions; here, external actions are given by the input, output and communication labels of MIOs and the internal actions are given by the single label τ. Since communication labels are considered to be visible, they must be respected in the same way as input/output labels. This is important when we consider assembly refinement which should respect communications.

For denoting sequences of transitions that abstract from silent transitions, we use the following notation. Let M be a MIO with I/O-labeling $L_M = (I_M, O_M, T_M)$.

1. We write $s \overset{\widehat{\tau}}{\dashrightarrow}_M s'$ if there is a (possibly empty) sequence of may-transitions from s to s' all labeled by τ, and likewise for must-transitions. For $l \in \bigcup L_M$, we write $s \overset{\widehat{l}}{\dashrightarrow}_M s'$ for $s \overset{\widehat{\tau}}{\dashrightarrow}_M r \overset{l}{\dashrightarrow}_M t \overset{\widehat{\tau}}{\dashrightarrow}_M s'$, and likewise for must-transitions.
2. To express that a sequence of transitions is obtained by an arbitrary order of single transitions involving only labels of a given set $X \subseteq \bigcup L_M$ or τ, we write $s \overset{\widehat{X}}{\dashrightarrow}_M s'$ for $s \overset{\widehat{l_1}}{\dashrightarrow}_M \cdots \overset{\widehat{l_n}}{\dashrightarrow}_M s'$ with $n \geq 0$ and $l_1, \ldots, l_n \in X$.

Let M and N be MIOs with the same I/O-labeling. A relation $R \subseteq S_M \times S_N$ is a *weak modal refinement relation* between M and N if for all $(s_M, s_N) \in R$ and for all $l \in \bigcup L_M = \bigcup L_N$ the following holds:

R1. $s_N \xrightarrow{l}_N s'_N \Rightarrow \exists s'_M \in S_M \,.\, s_M \xrightarrow{\widehat{l}}_M s'_M \wedge (s'_M, s'_N) \in R.$
R2. $s_N \xrightarrow{\tau}_A s'_N \Rightarrow \exists s'_M \in S_M \,.\, s_M \xrightarrow{\widehat{\tau}}_M s'_M \wedge (s'_M, s'_N) \in R.$
R3. $s_M \overset{l}{\dashrightarrow}_M s'_M \Rightarrow \exists s'_N \in S_N \,.\, s_N \overset{\widehat{l}}{\dashrightarrow}_N s'_N \wedge (s'_M, s'_N) \in R.$
R4. $s_M \overset{\tau}{\dashrightarrow}_M s'_M \Rightarrow \exists s'_N \in S_N \,.\, s_N \overset{\widehat{\tau}}{\dashrightarrow}_N s'_N \wedge (s'_M, s'_N) \in R.$

Recall that any must-transition is also a may-transition. Hence, by (R3) and (R4), must-transitions in M must be allowed by corresponding may-transitions in N.

M is a *weak modal refinement* of N, written $M \leq^*_{\mathrm{m}} N$, if there exists a weak modal refinement relation R between M and N such that $(s_{0,M}, s_{0,N}) \in R$. If all transitions of M and N are must-transitions, weak modal refinement coincides with weak bisimulation. Obviously, weak modal refinement is reflexive and transitive. Two MIOs M and N are *equivalent*, written $M \approx^*_{\mathrm{m}} N$, if $M \leq^*_{\mathrm{m}} N$ and $N \leq^*_{\mathrm{m}} M$, i.e. M co-simulates N.

Weak modal refinement is preserved by synchronous composition and by the hiding operator. The first statement extends the compositionality result of [16] to the case of products with visible communication labels. The second statement follows from the fact that any weak modal refinement relation witnessing $M \leq^*_{\mathrm{m}} N$ is also a weak modal refinement relation witnessing $M\xi \leq^*_{\mathrm{m}} N\xi$.

Proposition 1 (Preservation of weak modal refinement).

1. *For $i = 1, 2$, let M_i, N_i be MIOs such that $M_i \leq^*_{\mathrm{m}} N_i$ and let M_1 and M_2 (and hence N_1 and N_2) be composable. Then $M_1 \otimes^{\mathrm{sy}} M_2 \leq^*_{\mathrm{m}} N_1 \otimes^{\mathrm{sy}} N_2$.*
2. *Let M, N be MIOs such that $M \leq^*_{\mathrm{m}} N$. Then $M\xi \leq^*_{\mathrm{m}} N\xi$.* □

4 A Modal Assembly Theory

4.1 Modal Interfaces and Modal Assemblies

A *modal interface* F is given by a modal I/O-transition system (MIO), whose I/O-labeling $L_F = (I_F, O_F, \emptyset)$ does not show communication labels. The labeling restriction to the empty set of communication labels reflects the blackbox characteristics of modal interfaces abstracting from communication. The class of all modal interfaces is denoted by $\mathcal{F}^{\mathrm{m}}$. The notion of weak modal refinement (see Sect. 3) is directly applicable to define refinement for modal interfaces. A modal interface F *refines* a modal interface G, written $F \preccurlyeq^{\mathrm{m}} G$, if $F \leq^*_{\mathrm{m}} G$.[3]

We will now build a modal assembly theory over $(\mathcal{F}^{\mathrm{m}}, \preccurlyeq^{\mathrm{m}})$. Only those (finite) sets of modal interfaces are allowed to form a modal assembly, whose members are pairwise composable; see Sect. 3.

[3] The notation distinguishes between $\preccurlyeq^{\mathrm{m}}$ and $\leq^*_{\mathrm{m}}$ (Sect. 3), since $\preccurlyeq^{\mathrm{m}}$ is only applicable if the labelings do not show communication labels.

Definition 2 (Modal assemblies). *The class* $\mathcal{A}^{\mathrm{m}} \subseteq \wp_{\mathrm{fin}}(\mathcal{F}^{\mathrm{m}})$ *of* modal assemblies *consists of all non-empty, composable (and hence finite) subsets* $A \subseteq \mathcal{F}^{\mathrm{m}}$. □

This satisfies the requirements of Def. 1, since (1) $\emptyset \notin \mathcal{A}^{\mathrm{m}}$, (2) $\{F\} \in \mathcal{A}^{\mathrm{m}}$ for all $F \in \mathcal{F}^{\mathrm{m}}$, and (3) $\mathcal{A}^{\mathrm{m}}$ is closed under non-empty subsets. The composition of two modal assemblies A and B is denoted by $A \boxtimes^{\mathrm{m}} B$. Hence $A \boxtimes^{\mathrm{m}} B = A \cup B$ if $A \cup B \in \mathcal{A}^{\mathrm{m}}$ and undefined otherwise. Fig. 1 show the pictorial representation of a modal assembly consisting of three pairwise composable interfaces F_1, F_2, and F_3.

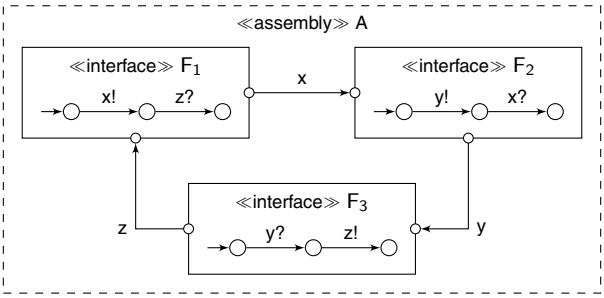


Fig. 1. A communication-safe modal assembly

4.2 Communication-Safety and Encapsulation of Modal Assemblies

In this work we define a communication-safety notion which is equivalent to the one in [15]. We will, however, use a different technical definition based on the explicit introduction of MIOs with error states. With this new definition we can generalize weak modal refinement to take into account errors states, which will lead to a new and powerful refinement notion for modal assemblies such that communication-safety is preserved. Our notion of communication-safe assembly is inspired by the notion of weak modal compatibility in [4]. This compatibility notion, as well as the compatibility notions in [10,12] and [17], rely on the assumption that outputs are autonomous and must be accepted by a communication partner while inputs are subject to external choice and need not to be served. Hence the discrimination of inputs and outputs is essential. Strong modal compatibility is based on the idea that whenever one component wants to send an output it finds the communication partner in a state, in which it must take the corresponding input *immediately*. Weak modal compatibility is more liberal, since it is sufficient if the communication partner must accept the message possibly after performing first some silent must-transitions. But in practice this compatibility requirement is still too strong. Therefore we generalize weak compatibility further and allow the communication partner to take the input only after performing silent must-transitions *and/or* mandatory communications with other components of the assembly *and/or* outputs on must-transitions which are directed outside of the assembly. This works well because, assuming communication-safe developments, these (open) outputs are again guaranteed to be taken, possibly after a delay, when an assembly is further extended.

For the technical definition of communication-safety we will first introduce a formal definition of the behavior of a modal assembly, which will be represented by a MIO

extended by explicit error states in the case that communication errors occur during execution of the assembly. Let $A = \{M_1, \ldots, M_n\}$ be a composable set of MIOs and let $M_j \in A$. Then the rest $A \setminus \{M_j\}$ plays the role of the environment for M_j. We must ensure that in any reachable state of the product $\bigotimes^{\mathrm{sy}} A$, whenever M_j wants to send an output l, then $Env_j = \bigotimes^{\mathrm{sy}} A \setminus \{M_j\}$ must be able to take l as an input possibly after some autonomous must-transitions which do not concern the communication with M_j. These autonomous transitions can be silent must-transitions of Env_j or must-communication transitions of Env_j obtained from communication inside $A \setminus \{M_j\}$, but also must-outputs of Env_j are admitted which are not shared with the inputs of M_j. If such a sequence of autonomous actions *cannot* be performed by Env_j there is a communication error.

Definition 3 (Communication errors). *Let $A = \{M_1, \ldots, M_n\}$ be a composable set of MIOs (with $n \geq 1$). If $n = 1$ there is no communication error. Otherwise, for each $1 \leq j \leq n$, let $Env_j = \bigotimes^{\mathrm{sy}} A \setminus \{M_j\}$. The* communication errors *$\mathscr{E}(A)$ are given by the set of pairs $((s_1, \ldots, s_n), l)$ such that $(s_1, \ldots, s_n) \in \mathscr{R}(\bigotimes^{\mathrm{sy}} A)$ and there is $1 \leq j \leq n$ with $l \in O_{M_j} \cap I_{Env_j}$, a state $s'_j \in S_{M_j}$ with $s_j \dashrightarrow_{M_j} s'_j$ but there are no transitions*

$$(s_1, \ldots, s_{j-1}, s_{j+1}, \ldots, s_n) \xrightarrow{\widehat{X}_j}_{Env_j} \cdot \xrightarrow{l}_{Env_j} (s'_1, \ldots, s'_{j-1}, s'_{j+1}, \ldots, s'_n)$$

with $X_j = T_{Env_j} \cup (O_{Env_j} \setminus I_{F_j})$.[4] □

Note that only communication errors occurring in the reachable part of the synchronous product of A are considered.

Definition 4 (MIOs with error states). *A* MIO with error states *(*EMIO*) is a pair (M, E) consisting of a MIO M and a set of* error states *$E \subseteq S_M$.* □

The error composition of MIOs is obtained by taking their synchronous product enriched by error states (if there are any) which are then reached by the un-accepted communication labels l. The idea is similar to the consent operator introduced in [1] to compose languages by indicating communication errors in traces.

Definition 5 (Error-composition of MIOs). *Let $A = \{M_1, \ldots, M_n\}$ be a composable set of MIOs and let $P = \bigotimes^{\mathrm{sy}} A$. The* error-composition *of A is given by the EMIO*

$$\bigotimes^{\mathrm{err}} A = ((L_P, S_P \cup \mathscr{E}(A), s_{0,P}, \dashrightarrow, \longrightarrow_P), \mathscr{E}(A))$$

with may-transition relation $\dashrightarrow = \dashrightarrow_P \cup \{(p, l, (p, l)) \mid (p, l) \in \mathscr{E}(A)\}$. □

The behavior of a modal assembly is given by the error composition of the modal interfaces of the assembly. It may also be considered as the semantics of the assembly. If no communication-error state appears in the assembly behavior, the assembly is communication-safe.

[4] Recall that T_{Env_j} are the communication labels of Env_j and $(O_{Env_j} \setminus I_{M_j})$ the output labels of Env_j unshared with the input labels of M_j, i.e., not used for communication between Env_j and M_j. The silent must-transitions of Env_j are anyway subsumed in the notation $\xrightarrow{\widehat{X}_j}_{Env_j}$; see Sect. 3.

Definition 6 (Behavior of a modal assembly and communication-safety). *Let* $A = \{F_1, \ldots, F_n\} \in \mathcal{A}^{\mathrm{m}}$ *be a modal assembly. The* behavior *of* A *is given by* $beh(A) = \bigotimes^{\mathrm{err}} A$. *$A$ is* communication safe*, written as* $cs^{\mathrm{m}}(A)$*, if* $\mathscr{E}(A) = \emptyset$. □

As an example, consider the assembly A in Fig. 1 and its (reachable) behavior shown in Fig. 2. The assembly is communication-safe since there is no error state. In fact all interfaces will be able to send their messages, possibly after a delay. For instance, F_1 can send x to F_2 after F_2 has communicated the message y to F_3.

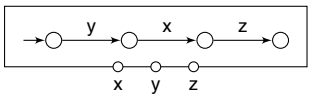


Fig. 2. Behavior of the assembly A in Fig. 1

Consider now a slight variation of the assembly in Fig. 1 such that the order of the input y? and output z! in F_3 is reversed. Let us call this assembly A'; see Fig. 3. The EMIO representing the (reachable) behavior of A' is shown in Fig. 4; it contains three error states. These are induced by the cyclic wait of the single interfaces in A'. Hence the assembly A' is not communication-safe. This example shows also that one cannot deduce from pairwise communication-safety of the interfaces of an assembly that the whole assembly is communication-safe. Indeed all pairs of interfaces in A' would form a communication-safe assembly.

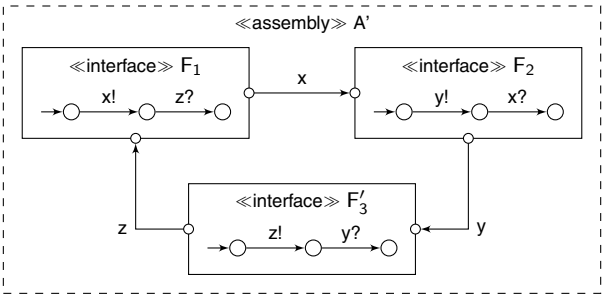


Fig. 3. Communication-safety does not follow from pairwise communication-safety

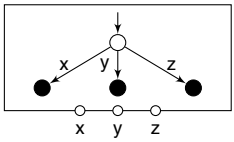
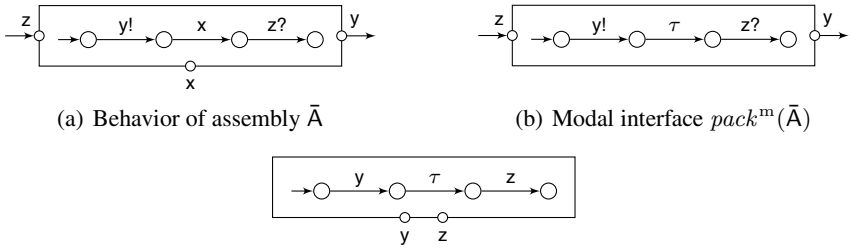


Fig. 4. Behavior of the assembly A'

The encapsulation of a modal assembly A by means of the modal pack operator is simply defined by hiding communication labels (see Sect. 3) in the behavior of A and forgetting the explicit discrimination of error states.



(a) Behavior of assembly $\bar{\mathsf{A}}$

(b) Modal interface $pack^{\mathrm{m}}(\bar{\mathsf{A}})$

(c) Behavior of assembly $\bar{\mathsf{C}}$

Fig. 5. Modal behaviors for assembly A in Fig. 1

Definition 7 (Encapsulation of modal assemblies). *The modal pack operator* $pack^{\mathrm{m}} : \mathcal{A}^{\mathrm{m}} \to \mathcal{F}^{\mathrm{m}}$ *is defined by* $pack^{\mathrm{m}}(A) = M\xi$ *with* $(M, E) = beh(A)$. □

We have to verify that the required properties (A1) and (A2) of an assembly theory are satisfied. First, we check (A1): (a) $pack^{\mathrm{m}}(\{F\}) = F$ holds by definition. (b) Let A and B be modal assemblies such that $A \boxtimes^{\mathrm{m}} B$ is defined; then $\{pack^{\mathrm{m}}(A), pack^{\mathrm{m}}(B)\} \in \mathcal{A}^{\mathrm{m}}$ since the MIOs underlying $beh(A)$ and $beh(B)$ are composable and since hiding preserves composability. (c) Now let additionally $cs^{\mathrm{m}}(A)$, $cs^{\mathrm{m}}(B)$, and $cs^{\mathrm{m}}(\{pack^{\mathrm{m}}(A), pack^{\mathrm{m}}(B)\})$ hold. Let $(M_A, E_A) = beh(A)$, $(M_B, E_B) = beh(B)$, $(M_{AB}, E_{AB}) = beh(\{M_A\xi, M_B\xi\})$, and $(M_{A\cup B}, E_{A\cup B}) = beh(A \cup B)$; then $\emptyset = E_A = E_B = E_{A\cup B} = E_{AB}$ and $M_A = \bigotimes^{\mathrm{sy}} A$, $M_B = \bigotimes^{\mathrm{sy}} B$, $M_{A\cup B} = \bigotimes^{\mathrm{sy}}(A \cup B) = M_A \otimes^{\mathrm{sy}} M_B$, and $M_{AB} = M_A\xi \otimes^{\mathrm{sy}} M_B\xi$ by the required communication-safety of A, B, and $\{pack^{\mathrm{m}}(A), pack^{\mathrm{m}}(B)\}$, and the resulting communication-safety of $A \boxtimes^{\mathrm{m}} B$ using (A2)(b). Thus

$$pack^{\mathrm{m}}(A \boxtimes^{\mathrm{m}} B) = M_{A\cup B}\xi = M_{AB}\xi = pack^{\mathrm{m}}(\{pack^{\mathrm{m}}(A)\} \boxtimes^{\mathrm{m}} \{pack^{\mathrm{m}}(B)\}) .$$

We now check (A2): (a) $cs^{\mathrm{m}}(\{F\})$ is obvious since no communication errors arise from a single modal interface. (b) Let A and B be modal assemblies such that $A \boxtimes^{\mathrm{m}} B$ is defined; then $\{pack^{\mathrm{m}}(A), pack^{\mathrm{m}}(B)\} \in \mathcal{A}^{\mathrm{m}}$ follows from (A1)(b). Now let additionally $cs^{\mathrm{m}}(A)$, $cs^{\mathrm{m}}(B)$, and $cs^{\mathrm{m}}(\{pack^{\mathrm{m}}(A), pack^{\mathrm{m}}(B)\})$ hold. Then $cs^{\mathrm{m}}(A \boxtimes^{\mathrm{m}} B)$ holds, since any communication error in $A \boxtimes^{\mathrm{m}} B$ would show up either in A or in B or at the boundary of A and B which would be captured by a communication error of $\{pack^{\mathrm{m}}(A), pack^{\mathrm{m}}(B)\}$.

Let us demonstrate how the principle of incremental design (see Sect. 2) works for the example assembly in Fig. 1. We start with the assembly $\bar{\mathsf{A}} = \{\mathsf{F}_1, \mathsf{F}_2\}$. The behavior of this assembly is shown in Fig. 5(a). Obviously, $\bar{\mathsf{A}}$ is communication-safe. We now want to add the interface F_3 to $\bar{\mathsf{A}}$. First, we pack the assembly $\bar{\mathsf{A}}$ which yields the modal interface $pack^{\mathrm{m}}(\bar{\mathsf{A}})$ shown in Fig. 5(b). Then we consider the assembly $\bar{\mathsf{C}} = \{pack^{\mathrm{m}}(\bar{\mathsf{A}}), \mathsf{F}_3\}$ whose behavior is shown in Fig. 5(c). Obviously $\bar{\mathsf{C}}$ is communication-safe and therefore, by the law of incremental design, the assembly $\mathsf{A} = \{\mathsf{F}_1, \mathsf{F}_2, \mathsf{F}_3\}$ is also communication-safe. The incremental communication-safety check would, in general, be much more efficient if we would minimize packed assemblies w.r.t. silent transitions.

Consider once more the assembly A' in Fig. 3 and assume that we want to construct it in an incremental way. Then we could start again with the assembly $\bar{\mathsf{A}} = \{\mathsf{F}_1, \mathsf{F}_2\}$ which is communication-safe. But now, for adding the interface F'_3, we have to consider the assembly $\bar{\mathsf{C}}' = \{pack^{\mathrm{m}}(\bar{\mathsf{A}}), \mathsf{F}'_3\}$ and to check communication-safety. The behavior of $\bar{\mathsf{C}}'$ is shown in Fig. 6; it has two error states. Hence, the incremental design step would not succeed and anyway, as we know from before, the assembly A' is not communication-safe.

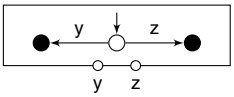
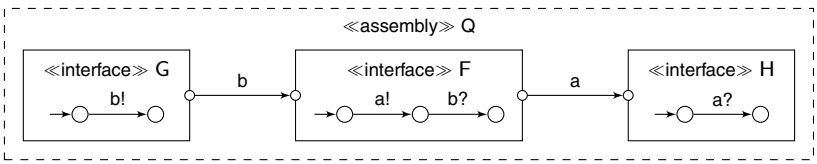
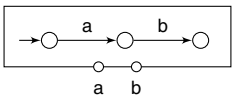


Fig. 6. Behavior of assembly $\bar{\mathsf{C}}'$

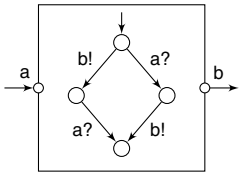
Conversely, we can not deduce from the communication-safety of an assembly $A \boxtimes^{\mathrm{m}} B$ that (i) $\{pack^{\mathrm{m}}(A), pack^{\mathrm{m}}(B)\}$ is communication-safe and we can also not deduce that (ii) the sub-assemblies A, B are communication-safe. Hence the converse direction of (A2)(b) does not hold. A counter-example for (i) is shown in Fig. 7(a). We can observe that the assembly Q is communication-safe; its (reachable) behavior, see Fig. 7(b), contains no error states. If we pack the sub-assembly $\{\mathsf{G}, \mathsf{H}\}$ we obtain the modal interface shown in Fig. 7(c). But the assembly $\{\mathsf{F}, pack^{\mathrm{m}}(\{\mathsf{G}, \mathsf{H}\})\}$ is not communication-safe. The reason is that $pack^{\mathrm{m}}(\{\mathsf{G}, \mathsf{H}\})$ has an output $\mathsf{b}!$ in its initial state, but the interface F can never accept this particular output as an input. It can only perform an a communication with $pack^{\mathrm{m}}(\{\mathsf{G}, \mathsf{H}\})$ and then accept "another" $\mathsf{b}!$ output of $pack^{\mathrm{m}}(\{\mathsf{G}, \mathsf{H}\})$ issued in another state.



(a) Modal assembly Q



(b) Behavior of assembly Q



(c) Modal interface $pack^{\mathrm{m}}(\{\mathsf{G}, \mathsf{H}\})$

Fig. 7. Counter-example for (i)

A counter-example for (ii) is shown in Fig. 8. The whole assembly R is communication-safe, but the sub-assembly $\{\mathsf{G}, \mathsf{F}'\}$ is not. The reason is that G has an output $\mathsf{b}!$ in its initial state, but F' has an open input $\mathsf{a}?$ before it can accept $\mathsf{b}?$ which is not allowed. (Inputs are not subject to internal choice and we cannot be sure that an environment will serve this input.)

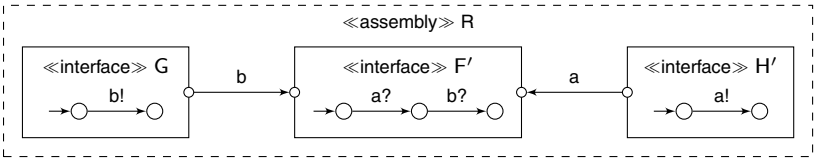


Fig. 8. Counter-example for (ii)

4.3 Refinement of Modal Assemblies

For the refinement of modal assemblies we compare their behaviors. Since assembly behaviors are MIOs with error states, we first extend the weak modal refinement notion for MIOs (defined in Sect. 3) to EMIOs, such that error states are respected by the refinement relation.

Definition 8 (Refinement of MIOs with error states). *Let* (M_A, E_A) *and* (M_B, E_B) *be two EMIOs.* (M_A, E_A) *is a* weak modal refinement *of* (M_B, E_B)*, if* $M_A \leq^*_{\mathrm{m}} M_B$ *is a weak modal MIO refinement witnessed by a refinement relation* $R \subseteq ((S_{M_A} \setminus E_A) \times (S_{M_B} \setminus E_B)) \cup (E_A \times E_B)$ *with* $(s_{0,M_A}, s_{0,M_B}) \in R$. □

Definition 9 (Refinement of modal assemblies). *A modal assembly* A refines *a modal assembly* B*, written as* $A \sqsubseteq^{\mathrm{m}} B$*, if* $beh(A)$ *is a weak modal refinement of* $beh(B)$. □

To get a modal assembly theory it remains to check that the conditions (A3), (A4) and (A5) of an assembly theory are satisfied. We will provide a short proof for each.

(A3): (a) That $F \preccurlyeq^{\mathrm{m}} G$ implies $\{F\} \sqsubseteq^{\mathrm{m}} \{G\}$ is obvious. (b) Now let $B_1 \boxtimes^{\mathrm{m}} B_2$ be defined and let $A_i \sqsubseteq^{\mathrm{m}} B_i$ for $i \in \{1,2\}$. Then $A_1 \boxtimes^{\mathrm{m}} A_2$ is defined, since A_i and B_i have the same I/O-labeling for $i \in \{1,2\}$. (c) Let $cs^{\mathrm{m}}(B_1)$, $cs^{\mathrm{m}}(B_2)$, and $cs^{\mathrm{m}}(\{pack^{\mathrm{m}}(B_1), pack^{\mathrm{m}}(B_2)\})$ hold additionally; then $cs^{\mathrm{m}}(B_1 \boxtimes^{\mathrm{m}} B_2)$ by (A2)(b). Let $(M_{A_i}, E_{A_i}) = beh(A_i)$ and $(M_{B_i}, E_{B_i}) = beh(B_i)$ for $i \in \{1,2\}$, $(M_{A_1 \cup A_2}, E_{A_1 \cup A_2}) = beh(A_1 \cup A_2)$, and $(M_{B_1 \cup B_2}, E_{B_1 \cup B_2}) = beh(B_1 \cup B_2)$. By the communication-safety of $B_1 \boxtimes^{\mathrm{m}} B_2$, $E_{B_1 \cup B_2} = \emptyset$ and hence $M_{B_1 \cup B_2} = \bigotimes^{\mathrm{sy}}(B_1 \cup B_2) = \bigotimes^{\mathrm{sy}} M_{B_1} \otimes^{\mathrm{sy}} \bigotimes^{\mathrm{sy}} M_{B_2}$. From the assumptions $A_i \sqsubseteq^{\mathrm{m}} B_i$, we obtain from (A5), to be proved momentarily, that $cs^{\mathrm{m}}(A_1)$ and $cs^{\mathrm{m}}(A_2)$. Hence, $M_{A_i} = \bigotimes^{\mathrm{sy}} A_i$ and $E_{A_i} = \emptyset$ for $i \in \{1,2\}$. By Prop. 1(1), we have $\bigotimes^{\mathrm{sy}} A_1 \otimes^{\mathrm{sy}} \bigotimes^{\mathrm{sy}} A_2 \leq^*_{\mathrm{m}} \bigotimes^{\mathrm{sy}} B_1 \otimes^{\mathrm{sy}} \bigotimes^{\mathrm{sy}} B_2$. It remains to ensure, that $E_{A_1 \cup A_2} = \emptyset$, i.e., $M_{A_1 \cup A_2} = \bigotimes^{\mathrm{sy}} A_1 \otimes^{\mathrm{sy}} \bigotimes^{\mathrm{sy}} A_2$. But if $(p,l) \in \mathscr{E}(A_1 \cup A_2)$, then $p \in \mathscr{R}(\bigotimes^{\mathrm{sy}} A_1 \otimes^{\mathrm{sy}} \bigotimes^{\mathrm{sy}} A_2)$ and there would be an output may-transition labeled l in one of the interfaces in $A_1 \cup A_2$, say in an interface of A_1. Then either $l \in O_{\bigotimes^{\mathrm{sy}} A_1}$ or $l \in T_{\bigotimes^{\mathrm{sy}} A_1}$. By $A_1 \leq^*_{\mathrm{m}} B_1$, such a transition also would have to be available in B_1. If $l \in O_{\bigotimes^{\mathrm{sy}} A_1}$, then the output would be accepted using a series of must-transitions in B_2 which do not affect B_1, since $cs^{\mathrm{m}}(\{pack^{\mathrm{m}}(B_1), pack^{\mathrm{m}}(B_2)\})$; this series of must-transitions would also have to be present in A_2 (up to must-τ's), as $A_2 \leq^*_{\mathrm{m}} B_2$, and hence $(p,l) \notin \mathscr{E}(A_1 \cup A_2)$. If $l \in T_{\bigotimes^{\mathrm{sy}} A_1}$, then the l would be accepted using a series of must-transitions in A_1 with possible outputs to A_2, since $cs^{\mathrm{m}}(A_1)$, and this series would be present also in B_1 by $A_1 \leq^*_{\mathrm{m}} B_1$. Again, these outputs are eventually accepted by B_2 by must-transitions, and thus by A_2; hence $(p,l) \notin \mathscr{E}(A_1 \cup A_2)$.

(A4): Let $A \sqsubseteq^{\mathrm{m}} B$ and let $cs^{\mathrm{m}}(B)$ hold. Let $(M_A, E_A) = beh(A)$ and $(M_B, E_B) = beh(B)$. Then $cs^{\mathrm{m}}(B)$ implies $E_B = \emptyset$, and thus, by (A5), $E_A = \emptyset$ since $A \sqsubseteq^{\mathrm{m}} B$. $M_A \leq^*_{\mathrm{m}} M_B$ implies $M_A\xi \leq^*_{\mathrm{m}} M_B\xi$ by Prop. 1(2), i.e., $pack^{\mathrm{m}}(A) \preccurlyeq^{\mathrm{m}} pack^{\mathrm{m}}(B)$.

(A5): Let $A \sqsubseteq^{\mathrm{m}} B$ and let $cs^{\mathrm{m}}(B)$ hold. Let $(M_A, E_A) = beh(A) = \bigotimes^{\mathrm{err}} A$ and $(M_B, E_B) = beh(B) = \bigotimes^{\mathrm{err}} B$. If an error state in E_A would be reachable in M_A, then $A \sqsubseteq^{\mathrm{m}} B$ would imply that some error state in E_B is also reachable in M_B since error states must be related to error states by a bisimulation. Thus $cs^{\mathrm{m}}(A)$ holds.

5 Conclusions

Our study is motivated by an extension of the abstract concepts of interface theories and interface languages, introduced by de Alfaro and Henzinger, to take into account interface assemblies. As a concrete formalism we have chosen modal I/O-transition systems which we have adapted to take into account not only blackbox interface behaviors but also assembly behaviors with distinguished (synchronous) communication actions. We have shown that the compositionality and compatibility requirements of an assembly theory are satisfied by modal assemblies.

In future work we are interested to study more instantiations of assembly theories, in particular assemblies which rely on asynchronous and multi-cast communication, and dynamic assemblies which may dynamically change the number of components and their connections. A concrete assembly theory using asynchronous communication via channel places can already easily be derived from the results for modal I/O-Petri nets in [14]. In this approach communication-safety is expressed by the property of a "necessarily consuming" Petri net. This property is compositional, decidable and preserved by refinement. We also plan to extend the MIO-workbench [4] to check not only MIOs but also modal assemblies and their communication-safety.

References

1. Adámek, J., Plasil, F.: Component composition errors and update atomicity: Static analysis. J. Softw. Maint. 17(5), 363–377 (2005)
2. ASCENS project, http://www.ascens-ist.eu
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proc. 4th IEEE Int. Conf. Software Engineering and Formal Methods (SEFM 2006), pp. 3–12. IEEE (2006)
4. Bauer, S.S., Mayer, P., Schroeder, A., Hennicker, R.: On weak modal compatibility, refinement, and the MIO workbench. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 175–189. Springer, Heidelberg (2010)
5. Bensalem, S., Griesmayer, A., Legay, A., Nguyen, T.-H., Sifakis, J., Yan, R.: D-Finder 2: Towards efficient correctness of incremental design. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 453–458. Springer, Heidelberg (2011)
6. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling dynamic architectures using Dy-BIP. In: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (eds.) SC 2012. LNCS, vol. 7306, pp. 1–16. Springer, Heidelberg (2012)
7. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM 30(2), 323–342 (1983)

8. Carmona, J., Kleijn, J.: Compatibility in a multi-component environment. Theor. Comput. Sci. 484, 1–15 (2013)
9. Cerná, I., Vareková, P., Zimmerova, B.: Component substitutability via equivalencies of component-interaction automata. Electr. Notes Theor. Comput. Sci. 182, 39–55 (2007)
10. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. 9th ACM SIGSOFT Ann. Symp. Foundations of Software Engineering (FSE 2001), pp. 109–120 (2001)
11. de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 148–165. Springer, Heidelberg (2001)
12. de Alfaro, L., Henzinger, T.A.: Interface-based design. In: Broy, M., Grünbauer, J., Harel, D., Hoare, C.A.R. (eds.) Engineering Theories of Software-intensive Systems. NATO Science Series: Mathematics, Physics, and Chemistry, vol. 195, pp. 83–104. Springer (2005)
13. Gößler, G., Sifakis, J.: Composition for component-based modeling. Sci. Comput. Program. 55(1-3), 161–183 (2005)
14. Haddad, S., Hennicker, R., Møller, M.H.: Specification of asynchronous component systems with Modal I/O-Petri nets. In: Abadi, M., Lluch Lafuente, A. (eds.) TGC 2013. LNCS, vol. 8358. Springer (to appear, 2014)
15. Hennicker, R., Knapp, A.: Modal interface theories for communication-safe component assemblies. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 135–153. Springer, Heidelberg (2011)
16. Hüttel, H., Larsen, K.G.: The use of static constructs in a modal process logic. In: Meyer, A.R., Taitslin, M.A. (eds.) Logic at Botik 1989. LNCS, vol. 363, pp. 163–180. Springer, Heidelberg (1989)
17. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O automata for interface and product line theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)
18. Larsen, K.G., Thomsen, B.: A modal process logic. In: Proc. 3rd Ann. IEEE Symp. Logic in Computer Science (LICS 1988), pp. 203–210. IEEE (1988)
19. Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.): The Common Component Modeling Example. LNCS, vol. 5153. Springer, Heidelberg (2008)
20. Sifakis, J.: Rigorous system design. Foundations and Trends in Electronic Design Automation 6(4), 293–362 (2013)

Constructive Collisions*

Edward A. Lee

EECS Department, UC Berkeley, Berkeley, CA, USA
eal@eecs.berkeley.edu

Abstract. This paper studies the semantics of models for discrete physical phenomena such as rigid body collisions. The paper combines generalized functions (specifically the Dirac delta function), superdense time, modal models, and constructive semantics to get a rich, flexible, efficient, and rigorous approach to modeling such systems. It shows that many physical scenarios that have been problematic for modeling techniques manifest as nonconstructive models, and that constructive versions of some of the models properly reflect uncertainty in the behavior of the physical systems that plausibly arise from the principles of quantum mechanics. The paper argues that these modeling difficulties are not reasonably solved by more detailed continuous models of the underlying physical phenomena. Such more detailed models simply shift the uncertainty to other aspects of the model. Since such detailed models come with a high computational cost, there is little justification in using them unless the goal of modeling is specifically to understand these more detailed physical processes. An implementation of these methods in the Ptolemy II modeling and simulation environment is described.

1 The Problem

Many physical phenomena are naturally modeled as being discrete rather than continuous. Modeling and simulating combinations of discrete and continuous dynamics, however, are challenging. Collisions of rigid objects and friction between moving objects are classic examples. Diodes and switches in electrical circuits present similar problems. All known solutions have significant limitations.

The difficulties stem from a number of sources. First, discontinuities make signals non-differentiable, which complicates simulation and analysis. Second, discrete phenomena can cause **chattering** around the discontinuity, where the solution repeatedly bounces across a discrete boundary. Third, discrete models more easily lead to Zeno conditions than continuous models, where an infinite number of events occur in a finite time. Finally, and perhaps most importantly, physical phenomena that are most naturally modeled as discrete are among the most poorly behaved and least understood. They frequently exhibit intrinsic nondeterminism and chaotic behaviors.

* This work was supported in part by the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), and the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley (supported by the National Science Foundation, NSF awards #0720882 (CSR-EHS: PRET), #1035672 (CPS: Medium: Ptides), and #0931843 (ActionWebs), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota).

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 161–176, 2014.

These sources of difficulty are worth separating. For example, it is not appropriate to condemn a model because it fails to deterministically model an intrinsically nondeterministic physical phenomenon. Nor is it fair to condemn a model for exhibiting Zeno behavior if the Zeno condition manifests outside the regime of parameters for which the model is suited.

Stewart [25] gives an excellent overview of approaches that have been used towards solving these problems for collisions and friction between macroscopic physical objects. In this regime, a solution that admits discrete behaviors can use generalized functions, most commonly the Dirac delta function, Lebesgue integration, measure theory, and differential inclusions. Stewarts argues for embracing discrete behaviors in models, and shows that a well-known paradox in the study of rigid body known as the Painlevé paradox can be resolved by admitting impulsive forces into the model.

A different (and more common) approach is to dispense with discrete models and insist on detailed modeling of the continuous dynamics. Collisions between rigid objects, for example, involve localized plastic deformation, viscous damping in the material, and acoustic wave propagation. Much experimental and theoretical work has been done to refine models of such phenomena, leading to considerable insight into the underlying physical phenomena. We contend, however, that such detailed modeling rarely helps in developing insight about macroscopic system behavior. So when the goal is, for example, to design robotic machinery, it is better to use simpler, more abstract models.

State-of-the-art design and simulation tools, however, do not support simpler models with discrete behaviors well. Modelica [26], for example, is a widely used language with well-supported libraries of models for a large variety of physical systems. Otter, et al. in [23] state that "at the moment, it is not possible to implement the solution with impulses ... in a generic way in Modelica." They offer continuous approximations as an alternative, categorizing three approaches for collisions: impulsive, spring-damper ignoring contact area, and spring-damper including contact area. They describe a library in Modelica that uses the latter two approaches.

Continuous models may indeed more accurately represent the physics, but they come at the price of greatly increased simulation cost and, perhaps more importantly, greatly increased modeling detail. The increased simulation cost is a consequence of the stiffness of the resulting differential equations. The increased modeling detail requires designers to specify much more detail about materials and systems than may be reasonable, particularly at early stages of design. Moreover, such detailed models may just shift the uncertainty from the modeling approximations to the determination of parameters. Is a robot designer able to characterize acoustic propagation in steel for a particular shape of robot arm in a particular range of temperatures and as the product ages? Probably not. So a detailed simulation model based on continuous physical processes may not be any more trustworthy than a much less detailed model.

In contrast, models that are created for the purpose of providing computer animations, like those described in Erleben et al. [9], are closer to what we need for understanding system dynamics. Computer animation has the very practical driving force that it must exhibit some behavior in reasonable time, so simulation efficiency is important.

The goal of this paper is improve the trustworthiness of less detailed, more abstract models. The approach is to put the semantics of the models on a solid foundation. If

the meaning of a model is absolutely clear, it is much easier to tell whether the model is faithful to the physical system it is modeling, and it is much easier to draw trusted conclusions from executions of the model.

To provide a solid foundation for abstract models, this paper embraces discrete phenomena modeled using generalized functions, and uses an extended model of time known as **superdense time** to cleanly mix discrete and continuous dynamics. In addition, the technique in this paper supports **modal models**, where a multiplicity of distinct abstract models, each with a well-defined regime of applicability, are combined to model the same system (as in **hybrid systems** [20,1]). Finally, the modeling framework is given a constructive fixed-point semantics [5], like that in synchronous-reactive languages [3]. We conjecture that nonconstructive models are suspect on physical grounds, and show that a number of well-known problematic scenarios with modeling discrete physical phenomena result in nonconstructive models. The techniques in this paper have been implemented as a Ptolemy II simulation tool [24], and the models displayed in this paper are all available online at `http://ptolemy.org/constructive/models`.

This paper is a shortened version of a technical report [13] that includes many more examples and more detailed analysis.

2 Time

Time is central to our approach to modeling. We require a model of time that combines a time continuum, over which physical dynamics can evolve, and discrete events, modeling abrupt changes in state of the system. In this section, we review the superdense model of time, the notion of discreteness, and the notion of piecewise continuity, which is essential for our models to work well with practical ordinary differential equation (ODE) solvers.

2.1 Superdense Time

We use a model of time known as **superdense time** [20,17]. A superdense time value is a pair (t,n), called a **time stamp**, where t is the **model time** and n is an **index** (also called a **microstep**). The model time represents the time at which some event occurs, and the microstep represents the sequencing of events that occur at the same model time. Two time stamps (t,n_1) and (t,n_2) can be interpreted as being **simultaneous** (in a weak sense) even if $n_1 \neq n_2$. Strong simultaneity requires the time stamps to be equal (both in model time and microstep).

To understand the role of the microstep, consider Newton's cradle, a toy with five steel balls suspended by strings. If you lift the first ball and release it, it strikes the second ball, which does not move. Instead, the fifth ball reacts by rising. Consider the momentum p of the second ball as a function of time. The second ball does not move, so its momentum must be everywhere zero. But the momentum of the first ball is somehow transferred to the fifth ball, passing through the second ball. So the momentum cannot be always zero.

Let $\mathbb{R}$ represent the real numbers. Let $p: \mathbb{R} \to \mathbb{R}$ be a function that represents the momentum of this second ball, and let τ be the time of the collision. Then

$$p(t) = \begin{cases} P & \text{if } t = \tau \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

for some constant P and for all $t \in \mathbb{R}$. Before and after the instant of time τ, the momentum of the ball is zero, but at time τ, it is not zero. Momentum is proportional to velocity, so

$$p(t) = Mv(t),$$

where M is the mass of the ball. Hence, combining with (1),

$$v(t) = \begin{cases} P/M & \text{if } t = \tau \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

The position of a mass is the integral of its velocity,

$$x(t) = x(0) + \int_0^t v(\tau) d\tau,$$

where $x(0)$ is the initial position. The integral of the function given by (2) is zero at all t, so the ball does not move, despite having a non-zero momentum at an instant.

The above physical model mostly works to describes the physics, but it has two flaws. First, it violates the basic physical principle of conservation of momentum. At the time of the collision, all three middle balls will simultaneously have non-zero momentum, so seemingly, aggregate momentum has magically increased.

Second, the model cannot be directly converted into a discrete representation (see Section 2.3 below). A discrete representation of a signal is a sequence of values that are ordered in time. Any such representation of the momentum in (1) or velocity in (2) is ambiguous. If the sequence does not include the value at the time of the collision, then the representation does not capture the fact that momentum is transferred through the ball. If the representation does include the value at the time of the collision, then the representation is indistinguishable from a representation of a signal that has a non-zero momentum over some interval of time, and therefore models a ball that does move. In such a discrete representation, there is no semantic distinction between an instantaneous event and a rapidly varying continuous event.

Superdense time solves both problems. Specifically, the momentum of the second ball can be unambiguously represented by a sequence of samples where $p(\tau, 0) = 0$, $p(\tau, 1) = P$, and $p(\tau, 2) = 0$, where τ is the time of the collision. The third ball has non-zero momentum only at superdense time $(\tau, 2)$. At the time of the collision, each ball first has zero momentum, then non-zero, then zero again, all in an instant. The event of having non-zero momentum is weakly simultaneous for all three middle balls, but not strongly simultaneous. Momentum is conserved, and the model is unambiguously discrete.

One could argue that the physical system is not actually discrete. Even well-made steel balls will compress, so the collision is actually a continuous process, not a discrete event. This may be true, but when building models, we do not want the modeling

formalism to force us to construct models that are more detailed than is appropriate. Such a model of Newton's cradle would be far more sophisticated, and the resulting non-linear dynamics would be far more difficult to analyze. The fidelity of the model may improve, but at a steep price in understandability and analyzability. Moreover, if the properties of the material and the dynamics of the collision are not well understood, the fidelity of the model may actually degrade as more detail is added.

The Newton's cradle example shows that physical processes that include instantaneous events are better modeled using functions of the form $p\colon \mathbb{R}\times\mathbb{N}\to\mathbb{R}$, where $\mathbb{N}$ represents the natural numbers, rather than the more conventional $p\colon \mathbb{R}\to\mathbb{R}$. The latter is adequate for continuous processes, but not for discrete events. At any time $t\in\mathbb{R}$, the signal p has a sequence of values, ordered by their microsteps. This signal cannot be misinterpreted as a rapidly varying continuous signal.

Superdense time is ordered lexicographically (like a dictionary), which means that $(t_1,n_1)<(t_2,n_2)$ if either $t_1<t_2$, or $t_1=t_2$ and $n_1<n_2$. Time stamps are a particular realization of **tags** in the tagged-signal model of [14].

2.2 Piecewise Continuity

So that we can leverage standard, well-understood numerical integration methods, we require signals to be piecewise-continuous in a specific technical sense. Consider a real-valued superdense-time signal $x\colon \mathbb{R}\times\mathbb{N}\to\mathbb{R}$. At each real-time $t\in\mathbb{R}$, we require that $x(t,n)$ settle to a final value and stay there. Specifically, we require that for all $t\in\mathbb{R}$, there exist an $m\in\mathbb{N}$ such that

$$\forall n>m,\quad x(t,n)=x(t,m). \tag{3}$$

A violation of this constraint is called a **chattering Zeno** condition. Such conditions would prevent an execution from progressing beyond real time t, assuming the execution is constrained to produce all values in chronological order.

Assuming x has no chattering Zeno condition, then there is a least value of m satisfying (3). We call this value of m the **final microstep** and $x(t,m)$ the **final value** of x at t. We call $x(t,0)$ the **initial value** at time t. If $m=0$, then x has only one value at time t.

Define the **initial value function** $x_i\colon \mathbb{R}\to\mathbb{R}$ by

$$\forall\, t\in\mathbb{R},\quad x_i(t)=x(t,0).$$

Define the **final value function** $x_f\colon \mathbb{R}\to\mathbb{R}$ by

$$\forall\, t\in\mathbb{R},\quad x_f(t)=x(t,m_t),$$

where m_t is the final microstep at time t. Note that x_i and x_f are conventional continuous-time functions. A **piecewise continuous** signal is defined to be a function x of the form $x\colon \mathbb{R}\times\mathbb{N}\to\mathbb{R}$ with no chattering Zeno conditions that satisfies three requirements:

1. the initial value function x_i is continuous on the left at all $t\in\mathbb{R}$;
2. the final value function x_f is continuous on the right at all $t\in\mathbb{R}$; and
3. x has only one value at all $t\in\mathbb{R}\backslash D$, where D is a discrete subset of $\mathbb{R}$.

The last requirement is a subtle one that deserves further discussion. First, the notation $\mathbb{R}\backslash D$ refers to a set that contains all elements of the set $\mathbb{R}$ except those in the set D. D is constrained to be a discrete set, as defined below. Intuitively, D is a set of time values that can be counted in temporal order. It is easy to see that if $D = \emptyset$ (the empty set), then $x_i = x_f$, and both x_i and x_f are continuous functions. Otherwise each of these functions is piecewise continuous.

Such piecewise-continuous signals coexist nicely with standard ODE solvers. At the time of a discontinuity or discrete event, the final value signal provides the initial boundary condition for the solver. The solver then works with an ordinary continuous signal until the time of the next discontinuity or discrete event, and the solver provides the initial value of the signal at the time of that next event.

2.3 Discreteness

A set D is a **discrete set** if it is a totally ordered set (for any two elements d_1 and d_2, either $d_1 \leq d_2$ or $d_1 > d_2$), and there exists a one-to-one function $f\colon D \to \mathbb{N}$ that is **order preserving**. Order preserving simply means that for all $d_1, d_2 \in D$ where $d_1 \leq d_2$, we have that $f(d_1) \leq f(d_2)$. The existence of such a one-to-one function ensures that we can arrange the elements of D *in temporal order*. Notice that D is a countable set, but not all countable sets are discrete. For example, the set $\mathbb{Q}$ of rational numbers is countable but not discrete. There is no such one-to-one function.

A **discrete-event** signal is a function defined on a lower subset[1] of superdense time to some value set, where the signal is non-absent only at a discrete subset of times. I.e., a discrete-event signal is absent almost everywhere, and the superdense times at which it is not absent form a discrete set. An **event** in a discrete-event signal is a time-value pair $((t,n),v)$, where (t,n) is a superdense time and v is a non-absent value. A discrete-event signal has a first event, a second event, etc., i.e. an ordered countable set of events.

The concept of piecewise continuity can be extended to discrete-event signals. Specifically, let ε represent the absence of an event. Then a discrete-event signal is a partial function

$$x\colon \mathbb{R} \times \mathbb{N} \to \{\varepsilon\} \cup U,$$

where U is some value set, where $x(t,n) = \varepsilon$ for all $(t,n) \in (\mathbb{R} \times \mathbb{N}) \setminus D$, where D is a discrete set. That is, the signal is absent almost everywhere, and is present only at a discrete subset of superdense times. A **piecewise-continuous discrete-event signal** is defined as a discrete-event signal whose initial value and final value functions always yield absent,

$$\forall t \in \mathbb{R}, \quad x_i(t) = \varepsilon, \quad x_f(t) = \varepsilon.$$

Such signals can coexist easily with numerical ODE solvers, since the signals seen by the solver, which are initial and final value signals, are simply absent. The solver ignores them.

The above definitions are used in [12]. Benveniste et al. in [4] define "discrete" differently to mean that "each instant has unique previous and next instants." This is a much weaker definition than ours here. We prefer our definition, because every event

[1] A **lower set** L is a subset of an ordered set S such that for all $x \in L$ and for all $y < x$, $y \in L$.

in a discrete-event signal has a finite number of predecessor events in the signal. This property is essential to being able to compute the events in a signal.

We define a **continuous-time signal** to be a signal whose value is not absent at any superdense time. Clearly, a continuous-time signal is not a discrete-event signal. But we can have signals that are neither continuous-time signals nor discrete-event signals. A signal that is not a discrete-event signal will need to either be represented symbolically or numerically approximated in any computation. Standard ODE solvers produce estimated **samples** of continuous-time signals. The time spacing between samples is determined by the step-size control of the solver. The samples themselves are defined on a discrete subset of superdense time.

2.4 An Alternative Model of Time

Note that an alternative model of time that can accomplish the same objectives as superdense time is studied in [4,22]. Their construction is based on nonstandard analysis [19], which, similarly to superdense time, has an infinite number of points at every real time point. These points are represented as convergent sequences, and a total order is induced over these sequences by means of a measure-theoretic construction. It has the property that every non-standard time has an immediate predecessor and an immediate successor, which the authors say provides an operational semantics. However, while an operational semantics does require the notion of a discrete step of computation, it also requires that the number of steps preceding any given step be finite. That is not automatically provided by the nonstandard semantics, and when it is provided, the solutions seem to be isomorphic with our much simpler superdense time construction. Hence, it does not appear to this author that anything is gained by going to a more complex mathematical formulation.

3 Constructive Fixed-Point Semantics

The next essential element in our rigorous modeling framework is the **constructive fixed-point semantics** [5], which defines the meaning of a model as a composition of components. The semantics we choose for hybrid systems is that of [18], which is implemented in Ptolemy II [6].

A model is assumed to be a graph of **actors**, as shown in Figure 1. An execution of the model (a simulation) will choose a discrete subset $D \subset \mathbb{R} \times \mathbb{N}$ of superdense time values at which to evaluate the model. The elements of D will be selected in order by a solver, beginning at some start time, say $(0,0)$. At each $(t,m) \in D$, the solver will find a value for each of the signals in the model, using a constructive procedure described below. For example, at the start time $(0,0)$, the solver will find the values $x(0,0), y(0,0)$, and $z(0,0)$ in Figure 1.

After finding the values of all signals at a time $(t,m) \in D$, the solver will choose the next time (t',m') at which to evaluate the model. To do this, it must first ensure that it has found the *final value* of each signal. If it has not, then it will increment only the microstep, so $(t',m') = (t,m+1)$. If it has found the final value of all signals, then it will consult an **event queue**, which contains a record of future discrete events,

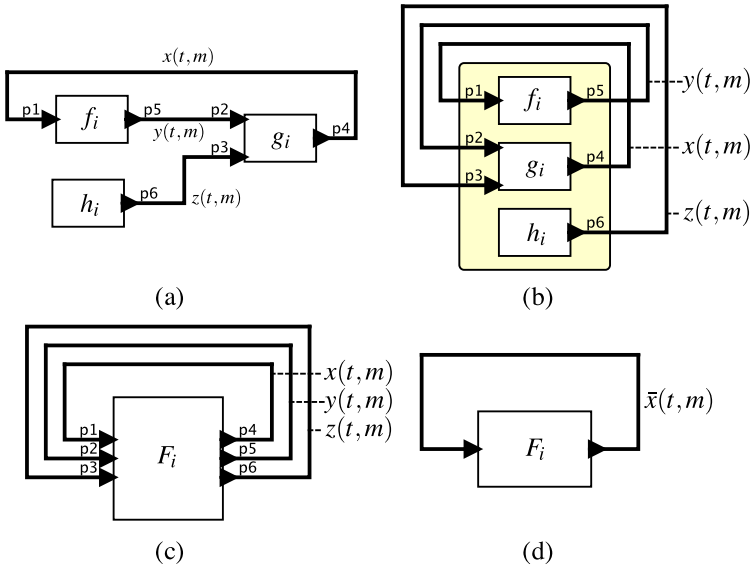


Fig. 1. A composition of actors with a constructive fixed-point semantics

and a **numerical ODE solver**, which determines a step size Δ that achieves a desired numerical accuracy. It then chooses the lesser of $(t+\Delta, 0)$ and (t_n, m), where (t_n, m) is the superdense time of the earliest event in the event queue.

Each superdense time in D is called a **tick**. The set D of ticks is discrete. The ticks can thus indexed by the natural numbers, so that

$$D = \{\tau_0, \tau_1, \cdots, \tau_i, \cdots\},$$

where $i < j \in \mathbb{N}$ implies that $\tau_i < \tau_j$.

At each $(t, m) \in D$, the solver needs to find the value of each signal. To support this, each actor provides a function from its input values to its output values. In brief, each function is required to be monotonic on a flat partial order and pointwise extensions of this order. In such partial orders, every chain is finite, and hence every monotonic function is continuous. The well-known Kleene fixed-point theorem [8] then provides a guarantee that a least fixed point exists and is unique, and provides a terminating procedure for finding that fixed point. For a more tutorial exposition, consult [13].

4 Collisions

We now consider the first of the families of discrete physical phenomena, collisions between rigid objects. Consider two objects with masses m_1 and m_2 colliding on a one-dimensional frictionless surface. We would like to treat the collision as an instantaneous event and are interested in determining the velocity after a collision. Newton's laws of motion imply that total momentum is conserved. If the velocities of the masses before

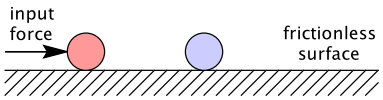


Fig. 2. Two balls on a frictionless surface

the collision are v_1 and v_2, and after the collision are v_1' and v_2', then conservation of momentum requires that

$$m_1 v_1' + m_2 v_2' = m_1 v_1 + m_2 v_2. \tag{4}$$

For notational simplicity, we leave off the dependence on time of the velocities, for now. Consider first perfectly elastic collisions, where no kinetic energy is lost. Conservation of kinetic energy requires that

$$\frac{m_1 (v_1')^2}{2} + \frac{m_2 (v_2')^2}{2} = \frac{m_1 (v_1)^2}{2} + \frac{m_2 (v_2)^2}{2}. \tag{5}$$

We have two equations and two unknowns, v_1' and v_2'. Because of the quadratic, there are two solutions to these equations. The trivial solution represents the absence of a collision, where $v_1' = v_1$ and $v_2' = v_2$. The second solution is

$$v_1' = \frac{v_1(m_1 - m_2) + 2m_2 v_2}{m_1 + m_2} \tag{6}$$

$$v_2' = \frac{v_2(m_2 - m_1) + 2m_1 v_1}{m_1 + m_2}. \tag{7}$$

Note that if $m_1 = m_2$, then the two masses simply exchange velocities.

In practice, most collisions of macroscopic physical objects lose kinetic energy. A common way to model this is to use an empirical quantity called the **coefficient of restitution**, denoted e and defined to be the relative speed after a collision divided by the relative speed before the collision. Using such a coefficient, the velocities after the collision are given by [2]

$$v_1' = \frac{e m_2 (v_2 - v_1) + m_1 v_1 + m_2 v_2}{m_1 + m_2} \tag{8}$$

$$v_2' = \frac{e m_1 (v_1 - v_2) + m_1 v_1 + m_2 v_2}{m_1 + m_2}. \tag{9}$$

The coefficient of restitution is determined experimentally for a particular pair of materials and must lie in the range $0 \leq e \leq 1$. Note that if $e = 1$, this reduces to elastic collision as given in (6) and (7). If $e = 0$, then momentum is still conserved, but the loss of kinetic energy is maximized. In this case, the resulting speeds of the two objects are identical. They collide and then travel together, not bouncing at all.

Note that if $m_1 = m_2$, then these equations reduce to

$$v_1' = (v_1(1-e) + v_2(1+e))/2 \tag{10}$$

$$v_2' = (v_2(1-e) + v_1(1+e))/2. \tag{11}$$

Another useful special case is where one of the masses is fixed (it cannot be moved), so the other will bounce off it. This follows from (8) and (9) if we let $v_2 = 0$ and determine the limit as $m_2 \to \infty$. In this case, we find

$$v_1' = -ev_1 \ . \tag{12}$$

4.1 Dirac Delta Functions

Stewart [25] points out that many of the difficulties in modeling collisions are a consequence of overly restricting the mathematical domains that are used. Impulsive forces, for example, can be naturally modeled using the **Dirac delta function**, a function $\delta : \mathbb{R} \to \mathbb{R}^+$ given by

$$\forall\, t \in \mathbb{R},\ t \neq 0, \quad \delta(t) = 0, \quad \text{and}$$
$$\int_{-\infty}^{\infty} \delta(\tau) d\tau = 1.$$

That is, the signal value is zero everywhere except at $t = 0$, but its integral is unity. At $t = 0$, therefore, its value cannot be finite. Any finite value would yield an integral of zero. This is indicated by $\mathbb{R}^+$ in the form of the function, $\delta : \mathbb{R} \to \mathbb{R}^+$, where $\mathbb{R}^+$ represents the **extended reals**, which includes infinity. Dirac delta functions are widely used in modeling continuous-time systems (see [16], for example), so it is important to be able to include them in simulations.

The Ptolemy II Integrator actor, used in Figure 3, directly supports Dirac delta functions. Specifically, the actor accepts a discrete-event signal at the input port labeled "impulse," and it interprets the real time of each event that arrives at that port as the time offset of the Dirac delta function and the value of the event as the weight of the Dirac delta function.

There are two reasons for providing a distinct input port for Dirac delta inputs vs. ordinary continuous-time inputs. First, the value of a Dirac impulse at the time it occurs is not a real number, so we would need some extended data type to include such weighted non-real values. But more importantly, at a superdense time (t, m), the output of the Integrator does not depend on the value of the input at the "derivative" input port, but it *does* depend on the value of the input at the "impulse" port! There is **direct feedthrough** from the impulse input to the output. The Integrator actor is both strict and nonstrict, depending on which input is being considered.

This causality distinction is essential to the soundness of our modeling approach. In fact, we will show below that many problematic modeling problems with discrete physical phenomena manifest as nonconstructive models. For example, any direct feedback from the output of the Integrator to the impulse input will result in a causality loop, and hence a nonconstructive model. Glockner [11] also advocates separately treating ordinary continuous-time signals and impulsive signals, whose models "split into the atomic and the Lebesgue part."

4.2 Modeling Collisions as Impulses

Figure 3 shows a Ptolemy II composite actor that models Newton's equations of motion, $F = ma$. The model has three parameter, the mass m of the ball, and the initial position

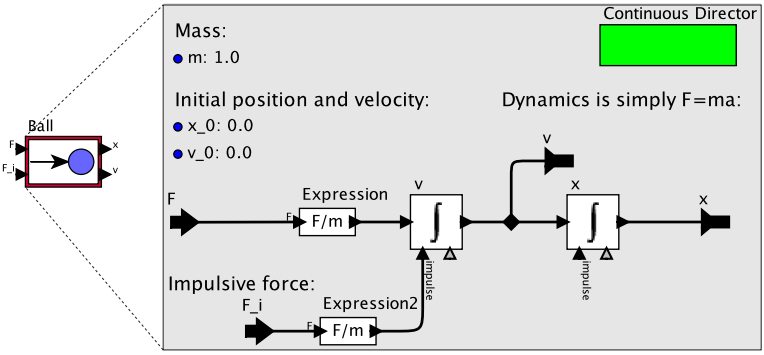


Fig. 3. A Ptolemy II model of a ball

x_0 and velocity v_0. The model uses Newton's second law to output the velocity v and position x as a function of time. There are two inputs, a real-valued force F and an impulsive force F_i. F_i is required to be a piecewise-continuous discrete-event signal, and its value represents the weight of a Dirac delta function.

A model that composes two instances of the ball model from Figure 3 is shown in Figure 4. The figure also shows a plot of the positions of two balls of diameter 1.0, where the left ball has an initial velocity of 1.0, and the right ball is standing still. After the collision, the situation is reversed. At the superdense time of the collision, the LevelCrossingDetector actor outputs an event, which enables execution of the **CalculateImpulsiveForce** composite actor. That actor is an instance of a subclass of **EnabledComposite**, which executes the inside model only when the *enable* port at the bottom has a present input with value true. The CalculateImpulsiveForce actor samples the current velocities of the balls and calculates the impulsive force that will change the velocities to those given by equations (8) and (9). The impulsive forces are then routed through a pair of MicrostepDelay actors, which apply the forces in the next microstep. Without these MicrostepDelay actors, we would have a causality loop, because the CalculateImpulsiveForce actor observes the velocities of the balls, and an impulsive force directly affects the velocities.

A collision occurs when the position of the right edge of the left ball coincides with the left edge of the right ball, and when the velocity of the left ball is greater than the velocity of the right ball. If ball 1 is on the left and ball 2 on the right, and the diameter of the left ball is d, then the collision occurs when

$$(x_1 + d \geq x_2) \wedge (v_1 > v_2).$$

However, this statement is fundamentally problematic. A collision occurs at the instant when the above predicate becomes true. First, there may be no such precise instant (suppose the balls are initially touching and we start applying a force to the left ball). Second, computational numerical methods have to approximate the continuums of time and position. In practice, to model such a collision as a discrete event, we need an error tolerance. Lower error tolerance will translate directly into increased computational cost.

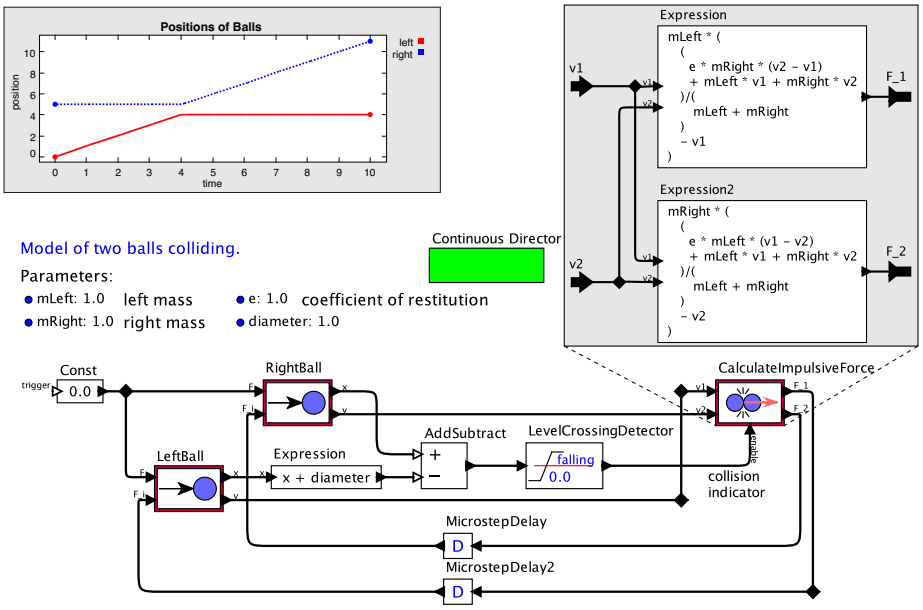


Fig. 4. A Ptolemy II model of two balls that collide

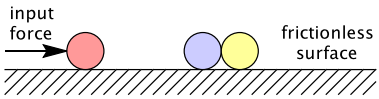


Fig. 5. Three balls on a frictionless surface

In Figure 4, the collision is detected by an actor labeled LevelCrossingDetector, which detects zero crossings of the distance between the two balls. This actor collaborates with the solver to adjust the step size of the numerical ODE solution so that the zero crossing is pinpointed with precision specified by a parameter.

Detecting collisions as zero crossings of the distance function, however, raises another difficulty. Specifically, if two balls are initially touching, the distance starts at zero. If a collision occurs, it does not cross zero. We consider this problem next.

4.3 Simultaneous Collisions

Consider the scenario shown in Figure 5, which is analogous to Newton's cradle. Two balls are initially touching, with zero distance between them, and a third ball approaches them from the left. At the instant of the collision, the left ball will transfer its momentum to the middle ball (assuming it has the same mass), which will then instantly transfer its momentum to the right ball. These two transfers occur at successive superdense time microsteps, so that the total momentum in the system is constant over time.

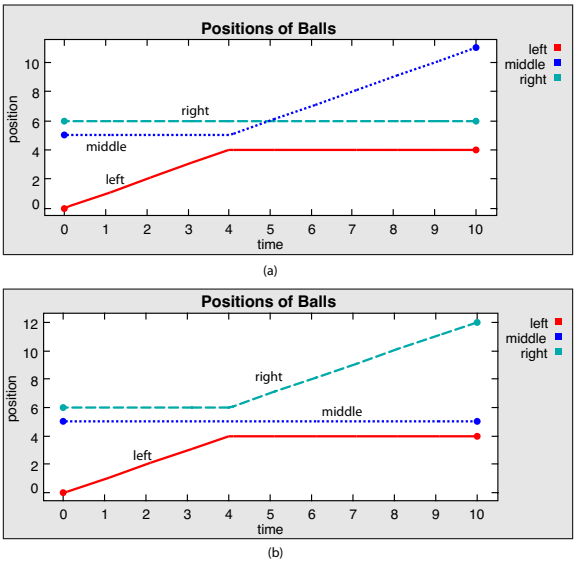


Fig. 6. (a) Second collision is not detected. (b) Second collision is detected.

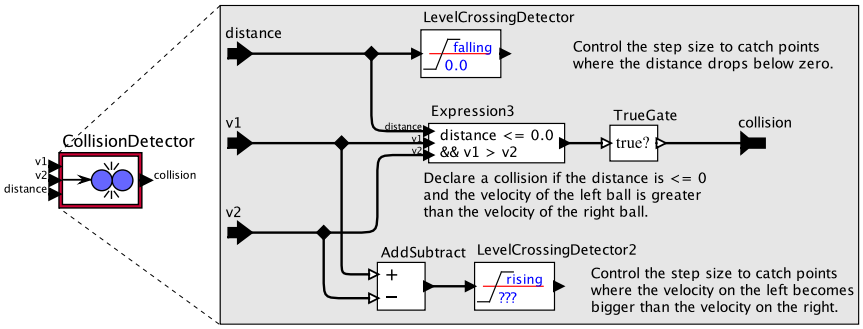


Fig. 7. A Ptolemy II model of a collision detector

The zero-crossing detection strategy in Figure 4, however, will not work for this scenario. It will fail to detect the second collision, because the distance between the middle and the right balls does not cross zero. It is initially zero, and a model like that in Figure 4 will show the left ball passing through the right ball, as shown in Figure 6(a). This difficulty is corrected by using a more sophisticated collision detection shown in Figure 7, which yields the plot in Figure 6(b). This correctly emulates Newton's cradle.

5 Discussion

Many more examples are considered in the extended version of this paper [13]. In that paper, a third scenario for the ball collisions is considered, where two balls

simultaneously collide with a stationary ball from opposite sides. This scenario is fundamentally problematic, and the Newtonian model of collisions given above has difficulty with it. A naive model superimposes the impulsive forces from the two simultaneous collisions, which cancel each other out in the middle ball. The result is a model where upon colliding, all three balls instantly stop. All the energy in the system is instantly lost.

One possible solution is to replace Newton's model with the **Poisson hypothesis**, which postulates that a collision consists of two distinct phases, a **compression phase** and a **restitution phase**. It is possible to construct a model where the two collisions have simultaneous compression phases, storing their kinetic energy as potential energy, and then, one superdense time index later, simultaneously release the potential energy as kinetic energy. Such a model would seem to solve the problem, but actually, it doesn't. There are many ways to assign kinetic energy such that both energy and momentum are conserved. In fact, such a solution simply masks a more fundamental physical problem.

An alternative solution is consider the two simultaneous collisions as being arbitrarily interleaved. That is, one occurs first, then the other. If the balls have the same mass, then it does not matter which one occurs first, and the model yields reasonable behavior. The outside balls bounce back with equal speed. However, if the balls have different masses, then the behavior depends on the order in which the collisions are handled, even though no time elapses between collisions.

In light of the Heisenberg uncertainty principle, these difficulties should not be surprising. The Heisenberg uncertainty principle states that we cannot simultaneously know the position and momentum of an object to arbitrary precision. But the reaction to these collisions depends on knowing position and momentum precisely. A direct expression of such simultaneous collisions results in a nonconstructive model. To get a constructive model, we have to insert microstep delays and tolerate nondeterminism. Nature, it seems, resolves nonconstructiveness with uncertainty. Chatterjee and Ruina suggest that indeed, a reasonable and practical approach to simulating such systems is to nondeterministically choose an ordering [7].

Note that doing more detailed modeling of the collisions does not solve the problem. It just shifts the uncertainty to other parts of the model. Unlike the two-ball collision, there are multiple solutions that conserve energy and momentum. We conjecture that defensible detailed models could yield the same (or more) variabilities in behaviors.

The extended paper [13] also studies Zeno conditions, using the classical bouncing-ball example. It shows that any finite precision model results in the ball eventually "tunnels" through the surface on which it is supposed to be bouncing. Again, it might seem odd to invoke quantum mechanics when considering macro phenomena such as collisions of balls. But the impulsive model we are using has infinite precision, and in physics, it is at high precisions where quantum mechanical effects become important. The solution is to avoid using models outside the their regime of applicability. Specifically, the idealized bouncing ball model with impulsive collisions becomes inappropriate when the extent of the bounce is comparable to the numerical precision of modeling tool. This is analogous to the situation in physics, where in different regimes, one might use classical Newtonian mechanics, quantum physics, or relativity, and failing to use the right models will yield misleading results.

The solution is to use **modal models** [10,15]. Such models give a multiple distinct models of the dynamics, and provide a state machine that transitions between these models. When the operating conditions exit the regime of applicability of a model of the dynamics, the state machine switches to a different model. Modal models provide an operational semantics for hybrid systems [17] that leverages superdense time.

The extended paper [13] also studies physical models of friction, which also exhibits discrete changes in behavior, combinations of friction and collisions, and electrical circuits with discrete behaviors, as in [21]. It shows that the same principles apply.

6 Conclusion

Constructive semantics gives a natural way to separate problems that can be solved with confidence from those that cannot. When, for example, the order of nearly instantaneous collisions is important, a constructive semantics forces us to either choose an order or explicitly choose nondeterminism. Building useful constructive models of combined continuous and discrete behaviors is facilitated by a superdense model of time, an explicit use of impulses (generalized functions), and modal models.

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
2. Beer Jr., F., Johnston, E.R.: Vector equations for Engineers: Dynamics, 6th edn. McGraw Hill (1996)
3. Benveniste, A., Berry, G.: The synchronous approach to reactive and real-time systems. Proceedings of the IEEE 79(9), 1270–1282 (1991)
4. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: The fundamentals of hybrid systems modelers. Journal of Computer and System Sciences 78(3), 877–910 (2012)
5. Berry, G.: The Constructive Semantics of Pure Esterel. Draft version, 3rd edn. (2003)
6. Cardoso, J., Lee, E.A., Liu, J., Zheng, H.: Continuous-time models. In: Ptolemaeus, C. (ed.) System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org, Berkeley (2014)
7. Chatterjee, A., Ruina, A.: A new algebraic rigid body collision law based on impulse space considerations. Journal of Applied Mechanics 65(4), 939–951 (1998)
8. Davey, B.A., Priestly, H.A.: Introduction to Lattices and Order, 2nd edn. Cambridge University Press (2002)
9. Erleben, K., Sporring, J., Henriksen, K., Dohlmann, H.: Physics-Based Animation. Charles River Media, Inc., Hingham (2005)
10. Feng, T.H., Lee, E.A., Liu, X., Tripakis, S., Zheng, H., Zhou, Y.: Modal models. In: Ptolemaeus, C. (ed.) System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org, Berkeley (2014)
11. Glockner, C.: Scalar force potentials in rigid multibody systems. In: Pfeiffer, F., Glockner, C. (eds.) Multibody Dynamics with Unilateral Contacts. CISM Courses and Lectures, vol. 421. Springer, Vienna (2000)
12. Lee, E.A.: Modeling concurrent real-time processes using discrete events. Annals of Software Engineering 7, 25–45 (1999)

13. Lee, E.A.: Constructive models of discrete and continuous physical phenomena. Technical report, EECS Department, UC Berkeley (February 2014)
14. Lee, E.A., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Trans. on Computer-Aided Design of Circuits and Systems 17(12), 1217–1229 (1998)
15. Lee, E.A., Tripakis, S.: Modal models in Ptolemy. In: 3rd Int. Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT), Oslo, Norway, vol. 47, pp. 11–21. Linköping University Electronic Press, Linköping University (2010)
16. Lee, E.A., Varaiya, P.: Structure and Interpretation of Signals and Systems, 2.0 edn. LeeVaraiya.org (2011)
17. Lee, E.A., Zheng, H.: Operational semantics of hybrid systems. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 25–53. Springer, Heidelberg (2005)
18. Lee, E.A., Zheng, H.: Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In: EMSOFT, Salzburg, Austria. ACM (2007)
19. Lindstrøm, T.: An invitation to nonstandard analysis. In: Cutland, N. (ed.) Nonstandard Analysis and Its Applications, pp. 1–105. Cambridge University Press (1988)
20. Maler, O., Manna, Z., Pnueli, A.: From timed to hybrid systems. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 447–484. Springer, Heidelberg (1992)
21. Mosterman, P.J., Biswas, G.: A theory of discontinuities in physical system models. Journal of the Franklin Institute 335(3), 401–439 (1998)
22. Mosterman, P.J., Simko, G., Zande, J.: A hyperdense semantic domain for discontinuous behavior in physical system models. In: Multi-Paradigm Modeling, MPM (2013)
23. Otter, M., Elmqvist, H., López, J.D.: Collision handling for the Modelica multibody library. In: Modelica Conference, Hamburg, Germany, pp. 45–53 (2005)
24. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org, Berkeley (2014)
25. Stewart, D.E.: Rigid-body dynamics with friction and impact. SIAM Review 42(1), 3–39 (2000)
26. Tiller, M.M.: Introduction to Physical Modeling with Modelica. Kluwer Academic Publishers (2001)

The Unmet Challenge of Timed Systems

Oded Maler

CNRS-VERIMAG, University of Grenoble
2, av. de Vignate, 38610 Gieres, France
Oded.Maler@imag.fr

Dedicated to Joseph Sifakis.

Abstract. *Timed systems* constitute a class of dynamical systems that live in an extremely useful level of abstraction. The paper stresses their importance in modeling without necessarily endorsing the orthodox approach for reasoning about them which is practiced in the theoretical and applied branches of formal verification.

1 Introduction

This paper is about *timed systems*, a formal model that combines discrete transitions and metric time. Joseph has been involved in studying such systems during several periods of his career including work on timed Petri nets, timed process algebra and, more effectively, in the context of algorithmic formal verification where he (together with students and collaborators) played an important role in the evolution of timed automata modeling and verification [18,30,15,12,5,27]. More recently, as a promoter of a compositional approach to embedded systems design [8,7], I guess he could observe how real-time and performance tend to pop-up and demonstrate yet another difference between a nice theory and practical reality. Performance is a non-compositional phenomenon per se because it involves sharing of limited resources and the performance of a single process in isolation typically degrades when it has to share resources with others.

Since I spent significant parts of my academic life working on timed systems at Joseph's VERIMAG, I find it appropriate to use this opportunity to reflect aloud on this topic, free from the usual ceremony of theorems, tools and case-studies found in standard publications. A large part of this paper will be situated on the abstract and meta level, but I will start by formulating a concrete motivating[1] problem more related to the expected audience.

Consider an application program such as a video decoder to be executed on a new mobile multi-core platform. The application is structured as a *task graph*, a collection of tasks,[2] partially-ordered according to precedence, and annotated by execution times and data transfer rates. We assume that these durations, as well as the arrival rates of new

[1] The term "motivating" is used in a very liberal and wide sense, covering psychological, sociological and metabolic aspects of scientific activity.

[2] Modules, components, actors, filters, functional blocks, stream transducers...

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 177–192, 2014.

jobs to execute, admit some *bounded uncertainty* which can be due to different factors ranging from variability in the type of image frames and the workload of the platform, down to the possibility that the software and/or hardware are still in the design stage. Executing the program on the architecture, in addition to the usual compilation chain, involves other orchestration decisions for which we use the collective term *deployment*: how to map tasks to processors, how to schedule them, how to allocate buffers and communication channels and how to transfer data among processors. All these questions involve non-trivial combinatorial problems and bad choices can have far reaching effects on performance to the point of making the difference between the feasible and the infeasible. Forcing application programmers to occupy themselves with these issues is like returning to the stone age of computing, undoing the impressive progress made over the years by isolating programmers from more and more low-level implementation details. Hence it is urgent to provide computer-aided support for multi-core deployment. Timed systems provide, in principle, the conceptual and mathematical foundations for evaluating, comparing and optimizing such deployment decisions.

The message of this paper can be summarized as follows. Models of timed systems are extremely important as they represent a level of abstraction which is used, explicitly or implicitly, in almost any domain of engineering and everyday life. In particular, timed models have a significant potential contribution to performance analysis and optimization for all sorts of systems. Unfortunately, sociological and cultural factors, both in academia and industry, as well as genuine complexity problems, prevent this potential from being fully realized.

The rest of the paper is organized as follows. Section 2 presents the timed level of abstraction and demonstrates how it can be used in modeling. Timed automata are introduced in Section 3 along with their non-scalable orthodox analysis in the tradition of formal verification and various attempts to make it scale up. Section 4 recounts some recent encounters with practice and provides retrospective reflections on timed and untimed verification without a decisive punch line. Anecdotes and strong opinions, not always well-founded, are interleaved in the text and should not be taken too seriously and certainly not offensively.

2 The Timed Level of Abstraction

It is a common knowledge that phenomena can be modeled at different levels of abstraction or granularity. Lower levels are more detailed, zoomed at more elementary entities and are supposed to be closer to *deep reality*.[3] I will start by discussing abstractions based on *aggregation* which are more common in Physics and its Applied Mathematics (and also in macro Economics) and then move to other types of abstractions, more relevant to our concerns.

The prime example of aggregation-type abstraction is found in Statistical Physics where the underlying detailed micro model consists of a huge number of particles whose respective momenta are aggregated and abstracted into coarse-grained macro entities such as temperature. The derivation of the macro model from the micro-level rules

[3] This is at least what reductionists (physicists, molecular biologists and even humble programmers) want us to believe.

often exists only in principle. My favorite example[4] is the *module of elasticity* which indicates the resistance of a beam to different loads and serves for reasoning about building stability. This characterization is inferred from macro level experiments on beams made of various materials and is *not* derived from a detailed model of zillions of interacting molecules.

When models are more detailed, you have to deal with more state variables and this raises two main problems. In the interface between the real world and the model you face the problem of observation/measurement: you need to keep track of many particles in order to identify the system dynamics and also to measure initial conditions if you want to use the model to predict the future. Even if this scientific model building and measurement problems are somehow miraculously solved and you have a model, you still have a problem in the virtual world of ideas and numbers because it takes much more effort to do *computations* with the model. This difficulty applies both to the (increasingly more rare) cases where you can apply analytic/symbolic methods, as well as to the modern way to reason about complex systems, namely, *simulation*. This computational difficulty applies even more to *algorithmic verification*, also known as *model-checking*, which is essentially a kind of extended muscular form of simulation, augmented sometimes with a more expressive temporal language for evaluating and classifying behaviors [22].

Another type of abstraction, perhaps more familiar to people from our close domains, is *model reduction* which eliminates some variables and simplifies the dynamics but remains within the same order of magnitude in terms of the number of variables. Such reductions are common in the treatment of models of continuous dynamical systems defined by ordinary differential equations (ODE) in Control and related engineering disciplines but also in discrete models where some variables are hidden. The formal relationship between the original and reduced models is based on the projection of the behaviors of both systems on common observable variables. For ODE models, the reduced system should produce behaviors (trajectories) which are *close* to the detailed one in some metric. In the Computer Science tradition, projecting away variables renders the system non-deterministic in a set-theoretic sense, first introduced in [26].[5] Hence the abstract system may have several behaviors emanating from the same initial state and the relation between the two levels is typically a *conservative approximation*: the set of behaviors of the abstract model is a superset of those of the concrete system.

In the compilation of high-level programs into machine code the relation between different levels is based on a combination of variable hiding and a change in the time scale. A program instruction which is considered atomic at high-level corresponds to a more detailed sequence of primitive instructions involving addressing and registers which are hidden from the high-level models. Proceeding downward, each of the machine code instructions is refined into a collection micro-code hardware transactions and so on. Going upwards, pieces of program code are grouped into *procedures* that can be used, in their turn, as atomic instructions. It is worth noting that software and

[4] Dedicated to my father, Ephraim Maler, a construction engineer.

[5] The adaptation to continuous and hybrid systems is elaborated in [23] where such systems are referred to as *under-determined* to avoid the term *non-deterministic* which is already overloaded with connotations.

digital hardware are remarkably exceptional in the sense of having a *formal equivalence* between so many levels. This is due to the aphysical (and sequential) nature of computational models, the underlying hardware infra-structure and the fact that we deal here with man-made artifacts rather than mother nature.

Finally there is a class of abstractions which transform a systems defined over real-valued numerical variables into a discrete-event system where ranges of continuous variables are quantized into abstract states. Two of the concrete examples given below in order to introduce the idea of timed systems are based on such a quantization while the third is taken from the software domain.

Example 1: Transistors and Gates. At a lower-level a logical gate, say inverter, is an electrical circuit composed of transistors whose voltage at the output port responds to the voltage in its input. Its behavior is a signal, a trajectory of a continuous dynamical system which has two stable states around voltages v_0 and v_1. At the abstract Boolean level we say that when the input goes down, the gate becomes excited and takes an observable transition from 0 to 1 as shown in Fig. 1-(a).

Example 2: Coming to Grenoble. Suppose you come to Grenoble from your home-town by airplane via Lyon airport. A low-level description can be given in terms of the trajectory of your center of mass on the spatial Earth coordinates. At a higher level you can describe it verbally as a sequence of events: fly to Lyon then wait for the bus that will take you to Grenoble, see Fig. 1-(b).

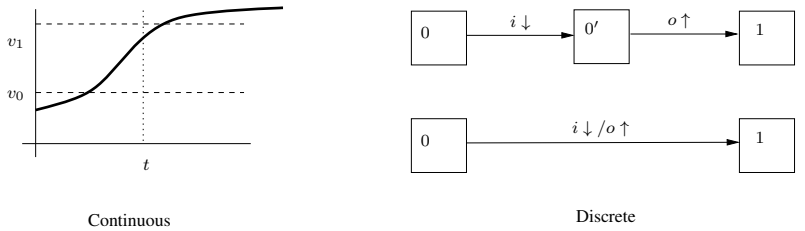
Example 3: Software. At a lower level we have piece of code, an algorithm that transforms some input to some output using instructions that run on some hardware platform. At a higher level of description we say that we process an image frame, e.g. decode or filter it (Fig. 1-(c)).

From the more abstract discrete point of view, in all these cases you have some kind of a (concrete, physical) process that you do not care too much about its *intermediate* states.The important thing is that at the end of the process you will be in Grenoble, the gate will switch properly from 0 to 1 according to Boole-Shannon rules and the image will be decoded. This is the essence of functional reasoning. But you cannot get rid completely of the underlying Physics.[6] In order to determine the clock rate that your computer can use, you need to know *how long* it takes to switch from 0 to 1.[7] To watch some stupid video on your smart phone you do care about the *execution time* of your decoding algorithm. To come on time to a conference you need to know the *duration* of the flight. The purely discrete automaton model does not distinguish between flying from Paris and flying from San Francisco: the flight is modeled simply as an abstract sequence of transitions: take-off $\rightarrow$ fly $\rightarrow$ land.

The timed level of abstraction offers a *compromise* between these two extremes, the fully continuous and the purely discrete. To understand what is going on, let us look closer at the nature of the discrete models and their relationships with the underlying continuous process. Such models observe the continuous variables through

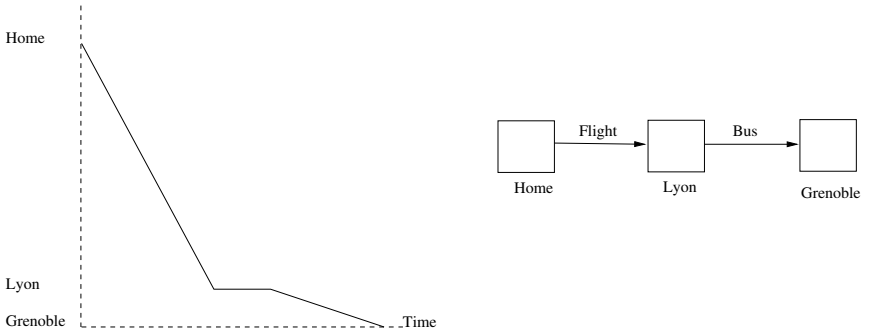
[6] The term "physics" is used here in a very loose sense as denoting all those quantitative things that preceded the abstract computer.

[7] In the development of digital circuits there is a phase called *timing closure* where these considerations, neglected in preceding design stages, are considered.



Continuous Discrete

(a)



(b)

```
for i=1 to 1024 do
  ...
  ...
  something
  ...
  ...
end
```

process an image

(c)

Fig. 1. Three examples of systems viewed in two adjacent levels of abstraction. (a) *Logical gate*. Left: a reaction of an inverter to a change in its input voltage; Right: The discrete model where from a stable state 0, the gate becomes excited (state $0'$) when its input goes down and then stabilizes into a new state 1. In some modeling styles the intermediate unstable state is ignored and the changes in input and output are part of the same transition. (b) *Coming to Grenoble*. Left: a simplified evolution of the distance from Grenoble during flight, waiting at the airport and taking the bus. Right: the trip viewed as a discrete sequence of steps. (c) A program and its abstract description.

an abstraction/quanization: rather than recording the exact liquid level in your glass, it classifies the states into a finite number of categories, say, empty, full and in-between. Likewise, the *dynamics* of the water level can be classified as filling, emptying or static, which can be viewed as quantizing the derivative. Discrete events indicate changes in states which correspond to threshold crossings of continuous variables as well as changes in the dynamics such as opening and closing valves. Timed systems augment the expressive power of the discrete model with a *metric temporal distance* between events or the *duration* of staying in states. As mathematical objects, timed behaviors are represented in two fundamental forms that correspond to the two types of timed monoids described in [4]:

- The first representation is based on *time-event sequences* where a behavior is viewed as an alternation of durations and instantaneous events. This is essentially equivalent to the timed traces used in [3] in which the events are embedded in the real-time axis and represented as sequences of time-stamped events. Such representations are produced in numerous domains including all sorts of event monitoring systems as well as numerical simulators for differential equations.
- The second representation is based on discrete-valued *signals* which are functions from real time to finite domains. Boolean signals are heavily used in the design of digital circuits. Similar mathematical objects are used in all planning and scheduling domains where they are called Gantt charts or time-tables.

Fig. 2 depicts timed descriptions of the trip to Grenoble in these two forms. Using such a representation we can distinguish short flights from long, fast gates from slow and efficient algorithms from less efficient ones.

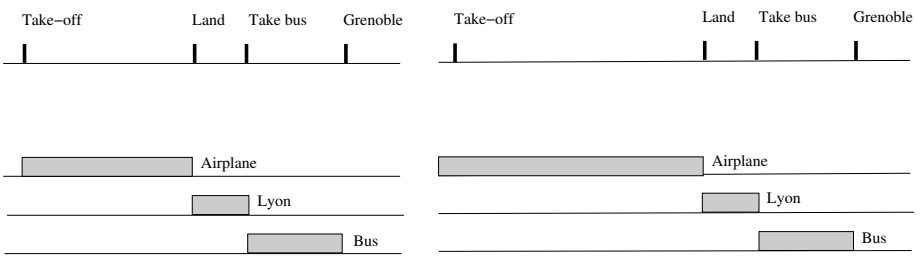


Fig. 2. The trip viewed as at a timed level of abstraction: as a sequence of events embedded in the real time axis (up) or as signals over discrete domains (down). At this level of abstraction one can tell the difference between a short flight (left) and a longer one (right).

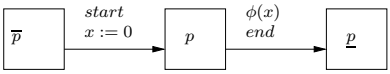
This gives rise to a new class of dynamical systems which, unfortunately, is a bit hard to digest. Recall that continuous behaviors are trajectories in a continuous state-space of dynamical systems specified using *differential equations* and that models based on *automata* are the dynamical systems that produce discrete behaviors, sequences of states and events. Likewise *timed automata* and similar timed models generate timed behaviors. These are the *dynamical systems of the timed level of abstraction.*

The basic atomic component of the timed world is the process that takes some time to conclude after having started. Such a process is modeled by the simple timed automaton depicted in Fig. 3-(a). It consists of an *idle* state $\overline{p}$ in which nothing happens and from which a *start* transition takes it to *active* state p while resetting clock x to zero. Clock x is a restricted type of state variable which progresses in the speed of time ($\dot{x} = 1$) so as to measure the time elapsed since an activity has started. The influence of x on the dynamics is via its participation in the precondition (the transition guard ϕ) for taking the *end* transition to *final* state $\underline{p}$. The condition ϕ can be deterministic, $x = d$, indicating that process duration is assumed to be precisely known. It can be non-deterministic, $x \in [a, b]$, meaning that the duration can be anywhere within the interval. The degenerate case where the condition is always true, that is, $x \in [0, \infty)$, is equivalent to an untimed model. The condition can also be made probabilistic, assuming some distribution on the duration, resulting in an expressively rich kind of a continuous-time stochastic process [24].

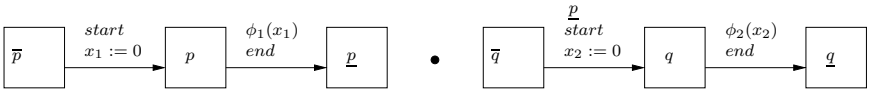
To my mind, the most important contribution of the theory of automata to humankind is in the notion of parallel composition (or products of automata). Such notions exist of course also for interacting continuous dynamical systems but only in automata you can visualize a global system and see the effect of interaction. When you compose two automata, you obtain a global automaton whose states are of the form $s = (s_1, s_2)$, elements of the Cartesian product of the individual state-spaces. Transitions available from s are either independent transitions taken from s_1 or s_2 or transitions of one automaton which are conditioned on the state of the other. Fig. 3-(b) shows how parallel composition can realize sequential composition: the *start* transition of the automaton of process q is conditioned the by the automaton for process p being in its final state. Sequential composition represents *precedence* relations between tasks where $p \prec q$ indicates that q cannot start before p has terminated. This is how you express statements like, *you can take the bus after you land*, *a gate switching triggers a change in the next gate* or *you can start processing the image only after having decoded it*.

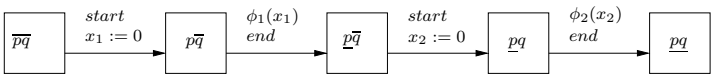
More generally, parallel composition can express processes that run concurrently, sometimes progressing independently and sometimes synchronizing. In timed models, independent progress is not as independent because Time is viewed as a shared variable that all processes interact with. The automaton of Fig. 3-(c) shows a fragment of the composition of automata for two timed processes. In state s we observe the fundamental phenomenon in timing analysis: two or more active processes running in parallel.[8] The winner in this *race*, a term used in continuous-time stochastic processes, is the one which takes its *end* transition first. The identity of the winner depends on the delay bounds $[a_1, b_1]$ and $[a_2, b_2]$ as well as the timing of the preceding transitions. Typically, timing constraints, due to sharing of the time variable, restrict the range of behaviors which would be otherwise possible in the untimed transition graph. Analyzing the possible behaviors of such concurrent timed processes is at the heart of almost anything we do when we hurry up to catch a bus or wait for our partner to come. The following list illustrates the universality of questions related to possible behaviors (paths) of networks of timed automata.

[8] For this reason I find papers that deal with timed automata with a *single* clock to be of a purely theoretical interest.

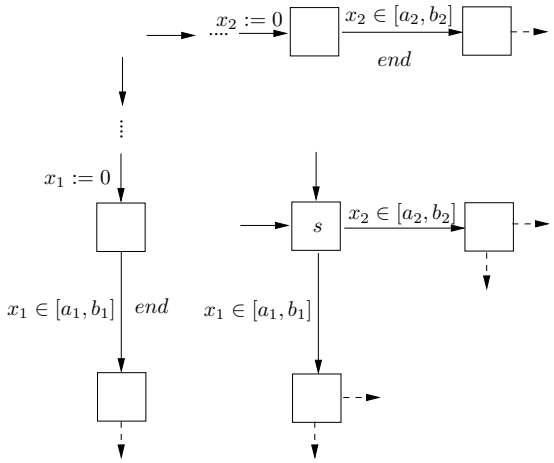


(a)





(b)



(c)

Fig. 3. Processes that take time: (a) The basic timed component; (b) Sequential composition; (c) Parallel composition. State s admits a race between two processes.

- Will there be a glitch in the circuit?
- Will he finish his boring talk by the coffee break?
- Will the meal be ready exactly when the guests arrive?
- Will my student finish the thesis before I run out of money?
- Will the image be processed before the arrival of the next one?
- Will the server answer the query before the attention span of the client expires?

I hope you are convinced by now that timed systems are important for *modeling* and you can use them to formulate all kinds of interesting questions in an extremely important level of abstraction. It is the level of abstraction that people use implicitly in scheduling, timing analysis, planning - you name it. It should be noted that timed automata (and other forms of timed transition systems such as timed Petri nets) are not unique in addressing this level of abstraction. For example, the work on Operations Research that involves scheduling deals with this level but it often jumps directly to a constrained optimization problem without passing through a dynamic model. Another example would be Queuing Theory which uses continuous-time stochastic processes that generate (distributions over) timed behaviors, but there, at least for a point of view of an innocent outsider, it seems that the heavy mathematical technology associated with probabilities over the reals dominates this work at the expense of the semantic view.[9] A dynamical system view of timed systems has been developed in the study of discrete-event system in Control [14] and also in the context of Max Plus algebra [6] but in the latter, due to linearity, the expressive power is rather limited.

The questions that remains is how can these timed models be used to provide *answers* to those interesting questions. To answer this particular question, let us have a retrospective look at *formal verification*, our home discipline.

3 Verification and Analysis of Timed Systems

A large part of verification is concerned with showing that components in a network of automata interact properly with each other. The term "properly" means that some sequences of events are considered acceptable while others violate the requirements. Violation means either that bad things happen, for example, two processes write simultaneously on a shared resource or an airplane crashes. Technically, such *safety* properties are violated by reaching an undesired part of the state-space. The other types of properties are called *liveness* properties and are violated when some good things do not happen, for example a client is starved to death without getting what he or she has requested.[10] The models used to verify such properties are discrete and often abstract away from data and focus on control/synchronization. The systems in question are open and under-determined and this means that a model will have *many* executions, some correct and some incorrect. Verification is a kind of *exhaustive simulation* which explores *all* the paths in a huge automaton.

Extending verification to timed systems means that, in addition to the under determination associated with external discrete actions, there is also a *dense* temporal nondeterminism concerning timing as we do not know execution times, propagation delays, inter-arrival times and process durations with precision but model them typically using *bounded intervals*. Following the pessimistic safety-critical spirit of verification, we

[9] I used to be a more zealous supporter of the semantic-dynamic approach [1] but like any other approach including those just mentioned, it has its advantages and shortcomings. Sometimes a semantically-correct "formal" approach stops at definitions and hardly computes anything.

[10] It is worth noting that for timed requirements, that is, when an upper-bound to an acceptable delay is specified, all properties can be viewed as safety properties [19].

attempt to reason *universally* about this uncertainty space, compute *all* possible behaviors under all choices of duration values and check the correctness of each of them. Without formal verification techniques this would amount to a non-countable number of simulations.

From the verification point of view we have a-priori a system with an infinite (and non-countable) state-space as the clocks are real-valued and the states of the timed automaton are of the form (s, x) with s being an automaton state and x a vector of clock valuations. Looking closer we can observe that clock valuations range practically over a bounded subset of $\mathbb{R}^n$ (clock values that go beyond the largest constant in the timing constraints are not interesting) and the restricted use of clocks in transition guards induces a finite-quotient property. More precisely, there is an equivalence relation $\sim$ over the set X of clock valuations such that $x \sim x'$ implies that the same sequences of transitions are possible from (s, x) and (s, x'). The relation is of finite index and hence a timed automaton is equivalent to a finite-state automaton with states of the from (s, R) where R is an equivalence class of $\sim$, also known as a *region*, and transitions correspond either to the passage of time from one region to another or to discrete transitions. This region graph was introduced in the seminal paper of Alur and Dill [3] which put timed automata on the map, and was used to prove decidability of the basic verification problem. Beyond this theoretical use, the region automaton is completely impractical and is not used by any living verification tool. The region equivalence is unnecessarily fine and makes distinctions between clock valuations that differ only by durations of time steps, but still admit the same sequences of transitions.

The other approach which uses a coarser equivalence has several origins [18,28,29] and in its contemporary form it computes a finite-state automaton, the *reachability graph* also known as the *simulation graph*, on the fly in a manner similar to the algorithmic analysis of hybrid systems described in [2]. The symbolic states are of the form (s, Z) where Z is a set of clock valuations belonging to a restricted type of polyhedra called *zones*, definable by conjunctions of inequalities of the form $c_1 \leq x_i \leq c_2$ or $c_1 \leq x_i - x_j \leq c_2$. Such difference constraints are fundamental to all sorts of timing and scheduling problems and they admit an efficient representation using difference-bound matrices (DBMs) invented by Bellman and proposed in the verification context in [17]. The regions of the region graph are the smallest zones possible.

To avoid large n-tuples let me illustrate zone-based reachability computation on the 2-clock automaton of Fig. 4, giving priority to clarity over precision. It starts at state q_1 with $x = (0, 0)$. Then the time elapse operator is used to compute all states reachable by letting time pass where the staying condition (invariant) $x \leq 3$ restricts the clock values with which it is possible to stay in q_1. Then this set is intersected with the transition guard $x \geq 1$ to yield the clock values with which the transition to q_2 is possible. The transition resets clock x_1 and hence the possible starting points at q_2 are $(0, y)$ where $y \in [1, 3]$. Then the time elapse operator is applied at q_2 and so on and so forth. The process is guaranteed to terminate and compute all state reachable by any choice of delay values. It has however an annoying non-intuitive feature that it shares also with verification of hybrid systems and which makes it hard to explain to potential clients of this technology. Unlike simulation of differential equations (and simulation in general) there is no simple relationship between the steps of the algorithm and the flow of time.

For example, after the first transition we are in q_2 with a set of initial clock valuations $1 \leq y \leq 3$ such that each of them has been reached at a different absolute time $t = y$, and this becomes more complicated for automata having several transitions outgoing from the same state.

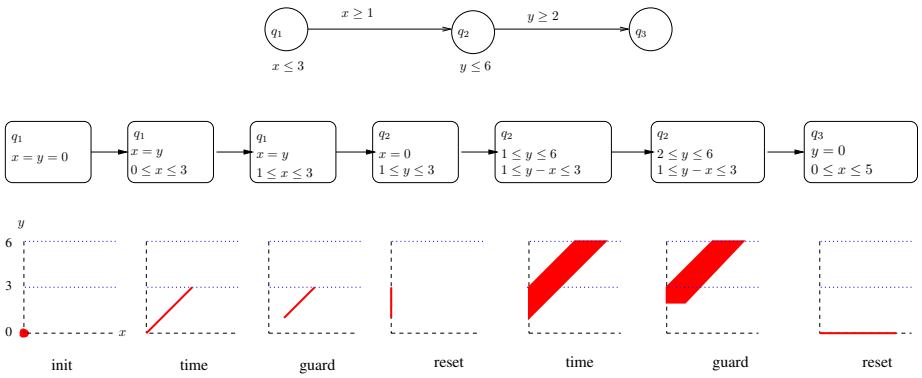


Fig. 4. First steps in computating a reachability graph

It is worth giving here two small lessons in the methodology and history of science. The first is about the mathematician's obsession with being the *first* to prove a theorem. The finite quotient property of timed systems was described in [11] in the context of timed Petri nets, six years before the conference version of [3], and it already involved zones. This went mostly unnoticed and only the later exposure of the verification community, which was already working on real-time models [20], to these ideas created the impact. So it is not only the "result" that counts but also the language that you use, the attitude and capabilities of the community to which you present your work, the right timing and more. The story told in [16] about the discovery of planets is relevant.

The second lesson is about complexity: the size of the region graph can be exponential in the number of clocks while that of the reachability graph is potentially double-exponential, but in reality the latter is almost always smaller. So proving complexity bounds on this or that problem can sometimes be no more than a sterile exercise.

Verification algorithms have been implemented and improved in a series of theses and tools. At VERIMAG, under the guidance of Joseph, this lead to the tools Kronos, Open-Kronos and IF with contributions of Sergio Yovine, Alfredo Olivero, Conrado Daws, Stavros Tripakis and Marius Bozga. The most celebrated and actively maintained tool these days is tool UPPAAL, started by Wang Yi, Paul Pettersen, and Kim Larsen as a collaboration between Uppsala and Aalborg and continued under the ongoing enthusiastic leadership of Kim with major contributions by G. Behrman and A. David. Despite enormous investments and some impressive achievements, I think it is fair to say that this approach rarely scales up beyond toy problems (and is also PSPACE-hard). Being convinced in the importance of timed systems for modeling and analysis I also spent around ten years of my life (and also those of students and collaborators) in trying to scale up and fight the clock explosion. Below is a short description of these attempts.

Numerical Decision Diagrams (NDD). One of the main problems in the verification of timed automata is the lack of a unified symbolic representation both for the discrete state-space and the clock-space. The representation is enumerative in the former and hence not suited for handling systems consisting of many components. NDDs give such a unified symbolic representation for discretized clocks encoded in binary. The technique worked well on one example but it had the deficiency of losing the metric structure of numbers via the binary encoding (known as bit blasting in the satisfiability jargon). Nevertheless it helped dispel some naive beliefs in the universal power of BDDs and clarify an important issue: dense time is *not* the main issue in timed automata but rather the symbolic representation of sets of clock valuations by inequalities.

Timed Polyhedra. Another canonical representation for unions of zones was obtained as an extension of a canonical representation for orthogonal polyhedra. This was nice theoretically but at the end did not work because of lack of efficient representation of sets of permutations.

Heuristic Search for Scheduling. In scheduling under certainty (durations are known) you have a synthesis rather than verification problem and an optimal solution corresponds to the shortest path (in terms of duration) in a timed automaton. If you do not insist on optimality you need not be exhaustive. Moreover, under certain general conditions that apply, for example, to job-shop scheduling, there is a finite and discrete set of paths to consider and you need not handle zones. This does not solve the general verification problem, but similar ideas have been tried under the title of guided search, with the goal of finding bad behaviors quickly.

Bounded Model-Checking with SMT Solvers. Bounded model-checking for timed automata, that is, the existence of a run with a bounded number of transitions, can be expressed by a formula in the logic of difference constraints. It turned out that even very strong solvers developed for this logic did not help in verifying timed automata and they even had a poorer performance than standard zone-based reachability.

Interleaving Reduction. One problem that adds to the high cost of zone-based reachability is that commuting paths do not really commute because the zone constraints remember the past history. Having shown that the union of zones reached by all interleavings of the same set of local transitions is convex, we developed a breadth-first reachability algorithm that merges such zones and for some period we held the olympic record in verifying Fisher's protocol.

Compositional Timing Analysis. This last heroic effort [10,9] was based on a divide-and-conquer methodology applied to Boolean circuits with bi-bounded delays with each gate modeled as a timed automaton according the principles explained in [25]. The automata for a sub-circuit were analyzed and then abstracted by hiding internal clocks and transition resulting in a small-size over-approximation which was plugged to the rest of the circuit. This technique could analyze a wave pipelining scheme for 3 waves of input to a non-trivial circuit of 36 gates, still a far cry from the needs of circuit timing analysis.

So was all this a waste of time? Before giving a hopefully negative answer let me reflect a bit on the state of science. Ideally one would like to apply noble first

class science and mathematics to solve real problems. For example, Formal Language Theory and Compilation, Information Theory and Telecommunication, Number Theory and Cryptography. We accept good mathematics for its own sake as well as technological innovations produced by people who do not formulate themselves in a clean mathematical way. However, we should be careful not to commit the double sin of doing mediocre mathematics over marginal questions under the pretext of hypothetical applications, but this seems to be unavoidable in the current state of affairs and the structure of scientific communities (and industry). Of course, these defects are more easily detected on others than on oneself.

4 Strange Encounters with Reality

In the sequel I report some impressions from participating in an industrial-academic project where we promised to progress toward solving the multi-core deployment problem mentioned in the introduction. The self-confidence was based partly on our acquaintance with timed automata, scheduling and SMT solvers. Most of the observations are known to many people but each person discovers the facts of life in his own path, pace and order.

Between Theoreticians and (real) Practitioners. The theoretician has the liberty to choose the problems and ignore aspects which are outside the scope of his interest and his capabilities. The real[11] practitioner does not have this choice, his deadlines are not self-imposed and his time is measured. The theoretician solves *general* problems: verification applies, in principle, to all automata, all temporal logic formulae, etc. The practitioner solves *one* problem at a time. Consequently the real-life scope of a theoretical solution is any number of problems in $[0, \infty)$. It is zero if the compromise with reality was too aggressive, and infinity if it was a clever one. As a theoretician I can observe that $[0, \infty)$ and 1 are not comparable.

Correctness and Performance. I hold the view that correctness is a special (Boolean) case of a *performance measure*, which is a way to associate cost/utility with individual system behaviors and with the system as a whole. We can measure elapsed time, associate costs with states and transitions and accumulate them along runs. We need not necessarily Booleanize them via inequalities such as deadline conditions - we can remain *quantitative*. We should provide real numbers (and vectors of real numbers when we have multiple evaluation criteria) as answers. Many people will agree on that and performance evaluation is a major issue in the embedded world and elsewhere.

Who Needs Universal Quantification? Due to safety-criticality or cost-criticality (hardware errors) verification always aspired to cover *all* possible points in the uncertainty space, in other words, a pessimistic *worst-case* attitude. This is, at the same time, *too much* and *too little* for most systems (soft real-time, best effort, mixed criticality). It is too much because if the worst-case is rare we can live with it, as we do throughout our daily life where major catastrophes are never fully excluded. This is too little because we really want to know what will *typically* happen, not only what is possible in principle. The solution in the context of timing performance is not new: replace duration

[11] The distinction between real and less real is due to Paul Caspi [13].

bounds which are without measure, that is, non-deterministic in the CS sense without probabilities, with probability distributions. This corresponds to the difference between Minkowski sum and convolution as shown in Fig. 5. In the set-theoretic tradition, when two tasks, both with an uncertain duration in $[a, b]$ each, are executed sequentially, the total duration can be anywhere in $[2a, 2b]$. Probabilistically, assuming a uniform distribution of the durations over $[a, b]$, a duration of $a + b$ is more likely than $2a$ or $2b$.

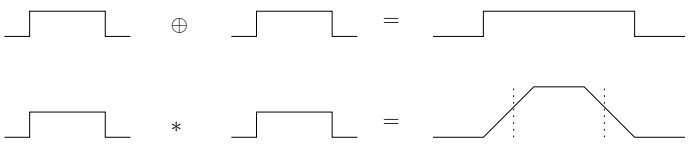


Fig. 5. From set-theoretic to probabilistic non-determinism. When two processes of duration $[a, b]$ execute one after the other, the total duration can be anywhere in their Minkowski sum $[2a, 2b]$. When the durations are assumed to be uniformly distributed, the total duration is distributed according to the convolution which still non-zero in $[2a, 2b]$ but its density is larger around the center and smaller in the tails.

The Late Discovery of Monte-Carlo Simulation. But what can you really do with such duration-probabilistic automata? Probabilizing the timing uncertainty does not alleviate the scalability problem - on the contrary, computing probabilities over sets is typically much harder than computing the sets themselves and this is what makes probabilistic verification so difficult and essentially theoretical. One direction to think about which has not been explored to the best of my knowledge is to employ *fat first search*, exploring only reachable sets of high probability. The other solution is simply to run Monte-Carlo simulations, sample the uncertainty space and collect statistics. Then we can call it *statistical model checking* to hide the fact that after 20 years we resort finally to what practitioners have always been doing. Kurt Vonnegut had an amazing observation on these matters in *Cat's Cradle*:

> *"Beware of the man who works hard to learn something, learns it, and finds himself no wiser than before... He is full of murderous resentment of people who are ignorant without having come by their ignorance the hard way."*

If we replace exhaustive verification by Monte Carlo simulation what was the worth of the exhaustive formal verification episode? One answer is that there are still systems which are critical and require exhaustive coverage. A second answer is that every domain can always benefit from a fresh look by researchers from a different culture.

The other answer is that abstract and semantically-correct modeling, if not abused, does have advantages and can help system builders that do not possess these capabilities (rather than impress them with your knowledge of Greek letters). System builders use concrete formalisms such as C and Verilog to build their systems and this coding is unavoidable if you want the system to be constructed. Abstract models such as those used in verification or performance analysis are, first of all, considered by them as an extra burden. By the way, I cannot blame them: I don't want anyone to tell me how to hack

my LaTeX code or use UML to structure my research. Secondly, many of them may have difficulties in building *abstract* models that do not correspond to something concretely executable. Consequently they use their designs as models for simulation: the software or the hardware *models itself*. When both of those already exist, this is the most efficient way to evaluate the performance (and check functional correctness). But in stages of design-space exploration when the hardware architecture or configuration is not realized, the software is run on a hardware simulator at some granularity, for example a cycle accurate simulator, and this is extremely slow. To explore different deployments it is much more efficient to use a discrete event simulator, where computations and data transfers are modeled as *timed processes* that take some time and immobilize some resource during that time. Of course you need to fill in the numbers (profiling, estimation, past experience) but recall that you need *not* be precise and deterministic.

Following these principles, the *Design-Space Explorer* prototype tool has been developed by J.-F. Kempf with help from M. Bozga and O. Lebeltel [21]. It has four components: 1) *Application description*: task-graphs annotated with execution times and data transfer rates; 2) *Input generators*: models of task arrivals (periodic, jitter, delayed periodic, bounded variability); 3) *Architecture description*: processors and their speeds, memories, busses and 4) *Deployment*: mapping and scheduling policies. From these descriptions timed automaton models are built which represent all the possible behaviors under all timing uncertainties. Then the system is analyzed using formal verification (when feasible and useful) and mostly via statistical simulation. It has been applied to compare different deployment policies for a video algorithm on a simplified model of an experimental multi-core platform developed by ST Microelectronics. Time will tell whether such modeling and analysis techniques will find their way to the design flow of dedicated multi-cores architectures and their software.

To wrap up, let me repeat once more that I consider timed automata to be one of the best inventions since the cut-and-paste. They also served as a launch pad for the study of hybrid systems. Despite the fact that they are n-tuples they can be useful, not only for the paper industry or the tool-paper industry, but to real applications. This requires more blood, sweat and tears, less theorem hunting and less bibliometry-guided research.

Acknowledgment. I would like to thank Eugene Asarin, Marius Bozga, Eric Fanchon Thomas Ferrère, Charles Rockland and P.S. Thiagarajan for commenting on various versions of this manuscript. This work was partially supported by the French project EQINOCS (ANR-11-BS02-004).

References

1. Abdeddaïm, Y., Asarin, E., Maler, O.: Scheduling with timed automata. Theoretical Computer Science 354(2), 272–300 (2006)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
3. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
4. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. J. ACM 49(2), 172–206 (2002)

5. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: Proc. IFAC Symposium on System Structure and Control, pp. 469–474 (1998)
6. Baccelli, F., Cohen, G., Olsder, G.J., Quadrat, J.-P.: Synchronization and linearity. Wiley, New York (1992)
7. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)
8. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM, pp. 3–12 (2006)
9. Ben Salah, R.: On Timing Analysis of Large Systems. PhD thesis, INP Grenoble (2007)
10. Ben-Salah, R., Bozga, M., Maler, O.: Compositional timing analysis. In: EMSOFT (2009)
11. Berthomieu, B., Menasche, M.: An enumerative approach for analyzing time petri nets. In: Proceedings IFIP (1983)
12. Bozga, M., Graf, S., Mounier, L.: IF-2.0: A validation environment for component-based real-time systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 343–348. Springer, Heidelberg (2002)
13. Caspi, P.: Réflexions sur la recherche appliquée (2004), Unpublished manuscript available at `http://perso.numericable.fr/bgrardca/TEXTES/recherche.pdf`
14. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Springer (2008)
15. Daws, C., Olivero, A., Tripakis, S., Yovine, S.: The tool KRONOS. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 208–219. Springer, Heidelberg (1996)
16. de Saint-Exupéry, A.: Le petit prince. Gallimard (1943)
17. Dill, D.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
18. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Information and Computation 111(2), 193–244 (1994)
19. Henzinger, T.A.: Sooner is safer than later. Information Processing Letters 43(3), 135–141 (1992)
20. Henzinger, T.A., Manna, Z., Pnueli, A.: Timed transition systems. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 226–251. Springer, Heidelberg (1992)
21. Kempf, J.-F.: On Computer-Aided Design-Space Exploration for Multi-Cores. PhD thesis, University of Grenoble (October 2012)
22. Maler, O.: Amir Pnueli and the dawn of hybrid systems. In: HSCC, pp. 293–295 (2010)
23. Maler, O.: On under-determined dynamical systems. In: EMSOFT, pp. 89–96 (2011)
24. Maler, O., Larsen, K., Krogh, B.: On zone-based analysis of duration probabilistic automata. In: INFINITY, pp. 33–46 (2010)
25. Maler, O., Pnueli, A.: Timing analysis of asynchronous circuits using timed automata. In: Camurati, P.E., Eveking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 189–205. Springer, Heidelberg (1995)
26. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM Journal of Research and Development 3(2), 114–125 (1959)
27. Sifakis, J., Yovine, S.: Compositional specification of timed systems. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046, pp. 347–359. Springer, Heidelberg (1996)
28. Tripakis, S., Courcoubetis, C.: Extending Promela and Spin for real time
29. Yovine, S.: Methods and tools for the symbolic verification of real-time systems. PhD thesis, INP, Grenoble (1993) (in French)
30. Yovine, S.: Kronos: A verification tool for real-time systems. STTT 1(1-2), 123–133 (1997)

Let's Get Physical: Computer Science Meets Systems

Pierluigi Nuzzo and Alberto Sangiovanni-Vincentelli

University of California at Berkeley, Berkeley CA 94720, USA
{nuzzo,alberto}@eecs.berkeley.edu

Abstract. In cyber-physical systems (CPS) computing, networking and control (typically regarded as the "cyber" part of the system) are tightly intertwined with mechanical, electrical, thermal, chemical or biological processes (the "physical" part). The increasing sophistication and heterogeneity of these systems requires radical changes in the way sense-and-control platforms are designed to regulate them. In this paper, we highlight some of the design challenges due to the complexity and heterogeneity of CPS. We argue that such challenges can be addressed by leveraging concepts that have been instrumental in fostering electronic design automation while dealing with complexity in VLSI system design. Based on these concepts, we introduce a design methodology whereby platform-based design is combined with assume-guarantee contracts to formalize the design process and enable realization of CPS architectures and control software in a hierarchical and compositional manner. We demonstrate our approach on a prototype design of an aircraft electric power system.

Keywords: Cyber-physical systems, embedded systems, VLSI systems, electronic design automation, platform-based design, contract-based design, assume-guarantee contracts, aircraft electric power system.

1 Emerging Information Technology Trends

The emerging information technology scenario features a large number of new applications which go beyond the traditional "compute" or "communicate" functions. The majority of these applications build on *distributed sense and control systems* destined to run on highly heterogeneous platforms, combining large, high-performance compute clusters (the infrastructure core or "cloud") with broad classes of mobiles, in turn surrounded by even larger swarms of microscopic sensors [16]. Such *cyber-physical systems* (CPS) [19,8,6] are characterized by the tight integration of computation with mechanical, electrical, and chemical processes: networks monitor and control the physical processes, usually with feedback loops where physics affects computation and vice versa.

CPS have the potential to radically influence how we deal with a broad range of crucial problems facing our society today, from national security and safety, to energy management, efficient transportation, and affordable health care. However, CPS complexity and heterogeneity, originating from combining what in

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 193–208, 2014.

the past have been separate worlds, tend to substantially increase *system* design and verification challenges. The cost of being late to market or of product malfunctioning is staggering as witnessed by the recent recalls and delivery delays that system industries had to bear. Toyota's infamous recall of approximately 9 million vehicles due to the sticky accelerator problem, Boeing's Airbus delay bringing an approximate toll of $6.6 billion are examples of devastating effects that design problems may cause. If this is the present situation, the problem of designing planetary-scale swarm systems appears insurmountable unless bold steps are taken to advance significantly the *science of design*.

While in traditional embedded system design the physical system is regarded as a given, the emphasis of CPS design is instead on managing dynamics, time, and concurrency by *orchestrating* networked, distributed computational resources *together* with the physical systems. Functionality in CPS is provided by an ensemble of sensing, actuation, connectivity, computation, storage and energy. Therefore, CPS design entails the convergence of several sub-disciplines, ranging from computer science, which mostly deal with computational aspects and carefully abstracts the physical world, to automatic control, electrical and mechanical engineering, which directly deals with the physical quantities involved in the design process. The inability to rigorously model the interactions among heterogeneous components and between the "physical" and the "cyber" sides is a serious obstacle to the efficient realization of CPS. Moreover, a severe limitation in common design practice is the lack of formal specifications. Requirements are written in languages that are not suitable for mathematical analysis and verification. Assessing system correctness is then left for simulation and, later in the design process, prototyping. Thus, the traditional heuristic design process based on informal requirement capture and designers' experience can lead to implementations that are inefficient and sometimes do not even satisfy the requirements, yielding long re-design cycles, cost overruns and unacceptable delays.

In this paper, we highlight the main *design challenges* for the realization of embedded systems caused by the complexity and heterogeneity of CPS. Resting on the successful achievements of electronic design automation (EDA) in taming design complexity of VLSI systems [15], we argue that such challenges can only be addressed by employing *structured and formal design methodologies* that seamlessly and coherently combine the various dimensions of the multi-scale design space and provide the appropriate abstractions. We then introduce and demonstrate a CPS design methodology by combining the platform-based design [17] and contract-based design [16] paradigms.

2 Cyber-Physical System Design Challenges

In this section we highlight the main CPS design challenges, based on the elaborations in [6] and [16]. In particular, we categorize them in terms of modeling, integration and specification challenges.

2.1 Modeling Challenges

Model-based design (MBD) [20,18] is today generally accepted as a key enabler for the design and integration of complex systems. However, CPS tend to stress all existing modeling languages and frameworks. While in computer science logic is emphasized rather than dynamics, and processes follow a sequential semantics, physical processes are generally represented using continuous-time dynamical models, expressed as differential equations, which are acausal, concurrent models. Therefore, most of the modeling challenges stem by the difficulty in accurately capturing the interactions between these two worlds.

Challenge 1—Modeling Timing and Concurrency. A first set of technical challenges in analysis and design of real-time embedded software stems from the need to bridge its inherently sequential semantics with the intrinsically concurrent physical world. All the general-purpose computation and networking abstractions are built on the premise that execution time is just an issue of performance, not correctness. Therefore, timing of programs is not repeatable, except at very coarse granularity, and programmers have hard time to specify timing behaviors within the current programming abstractions. Moreover, concurrency is often poorly modelled. Concurrent software is today dominated by threads, performing sequential computations with shared memory. Incomprehensible interactions between threads can be the sources of many problems, ranging from deadlock and scheduling anomalies, to timing variability, nondeterminism, buffer overruns, and system crashes. Finally, modeling distributed systems adds to the complexity of CPS modeling by introducing issues such as disparities in measurements of time, network delays, imperfect communication, consistency of views of system state, and distributed consensus [6].

Challenge 2—Modeling Interactions of Functionality and Implementation. To evaluate a CPS model, it is necessary to model the dynamics of software and networks. In fact, computation and communication do take time. However, pure functional models implicitly assume that data are computed and transmitted in zero time, so that the dynamics of the software and networks have no effect on system behavior. It is then essential to provide a mechanism to capture the interactions of functionality and implementation. Implementation is largely orthogonal to functionality and should therefore not be an integral part of a model of functionality. Instead, it should be possible to conjoin a functional model with an implementation model. The latter allows for design space exploration, while the former supports the design of control strategies. The conjoined models enable evaluation of interactions across these domains.

2.2 Integration Challenges

CPS integrate diverse subsystems, by often composing pieces that have been pre-designed or designed independently by different groups or companies. This is done routinely, for example, in the avionics and automotive sectors, albeit in a

heuristic and ad hoc way. Yet, integrating component models to develop holistic views of the system becomes very challenging, as summarized below.

Challenge 3—Keeping Model Components Consistent. Inconsistency may arise when a simpler (more abstract) model evolves into a more complex (refined) one, where a single component in the simple model becomes multiple components in the complex one. Moreover, non-functional aspects such as performance, timing, power or safety analysis are typically addressed in dedicated tools using specific models, which are often evolved independently of the functional ones (capturing the component dynamics), thus also increasing the risk of inconsistency. In a modeling environment, a mechanism for maintaining model consistency allows components to be copied and reused in various parts of the model while guaranteeing that, if later a change in one instance of the component becomes necessary, the same change is applied to all other instances that were used in the design. Additionally, such a mechanism is instrumental in maintaining consistency between the results of specialized analysis and synthesis tools using different representations of the same component.

Challenge 4—Preventing Misconnected Model Components. The bigger a model becomes, the harder it is to check for correctness of connections between components. Typically model components are highly interconnected and the possibility of errors increases. Errors may be due to different units between a transmitting and a receiving port (unit errors), different interpretation of the exchanged data (semantic errors), or just reversed connections among ports (transposition errors). Since none of these errors would be detected by a type system, specific measures should be enabled to automatically check for them [6].

Challenge 5—Improving Scalability and Accuracy of Model Analysis Techniques. Conventional verification and validation techniques do not scale to highly complex or adaptable systems (i.e., those with large or infinite numbers of possible states or configurations). Simulation techniques may also be affected by modeling artifacts, such as solver-dependent, nondeterminate, or Zeno behaviors [6]. In fact, CPS may be modeled as hybrid systems integrating solvers that numerically approximate the solutions to differential equations with discrete models, such as state machines, dataflow models, synchronous-reactive models, or discrete event models. Then, when a threshold must be detected, the behavior defined by a model may depend on the selected step size, which is dynamically adjusted by the numerical solver to increment time.

2.3 Specification Challenges

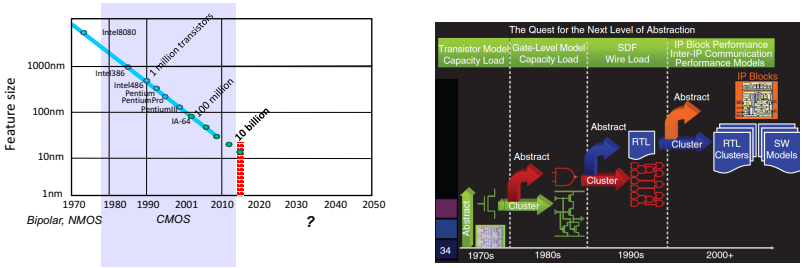
Depending on application domains, up to 50% of all errors result from imprecise, incomplete, or inconsistent and thus unfeasible requirements. The overall system product specification is somewhat of an art today, since to verify its completeness and its correctness there is little that it can be used to compare with.

Challenge 6—Capturing System Requirements. Among the many approaches taken in industry for getting requirements right, some of them are meant for initial system requirements, mostly relying on ISO 26262 compliant approaches. To cope with the inherently unstructured problem of (in)completeness of requirements, industry has set up domain- and application-class specific methodologies. As particular examples, we mention learning processes, such as the one employed by Airbus to incorporate the knowledge base of external hazards from flight incidents, and the Code of Practice proposed by the Prevent Project, using guiding questions to assess the completeness of requirements in the concept phase of the development of advanced driver assistance systems. Use-case analysis methods as advocated for UML based development processes follow the same objective. A common theme of these approaches is the intent to systematically identify those aspects of the environment of the system under development whose observability is necessary and sufficient to achieve the system requirements. However, the most efficient way of assessing completeness of a set of requirements is by executing it, which is only possible if semi-formal or formal specification languages are used, where the particular shape of such formalizations is domain dependent.

Challenge 7—Managing Requirements. Design specifications tend to move from one company (or one division) to the next in non-executable and often unstable and imprecise forms, thus yielding misinterpretations and consequent design errors. In addition, errors are often caught only at the final integration step as the specifications were incomplete and imprecise; further, nonfunctional specifications (e.g., timing, power consumption, size) are difficult to trace. It is common practice to structure system level requirements into several "chapters", "aspects", or "viewpoints", quite often developed by different teams using different skills, frameworks, and tools. Without a clean approach to handle multiple viewpoints, the common practice today is to discard some of the viewpoints in a first stage, e.g., by considering only functions and safety. Designs are then developed based on these only viewpoints. Other viewpoints are subsequently taken into account (e.g., timing, energy), thus resulting in late and costly modifications and re-designs.

3 Coping with Complexity in VLSI Design: Lessons Learned

Over the past decades, a major driver for silicon microelectronics research has been Moore's law, which conjectures the continued shrinkage of critical chip dimensions (see Fig. 1 (a)). Microelectronic progress became so predictable that the Semiconductor Industry Association (SIA) developed a road-map to help defining critical steps and sustaining progress; the material science and material processing research community has successfully met the challenges, maintaining a steady stream of results supporting continued scaling of CMOS devices to smaller dimensions. By taking full advantage of the availability of billion-transistor chips, increasingly higher performance Systems-on-Chip (SoC) are



(a) (b)

Fig. 1. (a) Reduction of minimum feature size with time in VLSI systems and (b) Levels of abstraction in VLSI design (Figure from [15])

being fabricated today, thus enabling new architectural approaches to information processing and communication. The appearance of new nano-scale devices is expected to revolutionize the way information is processed on chip, and perhaps more significantly, have a major impact on emerging applications at the intersection of the biological, information and micro-electro-mechanical worlds.

3.1 Dealing with Moore's Law

Dealing with steady increase in complexity over the decades has been made possible only because of the continuous increase of productivity brought by electronic design automation.

By looking back at the history of design methods, we can infer how the changes in design productivity have always been associated with a *rise in the level of abstraction of design capture*. As can be seen in Fig. 1 (b), in 1971, the highest level of abstraction for digital integrated circuits was the schematic of a transistor; ten years later, it became the digital gate; by 1990, the use of hardware description languages (HDL) was pervasive, and design capture was done at the register transfer level (RTL). Dealing with blocks of much coarser granularity than in the past has become essential in order to cope with the productivity increase the industry is asked to provide. The recent emphasis on SoC, bringing system-level issues into chip design, is a witness to this trend.

One of these issues relates to the concept of *system decomposition* and *integration out of pre-designed intellectual property (IP) blocks*. Although top-down decomposition has been customarily adopted by the semiconductor industry for years, it presents some limitations as a designer or a group of designers has to fully comprehend the entire system and to partition appropriately its various parts, a difficult task given the enormous complexity of today's systems. As mentioned in Section 2.2, an alternative is to develop systems by composing pre-designed pieces, whereby preserving compositionality is essential: the building blocks should be designed so that their properties are maintained when connected together to allow reuse without the need for expensive verification steps.

Decomposition and abstraction have been two basic approaches traditionally used to manage design complexity. However, complexity has been also managed *by "construction"*, i.e. by "artificially" constraining the space to regular, or modular design styles that can ease design verification (e.g. by enforcing regular layout and synchronous design), and by *structured methodologies*, which start high in the abstraction layers and define a number of refinement steps that go from the initial description to the final implementation.

The design problems faced in SoC design are very similar to the ones discussed in Section 2, the main difference between them being the importance given to time-to-market and to the customer appeal of the products versus safety and hard-time constraints. Several languages and design tools have been proposed over the years to enable checking system level properties or explore alternative architectural solutions for the same set of requirements. Among others, we recall generic modeling frameworks, such as Matlab/Simulink[1] or Ptolemy II[2], hardware description languages, such as Verilog[3] or VHDL[4], transaction-level modeling tools, such as SystemC[5], together with their respective analog-mixed-signal extensions[6], modeling languages for architecture modeling, such as SysML[7] and AADL[8]. Some of these tools focus on simulation while others are geared towards performance modeling, analysis and verification. However, the design technology challenge is to address the entire system design process and not to consider only point solutions of methodology, tools, and models that ease part of the design. This calls for new modeling approaches that can mix different physical systems, control logic, and implementation architectures. In doing so, existing approaches, models, and tools should be subsumed and not eliminated in order to be smoothly incorporated in current design flows. A design platform should then be developed to host the new techniques and to integrate a set of today's poorly interconnected tools.

3.2 System Design Methodology

The considerations above motivate the view that a unified methodology and framework could be used across several different industrial domains. Among the lines of attack developed by research institutions and industry to cope with the exponential complexity growth, a design paradigm of particular interest to the development of embedded systems is the V-model, a widely accepted scheme in the defense and transportation domains.

[1] http://www.mathworks.com/products/simulink
[2] http://ptolemy.eecs.berkeley.edu
[3] http://www.verilog.com/
[4] http://www.vhdl.org
[5] http://www.accellera.org/downloads/standards/systemc
[6] http://www.eda.org/verilog-ams/
http://www.eda.org/vhdl-ams/
http://www.accellera.org/downloads/standards/systemc/ams
[7] http://www.omg.org/spec/SysML
[8] http://www.aadl.info/aadl/currentsite

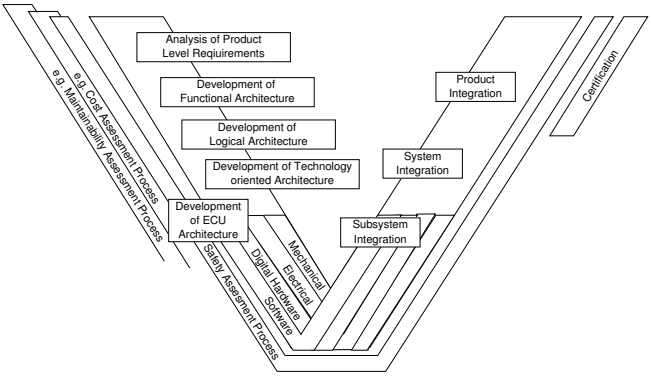


Fig. 2. The V-Model

The *V-model* was originally developed for defense applications by the German DoD.[9] It structures the product development processes into a *design* and an *integration* phase along variations of the V diagram shown in Fig. 2. Specifically, following product level requirement analysis, subsequent steps would first evolve a functional architecture supporting product level requirements. Sub-functions are then re-grouped taking into account re-use and product line requirements into a logical architecture, whose modules can be developed independently, e.g., by different subsystem suppliers. The realization of such modules often involves mechatronic design. The top-level of the technology-oriented architecture would then show the mechatronic architecture of the module, defining interfaces between the different domains of mechanical, hydraulic, electrical, and electronic system design. Subsequent phases would then unfold the detailed design for each of these domains, such as the design of the electronic subsystem involving among others the design of electronic control units (ECU). These design phases are paralleled by integration phases along the right-hand part of the V, such as integrating basic and application software on the ECU hardware to actually construct the electronic control unit, integrating the complete electronic subsystems, integrating the mechatronic subsystem to build the module, and integrating multiple modules to build the complete product. An integral part of V-based development processes are testing activities, where at each integration level test-suites developed during the design phases are used to verify compliance of the integrated entity to their specification. Since system integration and validation may often be performed too late in the design flow, there is limited ability to predict, early in the design process, the consequences of a design decision on system performance and the cost of radical departures from known designs. Therefore, design-space exploration is rarely performed adequately, yielding suboptimal designs where the architecture selection phase does not consider extensibility, re-usability, and fault tolerance to the extent that is needed to reduce cost, failure rates, and time-to-market.

[9] `http://www.v-model-xt.de`

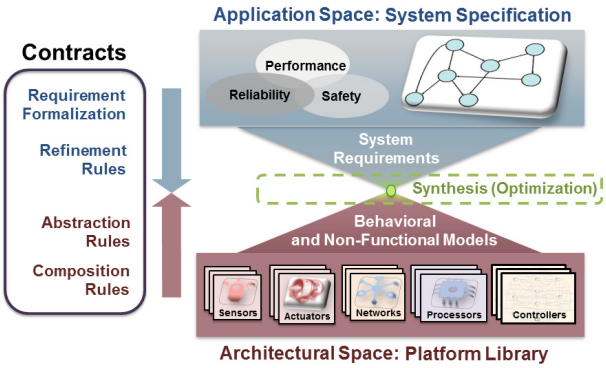


Fig. 3. Platform-based design and the role of contracts

Platform-based design (PBD) was introduced in the late 1980s to capture a design process that could encompass horizontal and vertical decompositions, and multiple viewpoints and in doing so, support the supply chain as well as multi-layer optimization [17].

Platform-based design addresses already many of the challenges outlined in Section 2. Its concepts have been applied to a variety of very different domains: from automotive, to System-on-Chip, from analog circuit design, to building automation, to synthetic biology. By limiting the design space to the platform library, it allows efficient design exploration and optimization, aiming to correct-by-construction solutions (Challenge 5). Moreover, the meet-in-the-middle approach, where functional models are combined with non-functional models, and successive top-down refinements of high-level specifications across design layers are mapped onto bottom-up abstractions and characterizations of potential implementations, allows effectively coupling system functionality and architecture (Challenge 1, 2 and 5), as represented in Fig. 3. However, to successfully deploy such a methodology, we need rigorous mechanisms for (i) determining *valid* compositions of compatible components so that when the design space is explored, only *legal* compositions are taken into consideration; (ii) guaranteeing that a component at a higher level of *abstraction* is an accurate representation of a lower level component (or aggregation of components); (iii) checking that an architecture platform is indeed a correct *refinement* of a specification platform, and (iv) formalizing top-level system *requirements*. In the following section, we show how such goals can be achieved by combining platform-based design with the concept of *contracts*.

4 Platform-Based Design with Contracts

The notion of contracts originates in the context of assume-guarantee reasoning. Informally, a contract is a pair $\mathcal{C} = (A, G)$ of properties, assumptions and guarantees, respectively representing the assumptions on the environment and the

promises of the system under these assumptions. The essence of contracts is a compositional approach, where design and verification complexity is reduced by decomposing system-level tasks into more manageable subproblems at the component level, under a set of assumptions. System properties can then be inferred or proved based on component properties.

Compositional reasoning has been known for a long time, but it has mostly been used as a verification mean for the design of software. Rigorous contract theories have then been developed over the years, including assume-guarantee (A/G) contracts [4] and interface theories [1]. However, their concrete adoption in CPS design is still at its infancy. Examples of application of A/G contracts have only been recently demonstrated in the automotive [5] and consumer electronics [12] domains. The use of A/G contracts for control design in combination with PBD has been advocated in [16], while in [13,11], a PBD methodology is first introduced that uses contracts to integrate heterogeneous modeling and analysis frameworks for synthesis and optimization of CPS architectures and control protocols. The design flow is demonstrated on a real-life example of industrial interest, namely the design of system topology and supervisory control for aircraft electric power systems (EPS).

4.1 Contracts

We summarize the main concepts behind our methodology by presenting a simple generic contract model centered around the notion of platform *component*. A platform component $\mathcal{M}$ can be seen as an abstraction representing an element of a design, characterized by a set of *attributes*, including: *variables* (input, output and internal), configuration *parameters*, and *ports* (input, output and bidirectional); a *behavioral model*, uniquely determining the values of the output and internal variables given the values of the input variables and configuration parameters, and a set of *non-functional models*, i.e. maps that allow computing non-functional attributes of a component, corresponding to particular valuations of its input variables and configuration parameters. Components can be connected together by sharing certain ports under constraints on the values of certain variables. In what follows, we use variables to denote both component variables and ports. A component may be associated with both implementations and contracts. An *implementation* M is an instantiation of a component $\mathcal{M}$ for a given set of configuration parameters. In the following, we also use M to denote the set of behaviors of an implementation, which assign a history of "values" to ports. Behaviors are generic and abstract. For instance, they could be continuous functions that result from solving differential equations, or sequences of values or events recognized by an automata model.

A *contract* $\mathcal{C}$ for a component $\mathcal{M}$ is a pair of assertions (A, G), called the *assumptions* and the *guarantees*, each representing a specific set of behaviors over the component variables [4]. An implementation M satisfies an assertion B whenever M and B are defined over the same set of variables and all the behaviors of M satisfy the assertion, i.e. when $M \subseteq B$. An implementation of a component satisfies a contract whenever it satisfies its guarantee, subject to

the assumption. Formally, $M \cap A \subseteq G$, where M and $\mathcal{C}$ have the same variables. We denote such a *satisfaction* relation by writing $M \models \mathcal{C}$. An implementation E is a legal *environment* for $\mathcal{C}$, i.e. $E \models_E \mathcal{C}$, whenever $E \subseteq A$. Two contracts $\mathcal{C}$ and $\mathcal{C}'$ with identical variables, identical assumptions, and such that $G' \cup \neg A = G \cup \neg A$, where $\neg A$ is the complement of A, possess identical sets of environments and implementations. Such two contracts are then *equivalent*. In particular, any contract $\mathcal{C} = (A, G)$ is equivalent to a contract in *saturated form* (A, G'), obtained by taking $G' = G \cup \neg A$. Therefore, in what follows, we assume that all contracts are in saturated form. A contract is *consistent* when the set of implementations satisfying it is not empty, i.e. it is feasible to develop implementations for it. For contracts in saturated form, this amounts to verify that $G \neq \emptyset$. Let M be any implementation, i.e. $M \models \mathcal{C}$, then $\mathcal{C}$ is *compatible*, if there exists a legal environment E for M, i.e. if and only if $A \neq \emptyset$. The intent is that a component satisfying contract $\mathcal{C}$ can only be used in the context of a compatible environment.

Contracts associated to different components can be combined according to different rules. Similar to parallel composition of components, *parallel composition* ($\otimes$) of contracts can be used to construct composite contracts out of simpler ones. Let M_1 and M_2 two components that are composable to obtain $M_1 \times M_2$ and satisfy, respectively, contracts $\mathcal{C}_1$ and $\mathcal{C}_2$. Then, $M_1 \times M_2$ is a valid composition if M_1 and M_2 are *compatible*. This can be checked by first computing the contract composition $\mathcal{C}_{12} = \mathcal{C}_1 \otimes \mathcal{C}_2$ and then checking whether $\mathcal{C}_{12}$ is compatible. To compose multiple views of the same component that need to be satisfied simultaneously, the *conjunction* ($\wedge$) of contracts can also be defined so that if $M \models \mathcal{C}_1 \wedge \mathcal{C}_2$, then $M \models \mathcal{C}_1$ and $M \models \mathcal{C}_2$. Contract conjunction can be computed by defining a preorder on contracts, which formalizes a notion of *refinement*. We say that $\mathcal{C}$ refines $\mathcal{C}'$, written $\mathcal{C} \preceq \mathcal{C}'$ if and only if $A \supseteq A'$ and $G \subseteq G'$. Refinement amounts to relaxing assumptions and reinforcing guarantees, therefore strengthening the contract. Clearly, if $M \models \mathcal{C}$ and $\mathcal{C} \preceq \mathcal{C}'$, then $M \models \mathcal{C}'$. On the other hand, if $E \models_E \mathcal{C}'$, then $E \models_E \mathcal{C}$. Mathematical expressions for computing contract composition and conjunction can be found in [4].

Horizontal and Vertical Contracts. Since compatibility is assessed among components at the same abstraction layer, the first category of contracts presented above can be denoted as *horizontal contracts*. On the other hand, *vertical contracts* can also be used to verify whether the system obtained by composing the library elements according to the horizontal contracts satisfies the requirements posed at the higher level of abstraction. If these sets of contracts are satisfied, the mapping mechanism of PBD can be used to produce design refinements that are correct by construction. Vertical contracts are tightly linked to the notions of mapping of an application onto an implementation platform [12]. However, compositional techniques that check correct refinement on each subsystem independently are not effective, in general, since the specification architecture at level $l+1$ may be defined in an independent way, and does not generally match the implementation architecture at level l. Let $\mathcal{S} = \otimes_{i \in I} \mathcal{S}_i$ and $\mathcal{A} = \otimes_{j \in J} \mathcal{A}_j$ be the two contracts describing the specification and implementation platforms,

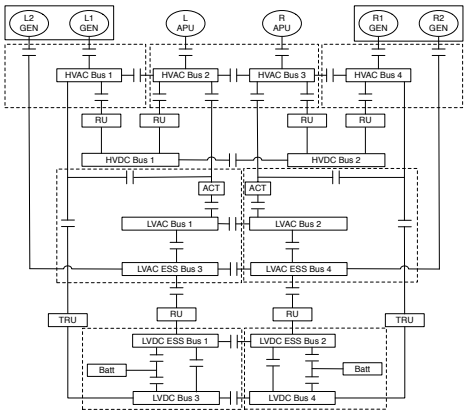


Fig. 4. Single-line diagram of an aircraft electric power system (Figure from [13])

respectively. Therefore, verification of vertical contracts can be performed through mapping of the application to the implementation platform as follows. In this example, the specification and implementation contracts are compositionally defined out of I and J components, which may not directly match. Then, the mapping of the specification over the implementation can be modelled by the composition $\mathcal{S} \otimes \mathcal{A}$, and checking vertical contracts becomes equivalent to checking that $\mathcal{S} \otimes \mathcal{A}$ refines $\mathcal{S}$, which can be performed compositionally.

4.2 A Contract-Based Design Flow for CPS

We now show how the challenges discussed in Section 2 can effectively be addressed by a platform-based design methodology using contracts. As an example, we consider the embedded control design problem in [13]. Fig. 4 shows a sample structure of an aircraft EPS in the form of a single-line diagram, a simplified notation for three-phase power systems. Generators deliver power to the loads via buses. Essential loads cannot be unpowered for more than a predefined period t_{max}, i.e. a typical specification would require that the failure probability for an essential load (i.e., the probability of being unpowered for longer than t_{max}) be smaller than 10^{-9} per flight hour. Contactors are electromechanical switches that are opened or closed to determine the power flow from sources to loads. The goal is to design the system topology (e.g. number and interconnection of components) and the controller to accommodate all changes in system conditions or failures, and reroute power by appropriately actuating the contactors, so that essential buses are adequately powered.

In our design flow, pictorially represented in Fig. 5, platform component design and characterization is completely orthogonalized from system specification and algorithm design.

Platform Library Generation. In the bottom-up phase of the design process, a library of components (and contracts) is generated to model (or specify) both

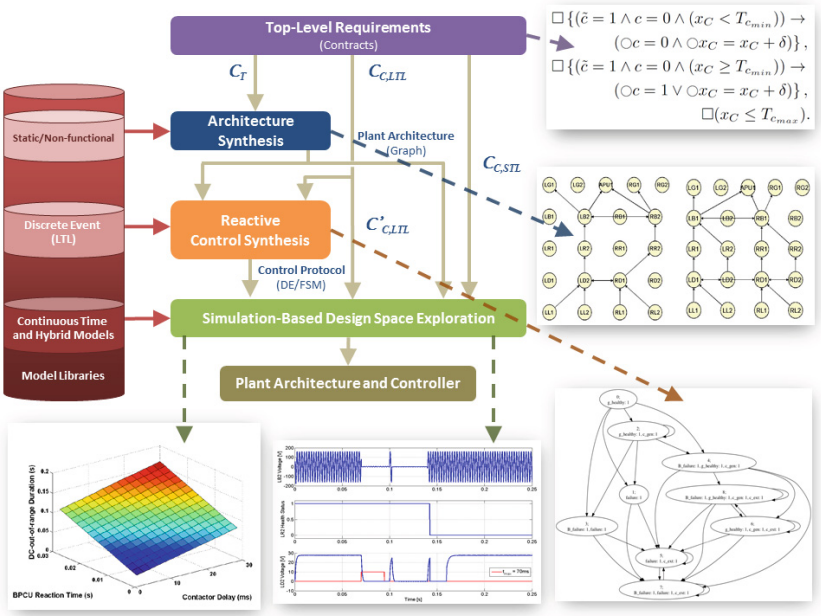


Fig. 5. Contract-based CPS design flow and its demonstration to an aircraft electrical power system [13]

the plant architecture (e.g. the power system topology in Fig. 5) and the controller. Components can be hierarchically organized to represent the system at different levels of abstraction, e.g. *steady-state* (static), *discrete-event* (DE), and *hybrid* levels. At each level of abstraction, components are also capable of exposing multiple, complementary *views*, associated with different design concerns (e.g. safety, performance, reliability) and with models that can be expressed via different formalisms (e.g. graphs, linear temporal logic, differential equations in Fig. 5), and analyzed by different tools. Such models include non-functional (performance) metrics, such as timing, energy and cost. Contracts allow checking consistency among models as the library evolves, which addresses Challenge 3.

Requirement Formalization. In the top-down phase of the design process, top-level system requirements are formalized as contracts. Responsibilities of achieving requirements are split into those to be established by the system (guarantees) and those characterizing admissible environments (assumptions). As an example, controller requirements can be expressed as a contract $\mathcal{C}_C = (A_C, G_C)$, where A_C encodes the allowable behaviors of the environment (physical plant) and G_C encodes the top-level system requirements. To define $\mathcal{C}_C$, formal specification languages can be used, e.g. linear temporal logic (LTL) [14] and signal temporal logic (STL) [9] in Fig. 5, to allow reasoning about temporal aspects of systems at different levels of abstraction. Using contracts resting on logic-based formalisms comes with the advantage that "spurious" unwanted behaviors can be excluded

by "throwing in" additional contracts, or strengthening assumptions, or by considering additional cases for guarantees, thus addressing Challenge 6. A second advantage rests in the capability of checking for consistency by providing effective tests, whether a set of contracts is realizable, or whether, in contrast, facets of these are inherently conflicting, and thus no implementation is feasible, which addresses Challenge 7. By reflecting the model library, the particular shape of requirement formalizations is also viewpoint and domain dependent. To address Challenge 1, such system-level models should still come with a rigorous temporal semantics that allows specifying the interaction between the control program and the physical dynamics so as to model timing and concurrency at a higher abstraction level in a way that is largely independent of underlying hardware details. For instance, LTL allows reasoning about the temporal behaviors of systems characterized by Boolean, discrete-time signals or sequences of events (DE abstraction). On the other hand, STL deals with dense-time real signals and hybrid dynamical models that mix the discrete dynamics of the controller with the continuous dynamics of the plant (hybrid abstraction).

Mapping Functions to Implementations. By leveraging models expressed in different formalisms, the design is cast as a set of problems mapping functions over implementations. The mapping problem is a synthesis problem that can be solved by either leveraging pre-existing synthesis tools, or by casting an optimization problem that uses information from both the system and the component levels to evaluate global tradeoffs among components.

In the example of Fig. 5, contract $\mathcal{C}_T$ is first used together with steady-state models of the plant components and a template of the EPS topology (represented as a graph) to synthesize a final topology that minimize component cost subject to connectivity, power flow and reliability constraints, all expressed as mixed integer-linear constraints. $\mathcal{C}_{C,LTL}$ is then used together with DE models of the plant components (also described by LTL formulas) and the EPS topology, to synthesize a reactive control protocol in the form of one (or more) state machines. Reactive synthesis tools [10,7] can be used to generate control logic from LTL A/G contracts. The resulting controller will then satisfy $\mathcal{C}_{C,LTL}$ by construction. Satisfaction of $\mathcal{C}_{C,STL}$ is then assessed on a hybrid model, including both the controller and an acausal, equation-based representation of the plant, by monitoring simulation traces while optimizing a set of system parameters. Contracts are captured as optimization constraints. The resulting optimal controller configuration is returned as the final design.

Horizontal contracts allow checking or enforcing compatibility and correct connections among components, thus addressing Challenge 4. Vertical contracts allow checking or enforcing correct abstraction and refinement relations, thus maintaining consistency among platform instances, models and requirements at different abstraction levels (Challenge 4 and 7). Moreover, in control design, vertical contracts define relations between the properties of the controller and the ones of its execution platform, which helps address Challenge 2. Typically, the controller defines requirements in terms of several aspects that include the timing behavior of the control tasks and of the communication between tasks,

their jitter, the accuracy and resolution of the computation, and, more generally, requirements on power and resource consumption. These requirements are taken as assumptions by the controller, which in turn provides guarantees in terms of the amount of requested computation, activation times and data dependencies.

The association of functionality to architectural services to evaluate the characteristics (such as latency, throughput, power, and energy) of a particular implementation by simulation (Challenge 2) can be supported by frameworks such as Metropolis [2,3], which is founded on design representation mechanisms that can be shared across different models of computation and different layers of abstraction. A typical design scenario would then entail a front-end *orchestrator* routine responsible for coordinating a set of back-end specialized synthesis and optimization frameworks, each dealing with a different representation of the platform, and consistently processing their results. To maintain such consistency and improve on the scalability of the specific synthesis and optimization algorithms (Challenge 5), such an orchestrator should maximally leverage the modularity offered by contracts, by directly working on their representations to perform compatibility, consistency and refinement checks on system portions of manageable size and complexity.

5 Conclusions

Dealing with the heterogeneity and complexity of cyber-physical systems requires innovations in design technologies and tools. In this paper, we have advocated the need for a design and integration platform that can operate at different levels of abstraction, orchestrate hardware and software, digital and analog, cyber and physical subsystem design, as well as facilitate the integration of IP blocks and tools. Then, we have introduced a platform-based design methodology enriched with contracts and demonstrated its potential to provide the foundations for such a framework.

Acknowledgments. This work was supported in part by IBM and United Technologies Corporation (UTC) via the iCyPhy consortium and by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

1. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. Symp. Foundations of Software Engineering, pp. 109–120. ACM Press (2001)
2. Balarin, F., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L., Watanabe, Y.: Metropolis: an integrated electronic system design environment. Computer 36(4), 45–52 (2003)

3. Balarin, F., Davare, A., D'Angelo, M., Densmore, D., Meyerowitz, T., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A., Simalatsar, A., Watanabe, Y., Yang, G., Zhu, Q.: Platform-based design and frameworks: METROPOLIS and METRO II. In: Nicolescu, G., Mosterman, P.J. (eds.) Model-Based Design for Embedded Systems, ch. 10, p. 259. CRC Press, Taylor and Francis Group, Boca Raton, London (2009)
4. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple Viewpoint Contract-Based Specification and Design. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 200–225. Springer, Heidelberg (2008)
5. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., et al.: Contracts for System Design. Rapport de recherche RR-8147, INRIA (November 2012)
6. Derler, P., Lee, E.A., Sangiovanni-Vincentelli, A.: Modeling cyber-physical systems. Proc. IEEE 100(1), 13–28 (2012)
7. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science, vol. 2, pp. 995–1072 (1990)
8. Lee, E.A.: Cyber physical systems: Design challenges. In: Proc. IEEE Int. Symposium on Object Oriented Real-Time Distributed Computing, pp. 363–369 (May 2008)
9. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT 2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004)
10. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems: Specification, vol. 1. Springer (1992)
11. Nuzzo, P., Finn, J.B., Iannopollo, A., Sangiovanni-Vincentelli, A.L.: Contract-based design of control protocols for safety-critical cyber-physical systems. In: Proc. Design, Automation and Test in Europe (March 2014)
12. Nuzzo, P., Sangiovanni-Vincentelli, A., Sun, X., Puggelli, A.: Methodology for the design of analog integrated interfaces using contracts. IEEE Sensors J. 12(12), 3329–3345 (2012)
13. Nuzzo, P., Xu, H., Ozay, N., Finn, J., Sangiovanni-Vincentelli, A., Murray, R., Donze, A., Seshia, S.: A contract-based methodology for aircraft electric power system design. IEEE Access 2, 1–25 (2014)
14. Pnueli, A.: The temporal logic of programs. In: Symp. Foundations of Computer Science, vol. 31, pp. 46–57 (November 1977)
15. Sangiovanni-Vincentelli, A.: Corsi e ricorsi: The EDA story. IEEE Solid State Circuits Magazine 2(3), 6–26 (2010)
16. Sangiovanni-Vincentelli, A., Damm, W., Passerone, R.: Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. European Journal of Control 18(3), 217–238 (2012)
17. Sangiovanni-Vincentelli, A.: Quo vadis, SLD? reasoning about the trends and challenges of system level design. Proceedings of the IEEE 95(3), 467–506 (2007)
18. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
19. Sztipanovits, J.: Composition of cyber-physical systems. In: Proc. IEEE Int. Conf. and Workshops on Engineering of Computer-Based Systems, pp. 3–6 (March 2007)
20. Sztipanovits, J., Karsai, G.: Model-integrated computing. IEEE Computer 30(4), 110–112 (1997)

What Can be Computed in a Distributed System?

Michel Raynal

Institut Universitaire de France
& IRISA, Université de Rennes, France
& Department of Computing, Polytechnic University, Hong Kong
raynal@irisa.fr

Abstract. Not only the world is distributed, but more and more applications are distributed. Hence, a fundamental question is the following one: What can be computed in a distributed system? The answer to this question depends on the environment in which evolves the considered distributed system, i.e., on the assumptions the system relies on. This environment is very often left implicit and nearly always not formulated in terms of precise underlying requirements. In the extreme case where the environment is such that there is no synchrony assumption and the computing entities may commit failures, many problems become impossible to solve (in these cases, a network of Turing machines where some machines may crash, is less powerful than a single reliable Turing machine). Given a distributed computing problem, it is consequently important to know the weakest assumptions (lower bounds) that give the limits beyond which the considered distributed problem cannot be solved. This paper is a short introduction to this kind of issues. It first presents a few of elements related to distributed computability, and then briefly addresses distributed complexity issues. The style of the paper is voluntarily informal.

Keywords: Agreement, Asynchronous system, Atomicity, Concurrency, Consensus, Crash failure, Distributed complexity, Distributed computability, Distributed computing, Environment, Fault-tolerance, Impossibility, Indulgence, Message adversary, Message-passing system, Progress condition, Read/write system, Synchronous system, Universal construction, Wait-freedom.

1 Definitions

Distributed Computing. Distributed computing was born in the late seventies when researchers and engineers started to take into account the intrinsic characteristic of physically distributed systems [39]. *Distributed computing* arises when one has to solve a problem involving physically distributed entities (called processes, processors, agents, actors, sensors, peers, etc.), such that each entity (a) has only a partial knowledge of the many input parameters of the problem to be solved, and (b) has to compute local outputs which may depend on some non-local input parameters. It follows that the computing entities have necessarily to exchange information and cooperate [52].

Distributed System. A (static) distributed system is made up of n sequential deterministic processes, denoted $p_1, \ldots, p_n$. These processes communicate and synchronize

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 209–224, 2014.

through a communication medium, which is either a network that allows the processes to send and receive messages, or a set of atomic read/write registers (atomic registers could be replaced by "weaker" safe or regular registers, but as shown in [40] –where these registers are defined– safe, regular and atomic registers have the same computational power).

Deterministic means here that the behavior of a process is entirely determined from its initial state, the algorithm it executes, and –according to the communication medium– the sequence of values read from atomic registers or the sequence of received messages (hence, obtaining different sequences of values or receiving messages in a different order can produce different behaviors).

Asynchronous Read/Write or Message-Passing System. In an asynchronous (also called time-free) read/write system, the processes are asynchronous in the sense that, for each of them, there is no assumption on its speed (except that it is positive).

If the communication is by message-passing, the network also is asynchronous, namely, the transfer duration of any message is finite but arbitrary.

Synchronous Message-Passing System. Differently, the main feature of a synchronous system lies in the existence of an upper bound on message transfer delays. Moreover, (a) this bound is known by the processes, and (b) it is assumed that processing durations are negligible with respect to message transfer delays; consequently processing are assumed to have zero duration.

This type of synchrony is abstracted by the notion of round-based computation. The processes proceed in rounds during which each process first sends messages, then, receive messages, and executes local computation. The fundamental assumption which characterizes a synchronous message-passing system is that a message sent during a round is received by its destination process during the very same round.

Process Crash Failure. The most common failure studied in distributed computing is the process crash failure. Such a failure occurs when a process halts unexpectedly. Before crashing it executes correctly its algorithm, and after having crashed, it never recovers.

Let t be the maximal number of processes that may crash; t is a model parameter and the model is called t-resilient model. The asynchronous distributed computing (read/write or message-passing) model in which all processes, except one, may crash is called *wait-free* model. Hence, *wait-free model* is synonym of $(n-1)$-resilient model.

The Notion of Environment and Non-determinism. The *environment* of a distributed system is the set of failures and (a)synchrony patterns in which the system may evolve. Hence, a system does not master its environment but suffers it.

As processes are deterministic, the only non-determinism a distributed system has to cope with is the non-determinism created by its environment.

Complexity vs. Computability Issues. Computability and complexity are the two lenses that allows us to understand and master computing. The following table presents

the main issues encountered in distributed computing, when considering these two lenses.

	Synchronous	Asynchronous
Failure-free	complexity	complexity
Failure-prone	complexity	computability

The rest of this paper illustrates and develops this table. It first addresses computability issues in asynchronous crash-prone distributed systems and presents several ways that have been proposed to circumvent these impossibilities. It then addresses the case of crash-prone synchronous systems, and finally the case of crash-free systems.

2 Are Asynchronous Crash-Prone Distributed Systems Universal?

On the Notion of a Universal Construction. In sequential computing, computability is understood through the Church-Turing's thesis (namely, anything that can be computed, can be computed by a Turing machine). Moreover, when considering the notion of a *universal algorithm* encountered in sequential computing, such an algorithm "has the ability to act like any algorithm whatsoever. It accepts as inputs the description of *any* algorithm A and *any* legal input X, and simply runs, or simulates, A on X. [...] In a sense, a computer [...] is very much like a universal algorithm." [25].

Hence, the question: Is it possible to design a universal algorithm/machine on top of an asynchronous crash-prone distributed system? As we are about to see, it happens that, due the environment (asynchrony and process failures) of a distributed system, and the fact that it cannot control it, distributed computability has a different flavor than computability in sequential computing. Moreover, this is independent of the fact that the communication is by read/write registers or message-passing.

Due to its very nature, distributed computing requires cooperation among the processes. Intuitively, the computability issues come from the fact that, due to the net effect of asynchrony and failures, a process can be unable to know if another process has crashed or is only slow (or equivalently if the channel connecting these processes is slow). Moreover, this is true whatever the individual power of each process. To cite [30], "It follows that the limits of computability reflect the difficulty of making decisions in the face of ambiguity, and have little to do with the inherent computational power of individual participants".

A Universality Notion for Distributed Computing. A *concurrent* object is an object that can be accessed by several processes. Let us consider a concurrent object Z defined by a sequential specification on a set of total operations. An operation is *total* is, when executed alone, it always returns a result. A specification is sequential, if all the correct behaviors of the object can be described by sequences of operations.

The notion of universality we are interested in concerns the possibility to implement any concurrent object such as Z, despite asynchrony and crashes. If it exists, such an implementation, which takes the sequential specification of Z as input and builds a corresponding concurrent object, is called a *universal construction*. This is depicted in Fig. 1.

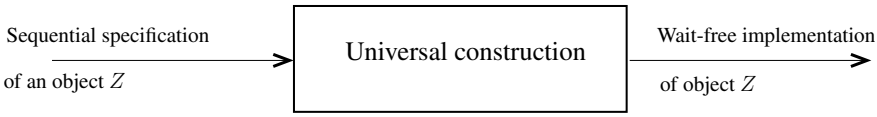


Fig. 1. From a sequential specification to a wait-free implementation

In some cases the object Z encapsulates a service which can be abstracted as a state machine. A replication-based universal construction of such an object Z is usually called a *state machine replication* algorithm [39]. Let us remark that the object Z could also be a Turing machine.

On the Liveness Property Associated with the Constructed Object. Several liveness properties can be associated with the constructed object Z. We consider here *wait-freedom* [27]. The term "wait-freedom" has here a meaning different from the one used in "wait-free model". More precisely, it is here a progress condition for the operations of the constructed objects Z, namely, it states that a universal construction satisfies the "wait-freedom progress condition" if the invocation of an operation on Z by a process can fail to terminate only if the invoking process crashes while executing the operation. We say then (by a slight abuse of language) that the implementation is wait-free. This means that the operation has to terminate even if all the processes, except the invoking process, crash. Let us observe that a wait-free implementation prevents the use of locks (a process that would crash after acquiring a lock could block the system, thereby preventing wait-freedom).

Wait-freedom is the strongest possible progress condition that can be associated with a universal construction (object implementation) in the wait-free model. Other progress conditions suited to the wait-free model are *obstruction-freedom* [29] and *non-blocking* [33]. (See chapter 5 of [51] for a guided tour on progress conditions.)

The Consensus Object. A *consensus* object is a one-shot concurrent object defined by a sequential specification, that provides the processes with a single operation denoted $\mathsf{propose}(v)$ where v is an input parameter (called "proposed value"). "One-shot" means that, given a consensus object, a process invokes at most once the operation $\mathsf{propose}()$. If it terminates, the operation returns a result (called "decided" value). This object can be defined by the three following properties.

- Validity. If a process decides a value, this value has been proposed by a process.
- Agreement. No two processes decide different values.
- Termination. An invocation of $\mathsf{propose}()$ by a process that does not crash terminates.

Consensus-Based Universal Construction Several universal constructions based on atomic registers and consensus objects have been proposed, e.g., [27] (see also chapter 14 of [51]). In that sense, and as depicted in Figure 2, consensus is a universal object to design wait-free universal constructions, i.e., wait-free implementations of any concurrent object defined by a sequential specification. This is depicted in Fig. 2, which completes Fig. 1.

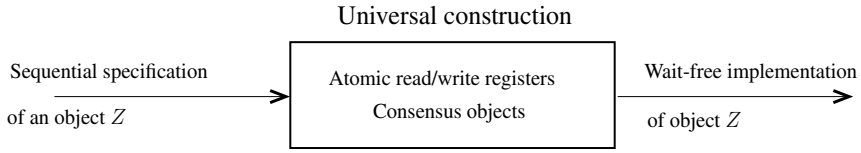


Fig. 2. Universal construction from atomic registers and consensus objects

In a universal construction, consensus objects are used by the processes to build a single total order on the operation invocations applied to the constructed object Z. This is the method used to ensure that the internal representation of Z remains always consistent, and is consequently seen the same way by all processes.

A Fundamental Result. One of the most important of the theoretical results of distributed computing is the celebrated FLP result (named after its authors Fischer, Lynch, and Paterson) [17]), which states that no binary consensus object (a process can only propose $v \in \{0, 1\}$) can be built in an asynchronous message-passing system whose environment states that (even only) one process may crash.

To prove this impossibility result, the authors have introduced the notion of *valence* associated with a global state (also called configuration). Considering binary consensus, a global state is 0-valent (1-valent) if only 0 (1) can be decided from this global state; 0-valent and 1-valent states are univalent states. Otherwise, "the dice are not yet cast", and any of 0 or 1 can be still decided. This is due to the uncontrolled and unpredictable behavior of the environment (i.e., asynchrony and failure pattern of the considered execution). A *decision step* of a construction is one that carries the construction from a bivalent state to a univalent state. The impossibility proof shows that (a) among all possible initial states, there is a bivalent state, and (2) among all possible executions in all possible environments, there is at least one execution that makes the construction always progress from a bivalent state to another bivalent state. It is easy to see that, the impossibility to implement a consensus object is related to the impossibility to break non-determinism (i.e., the impossibility to ensure that, in any execution, there is eventually a transition from a bivalent state to a univalent state).

This message-passing result has then been extended to asynchronous systems in which processes communicate only by reading and writing atomic registers instead of sending and receiving messages [27,42].

Sequential vs Distributed Computing: A Computability Point of View. It follows from the previous impossibility results that a network of Turing machines, that progress asynchronously and where at most one may crash (which are two reasonable assumptions) connected by a message-passing facility, or a read/write shared memory, is computationally less powerful than a single reliable Turing machine. As announced in the first section, this shows that the nature of distributed computability issues is different from the nature of Turing's computability issues, namely, it is not related to the computational power of the individual participants.

The Notion of a Consensus Number of an Object. The notion of *consensus number* of a concurrent object has been introduced by M. Herlihy [27]. The consensus number

of an object X is the largest integer n for which consensus can be be wait-free implemented in a read/write system of n processes enriched with any number of objects X. If there is no largest n, the consensus number is said to be infinite.

It is shown in [27] that the consensus numbers define an infinite hierarchy (hence the name Herlihy's hierarchy, or consensus hierarchy) where we have at levels 1, 2 and $+\infty$:

- Consensus number 1: read/write atomic registers, ...
- Consensus number 2: test&set, swap, fetch&add, stack, queue, ...
- Consensus number $+\infty$: compare&swap, LL/SC, mem-to-mem swap, ...

Universal Objects in the Read/Write Wait-Free System Model. An object is *universal* in the asynchronous read/write wait-free n-process system model, if it allows for the design of a wait-free universal construction in this system model (i.e., an algorithm implementing any concurrent object defined by a sequential specification). It is shown in [27] that any object with consensus number n is universal in a system of n (or less) processes.

From Asynchronous Read/Write Systems to Asynchronous Message-Passing Systems. The previous presentation was focused on asynchronous crash-prone systems in which the processes communicate by reading and writing a shared memory. Hence, the following question: are the impossibility results the same when the processes communicate by sending and receiving messages through a fully connected communication network? This question translates immediately into the following one: While it is easy to simulate the asynchronous wait-free message-passing system model on top of the asynchronous wait-free read/write system model, is the simulation in the other direction possible?

Let us remind that t is a model parameter denoting the maximal number of processes that may crash in an execution. It is shown in [3] that it is not possible to simulate the asynchronous wait-free read/write model on top of the wait-free message-passing model when $t \geq n/2$. This is called the ABD impossibility (named after its authors Attiya, Bar-Noy, Dolev).

The intuition that explains the ABD impossibility can be captured by what is called an *indistinguishability* argument, which relies on the fact that, when $t \geq n/2$, half or more processes may crash in a run. More precisely, as any number of processes may crash, it is possible to construct an execution in which the system "partitions" in two set of processes such that, while there is no crash, the messages between the two partitions take – in both directions– an arbitrarily long time, making the processes in each partition believe that the processes in the other partition have crashed. The system can then progress as two disconnected subsystems.

What is called in some papers [9,20] the "CAP theorem" (where CAP is a shortcut for "Consistency, Availability, Partition-tolerance") can be seen as an impossibility variant combining the ABD and the FLP impossibilities. This theorem, proved in [20], states that, when designing distributed services, it is impossible to design an algorithm that simultaneously ensures the three previous properties.

Playing with Progress Conditions and Consensus Objects. The progress condition called *obstruction-freedom* [29] is weaker than wait-freedom. It states that a process

that invokes an operation on an object is guaranteed to terminate its invocation only when it executes alone for a "long enough period".

The following asymmetric progress condition has been introduced in [35]. An object satisfies (y, x)-*liveness* if it can be accessed by a subset of $y \leq n$ processes only, and wait-freedom is guaranteed for $x \leq y$ processes while obstruction-freedom is guaranteed for the remaining $y - x$ processes. Notice that, (n, n)-liveness is wait-freedom while $(n, 0)$-liveness is obstruction-freedom. Among other results, it is shown in [35] that it is impossible to build a $(n, 1)$-live consensus object from read/write atomic registers and $(n - 1, n - 1)$-live consensus objects. Formulated differently, this states that wait-free consensus objects for $(n-1)$ processes are not computationally strong enough to implement a consensus object for n processes that ensures wait-freedom for only one process (be this process statically or dynamically defined) and the very weak obstruction freedom progress condition for the other processes. Moreover, the paper also shows that (n, x)-live consensus object with $x < n$ has consensus number $x+1$, which thereby establishes a hierarchy for (n, x)-liveness.

3 How to Circumvent Consensus Impossibility

As seen previously, one way to overcome consensus impossibility in a "pure" read/write asynchronous system composed of n processes consists in enriching the system with an object whose consensus number is equal to, or greater than, n. As simulating read/write registers on top of an asynchronous system requires $t < n/2$, this approach does not work for message-passing systems where $t \geq n/2$. Let us also notice that there is no notion of objects with a consensus number in message-passing.

As we are about to see, another approach (which works for both asynchronous read/write systems and asynchronous message-passing systems) consists in enriching the system with an oracle that provides processes with –possibly unreliable– information on failures. This is the failure detector approach.

A third approach consists in restricting the set of possible input vectors (an input vector is a vector –unknown to the processes– whose i-th entry contains the value proposed by process p_i). Hence, instead of enriching the system, this approach considers only a predefined subset of input vectors.

Another type of oracle that can be added to asynchronous systems to circumvent impossibility results, consists in allowing processes to use random numbers.

The Notion of an Unreliable Failure Detector. Failures detectors have been introduced in [12]. From an operational point of view, a failure detector can be seen as a set of n modules, each attached to a process. Failure detectors are divided into classes, according to the particular type of information they give on failures. Different problems (impossible to solve in asynchronous crash-prone systems) may require different classes of failure detectors. Two dimensions can be associated with failure detectors.

– The software engineering dimension of failure detectors. As any type of computer science object (e.g., stack or lock), a failure detector class is defined by a set of abstract properties, i.e., independently of the way these properties are implemented.

Hence, the design and the proof of an algorithm that uses a failure detector are based only on its properties. The implementation of a failure detector is an independent activity. The fact a failure detector can be implemented depends on additional behavioral properties that th environment has to satisfy.

– The ranking dimension of failure detectors. The failure detector approach makes possible the investigation and the statement of the weakest information on failures that allows a given problem to be solved [36]. This permits to rank the difficulty of distributed computing problems, according to the weakest failure detector they need to be solved. If $C1$ and $C2$ are the weakest failure detector classes to solve the problems $Pb1$ and $Pb2$, respectively, and if $C1$ is "strictly stronger" than $C2$ (i.e., gives more information on failures than $C2$), then we say that $Pb1$ is "strictly stronger" than $Pb2$. (The "strictly stronger" notion on failure detectors is a partial order, and consequently some problems cannot be compared with the help of the weakest failure detectors that allow them to be solved).

The Weakest Failure Detector to Solve Consensus in the Wait-Free Read/Write Model. The weakest failure detector class to solve consensus in the wait-free read/write model is denoted Ω. It has been introduced and proved to be minimal in [13]. Ω provides each process p_i with a read-only local variable denoted $leader_i$ that always contains a process identity, and satisfies the following property: there is an unknown but finite time τ after which the variable $leader_i$ of all the non-faulty processes contain the same identity, which is the identity of a non-faulty process.

It is important to notice that (a) there is no way for a process to know if time τ has occurred, and (b) before time τ occurs, there is an anarchy period during which the leader variables can have arbitrary values (e.g., some processes are their own leaders, while the leaders of the others are crashed processes).

Several algorithms implementing Ω, each assuming specific behavioral properties of the underlying system are described in Chapter 6 of [49]. So far, the best algorithm implementing Ω is the one described in [16]. ("Best" means here "with the weakest behavioral assumptions known so far".)

The Notion of an Indulgent Distributed Algorithm. This notion has been introduced in [21], and then investigated for consensus algorithms in [24], and formally characterized in [23]. It is on distributed algorithms that rest on failure detectors.

More precisely, a distributed algorithm is *indulgent* with respect to the failure detector FD it uses to solve a problem Pb if it always guarantees the safety property defining Pb (i.e., whatever the correct/incorrect behavior of FD), and satisfies the liveness property associated with Pb at least when FD behaves correctly. Hence, when the implementation of FD does not satisfies its specification, the algorithm may not terminate, but if it terminates its results are correct.

It is shown in [21] that all the failure detectors defined by an eventual property ("there is a finite time after which ...") are such that the algorithms that use them are indulgent. Ω is such a failure detector. In addition to its theoretical interest, indulgence is very important in practice. This follows from the observation that environments are usually made up of long "stable" periods followed by shorter "unstable" periods. These periods are such that the implementation of an "eventual" failure detector always satisfies its

specification during stable periods, while it does not during unstable periods. Hence, the liveness property of an indulgent algorithm is ensured during the "long enough" stable periods.

The Weakest Failure Detector to Simulate Read/Write on Top of Message-Passing. We have seen that it is possible to simulate an asynchronous crash-prone read/write system on top of an asynchronous crash-prone message-passing system only if $t < n/2$ (let us remind that t denotes the maximal number of processes that may crash in the considered model). As we have seen, intuitively this means that the system cannot partition.

The weakest failure detector to simulate read/write on top of message-passing whatever the value of t, has been introduced in [14]. Denoted Σ, and called *quorum* failure detector, it provides each process p_i with a read-only local variable $quorum_i$, which is a set containing process identities, and is such that (a) the values of any two quorums, each taken at any time, always intersect, and (b) the quorum of any non-faulty process eventually contains only non-faulty processes. The intersection property (a) is a perpetual property, while property (b) is an eventual property.

A simple proof of the minimality of Σ to implement a register on top of an asynchronous message-passing system prone to any number of process crashes can be found in [8]. A generalization of Σ to hybrid communication systems can be found in [34]. "Hybrid" means here that (a) all processes can communicate by sending and receiving messages, and (b) the processes are statically partitioned into clusters and inside each cluster the processes can communicate through read/write registers.

The Weakest Failure Detector to Solve Consensus in the Crash-Prone Message-Passing Model. The weakest failure detector class to solve consensus in the wait free (i.e., when $t = n - 1$) asynchronous message-passing model is the pair (Σ, Ω) [14].

If the system model is such that $t < n/2$, the weakest failure detector class to solve consensus in a message-passing model is Ω [13]. As an exercise, the reader can design a distributed algorithm that builds an atomic register in an asynchronous crash-prone message-passing system model where $t < n/2$ (solutions in [8,14,49]).

Other Weakest Failure Detectors. It is shown in [36] that every distributed computing problem, which can be solved with a failure detector, has a weakest failure detector.

As a simple example, it shown in [26] that the weakest failure detector to solve the *interactive consistency* problem [47] is the *perfect* failure detector defined in [12]. As another example, [15] exhibits the minimum information about failures for solving non-local tasks. (Roughly speaking, "tasks" in distributed computing corresponds to "functions" in sequential computing. "Local" means here that each process can compute its output from its input only.) The weakest failure detector to boost the obstruction-freedom progress condition to wait-freedom is described in [22].

Restricting the Set of Input Vectors for the Consensus Problem. A totally different approach to solve consensus in a read/write system in which up to $t < n$ processes may commit crashes has been proposed in [44]. This approach, called *condition-based*

approach, is related to error-detecting codes [19]. Intuitively an input vector "encodes" a decided value, and the aim of a distributed condition-based consensus algorithm is to "decode" it.

From a more operational point of view, it consists in favoring one of proposed values, while ensuring that this value can be selected by all the processes that decide. To that end, the "favored" value has to appear enough times in the input vector. As a simple example, a condition (set of allowed input vectors) can favor the greatest value present in a vector. To this end, the greatest value in each input vector has to appear at least $t + 1$ times.

Interestingly, the condition-based approach allows to establish a meeting point between computability in crash-prone asynchronous read/write systems and complexity in crash-prone synchronous message-passing systems [45]. Namely, considering systems in which up to t processes may crash, the weakest condition that allows consensus to be solved in an asynchronous read/write system is also the weakest condition that allows consensus to be solved optimally (with respect to the number of rounds) in a synchronous message-passing system.

Breaking Non-determinism with Random Numbers. Let us finally notice that random numbers have been used to solve binary consensus [7]. (There are then algorithms to go from binary consensus to multivalued consensus, both in asynchronous read/write systems [51] and asynchronous message-passing systems [49].) Randomization is used to break the non-determinism which makes the problem impossible to solve without additional computational power.

The termination property of consensus has to be slightly modified to take into account randomization. The corresponding algorithms are round-based, and the termination property becomes: the probability that a non-faulty process has decided by round r tends to 1 when r tends to $+\infty$.

4 Examples of Objects That Can Be Wait-Free Implemented in the Read/Write Wait-Free Model

This section presents two non-trivial objects which can be implemented in the asynchronous read/write wait-free system model (i.e., in which any number of processes may crash). It follows from the previous section that the synchronization and cooperation needed to wait-free implement these objects is relatively weak as it does not require underlying consensus objects. The reader interested in a more global view on objects which can be implemented in read/write systems prone to process crashes can consult [32,51,57].

The Snapshot Object. Snapshot objects have been introduced in [1]. A snapshot object is an array of registers $A[1..n]$, where $A[i]$ can be written only by p_i. It provides the processes with two operations. The first is a write that allows a process p_i to write (only) in $A[i]$. The second operation, denoted snapshot(), can be invoked by any process. It returns the value of the whole array. Moreover, both operations are *atomic*, which means that they appear as if they were executed instantaneously at some point of the time line, each between its start event and its end event [33,40].

A snapshot object can be wait-free implemented in the asynchronous read/write system model where any number of processes may crash (see for example [1,6,43,51]). Hence, it adds no computational power. Its interest lies in the abstraction level (programming comfort) it provides to programmers.

From a complexity point of view, the cost of an implementation of a snapshot object is measured by the number of accesses to basic read/write atomic registers [5]. While the cost of a write into its entry of the array A by a process is 1, the best algorithm designed so far to implement the operation snapshot() is $O(n \log_2 n)$. It is still an open problem to know which is the lower bound associated with the implementation of such an object.

The Renaming Object. This object is a one-shot object which has been introduced in [4], in the context of message-passing systems in which a majority of processes are assumed not to crash. It has then received a lot of attention in the context of read/write wait-free systems (i.e., in systems where any number of processes may crash) [6,11,51].

It is assumed that each process p_i has a name id_i taken from a large name space, whose size is N; the subscript i is then called the index of p_i. Initially a process knows only n and its initial identity id_i. The aim of a renaming object is to allow the processes to obtain new names in a smaller new name space, whose size M is much smaller than N. More precisely, an M-renaming object has a single operation denoted new_name(id) where the input parameter is the identity of the invoking process. An invocation of new_name() returns a new name to the invoking process. More precisely, an M-renaming object is defined by the following properties.

- Validity. A new name is an integer in the set $[1..M]$.
- Agreement. No two processes obtain the same new name.
- Index independence. $\forall\ i, j$, if a process whose index is i obtains the new name v, that process could have obtained the very same new name v if its index had been j.
- Termination. If a process invokes new_name() and does not crash, it eventually obtains a new name.

The index independence property states that, for any process, the new name obtained by that process is independent of its index. This means that, from an operational point of view, the indexes define only an underlying communication infrastructure, i.e., an addressing mechanism that can be used only to access entries of shared arrays. Indexes cannot be used to compute new names. This property prevents a process p_i from choosing i as its new name without any communication.

If only a (non-predetermined) arbitrary subset of processes invoke new_name(), the renaming object is *size-adaptive*. In that case, we have $M = f(p)$ where p is the number of processes which invokes the object operation. On the contrary, if all the processes are assumed to invoke new_name(), the renaming object is not size-adaptive. In that case, $M = f(n)$.

The lower bound on the size of the new name space is $M = 2p - 1$ for size-adaptive renaming. For renaming objects which are not size-adaptive, we have the following. The bound is $M = 2n - 1$ [31], except for an infinity number of values of n for which it is only known that $M \leq 2n - 2$ [10] (this infinite set of values of n includes the values that are not the power of a prime number).

5 On the Complexity Side: A Glance at Synchronous Systems

5.1 The Case of Crash-Prone Synchronous Systems

Synchronous distributed systems do not suffer the same kind of impossibility results as the ones encountered in asynchronous systems. (As a simple example, these systems are computationally strong enough to build a *perfect* failure detector [12].)

The nature of the impossibility results encountered in these systems is similar to the nature of the impossibility results encountered in sequential computing. To illustrate it, let us consider the consensus problem and three types of process failures: crash, send or receive omission, and Byzantine failure.

A process commits a send (receive) omission failure if it "forgets" to send (receive) messages. A process commits a general omission failure if it forgets to send or receive messages. A process commits a Byzantine failure if its behavior does not respect the algorithm it is supposed to execute. Moreover, for the consensus problem to be meaningful in presence of Byzantine failures, its validity and agreement properties have to be restricted as follows. Validity: If all the processes which are not faulty propose the same value, no other value can be decided. Agreement: two non-faulty processes cannot decide different values.

The following upper bounds on the model parameter t are attached to the consensus problem in presence of process failures.

Process failure model	Upper bound on t
crash failure	$t < n$
send omission failure	$t < n$
general omission failure	$t < n/2$
Byzantine failure	$t < n/3$

In all cases, the lower bound on the number of rounds that the processes have to execute is $t + 1$.

5.2 The Case of Crash-Free Synchronous Systems with a Message Adversary

The Notion of a Message Adversary. This notion has been introduced in [55,56] under the name *mobile fault*. A message adversary is a daemon which, at every round, is allowed to suppress messages. Of course, at any round, no process knows in advance which are the links on which messages are suppressed during this round.

If the adversary cannot suppress messages, we have a reliable synchronous system. If, at every round, it suppresses all messages, only local tasks can be computed (and the system is no longer a distributed system). Hence, it is important to characterize an adversary by a property defining its "worst behavior" during each round.

It has been shown in [2] that when the message adversary is constrained by a property denoted TOUR (four tournament), the corresponding synchronous message-passing system model and the asynchronous crash-prone read/write system model have the same computational power for distributed tasks. The adversary TOUR is such that, in each round, and for each pair of processes (p_i, p_j), the adversary is allowed to suppress the

message sent by p_i to p_j or the message sent by p_j to p_i, but not both. More general results connecting synchrony weakened by message adversaries vs asynchrony restricted by failure detectors have been recently established in [53]. An adversary related to message broadcast was introduced in [37].

More generally, the aim of message adversaries is to consider message losses as a normal behavior and not as a faulty behavior from the underlying communication environment. This is strongly related to *dynamic* systems where processes are allowed to move from a location to another location, thereby naturally modifying their neighboring connections (e.g., [54]).

5.3 A Glance at Crash-Free Synchronous Systems with an Arbitrary Network

Crash-Free Synchronous Systems with an Unknown Network. This section discusses briefly synchronous systems in which the communication graph is connected but is not a clique. Moreover, initially a process knows only its neighbors. It is easy to see that if the number of rounds that the processes are allowed to execute is equal to the network diameter, each process can learn all the inputs and then compute its result.

The Notion of Locality in Synchronous Systems. This notion has been introduced in [41], and the associated $\mathcal{LOCAL}$ model has been investigated in [48]. The locality notion has first been used to study complexity issues of distributed algorithms on graphs, and has then addressed more general decision problems.

In a local algorithm, a process is restricted to collect data from other processes which are at distance at most x (i.e., in at most x rounds), where x is smaller than the network diameter.

The main question is then: given a distributed graph problem, is it possible to solve it with a local algorithm? This question is fundamental from a scalability point of view. As a simple example, the optimal vertex coloring problem is not local, while verifying if an arbitrary vertex coloring is such that no two neighbor vertices have the same color can be solved in one round. The interested reader will find in [18,38,41,46,48] complexity results related to locality (associated with problems such as vertex coloring, minimum independent set, minimum vertex cover, etc.).

6 Conclusion

The aim of this paper was to be a short introduction to decidability issues in distributed computing. For more information the reader can consult the following books, some parts of which address distributed computability issues.

- [6,43] are on distributed algorithms in both read/write and send/receive systems where processes can commit failures.
- [51] is on algorithms in *asynchronous* shared memory systems where processes can commit *crash failures*. It focuses on the construction of reliable concurrent objects in the presence of process crashes.
- [57] is on synchronization in shared memory systems and associated complexity bounds.
- [32] is on the design of concurrent objects in shared memory systems.

- [49] is on *asynchronous message-passing* systems where processes are prone to *crash failures*. It presents communication and agreement abstractions for fault-tolerant asynchronous distributed systems. Failure detectors are used to circumvent impossibility results encountered in pure asynchronous systems.
- [50] is on *synchronous* message-passing systems, where the processes are prone to *crash failures, omission failures, or Byzantine failures*. It focuses on the following distributed agreement problems: consensus, interactive consistency, and non-blocking atomic commit.
- [52] is on elementary distributed computing for *failure-free asynchronous message-passing* systems.
- [48] is mainly on the $\mathcal{LOCAL}$ (synchronous) model and associated complexity issues.
- [28] is an introduction to distributed computing based on combinatorial topology.

Acknowledgments. The author wants to thank his colleagues Carole Delporte, Hugues Fauconnier, Eli Gafni, Damien Imbs, Achour Mostéfaoui, Sergio Rajsbaum, Julien Stainer, and Gadi Taubenfeld for stimulating discussions on the nature, the power, and the limits of distributed computing. He wants also to thank François Taïani for constructive comments.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. Journal of the ACM 40(4), 873–890 (1993)
2. Afek, Y., Gafni, E.: Asynchrony from synchrony. In: Frey, D., Raynal, M., Sarkar, S., Shyamasundar, R.K., Sinha, P. (eds.) ICDCN 2013. LNCS, vol. 7730, pp. 225–239. Springer, Heidelberg (2013)
3. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message passing systems. Journal of the ACM 42(1), 121–132 (1995)
4. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. Journal of the ACM 37(3), 524–548 (1990)
5. Attiya, H., Ellen, F., Fatourou, P.: The complexity of updating snapshot objects. Journal of Parallel and Distributed Computing 71(12), 1570–1577 (2010)
6. Attiya, H., Welch, J.L.: Distributed computing: fundamentals, simulations and advanced topics, 2nd edn., 414 pages. Wiley-Interscience (2004) ISBN 0-471-45324-2
7. Ben-Or, M.: Another advantage of free choice: completely asynchronous agreement protocol. In: Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC 1983), pp. 27–30. ACM Press (1983)
8. Bonnet, F., Raynal, M.: A simple proof of the necessity of the failure detector Σ to implement an atomic register in asynchronous message-passing systems. Information Processing Letters 110(4), 153–157 (2010)
9. Brewer, E.A.: Pushing the CAP: strategies for consistency and availability. IEEE Computer 45(2), 23–29 (2012)
10. Castañeda, A., Rajsbaum, S.: New combinatorial topology bounds for renaming: The upper bound. Journal of the ACM 59(1), Article 3, 49 pages (2012)
11. Castañeda, A., Rajsbaum, S., Raynal, M.: The renaming problem in shared memory systems: an introduction. Elsevier Computer Science Review 5, 229–251 (2011)

12. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
13. Chandra, T., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM 43(4), 685–722 (1996)
14. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Tight failure detection bounds on atomic object implementations. Journal of the ACM 57(4), Article 22 (2010)
15. Delporte-Gallet, C., Fauconnier, H., Toueg, S.: The minimum information about failures for solving non-local tasks in message-passing systems. Distributed Computing 24, 255–269 (2011)
16. Fernández, A., Jiménez, E., Raynal, M., Trédan, G.: A timing assumption and two t-resilient protocols for implementing an eventual leader service in asynchronous shared-memory systems. Algorithmica 56(4), 550–576 (2010)
17. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
18. Fraigniaud, P., Korman, A., Peleg, D.: Towards a complexity theory for local distributed computing. Journal of the ACM 60(5), Article 35, 16 pages (2013)
19. Friedman, R., Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Asynchronous agreement and its relation with error-correcting codes. IEEE Transactions on Computers 56(7), 865–875 (2007)
20. Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33(2), 51–59 (2002)
21. Guerraoui, R.: Indulgent algorithms. In: Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC 2000), pp. 289–298. ACM Press (2000)
22. Guerraoui, G., Kapalka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. Distributed Computing 20(6), 415–433 (2008)
23. Guerraoui, R., Lynch, N.A.: A general characterization of indulgence. ACM Transactions on Autonomous and Adaptive Systems 3(4), Article 20 (2008)
24. Guerraoui, R., Raynal, M.: The information structure of indulgent consensus. IEEE Transactions on Computers 53(4), 453–466 (2004)
25. Harel, D., Feldman, Y.: Algorithmics, the spirit of computing, 572 pages. Springer (2012)
26. Hélary, J.-M., Hurfin, M., Mostéfaoui, A., Raynal, M., Tronel, F.: Computing global functions in asynchronous distributed systems with perfect failure detectors. IEEE Transactions on Parallel and Distributed Systems 11(9), 897–909 (2000)
27. Herlihy, M.P.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
28. Herlihy, M.P., Kozlov, D., Rajsbaum, S.: Distributed computing through combinatorial topology, 336 pages. Morgan Kaufmann/Elsevier (2014) ISBN 9780124045781
29. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: Proc. 23rd Int'l IEEE Conference on Distributed Computing Systems (ICDCS 2003), pp. 522–529. IEEE Press (2003)
30. Herlihy, M.P., Rajsbaum, S., Raynal, M.: Power and limits of distributed computing shared memory models. Theoretical Computer Science 509, 3–24 (2013)
31. Herlihy, M.P., Shavit, N.: The topological structure of asynchronous computability. Journal of the ACM 46(6), 858–923 (1999)
32. Herlihy, M.P., Shavit, N.: The art of multiprocessor programming, 508 pages. Morgan Kaufmann (2008) ISBN 978-0-12-370591-4
33. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
34. Imbs, D., Raynal, M.: The weakest failure detector to implement a register in asynchronous systems with hybrid communication. Theoretical Computer Science 512, 130–142 (2013)

35. Imbs, D., Raynal, M., Taubenfeld, G.: On asymmetric progress conditions. In: Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC 2010), pp. 55–64. ACM Press (2010)
36. Jayanti, P., Toueg, S.: Every problem has a weakest failure detector. In: Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC 2008), pp. 75–84. ACM Press (2008)
37. Kuhn, F., Lynch, N.A., Oshman, R.: Distributed computation in dynamic networks. In: Proc. 42nd ACM Symposium on Theory of Computing (STOC 2010), pp. 513–522. ACM Press (2010)
38. Kuhn, F., Moscibroda, T., Wattenhofer, R.: What cannot be computed locally! In: Proc. 23rd ACM Symposium on Principles of Distributed Computing (PODC 2004), pp. 300–309. ACM Press (2004)
39. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
40. Lamport, L.: On inter-process communications, Part I: Basic formalism. Distributed Computing 1(2), 77–85 (1986)
41. Linial, N.: Locality in distributed graph algorithms. SIAM Journal on Computing 21(1), 193–201 (1992)
42. Loui, M., Abu-Amara, H.: Memory requirements for agreement among unreliable asynchronous processes. Advances in Computing Research 4, 163–183 (1987)
43. Lynch, N.A.: Distributed algorithms, 872 pages. Morgan Kaufmann (1996)
44. Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Conditions on input vectors for consensus solvability in asynchronous distributed systems. Journal of the ACM 50(6), 922–954 (2003)
45. Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Synchronous condition-based consensus. Distributed Computing 18(5), 325–343 (2006)
46. Naor, M., Stockmeyer, L.: What can be computed locally? In: Proc. 25th ACM Symposium on Theory of Computing (STOC 1993), pp. 184–193. ACM Press (1993)
47. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM 27, 228–234 (1980)
48. Peleg, D.: Distributed computing, a locally sensitive approach. SIAM Monographs on Discrete Mathematics and Applications, 343 pages (2000) ISBN 0-89871-464-8
49. Raynal, M.: Communication and agreement abstractions for fault-tolerant asynchronous distributed systems, 251 pages. Morgan & Claypool Pub. (2010) ISBN 978-1-60845-293-4
50. Raynal, M.: Fault-tolerant agreement in synchronous message-passing systems, 165 pages. Morgan & Claypool Publishers (2010) ISBN 978-1-60845-525-6
51. Raynal, M.: Concurrent programming: algorithms, principles, and foundations, 530 pages. Springer (2013) ISBN 978-3-642-32026-2
52. Raynal, M.: Distributed algorithms for message-passing systems, 515 pages. Springer, ISBN: 978-3-642-38122-5
53. Raynal, M., Stainer, J.: Round-based synchrony weakened by message adversaries *vs* asynchrony enriched with failure detectors. In: Proc. 33rd ACM Symposium on Principles of Distributed Computing (PODC 2013), pp. 166–175. ACM Press (2013)
54. Raynal, M., Stainer, J., Cao, J., Wu, W.: A simple broadcast algorithm for recurrent dynamic systems. In: Proc. 28th IEEE Int'l Conference on Advanced Information Networking and Applications (AINA 2014), 8 pages. IEEE Press (2014)
55. Santoro, N., Widmayer, P.: Time is not a healer. In: Cori, R., Monien, B. (eds.) STACS 1989. LNCS, vol. 349, pp. 304–316. Springer, Heidelberg (1989)
56. Santoro, N., Widmayer, P.: Agreement in synchronous networks with ubiquitous faults. Theoretical Computer Science 384(2-3), 232–249 (2007)
57. Taubenfeld, G.: Synchronization algorithms and concurrent programming, 423 pages. Pearson Education/Prentice Hall (2006) ISBN 0-131-97259-6

Toward a System Design Science

Joseph Sifakis

RiSD Laboratory, EPFL, Lausanne, Switzerland
joseph.sifakis@epfl.ch

1 About Design

Design is a universal concept. It links the immaterial world of concepts to the physical world. It is an essential area of human experience, expertise, and knowledge, which deals with our ability to mold our environment to satisfy material and spiritual needs.

Design has two different connotations. One is simply a plan or a pattern for assembling objects constituting a given artifact. The other is the creative process for devising plans or patterns and carrying them out to produce an artifact. For this paper we focus on the second interpretation. We are ultimately interested in putting design on a more scientific basis. Toward this end, we focus here on articulating a new structure for the design process, which we believe will support this goal.

We consider that design is the process that leads to an artifact meeting given requirements. The requirements include functional requirements describing the functionality provided by the artifact and extra-functional requirements dealing with the way in which resources are used for implementation and throughout the artifact's lifecycle.

Designers deal with two often antagonistic demands: 1) productivity, meaning cost-effectiveness; 2) correctness, meaning compliance to requirements. In pursuit of these demands, the design process moves through three stages. The first, requirements specification, describes the artifact's expected behavior and any applicable techno-economic constraints. The second, proceduralization, generates an executable description for realizing the anticipated behavior by executing sequences of elementary functions. The third, materialization, produces an artifact by following the procedure using the available physical resources. Design is an essential component of any engineering activity. By its nature, it is a "problem-solving process".

As a rule, requirements are declarative. They are usually expressed in natural languages. For some application areas, they can be formalized by using logics. When requirements are expressed by logical specifications, they can be treated as axioms; proofs that the artifact meets them can start from there. Proceduralization can be considered as a synthesis problem: procedures are executable models meeting the specifications. Unfortunately, model synthesis from logical requirements often runs into serious technical limitations such as non-computability or intrinsically high complexity.

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 225–234, 2014.

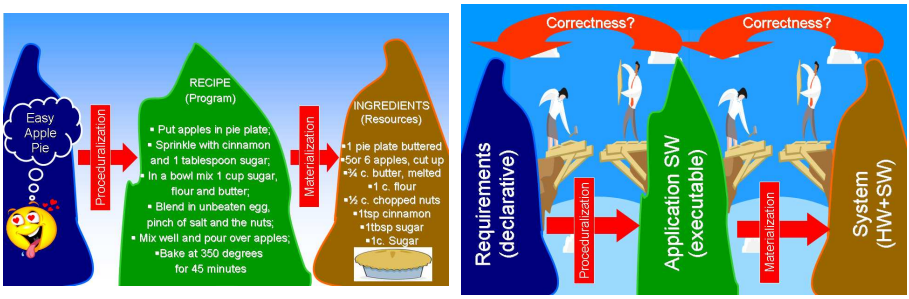


Fig. 1. Design is a universal concept applicable from cooking to computing systems

What happens when requirements are uncertain? To deal with uncertainty, engineers developed empirical approaches including requirements capture, incremental prototyping, incremental testing, etc. Some of these are done with solid science protocols focused on experimental design, data analysis, and hypothesis testing. Others are much less formal and involve many rules of thumb based on engineering experience. A design science would clarify the role of such empirical methods, and it would also address the issue of designing against uncertain requirements.

Design formalization raises a multitude of deep theoretical problems related to the conceptualization of needs in a given area and their effective transformation into correct artifacts. So far, it has attracted little attention from theoreticians. One reason is the predilection of the academic world for simple and elegant theories. Another reason is that design is by nature multi-disciplinary. Its formalization requires consistent integration of heterogeneous models supporting different levels of abstraction including logics, algorithms, programs, physical system models, risk models, statements about user practices and statements about esthetics.

Despite the challenges, providing systematic and well-founded design techniques is of paramount importance for two reasons. The first is that we need to construct artifacts of guaranteed quality and performance based on scientific evidence. This is the case for airplanes, cars, critical resource management systems as well as for critical computing and communication infrastructure. The second reason is the need to master as much as possible through automation the complexity and development costs of increasingly sophisticated artifacts. This need can be illustrated by numerous manufacturing setbacks experienced by the aircraft industry e.g. the A380 delivery delay or recent safety concerns with Boeing's Dreamliner.

Ideas for "scientizing" design emerged in the beginning of the 60's [1, 2]. There exists today an abundant literature on design science e.g. [3] and on design science and computing, in particular [4–7]. In this paper, we present our view about design science and propose a vision determining its scope and perimeter.

2 Bringing Science to Design

Science is a disciplined and systematic method for building, organizing and using knowledge about the world. We consider that scientific investigation intimately combines two interdependent processes. The first process is descriptive and intended to develop theory connecting some observed reality through abstractions to the world of concepts and mathematics. The second process is prescriptive and consists in applying a theory in order to assess its explicability and predictability as well as to invent things that do not yet exist. Interaction and cross-fertilization between these two processes is key to the progress of scientific knowledge. Today, more than ever, the two processes are involved in an accelerating virtuous cycle for the advancement of scientific knowledge.

The starting point in the scientific investigation cycle need not be observation. The theory of relativity was motivated by a series of thought experiments rather than direct observation. The development of computing as a scientific discipline started from prior knowledge about computation based on mathematics and logic.

We consider that design science studies design as a process for developing artifacts meeting given requirements for trust, function, safety, reliability, esthetics, cost containment, etc. Our definition significantly differs from others in the literature because it emphasizes the modeling-prediction cycle to support the application of technical means to aid the design processes. Design science is also concerned with deriving from the applied knowledge of the natural sciences appropriate information in a form suitable for the designer's use.

Inherent technical difficulties and limitations aggravate these problems. Nonetheless, we believe that their analysis and formalization in a flow leading from requirements to their materialization, can bring interesting insights about the very nature of artifact creation. For many aspects of design it will be impossible to achieve full automation. The main benefit from a scientific approach to design is rigor. Design can be sketched out as an iterative process consisting of steps supported by a methodology. While guaranteeing the correctness of designs may be an unattainable goal, a guarantee of accountability is realistic; accountability means that at each design step it should be possible to know which requirements hold, which ones do not hold and why.

3 Principles and Problems

We proposed a characterization of design science as a formal process encompassing the three stages of requirements expression, proceduralization and materialization. Each stage corresponds to a main type of problem to be solved by designers. Next, we discuss four principles that should drive the definition of a design process.

3.1 Four Driving Principles

Separation of Concerns. Design consistently integrates a sequence of three stages: requirements specification, proceduralization and materialization.

- Requirements express, usually in some declarative language, why we build an artifact. They express intention and needs motivating the design.
- Proceduralization consists in discovering what functionality should be ensured by the designed artifact to meet functional requirements; then it provides a procedure for composing atomic components, each component providing some elementary function or service.
- Materialization defines how function components can be implemented by using physical components. Implementation choices are driven by extra-functional requirements which are mainly trade-offs between cost and performance.

This three-stage decomposition is essential from a methodological point of view. It allows complexity to be tamed as it clearly separates concerns (why, what, how) by separately addressing three difficult problems. Furthermore, the distinction between proceduralization and materialization allows artifacts to be built providing the same functionality under different techno-economic constraints. This makes possible the development of families of artifacts with identical functional features and different performance and cost characteristics.

Each stage of a design process may be further decomposed into steps. At each step, the designed artifact is described at a certain level of abstraction by using an adequate modeling language. Each step progressively reduces abstraction by replacing conceptual constructs and primitives by more concrete ones. The final model is a blueprint for building the physical implementation.

Separation of concerns should be supported by an adequate design methodology. A design methodology identifies designer activities that can be supported by state-of-the-art tools to automate tedious and error-prone tasks. It also precisely determines where human intervention and ingenuity are needed to resolve design choices through requirements analysis and confrontation with experimental results. Identifying adequate design parameters and channeling the designers creativity are essential in this enterprise. The interaction between designer and supporting tools may involve iterations to eliminate design errors and determine optimized solutions.

Semantic Coherency. Designers use a variety of languages for the description of the behavior of the designed artifact at different abstraction levels. These include declarative languages for expressing requirements, and procedural languages for modeling, simulation, and performance analysis. Designers use, above all, domain-specific languages for example, for buildings, mechanical systems, electric systems, control-based systems, hardware description languages, and web-based systems.

Frequently, these languages only have informal semantics, which make it difficult not only to verify that the requirements capture their intended meanings, but also to reconcile different models that are brought together in a design process. This may be a source of design errors. To achieve semantic coherency, and minimize those errors, all these languages must be rooted in a common semantic model. The choice of the semantic model depends on the type of designed artifact for example, geometric model, differential equations, or abstract machines.

Semantic coherency enforcement may be completely transparent for the designer. It can be handled by translation tools such as compilers, interpreters, and model transformers.

A common semantic model is essential for rigor of design. It characterizes correctness through a semantic equivalence relation between artifact descriptions in the different languages that appear in each design stage. An essential requirement for a common semantic model is that it directly encompasses primitives and constructs of the hosted languages to avoid combinatorial explosion of the translation [8].

Component-Based Construction. Building larger structures from smaller components enhances productivity and correctness, and is essential in any design process. Components hide behavioral details behind interfaces that highlight their interactions with their environment. They can be assembled into composite components by partially composing their interfaces. Their semantics are defined by stating the rules of the composite component in terms of the behaviors of its constituent components. Component composition can use a large variety of mechanisms expressing how the behaviors of the composed components are restricted through mutual interaction.

Designers need a unified composition paradigm for describing and analyzing coordination between components in terms of tangible, well-founded, and well-organized concepts.

Correctness-by-Construction. Correctness means that a designed artifact meets its requirements specifications. Many designers consider it an ideal to establish correctness by checking that a design, once completed, meets its specifications. This ideal is usually impossible because automatic verification entails intractable computations. The size of state space to be examined by a verification method explodes exponentially with the number of components in the artifact. The best we can do is limit automatic verification to small or medium size models and to specific properties. Some researchers have investigated compositional verification techniques, which aim to decompose a global requirement for a composite artifact into sets of requirements for its constituent components. So far, compositional verification approaches have failed to make any significant breakthrough [9].

An alternative approach is to establish correctness-by-construction incrementally, as you go along the design process, through the combined application of three principles: property enforcement, property composability, and property preservation.

Property enforcement: Property enforcement is very common in engineering. Engineers extensively use principles for building complex artifacts from components so as to meet given properties. These principles can be embodied in patterns for buildings design or for software design, in communication protocols, in distributed algorithms, in hardware or system architectures. They are solutions to particular problems. For example, a communication protocol ensures reliable message transmission despite packet losses. A token-ring algorithm ensures mutual exclusion in a distributed system. Client-server architectures ensure atomicity of transactions and fault-tolerance. All these component coordination

mechanisms can be reused provided they are adequately formalized. They allow correctness almost for free.

Notice that property enforcement enables designers to ensure that compositions of components meet a specific global requirement. In contrast to compositional verification, it does not require breaking up the global requirement into sub-requirements to be met by components. For example, we do not have general compositionality theory for deadlock-freedom preservation. Nonetheless, specific protocols or architectures may be used to build deadlock-free systems from deadlock-free components.

Property composability: A key issue in this approach is maintaining coherence while combining multiple existing solutions to specific problems. For example, a database programmer might apply several instances of a lock algorithm, but their multiple application may contain a deadlock. How does the designer know that the combination of correct solutions might be unsafe?

Another illustration of this problem comes from fault-tolerant computing. Fault-tolerant systems combine multiple methods for protection against invalid actions, including: 1) triple modular redundancy mechanisms ensuring continuous operation in case of single component failure; 2) hardware checks to validate that programs use data only in their defined regions of memory; 3) default to least privilege (least sharing) to enforce file protection; 4) checkpoints that permit backing up to, and restarting from, a prior valid system state in case of a failure. If we combine all these methods, how can we be sure there are no unwanted interactions that make the system prone to new faults?

Guaranteeing non-interaction of features is essential for correct-by-construction design. Violations of this principle invariably cause trouble. For example, features of telecommunication systems frequently interact, causing user confusion and misuse. Interference among web services and among features in aspect programming are additional examples.

Property preservation: When a requirement holds for an artifact description at some design step, it is essential that it remains valid at all subsequent steps. This allows establishing correctness incrementally. Each new modeling step must not invalidate correctness of previous steps. Artifact models are progressively built by first ensuring validity of each functional requirement and then models are refined to satisfy additional extra-functional requirements. Model refinement can be characterized as a preorder relation between models. It can be implemented through a set of model transformation rules. The demand for property preservation means that these rules preserve the semantic equivalence of models.

3.2 Three Basic Problems

The three stages of the design process correspond to three types of basic problems. Their solution is aggravated by many factors including undecidability, overwhelming algorithmic complexity, conceptual ambiguity, and physical uncertainty. The objective is not to tackle these problems in their full generality but rather to identify avenues for their partial solution in specific application contexts by supporting the designer's ingenuity with automation.

Formalizing Requirements. Many design processes begin with an expression in a rigorous language that declares the needs to be met by an artifact and the associated techno-economic constraints. Several difficulties obstruct full formalization of requirements. Requirements are by their nature declarative; thus logic is, in principle, an adequate framework for their expression. However, requirements are initially expressed in natural languages, which usually allow ambiguities. Ambiguities inhibit translation into a formal language, limiting the designer's ability to be systematic and rigorous. In addition, many requirements are meant to describe the behavior of the artifact in context of its environment including its potential users. Formalization of an artifact's environment is no easy task it must be done at the right abstraction level, accounting for all the relevant behavioral properties. Today we lack theoretical approaches for tackling this problem.

The concept of correctness conjoins two types of requirements: 1) trustworthiness requirements ensuring that nothing bad could happen; 2) optimization requirements for performance, cost-effectiveness, and tradeoffs between them. Trustworthiness characterizes qualitative correctness. It means that the artifact can be trusted, and that it will behave as expected. It accounts for non-vulnerability to hazards such as: 1) design errors; 2) physical failures and defects; 3) interaction with potential users including erroneous use and threats; and 4) interaction with the physical environment including disturbances and unpredictable events.

Optimization requirements deal with the optimization of functions subject to constraints involving resources used for implementing and using the artifact. They deal with: 1) requirements on performance metrics such as throughput and response time, which characterize how well the artifact does with respect to user-defined criteria; 2) cost-effectiveness, which characterizes how well resources are used with respect economic criteria; 3) tradeoffs between performance and cost-effectiveness.

Trustworthiness and optimization requirements can be difficult to reconcile. As a rule, improving trustworthiness causes wasted resources. Conversely, resource optimization may jeopardize trustworthiness. Designers try to balance trustworthiness and optimization.

There is a limit to how far we can push a formal logic approach to requirements. The biggest problem is that users themselves often cannot articulate their deep concerns about trust and performance. How can we formalize what the customer cannot say? For example, with computer security, how can we be exhaustive and precise about threats from unseen or unsuspected adversaries? Here there is a real possibility that empirical approaches can help. We can build prototypes of systems and ask users to try them out and tell us about good points and problems. By systematically iterating between prototypes and customer assessments, we can converge on a set of requirements that earn their trust. The big challenge is to develop sound scientific methods for this process and reconcile the experimental results with the mathematical models.

Proceduralization. Proceduralization is a synthesis problem: find a procedure that builds functionality meeting given requirements from a set of predefined atomic components of known functional characteristics. Unfortunately, most non-trivial instances of this problem are computationally intractable for example, program synthesis from logical specifications.

A pragmatic approach for tackling this problem is to strive to bridge the gap between declarative and procedural languages by working in two directions.

One direction is to raise the abstraction of languages to get them as close as possible to the declarative style. This would simplify reasoning and relegate procedure generation to tools. Many approaches for enhanced abstraction proposing logical, constraint-based, and functional description languages already exist. They are equipped with interpreters or compilers that allow automatic synthesis of procedural descriptions.

The other direction is to develop adequate domain-specific languages allowing ease of description as well as enhanced safety and productivity. Examples include Matlab/Simulink, HTML, Logo, SQL, BPEL and hardware description languages.

Materialization. Materialization consists in exploring cost-performance tradeoffs among all the possible physical implementations of the desired functionality. It involves extensive empirical evaluation and hypotheses testing to determine designs better fitting cost-performance requirements. The exploration can be performed on an adequate model obtained from the procedural description by assigning to its elements models of functionally equivalent physical components. In addition to their functionality, this transformation should take into account physical characteristics, such as execution times, latency, and power consumption. The obtained model should faithfully describe the dynamic behavior of the artifact, including both the provided functionality and its global physical properties. A key issue to consider when building such models is their predictability that is the degree to which their quantitative properties can be asserted. For example, the materials laid down in layers during a 3-D print may not satisfy the continuity assumptions of a mathematical model; experimental validation of stress-bearing properties of those materials is essential. Building predictable models raises deep theoretical problems [10] and requires a marriage of formal methods and scientific validation methods.

Design space exploration techniques are intended to determine an optimal assignment of physical components that fits the user-defined cost-performance requirements. Currently, they are mostly ad hoc and consist in evaluating the impact of design parameters on the requirements. The challenge for designs science is to make the formal and experimental sides of design mutually compatible and reinforcing.

Design space exploration allows estimating combinations of parameters that better fit the requirements. The main challenge is the complexity of exploring the design space. State-of-the-art techniques for overcoming complexity combine symbolic representation of the design space and theoretical results for accelerating the exploration process [11].

4 Toward a Design Science

Even though forty years have passed since the first seminal ideas about design science, very little progress has been made toward defining its technical goals and clarifying its scope and limits. It is time to develop technical work contributing to the advancement of our knowledge about design as a universal paradigm amenable to both scientific analysis and rigorous practice. In this essay we have argued that design is a rigorous process involving the successive solution of three types of problems. We proposed principles and associated scientific challenges for putting design into practice.

We believe that achieving this goal is not only a matter of writing down a theory of design, it will require hard work with semantic models, empirical methods for dealing with uncertainty in requirements, flexibility for dealing with changing environments, testing, automated materialization, and the maturing of correct-by-construction principles. This vision is both intellectually challenging and culturally enlightening. It is at least of equal importance as the quest for scientific discovery in natural sciences.

Endowing design with scientific foundations is a huge intellectual challenge that would meet an urgent demand for cost-effectively building complex, trustworthy artifacts. Failure in this endeavor, would seriously limit our capability to master the techno-structure and its further development intended to address urgent global challenges for optimal resource management and enhanced services. It would also mean that designing is a definitely a-scientific activity [12] driven by predominant subjective factors that make for ineffectual rational treatment.

Meeting this challenge would significantly enhance our capability to build trustworthy artifacts, and would confirm that design is definitely a scientific activity.

Acknowledgments. Peter Denning and Richard Snodgrass contributed to significantly improving the paper through constructive comments and criticism.

References

1. Simon, H.A.: The Sciences of the Artificial, 3rd edn. MIT Press, Cambridge (1996)
2. Alexander, C.: Notes on the synthesis of form. Harvard University Press, Cambridge (1964); Autres tirages: 1968, 1971
3. Cross, N.: Designerly ways of knowing: Design discipline versus design science. Design Issues 17(3), 49–55 (2001)
4. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Q. 28(1), 75–105 (2004)
5. Winter, R., Zhao, J.L., Aier, S. (eds.): DESRIST 2010. LNCS, vol. 6105. Springer, Heidelberg (2010)
6. Peffers, K., Rothenberger, M., Kuechler, B. (eds.): DESRIST 2012. LNCS, vol. 7286. Springer, Heidelberg (2012)
7. Henzinger, T.A., Sifakis, J.: The discipline of embedded systems design. Computer 40(10), 32–40 (2007)

8. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 508–522. Springer, Heidelberg (2008)
9. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. ACM Trans. Softw. Eng. Methodol. 17(2), 7:1–7:52 (2008)
10. Thiele, L., Wilhelm, R.: Design for timing predictability. Real-Time Syst. 28(2-3), 157–177 (2004)
11. Mohanty, S., Prasanna, V.K., Neema, S., Davis, J.: Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In: Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems, LCTES/SCOPES 2002, pp. 18–27. ACM, New York (2002)
12. Grant, D.: Design methodology and design methods. Design Methods and Theories 13(1) (1979)

OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems

Janos Sztipanovits[1], Ted Bapty[1], Sandeep Neema[1],
Larry Howard[1], and Ethan Jackson[2]

[1] Institute for Software Integrated Systems (ISIS), Vanderbilt University
1025 16th Ave S, Suite 102, Nashville, TN 37212, USA
[2] Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA
{sztipaj,sandeep,bapty,howardlp}@isis.vanderbilt.edu,
ejackson@microsoft.com

Abstract. Model- and component-based design have yielded dramatic increase in design productivity in several narrowly focused homogeneous domains, such as signal processing, control and aspects of electronic design. However, significant impact on the design and manufacturing of complex cyber-physical systems (CPS) such as vehicles has not yet been achieved. This paper describes challenges of and solution approaches to building a comprehensive design tool suite for complex CPS. The primary driver for the OpenMETA tool chain was to push the boundaries of the "correct-by-construction" principle to decrease significantly the costly design-build-test-redesign cycles in design flows. In the discussions we will focus on the impact of heterogeneity in modeling CPS. This challenge is compounded by the need for rapidly evolving the design flow by changing/updating the selection of modeling languages, analysis and verification tools and synthesis methods. Based on our experience with the development of OpenMETA and with the evaluation of its performance in a complex CPS design challenge we argue that the current vertically integrated, discipline-specific tool chains for CPS design need to be complemented with horizontal integration layers that support model integration, tool integration and design process integration. This paper will examine the OpenMETA technical approach to construct the new integration layers, provides and overview of the technical framework we established for their implementation and summarize our experience with their application.

Keywords: Model-Based Design, Component-Based Design, Cyber Physical Systems, Design Automation, Model-Integrated Computing, Domain-Specific Modeling Language, Model Integration Language.

1 Introduction

Model- and component-based design have been recognized as key technologies for radically changing productivity of CPS design [1]. Model-based design uses formal and sufficiently complete models of physical and computational processes, their environment and their interactions. These models are mathematically and physically accurate for verifying and testing the behavior of the designed system against established

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 235–248, 2014.

requirements. The main promise of model-based design is a significant decrease or elimination of costly design-build-test-redesign iterations. The ultimate goal of model-based design is "correct-by-construction", where properties of the synthesized models of the designed system predict the properties of the implemented/manufactured system with sufficient accuracy.

Component-based design constructs systems from reusable components. A component is the superposition of two models: a behavior model and an interaction model [2]. In a model-based design flow, models of components are used for constructing valid system models. The promise of component-based design is the potentially massive productivity increase due to the reuse of design knowledge captured by the component models.

While model- and component-based design methods and tools have demonstrated significant success in several engineering domains, such as VLSI design, electronics design and specific segments of software design [6], success has been elusive for CPS. Among the reasons are the following technical barriers.

1. *Heterogeneity of CPS models*. Heterogeneity in CPS design has several dimensions such as physical phenomena, levels of abstraction used in modeling physical and computational structures and processes, and engineering disciplines involved in CPS design.
2. *Heterogeneity of design tools*. Tool chains that are used in traditional CPS design flows are discipline oriented, vertically integrated and cover "islands" in the overall design space. Integration across the tool suites is hard and usually not supported.
3. *Life-cycle heterogeneity*. A unique aspect of CPS design is the significant impact of manufacturing on system performance. In fact, design of the physical part of the system needs to be integrated with the design of manufacturing processes that will make those. Manufacturability constraints and properties of the system "as manufactured" require tradeoffs with the design even in the early conceptual design phase.

Separation of concerns is a widely used strategy to deal with heterogeneity in the design process. Its goal is to decrease design complexity by decomposing the overall design problem according to physical phenomena (electrical, mechanical, thermal, structural, etc…), level of abstraction (static, lumped parameter dynamics, distributed parameter dynamics, etc…) or engineering discipline (performance, systems engineering, software engineering, manufacturing, etc…). Consequences of this design strategy are quite significant both in terms of weakening the opportunity for correct-by-construction design, as well as performing cross-domain optimizations in CPS design flows. The chief reason is that discipline oriented design flows usually miss modeling interactions/interdependences among the various design views. The approach would work if the design concerns were orthogonal, but in tightly coupled CPS this is not the case. The price of the simplification is decreased predictability of properties of the implemented CPS and costly re-design cycles.

In 2010, the Defense Advanced Research Project Agency (DARPA) initiated the Adaptive Vehicle Make (AVM) program[1] to construct a fully integrated model- and

[1] http://www.darpa.mil/Our_Work/TTO/Programs/Adaptive_Vehicle_Make__(AVM).aspx

component-based design flow for the "make" process of complex cyber-physical systems (CPS) [1]. The resulting integrated tool suite, OpenMETA, provides a manufacturing-aware design flow, which covers both cyber and physical design aspects. In order to test and demonstrate the capabilities of the new design flow and the integrated tool suite in a real-life system, the AVM program also includes the Fast, Adaptable, Next-Generation Ground Vehicle (FANG) design challenge sequence[2]. While the target system for the AVM program is ground vehicle, the created infrastructure for model- and component-based design is generic and targets radical changes in the overall "make" process of large CPS systems. Through our work in leading the research on the open-source design tool suite, OpenMETA, the open-source model exchange and web-based collaborative design environment, VehicleForge, and the curation effort for the FANG component model library, we had the opportunity to gain experience with the challenges of using model- and component-based design methods in large-scale CPS.

In this paper we focus on the impact of heterogeneity on the OpenMETA tool architecture. We argue that the primary barriers to apply model- and component-based design flows for CPS are the lack of the following three integration frameworks:

1. *Model Integration Framework*. Model integration is required for expressing interactions across modeling domains - creating the need for multi-modeling. Semantic heterogeneity of domain specific modeling languages (DSMLs) used in different modeling views and the fact that DSMLs in CPS subdomains evolve more or less independently, further add to the modeling language and model integration challenge. The overall semantic complexity of CPS modeling domains and the differences among CPS product categories make the development and standardization of some form of unified CPS multi-modeling languages impractical. Instead, a different solution is needed that enables the semantically sound integration of modeling domains.

2. *Tool Integration Framework*. End-to-end tooling for complex CPS product lines such as automotive and aerospace systems is too heterogeneous and extends to too many technical areas for a single tool vendors to fully cover. In addition, significant part of the companies' design flow is supported by in-house tools that are proprietary and capture high value design IP. Integration of end-to-end tool chains for highly automated execution of design flows is such a complex task that successful examples are hard to find – even after massive investment by OEMs. Change demands robust tool integration frameworks that go well beyond the semantically weak and necessarily fragile ad-hoc connection among tools.

3. *Execution Integration Platform*. The dominant approach in current tool suites is desktop integration using platforms such as Microsoft's Visual Studio[3], or Eclipse[4]. However, overall complexity and heterogeneity of CPS tool chains increasingly demand the use of software as a service (SaaS) models, web-based tool integration

[2] http://www.darpa.mil/Our_Work/TTO/Programs/AVM/AVM_Design_Com petitions_(FANG).aspx
[3] http://www.visualstudio.com/
[4] http://www.eclipse.org/

platforms, high performance cloud-based back-ends for model repositories and web-based distributed collaboration services.

The rest of the paper has the following structure. First, we provide an overview of the three layers and their relationships. Next, we analyze the challenges and lessons learned in the design and implementation of the Model Integration Framework and show the significance of Model Integration Languages in CPS design flows. Finally we discuss application experience and summarize the ongoing research efforts.

2 OpenMETA Integration Layers

Achieving the goal of "correct-by-construction" design requires that models and analysis methods in the design phase predict with the required accuracy the behavior of the designed system. Our approach to improve the predictability of design has been the explicit modeling of multi-physics, multi-abstraction and multi-fidelity interactions and providing methods for composing heterogeneous component models.

The OpenMETA design flow is implemented as a multi-model composition/synthesis process that incrementally shapes and refines the design space using formal, manipulable models [3][19]. The model composition and refinement process is intertwined with testing and analysis steps to validate and verify requirements and to guide the design process toward the least complex, therefore the least risky and least expensive solutions. The design flow follows a progressive refinement strategy, starting with early design-space exploration covering very large design spaces using abstract, lower fidelity models and progressing toward increasingly complex, higher fidelity models and focusing on rapidly decreasing number of candidate designs.

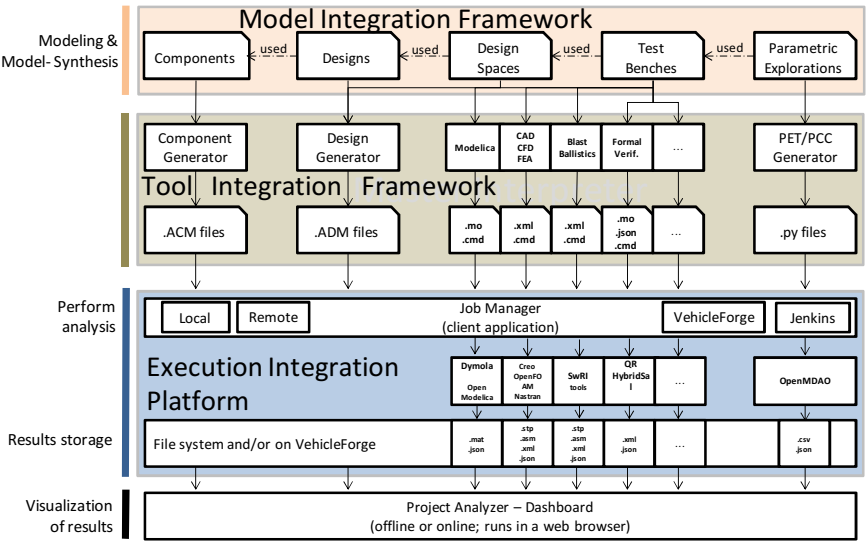


Fig. 1. Figure 1: OpenMETA integration frameworks

The META design flow proceeds in the following main phases:

1. Combinatorial design space exploration using static finite domain constraints and architecture evaluation.
2. Behavioral design space exploration by progressively deepening from qualitative discrete behaviors to precisely formulated relational abstractions and to quantitative multi-physics, lumped parameter hybrid dynamic models using both deterministic and probabilistic approaches.
3. Geometric/Structural Design Space Exploration coupled with physics-based nonlinear finite element analysis of thermal, mechanical and mobility properties.
4. Cyber design space exploration (both HW and SW) integrated with system dynamics.

As discussed before, automation of the design flow leads to complex integration challenges that we decomposed into three integration layers shown in Figure 1. Elements of the framework reflect primarily the FANG drive-train challenge, but the basic structure of the integration architecture remains the same for the hull design challenge as well, with larger emphasis on 3-D/CAD tools and a range of finite element analysis for verifying blast protection and hydrodynamic requirements.

In the following sections we describe each integration framework with more emphasis on model integration.

3 Model Integration Framework – Semantic Integration

The modeling and model-synthesis functions of the OpenMETA design flow is built on the introduction of the following model types:

1. AVM Component Models (ACM) with standard, composable interfaces
2. Design Models (DM) that describe component architectures and related constraints
3. Design Space Models (DSM) that define structural and architectural variabilities
4. Test Bench Models (TBM) representing environment inputs, composed system models connected to a range of testing and verification tools for key performance parameters, and
5. Parametric Exploration Models (PEM) for specifying regions in the design space to be used for optimization and models for complex analysis flows producing results such as Probabilistic Certification of Correctness (PCC).

In META, as well as in all other approaches to model-based design, modeling languages and their underlying semantics play a fundamental role in achieving compositionality. Heterogeneity of the multi-physics, multi-abstraction and multi-fidelity design space, and the need for rapidly evolving/updating design flows require the use of a rich set of modeling languages usually influenced/determined by existing and emerging model-based design, verification and simulation technologies and tools. Consequently, the language suite and the related infrastructure cannot be static; it will continuously evolve. To address both heterogeneity and evolvability simultaneously,

we departed from the most frequently used approach to address heterogeneity: the development or adoption of a very broad and necessarily hugely complex language standard designed for covering all relevant views of a multi-physics and cyber domains. Instead, we placed emphasis on the development of a model integration language – CyPhyML – with constructs limited to modeling the interactions among different modeling views (see Figure 2).

3.1 Model Integration Language and Semantic Interfaces

CyPhyML targets multi-modeling – it advances multi-modeling from a mere "ensemble" of models to a formally and precisely integrated, mathematically sound suite of models. Integration of the modeling language suite by CyPhyML is minimal in a sense that only those abstractions that are imported from the individual languages to CyPhyML are those required those for modeling cross-domain interactions. Since the suite of engineering tools is changing and the modeling languages of the individual tools (such e.g. Modelica) evolve independently from the model integration framework, CyPhyML is constructed as a light-weight, evolvable, composable integration language that is frequently updated and morphed. While these DSMLs may be individually quite complex (Modelica, Simulink, SystemC, etc.) ChyPhyML is relatively simple and easily evolvable. This "semantic interface" between CyPhyML and the domain specific modeling languages (DSML) (Figure 2) is formally defined, evolved as needed, and verified for essential properties (such as well-formedness and consistency) using the methods and tools of formal metamodeling [4][7]. By design, CyPhyML is moving in the opposite direction to unified system design languages, such as SysML or AADL. Its goal is specificity as opposed to generality, and heavy weight standardization is replaced by layered language architecture and specification of explicit semantics.

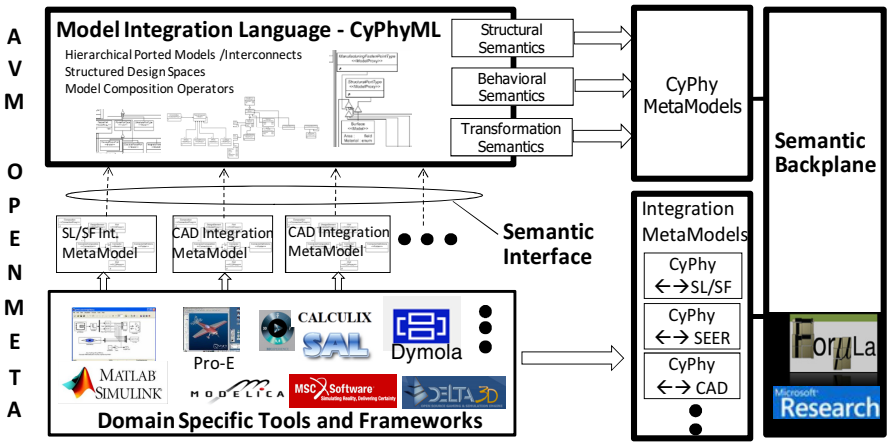


Fig. 2. Model Integration Framework

3.2 Semantic Backplane of Open META

The "cost" of introducing a dynamic model integration language is that a mathematically precise formal semantics for model integration had to be developed. The OpenMETA Semantic Backplane [4][22][23] is at the center of our semantic integration concept. The key idea is to define the structural [5] and behavioral semantics [8] of the CyPhy model integration language using formal metamodeling, and use a tool supported formal framework for updating the CyPhy metamodels and verifying its overall consistency and completeness as the modeling languages are evolving. The selected tool for formal metamodeling is FORMULA[5] from Microsoft Research [10]. FORMULA's algebraic data types (ADTs) and constraint logic programming (CLP) based semantics is rich enough for defining mathematically modeling domains, transformations across domains, as well as constraints over domains and transformations. In the followings we provide a brief summary of the basic elements of the mathematical framework used in the Semantic Backplane.

Structural semantics of modeling languages represents the domain of well-formed models [5]. Domains are modeled as a term algebra whose function symbols characterize the key sets and relations through uninterpreted functions. A syntactic instance of some structural semantics is a finite set of terms over its term algebra $TY(\Sigma)$, where Σ is an infinite alphabet of constants, Y is a finite set of n-ary function symbols (signature) standing for uninterpreted functions, and the algebra is inductively defined as the set of all terms that can be constructed from Σ and Y. A syntactic instance of some structural semantics is a finite set of terms over its term algebra $TY(\Sigma)$.The set of all syntactic instances is then the power set of its term algebra: $P(TY(\Sigma))$.

To avoid the many unintended instances of the syntax of a modeling language, FORMULA enriches term algebra semantics with types by reconstructing Σ as the union of smaller alphabets and alphabets are ordered by set inclusion. Structural semantics often contain complex conformance rules; these rules cannot be captured by simple-type systems. One common solution to this problem is to provide an additional constraint language for expressing syntactic rules such as the Object Constraint Language (OCL). Unlike other approaches, FORMULA choose Constraint Logic Programming (CLP) to represent syntactic constraints because it can extend term algebra semantics while supporting declarative rules and unlike purely algebraic specifications it provides a clear execution semantics for logic programs making it possible to specify model transformations in the same framework. FORMULA supports a class of logic programs with the following properties: (1) expressions may contain uninterpreted function symbols, (2) the semantics for negation is negation as finite failure, (3) all logic programs must be stratified and (4) supports fixpoint logic over theories [13][14].

Expressiveness of the formal framework discussed up to this point is sufficient for formalizing structural semantics, but does not support yet the specification of behavioral semantics. Since multi-physical modeling of systems requires modeling languages for continuous (DAE, PDE), discrete and hybrid dynamics, semantics need to be defined both denotationally and operationally [4]. Fortunately, the key to

[5] `http://research.microsoft.com/formula`

formalization in both cases is the development of precise specification of model transformations. In addition, formal modeling of model transformation are fundamental in all design automation frameworks, because they are used pervasively in integrating tool chains. In the OpenMETA Semantic Backplane, model transformations are encoded as logic programs where data types distinguish the inputs and outputs of the transformation [10]. For example:

```
Filter = out.MetaNode(x) :- in.MetaNode(x).
```

The constructor in.MetaNode() stands for primitives at the input of the transformation. Similarly, out.MetaNode() stands for primitives on the output of the transformation. A transformation is executed by providing an interpretation for the input primitives, and then computing the output primitives according to the CLP semantics. Specifying model transformations in the same CLP framework has fundamental advantages in allowing reasoning over the fully integrated representation of the input and output domains (e.g. proving that selected invariants will hold before and after the transformation).

While ADTs and CLP are sufficient for defining complex modeling domains and transformations, consistency checking and constructive modeling (model finding) [10] require the generation of automatic proofs from formal specifications by solving CLP satisability problems. Satisfiability is different from checking satisfaction of goals that be solved by simply running a logic program. It is to determine if a CLP program can be extended by a finite set of facts so that a goal is satisfied [9][16]. It requires searching through (infinitely) many possible extensions, which we achieve by efficient forward symbolic execution. FORMULA achieves this by efficient forward symbolic execution of logic program into the state-of-the-art satisfiability modulo theories (SMT) solver Z3 [15]. As a result, specifications can include variables ranging over infinite domains and rich data types (partial models). The method is constructive; it returns extensions of the CLP program witnessing goal satisfaction. An interesting application of this capability is design-space exploration [12].

The Model Integration Framework of OpenMETA (Figure 1) currently includes a large suite of modeling languages and tools for multi-physics, multi-abstraction and multi-fidelity modeling such as OpenModelica, Dymola, Bond Graphs, Simulink/Stateflow, STEP, ESMOL and many others. The CyPhyML model integration language provides the integration across this heterogeneous modeling space and the FORMULA - based Semantic Backplane provides the semantic integration for all OpenMETA composition tools.

4 Tool Integration Framework

The OpenMETA Tool Integration Framework (see Figure 1) comprises a network of model transformations that compose models for individual tools (e.g. Modelica models from ChyPhyML design models and component models) and integrate model-based design flows. Model-transformations are used in the following roles:

1. *Packaging.* Models are translated into a different syntactic form without changing their semantics. For example, AVM Component Models and AVM Design Models are translated into standard Design Data Packages (Figure 1, .ACM and .ADM files) for consumption by a variety of design analysis, manufacturability analysis and repository tools.
2. *Composition.* Model- and component-based technologies are based on composing different design artifacts (such as DAE-s for representing lumped parameter dynamics as Modelica equations [23], input models for verification tools [25], CAD models of component assemblies [19], design space models [25], and many others) from appropriate models of components and component architectures.
3. *Virtual prototyping.* Several test and verification methods (such as Probabilistic Certificate of Correctness – PCC) require test benches that embed a virtual prototype of the designed system executing a mission scenario in some environment (as defined in the requirement documents). We found distributed, multi-model simulation platforms the most scalable solution for these tests. We selected the High Level Architecture (HLA) as the distributed simulation platform and integrated FMI Co-Simulation components with HLA [26].
4. *Analysis flow.* Parametric explorations of designs (PET), such as analyzing effects of structural parameters (e.g. length of vehicle) on vehicle performance, or deriving PCC for performance properties frequently require complex analysis flows that include a number of intermediate stages. Automating design space explorations require that Python files controlling the execution of these flows on the Multidisciplinary Design Analysis and Optimization (OpenMDAO[6]) platform (that we currently use in OpenMETA) are autogenerated from the test bench and parametric exploration models (Figure 1).

Continuous evolution of the OpenMETA design flow makes it essential that the modeling tool suite for ChyPhyML is metaprogramable [4][17][20][18] and all model transformations used in the Composition Framework are formally specified as part of the Semantic Backplane. We believe that the lack of these capabilities are significant contributors to the failure of numerous large-scale model and tool integration efforts, due to the fact that semantic errors are all but impossible to detect without formal models.

Advantages of the Semantic Backplane and the logic-based formal framework is particularly important in the specification of composition semantics for mixed, multi-physical and computation modeling. For physical interactions, we chose acausal, power flow oriented modeling (e.g. Modelica, Simscape or Bond Graph modeling languages). In this approach, safe modeling of multi-physics interactions require rich typing for expressing and enforcing connectivity constraints. Beyond these static constraints, the semantics of acausal physical interconnections are expressed using algebraic constraints over the effort and flow variables [3]. This leads to a formal composition semantics that is simply the merging of the DAE equations representing component behaviors with the interconnection constraints [4]. Since our logic-based formal framework is expressive enough for describing typing and variables ranging

[6] `http://openmdao.org/`

over infinite domains and rich data types), description of composition semantics denotationally is quite straightforward in FORMULA.

5 Execution Integration Platform

The OpenMETA model and tool integration technology needs an infrastructure for creating and executing complex analysis flows including heterogeneous tool components. Our Analysis and Execution Framework (Figure 1) includes a wide range of cloud-deployed services such as component model repositories, ontology driven search engine, collaboration mechanisms, and cloud-deployed tools for design space exploration and data analytics. Our VehicleForge platform developed for DARPA's AVM program is essentially a gateway to shared resources and integrated services, not all of which are collocated. A central aim was to provide users secure and centrally managed access to these resources without the responsibility to individually respond to their evolution. An essential aspect of VehicleForge is its "software-as-a-service" delivery model. It allows the low-cost access of end users (individuals, research groups, and larger companies) to repositories, analytic services and design tools, without the very high cost of acquiring and maintaining desktop engineering tools.

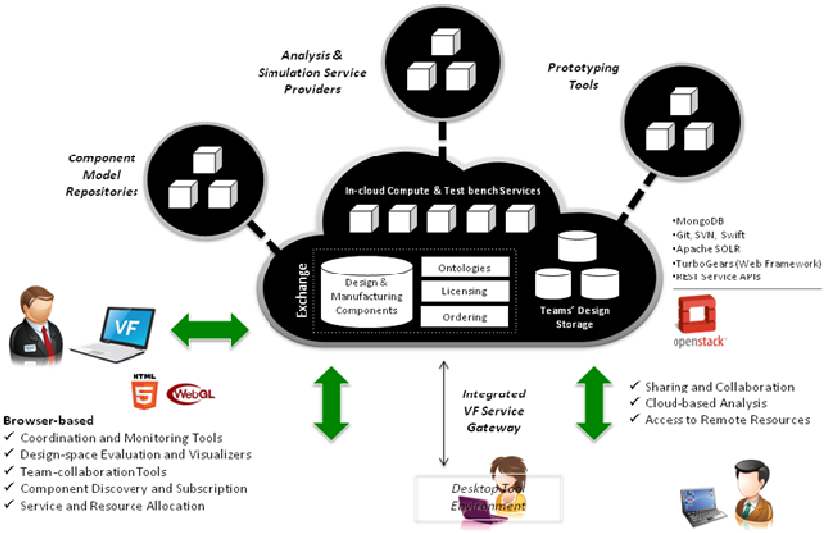


Fig. 3. VehicleForge Execution Integration Platform

Two key aspects of sustainability are addressed by this platform. The first is resource elasticity and service staging to address scalability and system-wide optimization. This is a fundamental cloud computing rationale, which is predicated on the dynamism of demand for various forms of infrastructure over time. The second, perhaps more vital aspect, is managing the evolution of data, data representations, and their use by services and service integrations over time.

6 Lessons Learned

The first release of the OpenMETA tool suite capable of model-based compositional design, design-space exploration, multi-physics analysis and virtual performance testing was used during the FANG Mobility/Drivetrain Challenge from January15 to April 15, 2013. During this competitive design event, over 1000 competitors organized into over 250 teams worked to design the drivetrain, suspension, propulsion elements and associated subsystems for FANG. The FANG component of the AVM program is currently building the winning design using the capabilities of the AVM program foundry, iFAB. Our team is currently expanding the OpenMETA tool suite with modeling and analysis capabilities required for hull design with strong focus on blast protection, structure, and fluid dynamics. This is preparation for the upcoming hull design challenge in February 2014.

Our work in the AVM program yields to two different kinds of results. First, we have created an end-to-end integrated tool suite, OpenMETA, that is now slated for transitioning to both the industry and the academic research communities. To ease transitioning and enable continued community-based development of OpenMETA, most of the tools integrated into the tool suite are open source with liberal BSD or MIT licensing. The second result is the insight we gained regarding promising directions and open problems in model- and component-based design. Below we list three essential points we have identified.

1. *Horizontal integration layers.* CPS companies face immense pressures to deliver safe and complex systems at low cost. End-to-end tooling for CPS industries is too heterogeneous and spans too many technical areas for any single tool vendor to fully support. In addition, CPS companies must develop, maintain and integrate in-house, proprietary tools to remain competitive. Consequently, integration of models and modeling languages in design flows, integration of tools into tool chains capable for the highly automated execution of design processes have emerged as a major challenge. Ad-hoc integration of models, tools and automated analysis threads is fragile, intractable, error prone and extremely costly. An essential insight of OpenMETA is that horizontal integration layers are fundamentally important in end-to-end CPS design flows. Their complexity requires the establishment of integration frameworks that provide the foundations and reusable, high-complexity components for rapid integration and evolution of CPS design tool chains.
2. *Component model repositories*. Component models capture reusable design knowledge, therefore model repositories play crucial role in improving design productivity. AVM components contain a suite of modeling views and their interactions: static properties, structural, behavioral, geometric, cyber on multiple levels of fidelity. Selected abstractions of these modeling views are exposed via the components' semantic interfaces for composition. Building reusable, composable model libraries requires deep domain understanding and semantic rigor. Open research topics include hard problems such as formal relationship among modeling abstractions, establishing multiple fidelity levels, and understanding methods for representing and managing uncertainties.

3. *Goal-driven model composition.* In real-life CPS, model composition methods frequently lead to extremely large models. For example, composition of lumped parameter physical dynamics easily produces models including tens of thousands of equations with algebraic loops and non-linearities. A common problem is that simulation and verification tools do not scale to this complexity. The most promising research direction we identified is goal directed composition, that composes models according to the system property under study.

Acknowledgements. Authors are grateful for the advice and directions they received from Prof. Joseph Sifakis and Prof. Alberto Sangiovanni-Vincentelli, members of the Senior Strategy Group of the program. The OpenMETA and VehicleForge projects involve a large group at ISIS/Vanderbilt. Authors recognize the exceptional contributions of Zsolt Lattman, Adam Nagel, Jason Scott to OpenMETA. This research is supported by the Defense Advanced Research Project Agency (DARPA) under award # HR0011-12-C-0008 and the National Science Foundation under award # CNS-1035655.

References

1. Eremenko, P.: Philosophical Underpinnings of Adaptive Vehicle Make. DARPA-BAA-12-15. Appendix 1 (December 5, 2011)
2. Gossler, G., Sifakis, J.: Composition for component-based modeling. Science of Computer Programming - Formal Methods for Components and Objects Pragmatic Aspects and Applications 55(1-3), 161–183 (2005)
3. Lattmann, Z., Nagel, A., Scott, J., Smyth, K., van Buskirk, C., Porter, J., Neema, S., Bapty, T., Sztipanovits, J.: Towards Automated Evaluation of Vehicle Dynamics in System-Level Design. In: Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2012, Chicago, IL, August 12-15 (2012)
4. Simko, G., Levendovszky, T., Neema, S., Jackson, E., Bapty, T., Porter, J., Sztipanovits, J.: Foundation for Model Integration: Semantic Backplane. In: Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2012, Chicago, IL, August 12-15 (2012); Sztipanovits J., Karsai G.: Model-Integrated Computing. IEEE Computer 30, 110–112 (1997)
5. Jackson, E., Sztipanovits, J.: Formalizing the Structural Semantics of Domain-Specific Modeling Languages. Journal of Software and Systems Modeling, 451–478 (September 2009)
6. Sangiovanni-Vincentelli, A.: Quo Vadis, SLD? Reasoning about the Trends and Challenges of System Level Design. Proc. of the IEEE 95(3), 467–506 (2007)
7. Jackson, E., Porter, J., Sztipanovits, J.: Semantics of Domain Specific Modeling Languages. In: Mosterman, P., Nicolescu, G. (eds.) Model-Based Design of Heterogeneous Embedded Systems, November 24, pp. 437–486. CRC Press (2009)
8. Chen, K., Sztipanovits, J., Neema, S.: Compositional Specification of Behavioral Semantics. In: Lauwereins, R., Madsen, J. (eds.) Design, Automation, and Test in Europe: The Most Influential Papers of 10 Years DATE. Springer (2008)

9. Jackson, E.K., Sztipanovits, J.: Constructive Techniques for Meta- and Model-Level Reasoning. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 405–419. Springer, Heidelberg (2007)
10. Jackson, E.K., Tiham, Balasubramanian, D.: Reasoning about Metamodeling with Formal Specifications and Automatic Proofs. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 653–667. Springer, Heidelberg (2011)
11. Sangiovanni-Vincentelli, A., Shukla, S., Sztipanovits, J., Yang, G.: Metamodeling: An Emerging representation Paradigm for System-Level Design. IEEE Design and Test of Computers (May/June 2009)
12. Jackson, E., Simko, G., Sztipanovits, J.: Diversely Enumerating System-Level Architectures. In: Proceedings of EMSOFT 2013, Embedded Systems Week, Montreal, CA, September 29-October 4 (2013)
13. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. 33(3), 374–425 (2001)
14. Jackson, E.K., Schulte, W.: Model Generation for Horn Logic with Stratified Negation. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 1–20. Springer, Heidelberg (2008)
15. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
16. Jackson, E.K., Sztipanovits, J.: Constructive Techniques for Meta- and Model-Level Reasoning. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 405–419. Springer, Heidelberg (2007)
17. Karsai, G., Maroti, M., Ledeczi, A., Gray, J., Sztipanovits, J.: Composition and cloning in modeling and meta-modeling. IEEE Transactions on Control Systems Technology 12(2), 263–278 (2004)
18. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. IEEE Computer 34(11), 44–51 (2001)
19. Wrenn, R., Nagel, A., Owens, R., Yao, D., Neema, H., Shi, F., Smyth, K., van Buskirk, C., Porter, J., Bapty, T., Neema, S., Sztipanovits, J., Ceisel, J., Mavris, D.: Towards Automated Exploration and Assembly of Vehicle Design Models. In: Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2012, Chicago, IL, August 12-15 (2012)
20. Karsai, G., Ledeczi, A., Neema, S., Sztipanovits, J.: The model integrated computing tool suite: Metaprogrammable tools for embedded control system design. In: Proceedings of the IEEE Joint Conference CCA, ISIC and CACSD, Munich, Germany (2006)
21. Sztipanovits, J.: Cyber Physical Systems: Convergence of Physical and Information Sciences. In: Information Technology, pp. 257–265. Oldenbourg Wissenschaftsverlag GmbH (June 2012)
22. Simko, G., Levendovszky, T., Maroti, M., Sztipanovits, J.: Towards a Theory for Cyber-Physical Systems Modeling. In: Proc. 3rd Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy 2013), Philadelphia, USA, April 08-11, pp. 1–6 (2013)
23. Simko, G., Lindecker, D., Levendovszky, T., Neema, S., Sztipanovits, J.: Specification of Cyber-Physical Components with Formal Semantics – Integration and Composition. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 471–487. Springer, Heidelberg (2013)

24. Eyisi, E., Zhang, Z., Koutsoukos, X., Porter, J., Karsai, G., Sztipanovits, J.: Model-Based Design and Integration of Cyber-Physical Systems: An Adaptive Cruise Control Case Studies. Journal of Control Science and Engineering, Special Issue on Embedded Model-Based Control 2013, Article ID 678016, 15 pages (2013)
25. Fritzson, P., Lattmann, Z., Pop, A., de Kleer, J., Janssen, B., Neema, S., Bapty, T., Koutsoukos, X., Klenk, M., Bobrow, D., Saha, B., Kurtoglu, T.: Verification and Design Exploration through Meta Tool Integration with OpenModelica. In: 10th International Modelica Conference 2014, Lund, Sweden, March 10-12 (2014)
26. Neema, H., Gohl, J., Lattmann, Z., Sztipanovits, J., Karsai, G., Neema, S., Bapty, T., Batteh, J., Tummescheit, H.: Model-Based Integration Platform for FMI Co-Simulation and Heterogeneous Simulations of Cyber-Physical Systems. In: 10th International Modelica Conference 2014, Lund, Sweden, March 10-12 (2014)

Feedback in Synchronous Relational Interfaces*

Stavros Tripakis[1,2] and Chris Shaver[1]

[1] University of California, Berkeley, USA
[2] Aalto University, Finland

Abstract. Synchronous relational interfaces is an interface theory which allows to specify the I/O interface of a component with input requirements and relational input-output guarantees. The theory allows to check interface compatibility during composition and to compute a composite interface from the atomic ones. It provides a refinement operator which allows to check whether a component can safely replace another one. This paper discusses the options and challenges in defining feedback composition in the context of this theory.

1 Introduction

Compositionality is not simply a desirable property in system design, but a "must" for building large and complex systems from smaller and simpler components. In his long career, Joseph Sifakis has pursued numerous research topics around compositionality, including, but not limited to [13,15,11,3,14,4,1,5].

The work presented in this paper is also on the general subject of compositionality. Our work approaches the subject following the framework of so-called *interface theories* [8,7]. In interface theories, components are captured by abstract models generically called interfaces. Such a theory also provides one or more interface composition operators, each allowing to obtain an interface for a composite component (i.e., a network of connected subcomponents) from the interfaces of the subcomponents. Finally, an interface theory provides a *refinement* relation between interfaces, which typically comes with two key theorems:

- Preservation of properties of interest by refinement: if interface I satisfies a given property ϕ (say, a safety property expressed in temporal logic), and interface I' refines I, then I' also satisfies ϕ.
- Preservation of refinement by composition: if I'_1 refines I_1, I'_2 refines I_2, and $\odot$ is a composition operator, then $I'_1 \odot I'_2$ refines $I_1 \odot I_2$.

Together the above theorems enable an incremental design methodology, which, for instance, allows to reduce the problem of checking substitutability (when can

* This work was partially supported by the the Academy of Finland and by the NSF via projects *COSMOI: Compositional System Modeling with Interfaces* and *ExCAPE: Expeditions in Computer Augmented Program Engineering*. This work was also partially supported by IBM and United Technologies Corporation (UTC) via the iCyPhy consortium, and by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 249–266, 2014.

a component with interface I' replace a component with interface I?) to that of checking whether I' refines I. Indeed, if I is used in a certain composition $I \odot C$, and I' refines I, then I' can replace I to obtain the new composition $I' \odot C$. By preservation of refinement by composition, $I' \odot C$ refines $I \odot C$. By preservation of properties by refinement, if $I \odot C$ satisfies a given ϕ, then so does $I' \odot C$. This means that, provided the existing system $I \odot C$ is known to be correct (i.e., to satisfy ϕ), the new system $I' \odot C$ need not be re-verified from scratch. We only need to check whether I' refines I to ensure substitutability.

A number of interface theories have been proposed over the years, starting from the original *interface automata* (IA) theory proposed in [7]. In this paper we consider the theory of *synchronous relational interfaces* proposed in [16]. Whereas interface automata use an asynchronous model of concurrency based on interleaving and input-output label synchronization, synchronous relational interfaces use synchronous composition similar to finite state machines of type Moore or Mealy.

In addition, compared to interface automata, synchronous relational interfaces offer a more compact and symbolic specification formalism. For instance, consider a component which receives as input an integer, adds one to it, and outputs it. This component could be modeled using the synchronous relational interface $(\{x\}, \{y\}, y = x + 1)$. Here, x and y are the input and output variables, respectively, and $y = x + 1$ is a formula capturing the input-output relation (or *contract*). The same component could be captured as an interface automaton, but this would most likely require an infinite number of states, transitions, and labels, to capture the infinite domain of possible input/output values.[1]

The theory of synchronous relational interfaces provides three composition operators: composition in series (connecting an output of one component to an input of another), in parallel (placing the two components next to each other without any connections), and in feedback (connecting one of the outputs of a component to one of its inputs). These are standard composition schemes found in synchronous systems such as, for instance, digital circuits. This work is concerned specifically with feedback composition. The work of [16] allows only a restricted form of feedback composition. This paper recalls the reasons why this is so, discusses why it would be desirable to extend the theory to allow a less restrictive version of feedback, and examines the challenges in doing so.

2 Background: Synchronous Relational Interfaces

For the purposes of this paper, it suffices to restrict ourselves to the simplest form of synchronous relational interfaces, namely, *stateless* interfaces, where the input/output contract is the same during the dynamic behavior of the component, i.e., at every synchronous cycle. *Stateful* interfaces are also considered

[1] We can imagine an interface automaton with transitions of the form $\xrightarrow{i_0?}\xrightarrow{o_1!}$, $\xrightarrow{i_1?}\xrightarrow{o_2!}$, etc., where $i_n?$ is the input action corresponding to reading input $x = n$, and $o_k!$ is the output action corresponding to writing output $y = k$.

in [16], where the contract may change from one cycle to the next. We only consider stateless interfaces in the sequel.

A (stateless synchronous relational) interface is a triple

$$I = (X, Y, \phi)$$

where X is a finite set of input variables, Y is a finite set of output variables, $X \cap Y = \emptyset$ (input and output variables are disjoint), and ϕ (the *contract*) is a relation between values of input and output variables, typically represented as a logical formula on the set of variables $X \cup Y$. We assume a universe U of possible values for all variables. $V(X)$ denotes the set of *assignments* (or *valuations*) over a set of variables X, that is, the set of all functions of the form $a : X \to U$. Semantically, a formula ϕ on $X \cup Y$ denotes the set of assignments over $X \cup Y$ which satisfy ϕ.

Assuming $Y = \{y_1, y_2, ..., y_n\}$, we define

$$\mathbf{in}(\phi) \quad := \quad \exists Y : \phi \quad := \quad \exists y_1 : \exists y_2 : \cdots \exists y_n : \phi.$$

That is, $\mathbf{in}(\phi)$ is syntactically a formula only on input variables X, characterizing the set of *legal* input assignments. For example, if ϕ is $x \neq 0 \wedge y \geq x$, and x is an input and y an output, then $\mathbf{in}(\phi) \equiv x \neq 0$, meaning that $x = 0$ is illegal.

ϕ, and in turn I, are called *input-complete* when $\mathbf{in}(\phi) \equiv \mathbf{true}$, i.e., when all input assignments are legal for I.

ϕ, and in turn I, are called *deterministic* when for every legal input assignment, i.e., for every function $a_X : X \to U$ satisfying $\mathbf{in}(\phi)$, there is a unique output assignment $a_Y : Y \to U$, such that the pair (a_X, a_Y) satisfies ϕ.

Parallel Composition. Parallel composition of relational interfaces can be defined by taking the conjunction of their corresponding contracts. Let $I_i = (X_i, Y_i, \phi_i)$, for $i = 1, 2$, where all sets X_1, X_2, Y_1, Y_2 are pair-wise disjoint. Then

$$I_1 || I_2 := (X_1 \cup X_2, Y_1 \cup Y_2, \phi_1 \wedge \phi_2)$$

Note that $\mathbf{in}(\phi_1 \wedge \phi_2) \equiv \exists Y_1, Y_2 : \phi_1 \wedge \phi_2 \equiv \exists Y_1 : (\phi_1 \wedge \exists Y_2 : \phi_2) \equiv (\exists Y_2 : \phi_2) \wedge (\exists Y_1 : \phi_1) \equiv \mathbf{in}(\phi_1) \wedge \mathbf{in}(\phi_2)$.

Serial Composition. For reasons thoroughly explained in [16], and not repeated here, serial composition of relational interfaces is defined using the principle of "demonic" non-determinism. In the simple case, where $I_1 = (\{x\}, \{y\}, \phi_1)$ and $I_2 = (\{y\}, \{z\}, \phi_2)$, the serial composition of I_1 and I_2, denoted $I_1 \rightsquigarrow I_2$, and consisting of connecting the output y of I_1 to the input y of I_2, is defined as follows:

$$I_1 \rightsquigarrow I_2 := \Big(\{x\}, \{y, z\}, \phi_1 \wedge \phi_2 \wedge \big(\forall y : \phi_1 \to \mathbf{in}(\phi_2)\big)\Big)$$

Note that if I_2 is input-complete, then $\big(\forall y : \phi_1 \to \mathbf{in}(\phi_2)\big) \equiv \mathbf{true}$ and the contract of $I_1 \rightsquigarrow I_2$ becomes $\phi_1 \wedge \phi_2$. The same is true when I_1 is deterministic.

When the contract of $I_1 \rightsquigarrow I_2$ is equivalent to **false** (i.e., unsatisfiable), we say that $I_1 \rightsquigarrow I_2$ is *invalid* and that I_1 and I_2 are *incompatible*. Otherwise, we say that $I_1 \rightsquigarrow I_2$ is *valid* and that I_1 and I_2 are *compatible*.

Example 1. Let $I_1 = (\{x\}, \{y\}, x \leq y)$ and $I_2 = (\{y\}, \{z\}, y \neq 0)$. Then

$$I_1 \rightsquigarrow I_2 = (\{x\}, \{y, z\}, x \leq y \wedge y \neq 0 \wedge x > 0)$$

where it is worth noting the additional input assumption $x > 0$ obtained thanks to the term $\forall y : x \leq y \rightarrow y \neq 0$. □

Refinement. Refinement between relational interfaces is defined as follows. Let $I_i = (X, Y, \phi_i)$, for $i = 1, 2$. Then, I_2 refines I_1, written $I_2 \sqsubseteq I_1$, iff

$$\mathbf{in}(\phi_1) \rightarrow \mathbf{in}(\phi_2) \qquad \text{and} \qquad \big(\mathbf{in}(\phi_1) \wedge \phi_2\big) \rightarrow \phi_1$$

are both valid formulas, i.e., equivalent to **true**.

It is shown in [16] that refinement preserves compatibility, that is, if $I_1 \rightsquigarrow I_2$ is valid, and $I_1' \sqsubseteq I_1$ and $I_2' \sqsubseteq I_2$, then $I_1' \rightsquigarrow I_2'$ is also valid.

Feedback. Suppose we want to connect an output $y \in Y$ of a relational interface $I = (X, Y, \phi)$ to one of its inputs $x \in X$. In the framework of [16], this is allowed only when I is so-called *Moore with respect to* x. For stateless interfaces, Moore with respect to x means that ϕ does not refer to x. In that case, connecting y to x results in a new interface where x is an output equal to y:

$$\mathit{feedback}_{y \rightsquigarrow x}(I) \quad := \quad (X - \{x\}, Y \cup \{x\}, \phi \wedge x = y).$$

The Problem: General Feedback Does Not Preserve Refinement. The reason why feedback is restricted to Moore interfaces is illustrated in the following example, borrowed from [9] and also discussed as Example 9.11 in [16].

Example 2. Let $I = (\{x, z\}, \{y\}, \mathbf{true})$ and $I' = (\{x, z\}, \{y\}, x \neq y)$. Then $I' \sqsubseteq I$. Suppose that we want to feed the output of I back to its first input, that is, we want to connect y to x. The straightforward way to define the resulting feedback composition is by adding the constraint $x = y$ to the contract of I. This constraint represents the fact that, once x and y are connected (imagine a wire connection between the two) their values become equal. Adding this constraint, that is, taking the conjunction of $x = y$ with the contract of I, which is **true**, we obtain the new interface $I_f = (\{z\}, \{x, y\}, x = y)$. In I_f, x is now an output, since it has been connected to y. Moreover, the new contract is $x = y$.

Let us try to do the same with I', that is, connect its output y to its input x. Doing so, we obtain the new interface $I_f' = (\{z\}, \{x, y\}, x \neq y \wedge x = y)$. Since the formula $x \neq y \wedge x = y$ is unsatisfiable, I_f' is equivalent to the interface $(\{z\}, \{x, y\}, \mathbf{false})$.

The problem now is that $I_f' \sqsubseteq I_f$ does **not** hold. This shows that the straightforward way of defining feedback results in refinement not being preserved by feedback (I' refines I, but I_f' does not refine I_f). □

3 Generalizing Feedback

In this section, we first discuss why a more general form of feedback would be desirable, and then the challenges that need to be overcome in order to achieve this more general form of feedback.

3.1 Why Generalize the Definition of Feedback?

One might say that Example 2 is too artificial to be of value, and that forbidding feedback for non-Moore interfaces makes sense. Consider, however, the following example:

Example 3. Take the parallel composition of two interfaces with contracts $y_i = x_i$, where x_i is an input and y_i is an output, for $i = 1, 2$. The resulting interface has contract $y_1 = x_1 \wedge y_2 = x_2$. This product interface is not Moore in neither x_1 nor x_2. Thus, we cannot form the feedback composition by connecting, say, y_2 to x_1. One might expect, however, that this feedback connection is the same as connecting the two original interfaces in series. □

What Example 3 illustrates is that the restriction to Moore interfaces results in serial composition not being equivalent to parallel composition followed by feedback. Can we relax the restrictions so as to obtain a definition of feedback which allows to express serial composition as parallel composition followed by feedback? We examine the challenges in achieving this goal next.

3.2 Challenge: Monolithic Order

Example 3 suggests that the definition of parallel composition is too *monolithic*, in the sense that it loses dependency information between inputs and outputs. This seems to be a fundamental problem, as illustrated with an even simpler example:

Example 4. Consider interfaces $I_1 = (\{\}, \{y\}, \mathbf{true})$ and $I_2 = (\{x, z\}, \{\}, \mathbf{true})$. I_1 has only an output y and I_2 has two inputs x, z. Clearly, the serial composition $I_1 \rightsquigarrow_{y \rightsquigarrow x} I_2$ formed by connecting y to x [2] is valid, since I_2 is input-complete.

Now let's try to form the same composition by first taking the parallel composition of I_1 and I_2, followed by feedback. The parallel composition of I_1 and I_2 is $I_1||I_2 = (\{x, z\}, \{y\}, \mathbf{true})$. This is exactly interface I which we saw in Example 2. If we forbid connecting I in feedback, as suggested above, then I_1 connected in series with I_2 would **not** be equivalent with $I_1||I_2$ connected in feedback, since the latter connection would be forbidden. □

The problem here seems to be the following. When we form the composition in series of I_1 and I_2, we interpret it as a game where I_1 plays first, choosing the output y, and I_2 plays second, accepting y as input x. Therefore, y is chosen first, and then assigned to x. (The point where z is chosen is irrelevant here.)

[2] This composition is defined after renaming x to y in I_2.

On the other hand, in a "monolithic" interface such as I, the interpretation of the game is different. First, the environment chooses the input x, and only afterwards does I reply with the output y. By forming the parallel composition of I_1 and I_2, we forced the order $x \to y$. Adding the feedback creates the opposite order $y \to x$, that is, a cycle. This is not the case with composition in series, which only has $y \to x$.

One might try to fix this by enriching the definition of interface to contain also dependency information between input and output variables. In our example, this means that there would be two versions of the interface $(\{x, z\}, \{y\}, \mathbf{true})$. One version where the output y depends on x (this would be I), and another version where y does not depend on x (this would be $I_1||I_2$).

But when we attempt to add such I/O dependency information, we run into new problems. This is explained next:

3.3 Interfaces with I/O Dependency Information

General Partial Orders on I/O Variables. Let us first try an approach where an interface is extended with a general partial order D on input and output variables. That is, an interface then becomes a quadruple $I = (X, Y, \phi, D)$ where X, Y, ϕ are the inputs, outputs and contract as previously, and D is a partial order on $X \cup Y$. The idea is that D represents dependencies between the variables, and also the order in which they are evaluated, as well as the possible ways for playing the game between the component and its environment. For example, if x is input and y is output, then dependency $x \to y$ means that, first the environment chooses x and then the component chooses y. $y \to x$ means that first the component chooses y and then the environment chooses x. If x, y are unrelated then they can be evaluated in any order.

This seems to solve the problems identified in §3.2, as it allows to distinguish I (which has the dependency $x \to y$) from $I_1||I_2$ (which has no dependency).

But consider another example:

Example 5. Let $A = (\{x\}, \{y\}, x = 0 \to y = 0, \{x \to y\})$ and $B = (\{z, u\}, \{\}, z = u, \{\})$. Suppose we wish to connect A and B in series, by connecting y to z. Is the connection valid? It should be, because the environment has two possible strategies for setting the free inputs x, u:

- either set $x = u = 0$, in which case A is forced to set $y = 0$, thus $z = 0$, thus $u = z$ and the input assumptions of B are satisfied;
- or set x to an arbitrary value, wait to observe output y of A, then set $u = y$, so that again $u = z$ is satisfied.

It seems that these two strategies cannot be represented with just a single contract ϕ and a single dependency relation D. Suppose they could. Then D would be $x \to y \to z$: notice that u is independent, since it could be given either at the same time as x, or after observing y.

Now, what would ϕ be? If u is given at the same time as x, then $x = 1, u = 0$ is not a possible assignment. On the other hand, if x is first set to 1, and then

y is set to 0, which means that also $z = 0$, then u must be set to 0. So, with the second strategy, $x = 1, u = 0$ is a valid assignment, whereas with the first strategy, it is not. □

This example appears to suggest that we need *sets* of pairs (ϕ, D), instead of just one pair, to represent the sets of possible strategies that may result during composition, even if the original interfaces had only a single strategy each. This option of using sets of pairs (ϕ, D) appears too complex, and we do not pursue it further here.

Restricted DAGs: Moore Outputs, Inputs, Non-moore Outputs. To simplify in order to avoid problems such as the one above, we may decide to restrict the I/O dependencies to a simpler form: $I = (X, Y, \phi, d)$ where $d \subseteq Y \times X$. d gives for each output the set of inputs it depends on. Those outputs that depend on no inputs are called Moore outputs. The game is played in 3 rounds: first the component chooses Moore outputs; then the environment chooses all inputs; then the component chooses non-Moore outputs.

This solves the problem of Example 5 because the second strategy, where the environment initially sets only x and then waits to observe y before setting u would be forbidden: both x, y should be set at the same time, since there is only one round to set all free inputs.

The problem with this approach is that feedback is non-commutative, as the following example illustrates.

Example 6. Let $I = (\{x_1, x_2, x_3\}, \{y_1, y_2\}, x_2 \neq y_2 \vee x_1 = y_1, \{(y_1, x_3), (y_2, x_3)\})$. In this example, both y_1, y_2 are non-Moore: x_3 is a "dummy" input that serves no other purpose except for providing dependencies for y_1, y_2 so that they are not Moore. The contract can also be read as $x_2 = y_2 \rightarrow x_1 = y_1$. Then, connecting y_1 to x_1 results in interface $\mathit{feedback}_{y_1 \rightsquigarrow x_1}(I)$ with contract $x_1 = y_1$. Following this, we can connect y_2 to x_2 to obtain the interface $\mathit{feedback}_{y_2 \rightsquigarrow x_2}(\mathit{feedback}_{y_1 \rightsquigarrow x_1}(I))$ with contract $x_1 = y_1 \wedge x_2 = y_2$. One would expect that if we do the same connections in the opposite order, i.e., first $y_2 \rightsquigarrow x_2$ and then $y_1 \rightsquigarrow x_1$, we should get the same result. But the contract of $\mathit{feedback}_{y_2 \rightsquigarrow x_2}(I)$ is **false**. Indeed, the following game is played here: first the environment chooses x_1, x_3 (there are no Moore outputs so their round is skipped); then the component chooses y_1, y_2; finally x_2 is set to y_2. The environment loses this game, since no matter what it picks for x_1, the component can always pick $y_1 \neq x_1$ and violate the contract of I, because of the feedback $x_2 = y_2$. □

Extracting Dependencies from Contracts. Note that there is an additional, mostly orthogonal complexity to the approach of extending interfaces with variable dependency information, and this has to do with where this information comes from. The simple approach is to expect the user to provide such information for atomic interfaces, and then compute it automatically for composite interfaces. This does not avoid the problems illustrated by the examples above.

Another approach is to try to extract dependencies automatically from the contract itself. For instance, if the contract is $y = x+1$, where x is the input and y is the output, then we could extract the dependency $x \rightarrow y$, i.e., y depends on x. This interpretation assumes that, first, the input x is given, and then the component computes the output y as $x + 1$. It is unclear, however, whether this interpretation is correct. An alternative interpretation is the following: first, an output y is chosen non-deterministically; then, the input x must be given, such that $x = y - 1$. Although the first interpretation may appear more natural, there is no reason why the second one should be considered invalid. This is more obvious in an interface with a slightly different contract, say, contract **true**. Here, as mentioned above, we should be able to distinguish the case where first the environment provides the input x and then the component replies with the output y, from the opposite case, where the component provides y first, and then expects x.

The problem of extracting variable dependencies from formulas is itself interesting, although it by itself does not resolve the issues raised by Examples 5 and 6. Let us briefly discuss the problem of extracting variable dependencies from formulas. Consider a formula ϕ on a set of variables V. We may assume no knowledge of "directionality" (input vs. output) for any of these variables, and seek to define a symmetric notion of dependency (see Table 1).

One idea is to define dependency based on the principle of "geometric orthogonality". Consider a ϕ over just two variables, say, x, y. Intuitively, x, y are independent in ϕ, if ϕ is a "rectangle", that is, if ϕ is equivalent to $(\exists x : \phi) \wedge (\exists y : \phi)$. For example, in both $x = y$ and $x \neq y$, x and y are dependent, whereas in $x = 0 \wedge y = 0$, they are independent, and so are they in $0 \leq x \leq 1 \wedge 0 \leq y \leq 1$.

Table 1. Dependency examples

ϕ	dependency
$x = y$	x, y dependent
$x \neq y$	x, y dependent
$x = 0 \wedge y = 0$	x, y independent
$0 \leq x \leq 1 \wedge 0 \leq y \leq 1$	x, y independent
$x = y \wedge y = z \wedge z = 0$	x, y independent
$x = y \wedge y = z$	x, y dependent
$x = y_1 \wedge x_2 = y$	x, y independent
$x < z \wedge z < y$	x, y dependent
$z < x \wedge z < y$	x, y independent ?

Let us try to generalize this idea to formulas with $n \geq 2$ variables. Let $\phi(\boldsymbol{y}, x_1, x_2)$ be over a set of variables $\boldsymbol{y} \cup \{x_1, x_2\}$. We can attempt the definition:

$$indep(\phi, x_1, x_2) \quad := \quad \phi \equiv \big((\exists x_1 : \phi) \wedge (\exists x_2 : \phi)\big)$$

Unfortunately this doesn't seem to work for the formula $x = y \wedge y = z$. In this case, we find that any pair of variables are independent according to the above definition. For instance, $\exists x : x = y \wedge y = z$ is $y = z$, and $\exists y : x = y \wedge y = z$ is $x = z$. This seems in contradiction with the fact that x, y are dependent in $x = y$, which is a weaker constraint than $x = y \wedge y = z$.

To capture the principle of geometric orthogonality, we may use the principle of *factorization*. Namely, if ϕ is over a set of variables X, we should be able to

find a partition of X into disjoint sets $X_1, ..., X_n$, and formulas $\phi_1, ..., \phi_n$, where ϕ_i is over X_i, such that ϕ is equivalent to $\bigwedge_{i=1}^{n} \phi_i$. Then, for given variables x and y, they are independent if they do not belong in the same set X_i in the partition. This is an interesting problem, although beyond the scope of this paper.

Non-preservation of Dependencies by Refinement. One fundamental problem with the conceptual requirements of feedback and refinement with regards to independence is the issue that an interface that has independent variables can have refinements in which the variables are dependent. Concretely, consider the case of the interface with the predicate

$$\phi(\bar{x}, \bar{y}) = \textbf{true}$$

This can always be written as a conjuction of functions on the individual variables, in particular because each component is similarly true.

$$\phi(\bar{x}, \bar{y}) = \phi_x^1 \wedge \phi_x^2 \wedge \ldots \phi_y^1 \wedge \phi_y^2 \wedge \ldots \quad \text{where } \phi_*^k = \textbf{true}$$

This would suggest that the interface is equivalent to a parallel composition of a series of ambivalent source and sink actors; they can be composed serially, etc... However, as in the above examples, clearly there are refinements of the original function that can introduce dependencies. If

$$\phi'(\bar{x}, \bar{y}) = x_k \neq y_j$$

refines ϕ, as it is shown to in the above example, the interface with independent variables that seems to be decomposable into parallel parts (that can be composed serially), can be refined into an interface that does not have any admissible definition for feedback.

Intuitively, there is a general sense behind this. Consider that one alteration to an interface to refine it is to take an input for which there are multiple satisfying outputs to choose from, non-determinism, and reduce the number of options it permits, all the way down to a unique option. If an input and output variable are independent, the choice of input has no effect on the choice of output. Consequently, if there are multiple possible outputs, for any given input, making the output depend on the input in a way that still permits a satisfiable choice legitimately refines the behavior even though it is no longer independent. That is, independence is not preserved by refinement.

3.4 Lifting to Powersets

Another idea is to treat contracts not as relations, but as functions. Feedback can be naturally defined on functions using fixpoint theory, so this appears to be a promising approach. Unfortunately, it is not as easy as it may appear to be at first.

To transform relations to functions, we can lift their domain and codomain to powersets. Specifically, let ϕ be a contract over I/O variable sets X and Y.

Then ϕ is semantically a relation $\phi \subseteq V(X) \times V(Y)$, where $V(X)$ denotes the set of valuations over X. ϕ defines a function

$$\tilde{\phi} : 2^{V(X)} \to 2^{V(Y)}$$

where, for $V_x \subseteq V(X)$, $\tilde{\phi}(V_x)$ is defined as follows:

$$\tilde{\phi}(V_x) := \{a_Y \in V(Y) \mid \exists a_X \in V_X : (a_X, a_Y) \in \phi\}.$$

$\tilde{\phi}$ is monotonic with respect to set inclusion, so it appears as if fixpoint theory can be readily applied. However, an element of $V(X)$ is an assignment over the entire set of input variables X, and an output of the function $\tilde{\phi}$ is an assignment over the entire set of output variables Y. As a result, it is unclear how to define feedback directly on $\tilde{\phi}$. For example, we may have $X = \{x_1, x_2\}$ and $Y = \{y\}$. In this case, the output of $\tilde{\phi}$ does not match its input, since there are two input variables, and only one output variable. But even when the number of input and output variables is the same, it is unclear how to define feedback of *individual* variables. For example, we may have $X = \{x_1, x_2\}$ and $Y = \{y_1, y_2\}$, and we may want to connect y_1 to x_2. It is not clear how to express this connection as a fixpoint operation on $\tilde{\phi}$.

We can attempt to define a feedback connection like the one above using projection and product functions, in addition to the $\tilde{\phi}$ function. For instance, a projection function could be used to extract an assignment over just y_1 from an assignment over $\{y_1, y_2\}$. This may solve the typing problems and allow to define a fixpoint that type checks. However, the above functions (including $\tilde{\phi}$, projection, and product) have the property that they return the empty set when given the empty set as input. As a result, the empty set would be a valid fixpoint of any composition of such functions. Moreover, the empty set would be the *least* fixpoint with respect to set inclusion, therefore the preferred solution chosen in typical fixpoint semantics approaches. Unfortunately, the empty set is not the value that one would expect as in particular it does not allow to capture serial composition.

3.5 Separating Input Assumptions

In the relational interface framework of [16], input assumptions and output guarantees are combined into a single contract ϕ. One idea is to separate them. Following this idea, an interface would be a quadruple

$$I = (X, Y, \phi, \psi)$$

where X is a finite set of input variables, Y is a finite set of output variables (as usual, we assume that $X \cap Y = \emptyset$), ϕ is a relation/predicate on X, and ψ is a relation/predicate on $X \cup Y$. ϕ captures the requirements on inputs only, while ψ is aimed at capturing guarantees on the outputs, with relation to inputs.

Ideally, we would like to have no redundancy between ϕ and ψ, for example, ψ should not impose more restrictions on the inputs than what ϕ imposes, as

for example in the case $\phi := x > 0$, $\psi := x > 1 \wedge y > x$. One way to achieve this which appears "canonical" is to attempt to enforce that ψ be *total* (i.e., input-complete), that is, to enforce $\mathbf{in}(\psi) \equiv \mathbf{true}$. As we shall see below, however, this requirement is not always compatible with our other desiderata.

Let us see first how the definitions of refinement and composition could be adapted to this setting.

Refinement. Refinement can be defined as follows. Let $I_i = (X, Y, \phi_i, \psi_i)$, for $i = 1, 2$. Then, I_2 refines I_1, written $I_2 \sqsubseteq I_1$, iff

$$\phi_1 \rightarrow \phi_2 \qquad \text{and} \qquad (\phi_1 \wedge \psi_2) \rightarrow \psi_1$$

are both valid formulas, i.e., equivalent to **true**.

With contract pairs, the refinement order gives a lattice, with top being the pair of contracts $(\mathbf{false}, \mathbf{true})$ and bottom being the pair $(\mathbf{true}, \mathbf{false})$.

Parallel Composition. Parallel composition of interfaces with contract pairs can be defined as usual, by taking the conjunction of their corresponding contracts. Let $I_i = (X_i, Y_i, \phi_i, \psi_i)$, for $i = 1, 2$, where all sets X_1, X_2, Y_1, Y_2 are pair-wise disjoint. Then

$$I_1 || I_2 := (X_1 \cup X_2, Y_1 \cup Y_2, \phi_1 \wedge \phi_2, \psi_1 \wedge \psi_2).$$

Note that $\phi_1 \wedge \phi_2$ is over $X_1 \cup X_2$, and $\psi_1 \wedge \psi_2$ is over $X_1 \cup X_2 \cup Y_1 \cup Y_2$.

Also note that

$$\begin{aligned} \mathbf{in}(\psi_1 \wedge \psi_2) &\equiv \exists Y_1, Y_2 : \psi_1 \wedge \psi_2 \qquad \equiv \exists Y_1 : (\psi_1 \wedge \exists Y_2 : \psi_2) \\ &\equiv (\exists Y_2 : \psi_2) \wedge (\exists Y_1 : \psi_1) \equiv \mathbf{in}(\psi_1) \wedge \mathbf{in}(\psi_2) \end{aligned}$$

Thus, if $\mathbf{in}(\psi_1) \equiv \mathbf{in}(\psi_2) \equiv \mathbf{true}$, we also have $\mathbf{in}(\psi_1 \wedge \psi_2) \equiv \mathbf{true}$, which means that parallel composition preserves our desiderata of no redundancy between ϕ and ψ.

Feedback. Feedback can be defined as follows. Let $I = (X, Y, \phi, \psi)$ and let $x \in X$ and $y \in Y$. Then connecting output y to input x yields the new interface

$$\mathit{feedback}_{y \rightsquigarrow x}(I) \quad := \quad (X - \{x\}, Y \cup \{x\}, \phi', \psi')$$

where

$$\phi' := \exists x : \Big(\phi \wedge \big(\forall Y, x : (\psi \wedge x = y) \rightarrow \phi \big) \Big)$$

and

$$\psi' := \psi \wedge x = y.$$

We can immediately see that ψ' is indeed a predicate over $Y \cup \{x\}$.

$\forall Y, x : (\psi \wedge x = y) \rightarrow \phi$ is a predicate over $X - \{x\}$. Therefore, ϕ' is equivalent to

$$(\exists x : \phi) \wedge \big(\forall Y, x : (\psi \wedge x = y) \rightarrow \phi \big).$$

and, since both conjuncts above are predicates over $X - \{x\}$, we can now see that ϕ' is a predicate over $X - \{x\}$. The idea behind the definition of ϕ' is to capture the notion of demonic non-determinism, as in the definition of serial composition, by strengthening the original input requirements ϕ with the additional term $\forall Y, x : (\psi \wedge x = y) \rightarrow \phi$.

Does this definition of feedback allow to reduce serial composition to parallel composition followed by feedback? This indeed appears to work on simple examples.

Example 7. Consider the interfaces defined in Example 1, $I_1 = (\{x\}, \{y\}, x \leq y)$, $I_2 = (\{y\}, \{z\}, y \neq 0)$, and their serial composition

$$I_1 \rightsquigarrow I_2 = (\{x\}, \{y, z\}, x \leq y \wedge y \neq 0 \wedge x > 0).$$

Transforming I_1 and I_2 into interfaces with pairs of contracts, we get $I_1' = (\{x\}, \{y\}, \mathbf{true}, x \leq y)$ and $I_2' = (\{y'\}, \{z\}, y' \neq 0, \mathbf{true})$, where we have also renamed y to y' in I_2'. We can now form the parallel composition of I_1' and I_2':

$$I_1' || I_2' = (\{x, y'\}, \{y, z\}, y' \neq 0, x \leq y)$$

and then connect y to y' in feedback, to obtain:

$$\mathit{feedback}_{y \rightsquigarrow y'}(I_1 || I_2) = (\{x\}, \{y, y', z\}, \phi', x \leq y \wedge y = y')$$

where

$$\phi' \quad := \quad (\exists y' : y' \neq 0) \wedge (\forall y, y', z : x \leq y \wedge y = y' \rightarrow y' \neq 0)$$

which is equivalent to $x > 0$. Therefore, after eliminating y' which is equal to y in $\mathit{feedback}_{y \rightsquigarrow y'}(I_1 || I_2)$, the latter simplifies to

$$\mathit{feedback}_{y \rightsquigarrow y'}(I_1 || I_2) = (\{x\}, \{y, z\}, x > 0, x \leq y)$$

which, as it can be seen, is the same as $I_1 \rightsquigarrow I_2$, except that the single contract is replaced by a contract pair. □

The above definition of feedback also appears to resolve the problem of non-preservation of refinement described in Example 2:

Example 8. Let $I = (\{x, z\}, \{y\}, \mathbf{true}, \mathbf{true})$ and $I' = (\{x, z\}, \{y\}, \mathbf{true}, x \neq y)$. Then $I' \sqsubseteq I$. Also, $\mathit{feedback}_{y \rightsquigarrow x}(I) = (\{z\}, \{y, x\}, \mathbf{true}, x = y)$ and $\mathit{feedback}_{y \rightsquigarrow x}(I') = (\{z\}, \{y, x\}, \mathbf{true}, \mathbf{false})$. As it can be seen, $\mathit{feedback}_{y \rightsquigarrow x}(I') \sqsubseteq \mathit{feedback}_{y \rightsquigarrow x}(I)$. □

Note that, as Example 8 demonstrates, the canonical form requirement for interfaces with contract pairs, namely that $\mathbf{in}(\psi)$ must be $\mathbf{true}$, is not generally preserved by feedback composition.

Summary. As it can be seen, interfaces with pairs of contracts seem to resolve several issues that exist in interfaces with single contracts. On the other hand, the interpretation of interfaces with contract pairs is unclear, and some new problems are introduced. First, regarding interpretation, it is unclear what the meaning of interfaces such as (**false**, **true**) and (**true**, **false**) is, and what the difference between the two is. The standard interpretation is that in (**false**, **true**) all inputs are illegal, and that in (**true**, **false**) all inputs are legal, but no output is produced. We find these interpretations unsatisfactory. What is the meaning of a "legal" input if no output can be produced? And why distinguish the contract (**false**, **true**) from, say, (**false**, $y \geq 0$)? After all, both accept no inputs, therefore, they cannot be used in any useful composition. In addition, interfaces with at least one contract being **false** do not seem to have valid implementations (say, by deterministic state machines).

Perhaps the most important problem with pairs of contracts is the fact that refinement does not preserve the "well-formedness" property that none of the two contracts be **false**, and therefore does not preserve implementability as discussed above. For instance, (**true**, **false**) refines every pair of contracts. This means that we could start from an implementable (well-formed) specification and reach a non-implementable one by successive refinements. This is clearly undesirable.

4 Refinement-Preserving Feedback

In this section a definition of feedback composition, derived from a set of desiderata, will be proposed for general relational interfaces. Given the interface

$$I := (\{\bar{u},\, \bar{v}\},\, \{\bar{x},\, \bar{y}\}, \phi)$$

where the barred variables indicate sequences of individual variables, and specifically, $\bar{x}$ is of the same length as $\bar{u}$, feedback composition will be defined

$$\mathit{feedback}_{\bar{x} \rightsquigarrow \bar{u}}(I) := (\{\bar{v}\},\, \{\bar{x},\, \bar{y}\}, \phi^*)$$

eliminating the set of inputs $\bar{u}$ from the signature of the interface. The definition of the new relation ϕ^* is what will be determined here from the desiderata. For brevity the interface will be expressed in terms of its relation, such as ϕ here for I. Sometimes the variables will be given with the relation with the input and output variables separated by a semicolon, as in $\phi(\bar{u},\, \bar{v};\, \bar{x},\, \bar{y})$. Also, unless otherwise noted, all interfaces will have this same general signature, and the asterisk, as in ϕ^* will indicate the feedback composition $\mathit{feedback}_{\bar{x} \rightsquigarrow \bar{u}}(I)$ (and its constituting relation).

For a relational interface characterized by a total function, the concept of a feedback connection between an output and an input can be defined straightforwardly by a fixed-point relation. Specifically, given a functional interface

$$f(\bar{u},\, \bar{v};\, \bar{x},\, \bar{y})$$

feedback composition can be defined as

$$f^*(\bar{v};\, \bar{x},\, \bar{y}) \leftrightarrow f(\bar{x},\, \bar{v};\, \bar{x},\, \bar{y}) \tag{1}$$

meaning that, for given input $\bar{v}$, $\bar{x}$ is an output of the feedback interface f^* iff $\bar{x}$ is a fixed-point of f.[3] A general definition for feedback over all relations therefore can be constrained to at least reduce to this one in the case of the relation being a graph of a total function; that is both input complete and deterministic. The formula 1 will then be the first desideratum.

A second qualification desirable for a feedback definition is that it preserves refinement relations in the same manner as serial and parallel composition. Given two interfaces ϕ and ψ with the same signature

$$\psi \sqsubseteq \phi \rightarrow \psi^* \sqsubseteq \phi^* \tag{2}$$

In other words, feedback composition is monotonic under the refinement order. This qualification is consistent with the first vacuously, since functional interfaces are minimal in the refinement order. The formula 2 will be the second desideratum.

A third qualification should then be given to determine the possible values taken on by the feedback edges of a feedback composition. The obvious possibility is to make the set of values some subset of the fixed-points of the relation between the connected output and input variables. In formal terms

$$\phi^*(\bar{v};\, \bar{x},\, \bar{y}) \rightarrow \phi(\bar{x},\, \bar{v};\, \bar{x},\, \bar{y}) \tag{3}$$

Here, the difference between the case for functional relations and general relations is that the feedback values for the former are exactly the fixed-points, whereas the latter could reject certain fixed points. Outside of fixed points, another possible choice for feedback values might be values that have some finite orbit through the relation. However, these points would have to be included in desideratum 1 for total functional relations, hence for consistency with this first qualification, this expanded set will not be considered. The formula 3 will be the third desideratum.

Using these three desiderata, building off of the third 3, a definition for feedback composition can be postulated. This definition will be of the form

$$\phi^*(\bar{v};\, \bar{x},\, \bar{y}) := \phi(\bar{x},\, \bar{v};\, \bar{x},\, \bar{y}) \wedge \textit{additional constraints} \tag{4}$$

where the *additional constraints* remove certain fixed-points. Clearly, based on 1, these constraints must reduce to **true** in the case of total functional relations. These constraints will be deduced in the following by considering particular cases of relations.

[3] Note that 1 says that *all* fixed-points of f must be solutions of the feedback f^*. This is different from the semantics of deterministic synchronous formalisms such as Esterel [2] or synchronous block diagrams [10], where feedback is defined by choosing from the set of fixed-points a unique representative, namely the *least* fixed-point. The least fixed-point solution relies on f being defined on an ordered set structure such as a complete partial order. We make no such assumption here.

4.1 Partiality

Consider first the case of the free inputs $\bar{v}$ in the formula being fixed to a particular value $\mathbf{a}$, and the remaining relation

$$\phi(\bar{u},\, \mathbf{a};\, \bar{x},\, \bar{y})$$

being *partial and deterministic* with respect to the input $\bar{u}$. Then, suppose the following relation is defined:

$$\psi(\bar{u},\, \mathbf{a};\, \bar{x},\, \bar{y}) := \phi(\bar{u},\, \mathbf{a};\, \bar{x},\, \bar{y}) \vee \left[\left(\neg \exists \bar{z}, \bar{w} :\, \phi(\bar{u},\, \mathbf{a};\, \bar{z},\, \bar{w})\right) \wedge \bar{u} = \bar{x} \wedge \bar{y} = \mathbf{b}\right]$$

where $\mathbf{b}$ is some arbitrary chosen constant tuple of the same length as $\bar{y}$. It can be verified that $\psi \sqsubseteq \phi$, intuitively, because ψ simply gives output values for inputs not in the domain of ϕ, thus refining it. Moreover, by the definition of ψ and the assumption that ϕ is deterministic, ψ returns a unique value for all input values. Thus, ψ is a total function. By desideratum 1, it follows that

$$\psi^*(\mathbf{a};\, \bar{x},\, \bar{y}) \leftrightarrow \psi(\bar{x},\, \mathbf{a};\, \bar{x},\, \bar{y}).$$

Using the above definition, along with 3, we obtain

$$\phi^*(\mathbf{a};\, \bar{x},\, \bar{y}) \rightarrow \psi^*(\mathbf{a};\, \bar{x},\, \bar{y}).$$

The only way the above and 2 can both be true is if

$$\phi^*(\mathbf{a};\, \bar{x},\, \bar{y}) := \mathbf{false}.$$

What can be concluded from this is that feedback composition over a relation that is partial with respect to the feedback input must exclude all fixed points. This can be accomplished by conjoining the following constraint to the definition for feedback composition

$$\forall \bar{u} :\, \exists \bar{x}, \bar{y} :\, \phi(\bar{u},\, \bar{v};\, \bar{x},\, \bar{y}).$$

4.2 Nondeterminism

Consider first the case of the free inputs $\bar{v}$ in the formula being fixed to a particular value $\mathbf{a}$, and the remaining relation

$$\phi(\bar{u},\, \mathbf{a};\, \bar{x},\, \bar{y})$$

being *total* or complete with respect to the input $\bar{u}$, but also being nondeterministic. Let

$$\hat{\phi}(\bar{u},\, \bar{v}) := \{(\bar{x},\, \bar{y}) \,|\, \phi(\bar{u},\, \bar{v};\, \bar{x},\, \bar{y})\}$$

denote the set of output values for an interface corresponding to the given input values.

Since ϕ is total, every refinement ψ will be total as well. More, it follows from the definition of refinement that

$$\hat{\psi}(\bar{u}, \bar{v}) \subseteq \hat{\phi}(\bar{u}, \bar{v}).$$

Suppose then that a particular relation ψ is defined such that $\hat{\psi}(\bar{u}, \bar{v})$ is a singleton for every set of input values. Moreover, if $\hat{\phi}(\bar{u}, \bar{v})$ contains more than one choice for the value of $\bar{x}$, the feedback output, let the choice of singleton value be the one where $\bar{x} \neq \bar{u}$; this is always possible if such a choice exists.

The above construction for ψ gives a refinement of ϕ for the above given reasons. This construction is also total functional, since a unique value was chosen for every input. By desideratum 1, the feedback values for $\bar{x}$ in ψ^* are exactly the fixed point solutions for $\bar{x}$ in ψ. However, if there are no such fixed point solutions for ψ, and consequently no feedback values for ψ^*, by desideratum 2, ϕ^* can have no feedback values for $\bar{x}$; that is, ϕ^* would have to be **false**. On the other hand, by the definition of refinement, for any set of inputs to a relation, if the set of outputs is unique, it must be unique in every refinement for the same inputs, thus the fixed point solutions that are also unique outputs for their corresponding inputs are preserved by refinement. The construction for ψ removes all of the other fixed point solutions from ϕ, arising from nondeterministic inputs.

What can be concluded from this is that the feedback composition, to be consistent with the desiderata, must be false unless there is at least one fixed point solution that is a unique output value for its corresponding input. This constraint can be formulated as the following term

$$\exists \bar{z} : \phi(\bar{z}, \bar{v}; \bar{z}, \bar{y}) \wedge (\forall \bar{w} : \phi(\bar{w}, \bar{v}; \bar{w}, \bar{y}) \rightarrow \bar{w} = \bar{z}) \tag{5}$$

which can be conjoined with the feedback composition definition. A simpler term to conjoin would be one that constrains the feedback values to only be deterministic ones; ones that are the unique outputs for their corresponding input values. This term would be

$$\forall \bar{z} : \phi(\bar{z}, \bar{v}; \bar{z}, \bar{y}) \rightarrow \bar{x} = \bar{z}. \tag{6}$$

From the perspective of the desiderata, this latter definition would make an unnecessary restriction, but nevertheless it would simplify the definition for feedback composition considerably. The decision to use the former or the latter would hinge on the presence of additional desiderata, perhaps regarding the preservation of feedback values by refinement; clearly, it is necessary that at least one feedback value should be preserved for others to exist, and the important question should be whether or not they all should be.

4.3 General Feedback Composition

Combining the above considerations and assembling the corresponding constraints, the following two definitions may be postulated for feedback composition on relational interfaces:

$$\begin{aligned}\phi^*(\bar{v};\, \bar{x},\, \bar{y}) := {} & \phi(\bar{x},\, \bar{v};\, \bar{x},\, \bar{y}) \\ & \wedge\, [\forall \bar{u} : \exists \bar{x}, \bar{y} : \phi(\bar{u},\, \bar{v};\, \bar{x},\, \bar{y})] \\ & \wedge\, [\exists \bar{z} : \phi(\bar{z},\, \bar{v};\, \bar{z},\, \bar{y}) \wedge (\forall \bar{w} : \phi(\bar{w},\, \bar{v};\, \bar{w},\, \bar{y}) \rightarrow \bar{w} = \bar{z})]\end{aligned} \tag{7}$$

$$\begin{aligned}\phi^*(\bar{v};\, \bar{x},\, \bar{y}) := {} & \phi(\bar{x},\, \bar{v};\, \bar{x},\, \bar{y}) \\ & \wedge\, [\forall \bar{u} : \exists \bar{x}, \bar{y} : \phi(\bar{u},\, \bar{v};\, \bar{x},\, \bar{y})] \\ & \wedge\, [\forall \bar{z} : \phi(\bar{z},\, \bar{v};\, \bar{z},\, \bar{y}) \rightarrow \bar{x} = \bar{z}]\end{aligned} \tag{8}$$

The full ramifications of these definitions of feedback warrant much further investigation. The property that the two definitions reduce to only the first term, the fixed-point relation, in the case of total functional relations means that at least, for the subclass of total functional relations (or simply functions), this definition is consistent with the usual notion of feedback in deterministic synchronous models of computation.

As an example, the above definitions both applied to the one input, one output **true** interface result in the one output **false** interface, consistent with the earlier example demonstrating that this must be the case. Suppose, instead, that a relation were defined

$$\phi(x;\, y) := x = 5 \rightarrow y = 5$$

which is similar to **true**, except on the input $x := 5$, which must be mapped deterministically to $y := 5$. Then, under the two definitions of feedback, the corresponding compositions would be

$$\phi^*(\cdot;\, y) := \mathbf{true}$$

and

$$\phi^*(\cdot;\, y) := y = 5$$

In both cases, the relation $x \neq y$ is not a refinement. Indeed, every refinement of both must at least include 5 as a feedback output value.

5 Conclusions

The definition of feedback composition in the context of synchronous relational interfaces has been investigated. Challenges were described in Section 3, and a systematic derivation of novel alternatives was proposed in Section 4. Future work includes a more thorough study of these new alternatives in the context of the full theory presented in [16], as well as in the context of recent work on *error-completion* [17]. In addition, it would be interesting to examine how the difficulties in defining feedback in synchronous interfaces are related to the problem of compositionality of relational semantics of non-deterministic dataflow networks and the so-called *Brock-Ackerman anomaly* [6,12].

References

1. Bensalem, S., Bozga, M., Nguyen, T.-H., Sifakis, J.: D-finder: A tool for compositional deadlock detection and verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 614–619. Springer, Heidelberg (2009)
2. Berry, G.: The foundations of Esterel, pp. 425–454. MIT Press (2000)
3. Bliudze, S., Sifakis, J.: The algebra of connectors: structuring interaction in bip. In: EMSOFT 2007, pp. 11–20. ACM (2007)
4. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 508–522. Springer, Heidelberg (2008)
5. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. Distributed Computing 25(5), 383–409 (2012)
6. Brock, J.D., Ackerman, W.B.: Scenarios: A model of non-determinate computation. In: Díaz, J., Ramos, I. (eds.) Formalization of Programming Concepts. LNCS, vol. 107, pp. 252–259. Springer, Heidelberg (1981)
7. de Alfaro, L., Henzinger, T.: Interface automata. In: Foundations of Software Engineering (FSE). ACM Press (2001)
8. de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 148–165. Springer, Heidelberg (2001)
9. Doyen, L., Henzinger, T., Jobstmann, B., Petrov, T.: Interface theories with component reuse. In: EMSOFT 2008, pp. 79–88 (2008)
10. Edwards, S., Lee, E.: The semantics and execution of a synchronous block-diagram language. Science of Computer Programming 48, 21–42 (2003)
11. Gössler, G., Sifakis, J.: Composition for component-based modeling. Science of Computer Programming 55(1), 161–183 (2005)
12. Jonsson, B.: A fully abstract trace model for dataflow and asynchronous networks. Distributed Computing 7(4), 197–212 (1994)
13. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S., Probst, D.: Property preserving abstractions for the verification of concurrent systems. Formal Methods in System Design 6(1), 11–44 (1995)
14. Poulhiès, M., Pulou, J., Rippert, C., Sifakis, J.: A methodology and supporting tools for the development of component-based embedded systems. In: Kordon, F., Sokolsky, O. (eds.) Monterey Workshop 2006. LNCS, vol. 4888, pp. 75–96. Springer, Heidelberg (2007)
15. Sifakis, J., Yovine, S.: Compositional specification of timed systems. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046, pp. 345–359. Springer, Heidelberg (1996)
16. Tripakis, S., Lickly, B., Henzinger, T.A., Lee, E.A.: A theory of synchronous relational interfaces. ACM TOPLAS 33(4) (July 2011)
17. Tripakis, S., Stergiou, C., Broy, M., Lee, E.A.: Error-Completion in Interface Theories. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 358–375. Springer, Heidelberg (2013)

Reasoning about Network Topologies in Space

Lenore D. Zuck[1] and Kenneth L. McMillan[2]

[1] University of Illinois at Chicago, USA
lenore@cs.uic.edu
[2] MSR Redmond, WA, USA
kenmcmil@microsoft.com

Abstract. "Traditional" satellite systems consist of special-purpose monolithic satellites. Future ones aim to comprise of a small number of inexpensive general-purpose spacecraft that communicate with one another to carry out missions, with a certification requirement. Such certification would guarantee the security and correctness of all mission software.

In this work we focus on proving correctness of a proposed protocol for aggregation of the data of member nodes in such a system. The modeling and verification of such a system is complicated by a number of factors, including real-time constraints and the unusual topology of the network, which does not fit well-studied cases such as clique, star and ring topologies.

We show how to use decomposition and abstraction to isolate the topology-dependent reasoning in the the proof into a simple lemma. This allows us to use finite-state model checking techniques to perform this reasoning, and to quickly assess classes of network topologies. The assumptions we made in abstracting the model (the premises of our lemma) can in principle be verified locally, without concern for the network topology.

This case study can be seen as an instance of a general proof strategy: separate the complicating aspects of the proof of a complex system so that each can be handled by an appropriate tool.

1 Introduction

"Traditional" satellite systems consist of special-purpose monolithic satellites. Future ones aim to comprise of a small number of inexpensive general-purpose spacecrafts that communicate with one another to carry out missions. The success of this futuristic satellite architecture is predicated on the reliability, fault tolerance, security, and timeliness properties of the software facilitating computation and communication within a cluster. The inability to handle transient faults, failure to maintain data confidentiality or integrity, or excessive propagation delays would immediately nullify the cost and flexibility benefits that these proposed systems have over more traditional monolithic architectures. Because of the potential sensitivity of data in the system, correctness is a major concern and software need to be certified.

Such certification implies verification. Here we focus on one of the tasks that such systems will routinely perform: aggregation of the data of member nodes a system by a designated "root" node. We use one of the protocols that has been proposed to accomplish this data aggregation. Our approach is an instance of a more general strategy:

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 267–277, 2014.

separate the complicating aspects of the proof of a complex system so that each can handled by an appropriate tool. In this case, the aspect we wish to isolate is the topology of the network.

The challenges to verifying such protocols include the size of the system, the number of possibly topologies, the presence of faults, the real time constraints imposed by the communication protocol, etc. We show that many of these issues can be abstracted away so that we may focus on the topology-related reason in a very simple model. Other aspects may then be dealt with locally, without the complication of reasoning about parameterized networks. Correctness can be proven by model checking techniques to system of a reasonable size (in the tens of nodes). This is on the scale of what can be expected in a real deployment.

The protocol at hand supports sense/compute/store/forward missions, in which physical phenomena are detected by nodes with sensory capabilities, post-processed or aggregated by nodes with computational capabilities, and forwarded to nodes with dedicated storage or downlinks to ground stations. It facilitates secure, reliable, in-network aggregation over a logical topology of communicating nodes. The correct aggregation of values from sensors to a "root" node in a cluster is an abstraction of the larger goal of communication protocols that share data and use the cluster to provide fault tolerance and savings in the constrained resources.

Our main conclusion that even for a rather complex system, the topology-related reasoning is simple and can be accomplished with automated techniques. Although we do not prove correctness for arbitrary-size networks, we can provide a guarantee that is sufficient for a real deployment of a system of fixed size.

A Specific Protocol and Its Expected Properties

We choose to focus on the *Secure Ride Sharing* protocol (SRS) of Lee and Mossé [6,9] that builds on the Ride Sharing protocol of [7], augmenting it with (additively homomorphic) encryption [3] to guarantee its security properties, which we assume are correct, and fixing some minor bugs. The assumption of correct cryptography allows us to focus on the communication, rather than the computational, aspects of the protocol. Our description of the protocol derives from [6].

SRS is a TDMA-based protocol that uses the redundancy present in the satellite cluster in order to provide for fault tolerance. TDMA (time division multiple access) is a communication mechanism that guarantee collision-free transmission of data by scheduling each satellite to transmit during its own prescribed time slots. This allows for savings in energy since there is no need for collision detection/retransmission, and individual satellites are free to power off their radios during periods in which they are not expected to transmit or receive. The deterministic nature of TDMA can also help facilitate protocols requiring real-time response. We assume for simplicity of description that satellites have omnidirectional antennas and that the connectivity of these satellites can be extracted from the cluster at the time of cluster formation. This omnidirectionality provides a multicast medium, which is inherently redundant and fault tolerant, without having to carry out extra transmissions.

The protocol should guarantee that the root node correctly aggregates the values sent by the other nodes (that could be acting as sensors or as computing agents) even in

the face of link failures. Moreover, no value should be received by the root more than once, and no node should be able to obtain the (secret) value contributed by any other node. The protocol should further be efficient in terms of memory, time, power, and communication cost, as well as fault tolerant to link failures.

At first glance the verification of this protocol seems well beyond state-of-the art verification lore. It allows for an arbitrary number of nodes to communicate over a network whose topology is determined at cluster formation time. The number of nodes is in the tens. Considering all allowable topologies of such a network rules out many, if not all, approaches to such verification (none of the "parameterized verification" approaches can handle properties that depend in non-trivial network topology constraints). Moreover, the number of nodes is not static—nodes leave and join clusters—and, consequently, the topologies are not fixed. In addition, there are real-time constraints on the protocol, adding to the modeling and verification complexity.

We will show, however, that the topology-dependent reasoning required to prove correct data aggregation is actually quite simple, provided we abstract away other aspects of the protocol such as real time and operations on data. This makes it possible to carry out this reasoning in an automated way exploiting the technique of Bounded Model Checking [2]. In this way, we can provide a correctness guarantee that is sufficient for satellite clusters of realistic size.

2 The Secure RideSharing Protocol (SRS)

The Secure RideSharing protocol (SRS) [9] is the secure successor of the RideSharing fault-tolerant in-network aggregation protocol originally proposed in [7]. This protocol provides a secure, fault-tolerant, and energy-efficient means of aggregating values detected by nodes within a hierarchical network topology. It further enables networks to reap the bandwidth and power-consumption benefits of in-network aggregation protocols without exposing the confidential data to outside observers or compromised nodes within the network itself.

Algorithm 1 describes in pseudo code the functionality of a single node in the network, and Algorithm 2 is the pseudo code for the root.

Just like its predecessor, SRS exploits the inherent redundancy of shared wireless medium to detect and correct communication errors. Nodes are organized in a *track* graph [5] as shown in Fig. 1, where the aggregation path forms a DAG with multiple paths through the track graph rather than a simple spanning tree. In Fig. 1, a directed edge from a child C to a parent P_1 indicates that P_1 and C are within range of each other and that P_1 is able to hear C's messages. Edges are *primary*, *backup*, or *side*; primary and backup edges are between adjacent tracks, and side edges are within the same track. Each sensor node has one primary edge (leading to its primary parent) and several backup edges. The primary edges form a spanning tree and are used to propagate data as long as no communication errors occur.

Every sensor node transmits its (encrypted) value according to a predefined TDMA (Time Division Multiple Access) schedule. With the use of omnidirectional antennas, and the assumption that a child has at least one non-failing parent link, we can guarantee that one or more receive each message. Hence, if the primary link fails, one of

Algorithm 1. Aggregation algorithm run by sensors within the network

```
input : PC, BC, SP, v
A := 0;
P := 0̄;
L.r := 0̄;
L.e := 0̄;
if v NOT NULL then                                          // Aggregate own value
    A := A + v + g_ID(k_ID) mod M;
    P[ID] := 1;
end
L := rcvL(SP);
foreach Child C in PC ∪ BC do
    if rcv(A_c,P_c) from Child C then
        if C ∈ PC OR (C ∈ BC AND L[C].e = 1 AND L[C].r = 0) then   // Aggregate received
            A := A + A_C mod M;
            P := P OR P_c;
        end
    end
    else                                                    // Propagate the error signal
        L[C].e := 1;
    end
end
Transmit(A,P,L);
```

Algorithm 2. Final aggregation and decryption algorithm used by the root

```
input : PC
output: FinalA
A := 0;
P := 0̄;
K := 0;
FinalA := 0;
foreach Child C in PC do
    if rcv(A_c,P_c) from Child C then
        A := A + A_C mod M;
        P := P OR P_c;
    end
end
foreach bit set to '1' in P do
    K := K + g_i(k_i) mod M;
end
FinalA := A − K mod M;
```

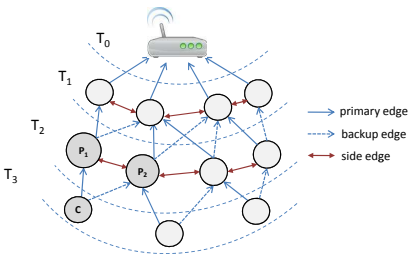


Fig. 1. Track Topology

the backup parents can aggregate the child's value (justifying the name "ridesharing"). Obviously, every value should be aggregated at most once, and the backup parents need to coordinate to ensure that this is the case.

Assume that node n has a primary parent p and backup ordered parents $b_1, \ldots, b_k$. Each parent maintains an L-vector with 2 bits corresponding to n, $L.e[n]$ and $L.r[n]$. When p receives a message from n, it sets $L.r[n]$. If an error occurs in the primary edge, p sets $L.e[n]$. Every parent attaches its L-vector to each message it sends. The first (in order above) backup parents b_i to receive an L-vector from p with $L.e[n] = 1$, aggregates this value and informs its neighbors that it had done so (by sending them its own L vector with $L.r[n]$ set).

As an example, Fig. 1 shows a node C in track T_3 with two parents, P_1 and P_2, in track T_2, where P_1 is primary and P_2 backup. Assuming no error, both P_1 and P_2 receive C's value, but only P_1 aggregates it. Now, assume a link error only in the primary edge. P_2 will receive the bit vector of P_1 over the side edge $P_1 \leftrightarrow P_2$, detect that C is missing, and correct the error by aggregating C's value into its own.

Each node also maintains a *Partaking-Vector* (P vector) that is used to keep track of which nodes successfully contributed their values in the final aggregate. Each node i is responsible for aggregating its own (encrypted) value into its local aggregate and setting $P[i]$ vector to '1'. The node then piggybacks both the L and the P vectors on the encrypted aggregate and sends them to all its parents. The aggregation of the P vectors is a bitwise OR operation of all the P vectors.

3 Modeling and Verification

We now consider the problem of modeling and formally verifying the protocol. The property of the protocol we would like to prove is a simple safety property. That is, upon termination of a TDMA cycle, assuming some upper bound on the number of link failures, the root node contains the the aggregated data from all nodes (and moreover each value is aggregated exactly once).

3.1 Modeling and Verification Approach

There are several challenges inherent in the modeling and verification. Our protocol is a real-time, parameterized system. That is, it consists of an arbitrary number of similar processes, and each must meet a real-time deadline for the system to carry out its function (that is, it must transmit its data to all of its parents before the end of its TDMA slot). Moreover, the protocol processes non-finite data, using cryptographic primitives. All three of these aspects present challenges for modeling and verification.

In addition, the correctness of the protcol depends on the interconnection topology. This topology is unique to the protocol and does not fit any well-studied class (for example, ring, star or clique).

Given these difficulties, one of two approaches has been taken in the past. The first is to give up on generic verification of the protocol and to verify only fixed finite configurations [6]. In this approach, random legal topologies were generated and verified at an abstract level using the timed finite-state model checker UPPAAL [1]. This is unsatisfactory in that we may not know the exact topology in advance. The other approach is to verify the protocol generically using a manually generated inductive invariant as in [9]. This manual process may solve the problem, but presents the difficult problem of

finding the invariant and performing the invariant checking using an automated theorem prover.

Here, we consider a third alternative, and perform some simple experiments to provide evidence for its feasibility. That is, we use abstraction and decomposition techniques to separate concerns in the verification process. When reasoning about the protocol as a whole, we can make assumptions to be discharged locally. For example, we can simply assume that each process meets its real-time deadline. This allows us to treat the protocol as an untimed synchronous system. The real-time proof obligation can be discharged locally by considering a single process. Thus we have decoupled the questions of topology and parametricity from the real-time aspect of the problem.

Further, at the protocol level we can abstract away the precise operations of aggregation and ecryption. We need only be concerned with the P vector that determines the subset of node data values that have been aggregated and the L vector that determines which values still need to be aggregated. The actual data correctness is again an assumption that can be discharged locally.

Having made these assumptions, we are left with an abstract model consisting of a network of nodes whoses edges meet a topology constraint. Since there are no real-time constraints, we can model time in the TDMA cycle by an integer counter.

At this point, we make one further assumption to allow us to perform the safety verification automatically: we assume the number of nodes is bounded by a fixed constant N. This assumption has two effects. It makes the model finite-state, and it makes the safety property a *bounded* safety property. That is, because the number of steps in a TDMA cycle is now bounded by N, it is sufficient to prove that the safety condition holds for only a bounded number steps. The result is that for any fixed N, we can check safety of the abstract model using bounded model checking (BMC) reducing the verification problme to a Boolean satisfiability (SAT) problem.

In a typical networking verification problem, this solution might be considered inadequate, as no bound N can be determined in advance. However, in the satellite domain, we might reasonably put a moderate upper bound on the number of nodes in the network (given the cost of launching satellites).

This approach provides some considerable advantages. We can verify the protocol quickly for arbitrary topologies meeting our topology constraint. Moreover, we can experiment with different topology classes and transmission error assumptions simply by changing the constraints on the model. Of course, we must ultimately verify our modeling assumptions (that is, that nodes aggregate data correctly and in a timely manner). However, we can do this without concern for the size or topology of the network.

3.2 The Abstract Model

We used Cadence SMV to specify and verify the abstract model. The model has three fixed parameters: the number of nodes `NODES`, the number of parents of each node `PARENTS` and the number of tracks `TRACKS`. The model has three immutable variables describing the the network topology:

1. An array `track` that assigns a track number to each node,
2. An array `parent` that gives the set of parents of each node.
3. An array `side` that gives the side channel connection of each node.

The state components of the abstract model are:

1. The TDMA frame counter `slot`,
2. A non-deterministic array `fail` indicating the set of links that fail at the current time,
3. An array `P` giving the P vector of each node.
4. An array `L` giving the L vector of each node.

To simplify the model, the nodes are indexed by their TDMA slots. That is, we assume that every node has exactly one slot in the TDMA schedule. The node of highest index is the root node. We wish to prove that each node's data is aggregated along one and only one path to the root. Using a standard decomposition trick, we choose a representive node and prove the property for this node only. This allows us to reduce the partaking vector to a single bit. The model computes a boolean array `aggregate` the indicates which nodes are aggregating the representative's data in the current TDMA slot. This array is used to update the P vectors.

Further, to simplify the protocol description, we will encode the L vector with a single bit that is true if the representative's data has already been aggregated by some parent (thus further aggregation should be inhibited). Further refinement of the model would be needed to compute this information using the actual L vector.

The resulting Cadence SMV model is shown in Figure 2. The linear temporal logic properties we wish to prove of this model are (translated into more familiar logical notation):

$$G\,(\text{slot} = \text{ROOT} \Rightarrow \text{P[ROOT]})$$
$$G\,\neg \wedge_{i \in \text{NODE}}\,(\text{aggregate}[i] \wedge \text{P}[i])$$

The first of these says that the representative's data must reach the root, while the second says it must never be aggregated twice in the same location.

Of course, this property will not be true if we do not make some assumptions about the network topology, the TDMA schedule and the occurrence of link failures. In the protocol design the network is a layered DAG in which each layer is connected by side channels. The side channel connections are defined by:

```
side[i] := (i < NODES-1 & track[i+1] = track[i]) ? i+1 : 0;
```

That is, we assume a side channel from each node to the next scheduled node in its track. Further, we require that all parents of a node in track t must be in track $t+1$:

$$\wedge_{i=0\ldots\text{NODES}-2}\,\wedge_{j \in \text{Parent}}\,(\text{parent}[i][j] \neq 0 \Rightarrow \text{track}[\text{parent}[i][j]] = \text{track}[i] + 1)$$

(note we use zero as a null value in the parent vector). We require that the TDMA schedule respect the track order:

$$\wedge_{i=0\ldots\text{NODES}-2}(\text{track}[i+1] \geq \text{track}[i])$$

Finally, we assume that each node succeeds in transmitting to at least one parent:

$$G\ \vee_{i \in \text{Parent}}\,(\text{parent}[\text{slot}][i] \neq 0 \wedge \neg\text{fail}[i])$$

With these assumptions on the network, we can prove the required safety property.

```
typedef Node 0..(NODES-1);
typedef Parent  0..(PARENTS-1);

module main(){

  /* Model of the network topology */
  parent : array Node of array Parent of Node;
  track : array Node of Track;
  side : array Node of Node;

  next(parent) := parent;
  next(track) := track;

  /* Model of the partaking vectors */
  P : array Node of boolean;
  representative : Node;
  forall(i in Node)
    init(P[i]) := i = representative;

  /* Model of the L vectors */
  L : array Node of boolean;
  forall(i in Node)
    init(L[i]) := 0;

  /* The slot counter counts up to NODES-1 */
  slot : Node;
  init(slot) := 0;
  next(slot) := (slot < NODES - 1) ? slot + 1 : slot;

  /* This models non-deterministic link failures */
  fail : array Parent of boolean;

  /* The update function for the P vectors */
  aggregate : array Node of boolean;
  forall(j in Node)
    aggregate[j] :=
      |[P[slot] & !fail[i] & parent[slot][i] = j : i in Parent]
       & !L[slot] & j!=0 & slot != NODES-1;

  next(P) := P | aggregate;

  /* The update function for the L vectors */
  if(L[slot] | P[slot])
    next(L[side[slot]]) := 1;

}
```

Fig. 2. Abstract network model in Cadence SMV language

3.3 Verification Performance

With the given topology constraint, for fixed values of the parameters, Cadence SMV can verify our safety specification using bounded model checking and a SAT solver. Bounded verification is complete in this case because the property being checked is a bounded-time property.

Figure 3 shows the run time of the solver as we increase the parameter NODES from 4 to 20 with TRACKS $= 4$. One line shows the case PARENTS $= 3$, while the other shows PARENTS $= 4$. We observe that the run time is inceasing exponentially with the number of nodes. However, because of the simplicity of the abstract model, we can handle a network of at least 20 nodes in moderate time. Provided the actual deployed network is of less than this size, no further guarantee is needed. Failing this, we would have to apply parameterized methods or some additional reduction to the abstract model. Because the abstract model is fairly simple, it is possible that an inductive invariant proving it could be derived with an acceptable level of effort.

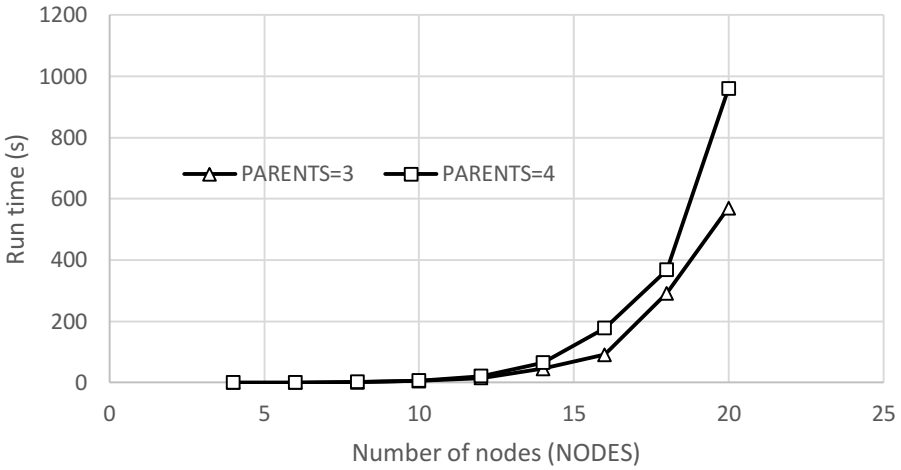


Fig. 3. Verification performance

We have made the assumption that each node has one slot in the TDMA cycle. This assumption is easily removed, however. We can, for example, interleave arbitrary nodes into the schedule without violating safety. This adds one further fixed parameter to the model: the length of the TDMA schedule. Since this parameter determines the BMC depth, we cannot verify arbitrarily large values of this parameter. However, due to energy considerations, the number of TDMA slots will not in practice be significantly greater than the number of nodes.

3.4 Refinement Verification

Having verified an abstract model of the protocol, we can proceed to verify increasingly detailed models of the protocol using refinement techniques, as in [4]. We can refine the model, for example, by adding introducing the actual data aggregation operations,

by adding details of message transmisison, and so on. In verifying these refinements, however, we need not consider the topology of the network. Thus, by abstracting the model, we have effectively isolated the topology-dependent reasoning.

4 Conclusion

Much of the existing research on automated verification of networks of processes has focused on symmetric connection topologies, such as ring, star and clique topologies. When faced with a network whose correctness depends on a more specific topology, we may find these methods inapplicable. In this paper, we have considered such a case: a proposed protocol for aggregation of data in networks of earth-orbiting satellites. For reasons of physical distribution, the network has an unusual shape: a layered DAG with added ring connections.

While we have not performed a full formal verification of this protocol in any detail, we observed that the topology-dependent reasoning in its proof can be isolated into a simple lemma that can be discharged by finite-state methods for networks of realistic size. In effect, we succeeded in enumerating all allowable network topologies up to a size bound in proving this simple property. The lemma we proved should make it possible to carry on the verification process without further reasoning about parameterized networks of processes.

With this approach we can prove a property of correct aggregation under assumptions about transmission failures. However, we do not obtain any quantitative information about reliability. This suggests a challenge problem: to prove probabilistic bounds on correct aggregation for all allowable topologies. This is a bounded probabilistic model checking problem (but not a statistical model checking problem because of the quantifier over topologies). In principle tools such as PRISM [8] can handle such problems, but in practice this might present a considerable challenge.

Our case study provides and example a general proof strategy: separate the complicating aspects of the proof of a complex system so that each can be handled by an appropriate tool. We have seen that this approach can allow us to apply model checking techniques to networks of processes whose correctness depends on network topology.

Acknolwedgements. We thank Adam Lee and Daniel Mossè for introducing us to the protocol and for explaining to us some of its finer details. This research was supported in part by DARPA contract NNA11AB36C.

References

1. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
2. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
3. Castelluccia, C., Chan, A.C.-F., Mykletun, E., Tsudik, G.: Efficient and provably secure aggregation of encrypted data in wireless sensor networks, pp. 1–36

4. Eiríksson, Á.T.: The formal design of 1M-gate ASICs. Form. Methods Syst. Des. 16(1), 7–22 (2000)
5. Felsner, S., Liotta, G., Wismath, S.K.: Straight-line drawings on restricted integer grids in two and three dimensions. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) GD 2001. LNCS, vol. 2265, pp. 328–342. Springer, Heidelberg (2002)
6. Feo-Arenis, S., Iskander, M.K., Lee, A.J., Mossé, D., Zuck, L.D.: Verifying protocols for f6. Inernal document, available upon request (November 2012)
7. Gobriel, S., Khattab, S., Mossé, D., Brustoloni, J., Melhem, R.: Ridesharing: Fault tolerant aggregation in sensor networks using corrective actions. In: The 3rd Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON), pp. 595–604 (2006)
8. Kwiatkowska, M.Z., Norman, G., Parker, D.: Probabilistic symbolic model checking with prism: a hybrid approach. STTT 6(2), 128–142 (2004)
9. Lee, A.J., Iskander, M.K., Mossé, D.: Confidentiality-preserving and fault-tolerant in-network aggregation for collaborative wsns. In: Proceedings of the 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom (October 2012)

Author Index