Marco Carbone
Jean-Marc Petit (Eds.)

LNCS 7176

Web Services and Formal Methods

8th International Workshop, WS-FM 2011
Clermont-Ferrand, France, September 2011
Revised Selected Papers

Springer

Lecture Notes in Computer Science 7176

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

Marco Carbone Jean-Marc Petit (Eds.)

Web Services and Formal Methods

8th International Workshop, WS-FM 2011
Clermont-Ferrand, France, September 1-2, 2011
Revised Selected Papers

Springer

Volume Editors

Marco Carbone
IT University of Copenhagen
Rued Langgaards Vej 7, 2300 Copenhagen, Denmark
E-mail: carbonem@itu.dk

Jean-Marc Petit
Université de Lyon – CNRS INSA Lyon
LIRIS
7 avenue Jean Capelle, 69621 Villeurbanne Cedex, France
E-mail: jean-marc.petit@insa-lyon.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-29833-2 e-ISBN 978-3-642-29834-9
DOI 10.1007/978-3-642-29834-9
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012936044

CR Subject Classification (1998): H.4, H.3, D.2, K.6, H.5

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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)

Preface

We are pleased to present the proceedings of the 8th International Workshop on Web Services and Formal Methods (WS-FM 2011), held in Clermont-Ferrand, France during September 1–2, 2011, and co-located with the 9th International Conference on Business Process Management (BPM 2011). The aim of the WS-FM workshop series is to bring together researchers working on service-oriented computing (SOC), cloud computing and formal methods in order to catalyze fruitful collaborations. The scope of the workshop is not limited to technological aspects: In fact, the WS-FM series have a strong tradition of attracting submissions on formal approaches to enterprise systems modeling in general, and business process modeling in particular. Potentially, this could have a significant impact on the on-going standardization efforts for SOC and cloud computing technologies.

SOC provides standard mechanisms and protocols for describing, locating and invoking services over the Internet. Although there are existing SOC infrastructures supporting specification of service interfaces, access policies, behaviors and compositions, there are still many active research areas in SOC such as management of interactions with stateful and long-running services, farms of services and quality of services. Moreover, the emerging paradigm of cloud computing provides a new platform for service delivery, enabling the development of services that are configurable based on client requirements, service level guarantee mechanisms, and extended services based on virtualization. The convergence of SOC and cloud computing is accelerating the adoption of both of these technologies, making the service dependability and trustworthiness a crucial and urgent problem. In this research area, formal methods can play a fundamental role. They can help us define unambiguous semantics for the languages and protocols that underpin existing Web service infrastructures, and they provide a basis for checking the conformance and compliance of bundled services. They can also empower dynamic discovery and binding with compatibility checks against behavioral properties and quality of service requirements. Formal analysis of security properties and performance is also essential in cloud computing and in application areas including e-science, e-commerce, business process management, etc. Moreover, the challenges raised by this new area can offer opportunities for extending the state of the art in formal techniques.

In this edition of the workshop, we received 14 submissions and each of them was reviewed by at least three members of the Program Committee. We decided to accept nine papers. We wish to express our gratitude to all authors of submitted papers, the Program Committee members and the additional reviewers for their efforts in evaluating the papers. We are also very grateful to the two world-class keynote speakers (Kohei Honda, Queen Mary University of London, UK, and Hassan Aït-Kaci, IBM Canada) who gave us two wonderful talks. We also

thank the local Organizing Committee, chaired by Farouk Toumani, for making the practical arrangements for the workshop. Last but not least, we wish to thank Andrei Voronkov, who allowed us to use the free conference software system EasyChair for carrying out the reviewing process of WS-FM 2011.

January 2012

Marco Carbone
Jean-Marc Petit

Organization

Program Committee Co-chairs

Marco Carbone	IT University of Copenhagen, Denmark
Jean-Marc Petit	University of Lyon/CNRS, France

Program Committee

Karthikeyan Bhargavan	INRIA, France
Maria Grazia Buscemi	IMT Lucca, Italy
Marco Carbone	IT University of Copenhagen, Denmark
Florian Daniel	Università di Trento, Italy
Pierre-Malo Daniélou	Imperial College London, UK
Giuseppe De Giacomo	SAPIENZA Università di Roma, Italy
Rocco De Nicola	Università di Firenze, Italy
Marlon Dumas	University of Tartu, Estonia
José Luiz Fiadeiro	University of Leicester, UK
Xiang Fu	Hofstra University, USA
Serge Haddad	ENS Cachan, France
Sylvain Hallé	Université du Québec à Chicoutimi, Canada
Thomas Hildebrandt	IT University of Copenhagen, Denmark
Manuel Mazzara	Newcastle University, UK
Luca Padovani	Università di Torino, Italy
Jean-Marc Petit	University of Lyon/CNRS, France
Steve Ross-Talbot	Pi4Tech, UK
Hagen Voelzer	IBM Research, Switzerland
Nobuko Yoshida	Imperial College London, UK
Fatiha Zaidi	CNRS, France
Gianluigi Zavattaro	Università di Bologna, Italy

Additional Reviewers

M. Bartoletti	R. Hu	P. Poizat
V. Bono	H. Melgratti	F. Tiezzi
S. Dal Zilio	A. Mukhamedov	
L. Fossati	A. Pironti	

Table of Contents

Understanding Distributed Services through the π-Calculus

Kohei Honda

Queen Mary, University of London, UK
kohei@eecs.qmul.ac.uk

Computing started from a simple but powerful abstract machine, Turing Machine, and theories of functions. On this basis we built core principles of hardware, programming languages and systems software (such as compilers and OSes). Later computing systems are linked by networks and, eventually, by Internet, building on new principles for networking. This has transformed computing, leading to many software-based services shared through remote communications. One of the key enablers of this transformation is World Wide Web, supported by Internet. Here each application is still essentially sequential, but they are linked by a fixed set of network protocols.

Computing is now undergoing another transformation, where software is built on communicating processes from the ground up, because of a deep structural change in computing environments exemplified by, among others, manycore chips and cloud computing. A significance of this transformation is that it is affecting fundamental software principles, notably those for programming languages and systems software, which now need to treat communication-based organisation of software in earnest. Here communication and concurrency are the norm rather than a marginal concern. An application, or a "service", is realised not by a single endpoint but through a coordination of multiple endpoints, either on a single chip, in clouds, or distributed among several continents. A collection of such services will constitute a distributed infrastructure. Their nascent form is already visible in the backend of popular portals such as Google, Facebook and Skype.

This new framework demands software to be modelled, designed, implemented, analysed, managed and, in effect, understood as communicating processes, posing exciting challenges to the principles of programming language and systems software as well as associated methodologies. It also means a great chance to link theories and practice. The talk illustrates some of these chances and challenges, drawing from our recent experience in the use of types and logics of the π-calculus for the practice of computing, including design of programming languages, specification and verification methods, and software development framework.

M. Carbone and J.-M. Petit (Eds.): WS-FM 2011, LNCS 7176, p. 1, 2012.

Reliable Contracts for Unreliable Half-Duplex Communications*

Étienne Lozes[1] and Jules Villard[2]

[1] University of Kassel DE
[2] Queen Mary, University of London UK

Abstract. Recent trends in formal models of web services description languages and session types focus on the asynchronicity of communications. In this paper, we study a core of these models that arose from our modelling of the Sing# programming language, and demonstrate correspondences between Sing# contracts, asynchronous session behaviors, and the subclass of communicating automata with two participants that satisfy the half-duplex property. This correspondence better explains the criteria proposed by Stengel and Bultan for Sing# contracts to be reliable, and possibly indicate useful criteria for the design of WSDL. We moreover establish a polynomial-time complexity for the analysis of communication contracts under arbitrary models of asynchronicity, and we investigate the model-checking problems against LTL formulas.

Introduction

Communication contracts are becoming commonplace in several information systems, like languages for web services (WSDL, abstract WS-BPEL, WSCL, or WSCI) and programming languages (like Sing# [12] or Axum). Theoretical foundations of communication contracts are often based on bi-partite [16,22] or multi-partite session types, with a recent trend on the asynchronicity of communications [17]. In a previous work [24], we modelled the Sing# programming language, which also features asynchronous communication contracts preventing communication errors, and proposed a formal definition of Sing# contracts that was independently discovered by Stengel and Bultan [21]. This definition is remarkably close to the one of session behaviors [3], a first-order fragment of session types.

The primary goal of session types is to ensure type-safety, *e.g.* that communications over a typed channel follow the scenario of their type at runtime. Type safety then possibly ensures other safety properties, most notably reception-safety, *i.e.* the guarantee that the received messages are always of one of the expected formats, but also buffer boundedness, absence of message orphans, etc. However, type-safe programs may go wrong under some situations. Indeed, a program may be successfully checked against a contract, and yet not be reception-safe, if the contract features either non determinism, or mixed choice. In these situations, the server and the client may follow different paths

* The European Research Council has provided financial support under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no 259267.The second author acknowledges support from EPSRC.

M. Carbone and J.-M. Petit (Eds.): WS-FM 2011, LNCS 7176, pp. 2–16, 2012.

on the contract and its dual, in which case the contract might not faithfully reflect the possible head message of the incoming queues. Such bugs occurred in two contracts of the original release of Singularity [21]. The reason for this problem is that reception safety is not an intrinsic property of the syntax of the contract, but rather of the set Post^* of reachable configurations. A formal link between the properties ensured on Post^* and the properties of programs that follow $\mathcal{C}$ can be established [23], remarkably without imposing syntactic constraints on contracts or a particular communication model like FIFO. More precisely, Villard showed [23] that the boundedness of the size of the channel, the reception safety, or the absence of message orphans on Post^* directly translate into the same properties for programs (whereas deadlocks and memory leaks require extra hypotheses).

Determinism and choice uniformity are quite standard conditions on sessions, but their intrinsic relation to FIFO communications is not fully acknowledged. Consider the contract $\mathcal{C} \coloneqq ?pin; !ack; (!ok \oplus !err); \mathcal{C}$ implemented by a server S: initially, S should wait for a pin on its input buffer, send an acknowledgement on its output buffer, and then check the pin and send either an authorisation or a denial to proceed. This server can be composed with any client that follows the dual specification $\overline{\mathcal{C}}$, where sending becomes receiving and vice-versa. Assume now that the communications are not perfect FIFO, and suffer from stuttering errors. Then, even for a type-safe program, it is possible that the ack message could be received twice, thus breaking reception-safety even for a contract-abiding client.

This observation raises the question whether communication contracts can be made reliable if asynchronocity is not FIFO, say for instance out-of-order, or with stuttering errors, as it is quite common in the world-wide web network. Answering this question requires first to precise what is meant by *reliable*. The literature on session types and contracts [16,17,21,5] basically gives two kinds of answers:

- the instrumental point of view: a contract is reliable if it guarantees that the programs it types are well-behaved,
- the multiparty and message sequence chart point of view: a contract is reliable if it is realisable, or inhabited, *i.e.* if there exists a program whose conversation is exactly the one described by the contract.

These two kinds of answers actually spawned a multiplicity of notions of "reliable" contracts, obtained by adopting several notions of "well-behaved" programs [3,19,23,24,12] different notions of "conversations" [8,17,21] and considering different communication models. This multiplicity of answers suggests that there is no robust notion of reliable contracts that would be meaningful for all kinds of properties and for all kinds of asynchronicity. Moreover, each proposal justifies the determinacy and uniform choice conditions as *sufficient* conditions for contracts being reliable, and while they usually suggest that these conditions might be relaxed, they do not show how to do it *effectively*.

The issue of effectively recognising whether a contract is reliable is however crucial. Contracts are naturally modelled as communicating automata, which in the FIFO case are Turing powerful even for only one communication buffer [7]. Under other asynchronous semantics, such *dialogue* systems are often no longer Turing powerful, but still exhibit a very high complexity for reachability questions, making them difficult to analyse in practise. Channel contracts, on the other hand, can be thought of as a

restricted form of such dialogues where each machine is the dual of the other. One could think that this restriction alone makes them easier to analyse but, as we show in this paper, dualised dialogues retain their Turing power from general FIFO dialogue systems, and their high complexity from other asynchronous semantics. In fact, we show that the complexity remains the same even if contracts enjoy one (but not both) of the two syntactic restrictions mentioned before (determinism and uniform choice).

In this paper, we propose to define reliable contracts as those that are *half-duplex*, *i.e.* those where the two communication buffers are never used simultaneously, similarly to walkie-talkie conversations. Cécé and Finkel first introduced this notion in the context of FIFO communications and showed that such communications enjoy a remarkable simplicity: the set of reachable configurations is regular, and their semantics is closely related to the synchronous semantics [9]. Although this does not scale to non-FIFO communications, we show that the polynomial-time complexity of the verification problems scales to a large class of communication models, including out-of-order, lossy and stuttering communications. Adopting the half-duplex property as a definition of reliable contracts has several advantages; first, it clarifies the determinacy and uniform choice conditions on contracts: these conditions are tight with respect to the half-duplex property. In comparison, other notions of reliable contracts suggest that these properties may be relaxed but, as we show in this paper, this quickly leads to undecidability. Second, it permits to effectively ensure a flexible notion of "well-behaved" programs that may or may not take into account unspecified receptions, orphan messages, boundedness, and any regular safety property that might be of interest. Indeed, once one has shown that a particular contract is half-duplex, these questions become efficiently decidable. Third, it does not rely on each of the two parties being the "dual" of the other, and thus avoids introducing a notion of subtyping in the situations where the two parties are typed against non-dual contracts.

Our contributions are the following:

1. We show that previous foundations of contracts neither convincingly explain the determinacy and polarisation conditions, nor do they provide arguments for making the analysis of contract communications effective.
2. We show that the half-duplex property is polynomial-time and that, for half-duplex systems, boundedness, absence of unspecified receptions and message orphans can be solved in polynomial time.
3. We investigate the LTL model-checking problem over traces of either configurations or actions, and show that the former is undecidable, even for half-duplex contract communications with safe receptions, whereas the latter is decidable.

In the first section, we introduce our model of asynchronous dialogues. The second section is dedicated to defining contracts and examining previous attempts at providing foundations for them. The third section is about half-duplex communications, and we establish the polynomial-time complexity of several problems. In the last section, we consider the model-checking problem for contracts and half-duplex dialogues against linear-time temporal logic.

1 Asynchronous Dialogues

Given a finite set Σ, we write Σ^* for the set of words over alphabet Σ, ranged over by w, w'; we write $w.w'$ for the concatenation of w and w', and ϵ for the empty word. The commutative (or Parikh) image of a word w is the set of words w' equal to w up to commuting letters. The semi-linear subsets S of $\mathbb{N}^\Sigma$ are the finite unions of sets of the form $\mathbb{N}\overrightarrow{a_1} + \cdots + \mathbb{N}\overrightarrow{a_n} + \overrightarrow{b}$, and correspond to the commutative images of regular languages. When we talk about a representation of a regular (resp. semi-linear) language, we mean a non-deterministic finite automaton (resp. a base-period decomposition). The class of decision problems $\mathbf{P}$ (resp. $\mathbf{NP}$) is the one of problems that can be decided in polynomial time in the size of the input by a deterministic (resp. non-deterministic) Turing machine. A decision problem is primitive recursive if it can be decided in time $O(f(n))$ for some primitive recursive function f. We write ⁉ to denote an element of $\{!, ?\}$. For a subset S of an ordered set $(\mathcal{S}, \succeq)$, we write $\downarrow S$ and $\uparrow S$ for respectively the downward $\{s' : \exists s \in S, s \succeq s'\}$ and upward $\{s' : \exists s \in S, s' \succeq s\}$ closures of S. We assume a fixed alphabet Σ, whose size is a parameter in all the complexity results.

Communicator. A communicator is a non-deterministic finite state automaton over an alphabet of the form $\{!, ?\} \times \Sigma$.

Definition 1 (Communicator). *A* communicator *is a tuple* $\mathcal{M}$=$(Q, \Sigma, \Delta, \dot{q}, F)$ *where:*

- Q *is a finite set of states;*
- Σ *is a finite set of messages (or letters);*
- $\Delta \subseteq Q \times (\{!, ?\} \times \Sigma) \times Q$ *is a finite set of transitions;*
- $\dot{q} \in Q$ *is called the initial state;*
- $F \subseteq Q$ *is called the set of final states.*

We range over Q with $q, q', \ldots$, and over Σ with $a, b, \ldots$ Elements of the set $\mathsf{Act}_\Sigma := \{!, ?\} \times \Sigma$ are called *actions*, ranged over by $\lambda, \lambda', \ldots$, and we write $\mathsf{pol}(\lambda) \in \{!, ?\}$ for their first projection (their *polarity*). As usual, we write $q \xrightarrow{\lambda}_{\mathcal{M}} q'$ if $(q, \lambda, q') \in \Delta$ (the subscript can be omitted). Let $\mathcal{M} = (Q, \Sigma, \Delta, \dot{q}, F)$ be a fixed communicator. A state q of $\mathcal{M}$ is *terminal* if there is no $(\lambda, q') \in (\{!, ?\} \times \Sigma) \times Q$ such that $q \xrightarrow{\lambda} q'$. A (non-empty) *path* in $\mathcal{M}$ is a finite sequence of states $q_0, \ldots, q_{n+1}$ such that $q_i \xrightarrow{\lambda_i} q_{i+1}$ for all $0 \leq i \leq n$. A path is *uniform* if $\mathsf{pol}(\lambda_i) = \mathsf{pol}(\lambda_j)$ for all $i, j \in \{0, \ldots, n\}$. A communicator is *connected* if for every non initial state q, there is a path from $\dot{q}$ to q. From now on, we will implicitly consider connected communicators only. A state q is *polarised* if there are no a, b, q_1, q_2 such that $q \xrightarrow{!a} q_1$ and $q \xrightarrow{?b} q_2$. A state q is *deterministic* if for all $\lambda \in \{!, ?\} \times \Sigma$, there is at most one q' such that $q \xrightarrow{\lambda} q'$. A state q is k*-bounded* if all uniform paths (possibly cyclic) starting from q have a length less than k. A communicator is *polarised* (resp. *deterministic*) if all its states are.

Dialogues. Communicators will be either communicating with themselves using a single communication buffer, and then called *monologues*, or they will be paired with another communicator using two buffers, and then called *dialogues*.

Definition 2 (Dialogue). *A dialogue is a tuple* $\mathcal{D} = (Q_0, Q_1, \Sigma, \Delta_0, \Delta_1, \dot{q}, F)$ *such that* $(Q_i, \Sigma, \Delta_i, \dot{q}, \{q_i : (q_0, q_1) \in F\})$ *is a communicator for each* i.

Dialogues can be used to describe the semantics of Singularity contracts, where one communicating automaton is used to describe both sides of the conversation [12,21,23]. In this case, a communicator $\mathcal{M}$ is paired with its *dual*, *i.e.* the communicator in which each transition $q \xrightarrow{a} q'$ is replaced by $q \xrightarrow{\bar{a}} q'$, for $\overset{\bullet}{?} \neq \overset{\bullet}{\bar{?}}$, and the final states of the system are the pairs (q, q) of final states of $\mathcal{M}$.[1] We write $\mathcal{M} \parallel \overline{\mathcal{M}}$ for this dialogue.

We will also use the notation $\mathcal{M}_1 \parallel \mathcal{M}_2$ to denote a dialogue between $\mathcal{M}_1$ and $\mathcal{M}_2$.

Interferences. The semantics of dialogue systems will interpret send and receive actions as respectively pushing messages into an outgoing buffer and popping messages from an incoming buffer. However, we will not restrict ourselves to perfect FIFO buffers, and we will rather consider that they may be subject to several kinds of interferences from the environment. We model these interferences by a preorder over words $\succeq \subseteq \Sigma^* \times \Sigma^*$ which will parametrise the semantics of dialogue systems. Intuitively, $w \succeq w'$ if w and w' are the contents of a buffer respectively before and after being submitted to interferences.

Definition 3 (Interference Model). *An interference model is a binary relation* $\succeq \subseteq \Sigma^* \times \Sigma^*$ *satisfying the following axioms:*

Reflexivity	Transitivity	Additivity	Integrity
$\dfrac{a \in \Sigma}{a \succeq a}$	$\dfrac{w \succeq w' \qquad w' \succeq w''}{w \succeq w''}$	$\dfrac{w_1 \succeq w_1' \qquad w_2 \succeq w_2'}{w_1.w_2 \succeq w_1'.w_2'}$	$\dfrac{\epsilon \succeq w}{w = \epsilon}$

Intuitively, these axioms capture the following assumptions on the interferences we consider: they may leave the communication buffers unchanged, they act locally, and they cannot fill an empty buffer. The last assumption, corresponding to the integrity axiom, will be later clarified by the half-duplex property.

The least interference model is $w \succeq w'$ if and only if $w = w'$; it models FIFO communications, *i.e.* communications without interferences. Let us review some standard forms of interferences.

Lossiness. Possible leaks of messages during transmission are modelled by adding the axiom $a \succeq \epsilon$.

Corruption. Possible transformation of a message a into a message b is modelled by adding the axiom $a \succeq b$.

Out-of-order. Out-of-order communications are modelled by adding axioms $a.b \succeq b.a$ for all $a, b \in \Sigma$.

Stuttering. Possible duplication of a message a is obtained by adding the axiom $a \succeq a.a$.

Some of these models can be put in correspondence with existing communication protocols. For instance, the FIFO model corresponds essentially to TCP, and the lossy

[1] This is actually a slight generalisation: in Sing# contracts, final states are also terminal, and every state leads to a final state via a special message ChannelClosed. [21]

stuttering out-of-order model to UDP. Note that the out-of-order model corresponds to one where buffers are just multisets, and is thus computationally equivalent to vector addition systems with states, or Petri nets. One notable exception to our definition of interferences is the model with insertion errors, where arbitrary messages can be inserted. It would be modelled by the axiom $\epsilon \succeq a$, which would not validate the integrity axiom. We may sometimes make the stronger hypothesis that the interference model is *non-expanding*, meaning that $w \succeq w'$ implies that the length of w' is smaller than the one of w. Barring the stuttering model, all the interference models we mentioned are non-expanding.

Configurations. Let $\mathcal{D}$ be a fixed dialogue. A configuration of $\mathcal{D}$ is a tuple

$$\gamma = (q_0, q_1, w_0, w_1) \in \mathsf{Confs}(\mathcal{D}) \coloneqq Q_0 \times Q_1 \times \Sigma^* \times \Sigma^*.$$

The initial configuration $\dot{\gamma}$ of a dialogue $\mathcal{D}$ is $(\dot{q}, \dot{q}, \epsilon, \epsilon)$; (q_0, q_1, w_0, w_1) is final if (q_0, q_1) is final. We will view a configuration (q_0, q_1, w_0, w_1) as the word $q_0.w_0.q_1.w_1$ over the alphabet $(Q_0 \cup Q_1) \uplus \Sigma$. Similarly, a set of configurations can be considered as a language over such an alphabet. A set of configurations is regular (resp. semi-linear) if its associated language is. For instance, the set of configurations with empty buffers is both regular and semi-linear, whereas the set of configurations with buffers having as many a and b messages is semi-linear but not regular. A configuration $\gamma = (q_0, q_1, w_0, w_1)$ is *stable* if both buffers are empty: $w_0.w_1 = \epsilon$, *a message orphan* if it is not stable and $(q_0, q_1) \in F$, *an unspecified reception* if there is i such that $w_i \neq \epsilon$ and $\gamma \overset{i\lambda}{\not\longrightarrow}$ for all $\lambda \in \{!, ?\} \times \Sigma$, and *a k-out-of-bound error*, for $k \in \mathbb{N}$, if the sum of the length of w_0 and w_1 is greater than k.

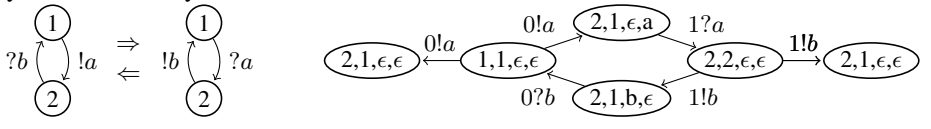
Semantics. The transition system associated to a dialogue $\mathcal{D}$ for the interference model $\succeq$ is defined by a binary relation $\xrightarrow{i\lambda}_{\succeq,\mathcal{D}}$ over configurations. We write

$$(q_0, q_1, w_0, w_1) \xrightarrow{i\lambda}_{\succeq,\mathcal{D}} (q_0', q_1', w_0', w_1')$$

if and only if there is a transition $q_i \xrightarrow{\lambda}_{\mathcal{M}_i} q_i'$ such that $q_{1-i} = q_{1-i}'$, and

(send case) either $\lambda = !a$, $w_{1-i}.a \succeq w_{1-i}'$ and $w_i \succeq w_i'$;
(receive case) or $\lambda = ?a$, $w_i \succeq a.w_i'$ and $w_{1-i} \succeq w_{1-i}'$.

Example 1. The following picture represents a dialogue and its associated transition system in the lossy semantics.



We often write $\xrightarrow{i\lambda}$ instead of $\xrightarrow{i\lambda}_{\succeq,\mathcal{D}}$ when $\succeq$ and $\mathcal{D}$ are clear from the context. We write $\gamma \to \gamma'$ if $\gamma \xrightarrow{i\lambda} \gamma'$ for some i, λ, and we write Post^* for the set of reachable configurations, *i.e.* the smallest set containing $\dot{\gamma}$ and such that, for all γ, γ', if $\gamma \to \gamma'$ and $\gamma \in \mathsf{Post}^*$ then $\gamma' \in \mathsf{Post}^*$.

The semantics of a monologue is defined similarly: it behaves as if it were in a dialogue with a forwarder communicator.

A sequence of configurations $(\gamma_i)_{i\geq 0} \in \mathsf{Confs}(\mathcal{D})^{\mathbb{N}}$ is called a *trace of configurations* of a dialogue (resp. monologue) $\mathcal{D}$ if $\gamma_0 = \dot{\gamma}$, and there is some $N \in \mathbb{N} \cup \{\infty\}$ such that for all $i \leq N$, $\gamma_i \to_{\mathcal{D}} \gamma_{i+1}$, and for all $i > N$, $\gamma_i = \gamma_{i+1}$ is a final configuration. A finite sequence $\rho = \big((pid_0, \lambda_0) \dots (pid_n, \lambda_n)\big) \in (\{0,1\} \times \mathsf{Act}_\Sigma)^*$ is a *trace of actions* of the dialogue $\mathcal{D}$ if there is a trace $(\gamma_i)_{i\geq 0}$ of $\mathcal{D}$ such that $\gamma_i \xrightarrow{pid_i \lambda_i} \gamma_{i+1}$ for all $i \leq n$. A trace of send actions is a trace of actions where receive actions are skipped; a trace of receive actions is defined similarly. A finite sequence $c = \lambda_0 \dots \lambda_n \in \mathsf{Act}_\Sigma^*$ is called a *conversation* of $\mathcal{D}$ if there is a trace of send actions ρ such that c is obtained from ρ by the substitution $(0, !a) \mapsto !a$, $(1, !a) \mapsto ?a$. For instance, a conversation of $!a; ?b; !c; \mathsf{end} \parallel ?a; !b; \mathsf{end}$ is $!a.?b.!c$, obtained from the trace of send actions $(0, !a).(1, !b).(0, !c)$, which is itself a subtrace of the trace of actions $(0, !a).(1, ?a).(1, !b).(0, ?b).(0!c).(1, ?c)$. We write $\mathsf{Conv}(\mathcal{D})$ to denote the set of all conversations.

Problems of interest. A safety property $P_{\mathcal{D}}$ of a dialogue $\mathcal{D}$ is a subset of $\mathsf{Confs}(\mathcal{D})$. We say that $\mathcal{D}$ satisfies $P_{\mathcal{D}}$ if $\mathsf{Post}^* \subseteq P_{\mathcal{D}}$. A safety property is polynomial-time regular if there is a polynomial-time function that associates to each dialogue system $\mathcal{D}$ a deterministic finite automaton that represents $P_{\mathcal{D}}$. Typical properties addressed by contracts are polynomial-time regular safety properties: safe receptions, *i.e.* the absence of unspecified receptions, *orphan-freedom*, *i.e.* the absence of messages in buffers at communication closure, and k-boundedness, *i.e.* absence of k-out-of-bound-errors. Boundedness, *i.e.* k-boundedness for some existentially quantified k, is also a typical property of interest, though it is not a safety property. All of these properties, as most properties depending on control state reachability, are known to be undecidable for dialogues and monologues in the FIFO semantics [7], and to be at least non-primitive recursive for the lossy semantics [18,20], and at least exponential space for the out-of-order semantics [11]. Finally, let us introduce two less standard properties. We say that a dialogue is *synchronous* if all stable reachable configurations can be reached without visiting a 2 out-of-bound configuration. In other words, a dialogue is synchronous if all stable configurations are reached in the synchronous semantics à la CCS; for instance, $!a; !b \parallel ?b; ?a$ is synchronous for the FIFO semantics (where it cannot run) but not for the out-of-order one. Following Stengel and Bultan [21], we say that a communicator $\mathcal{M}$ is *realisable* if, considered as finite state automaton, it exactly recognises $\mathsf{Conv}(\mathcal{M} \parallel \overline{\mathcal{M}})$. For instance, $!a + !b$ is realisable, whereas $!a + ?b$ is not.

2 Contracts

Contracts are communicators with certain syntactic conditions, similarly to Sing# contracts as formalised by Stengel and Bultan [21].[2]

[2] We omit the condition on final states being terminal, reached implictly in Sing# when receiving ChannelClosed messages. Moreover, Sing# requires another condition meant to ensure boundedness of the buffers, see Prop 1.5 below.

$\wedge$	$\neg$RS	$\neg$R	$\neg$S	$\neg$SC
RS	$-$	$!a + ?a$	$\mathcal{C}_1$	$!a + ?a$
R	$\mathcal{C}_2$	$-$	$\mathcal{C}_1$	$\mathcal{C}_2$
S	$!a; ?b + ?c$	$(!a; ?c^*; !b)^*$	$-$	$(!a; ?c^*; !b)^*$

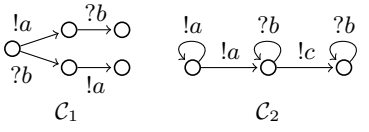


Fig. 1. Reception-safety (RS), realizability (R) and synchronism (S) are incomparable and generalise synched contracts (SC)

Definition 4 (Synched Contract). *A communicator $\mathcal{C}$ is called a* synched contract *if it is polarised, connected, and deterministic.*

Contracts are a slight generalisation of *session behaviours* [3], often presented as terms over the following grammar:

$$\mathcal{C} ::= \mathsf{end} \mid C; C \mid \oplus_{i=1,\ldots,n} !a_i; \mathcal{C}_i \mid \&_{i=1,\ldots,n} ?a_i; \mathcal{C}_i \mid \mathcal{X} \mid \mathsf{rec}\ \mathcal{X}\ \mathsf{in}\ \mathcal{C}$$

We will use this notation for concisely describing a contract, sometimes omitting trailing end. Note that session behaviours are contracts for which the final states are the terminal states. The semantics of $\mathcal{C}$ is that of the dialogue $\mathcal{C} \parallel \overline{\mathcal{C}}$. Unlike arbitrary dialogues, this semantics is very regular and thus quite easy to analyse because, at any time, at least one of the two buffers is empty, hence there is always one communicator that "follows" the other. Let us recall some facts already mentioned in the literature [15,21,24,14,5].

Proposition 1. *Let $\mathcal{C}$ be a synched contract. For $\mathcal{C} \parallel \overline{\mathcal{C}}$ in the FIFO semantics, the following properties are true:*

1. *$\mathcal{C} \parallel \overline{\mathcal{C}}$ is reception-safe;*
2. *$\mathcal{C}$ is realisable;*
3. *$\mathcal{C} \parallel \overline{\mathcal{C}}$ is synchronous;*
4. *Post^* is regular and effective;*
5. *$\mathcal{C} \parallel \overline{\mathcal{C}}$ is k-bounded if and only if all states are k-bounded;*
6. *if $\mathcal{C}$ is a session behaviour, $\mathcal{C} \parallel \overline{\mathcal{C}}$ is orphan-free.*

Properties 1, 4, 5 and 6 are of practical interest, and point out the gap between contracts and general dialogues: for the latter, they would be undecidable (see Thm. 1). Finding a good notion that generalises the contracts and justifies the determinacy and polarisation conditions first lead us to consider already discovered properties: reception-safety in the context of semantic subtyping [3], Stengel and Bultan's notion of realizability [21], and synchronism [9]. A first observation is that these properties are pairwise incomparable, and all strictly generalise the notion of synched contracts (see Fig. 1). A second observation is that they are not satisfied by synched contracts for non-FIFO communications: the synched contract $!a; !b \oplus !b; ?c$ is neither realisable, nor synchronous, nor reception-safe if communications are lossy or out-of-order.

We claim that a good candidate for being a "fundamental" property should satisfy two conditions: (1) be a decidable property, and (2) for dialogues with this property, other properties can be checked efficiently (*e.g.* boundedness can be checked efficiently for synched contracts simply by making sure that every loop in the contracts contains

	det	pol	both
FIFO	U	U	Yes
lossy	NPR	NPR	NPR

(a) Has the contract only safe receptions?

	det	pol	both
FIFO	$U^{(1)}$	$U^{(2)}$	Yes
lossy	NPR	NPR	$NPR^{(3)}$

(b) Is the contract with safe receptions orphan-free?

Fig. 2. Complexity results for relaxed hypotheses. "det" = deterministic; "pol" = polarised; "U" = undecidable; "Yes" = always true; "NPR" = non-primitive recursive.

at least one send and one receive, although synched contracts do not entail boundedness directly). Despite existing works [2,4] on related but slightly different properties, making such a call for synchronism or realizability has never been done, and it is rather unclear whether these notions satisfy requirements (1) and (2). The third candidate notion, reception-safety, is known to be undecidable for arbitrary dialogues, but Prop. 1 suggests that it could be decidable for dualised dialogues, for instance if the communicators are "almost" synched contracts. Similarly, as reception-safety plays an important role in subtyping and in the foundations of duality [3], it could be expected that properties such as boundedness or orphan-freedom are decidable for "almost synched contracts" that are reception-free. This is not the case, as the following theorem shows.

Theorem 1. *Reception-safe dialogues $\mathcal{M} \parallel \overline{\mathcal{M}}$ form neither an effective class (Fig. 2(a)), nor a class over which orphan freedom is decidable, even if $\mathcal{M}$ is assumed to be deterministic (resp. polarised), nor a tractable class if $\mathcal{M}$ is a synched contract but communications are lossy (Fig. 2(b)).*

Proof. Let us give a proof sketch for the problems marked (1), (2) and (3) in Fig. 2(b). The others are simple variants. First, let us observe that the reception-safety assumption can be lifted: a communicator C can always be extended with a sink node handling unspecified receptions without changing reachability issues (for instance, $!a; ?b; \mathsf{end}$ would be completed into $!a; (?b; \mathsf{end} \& ?a; \mathcal{C}_{\mathsf{sink}}) \oplus !b; \mathcal{C}_{\mathsf{sink}}$, where $\mathcal{C}_{\mathsf{sink}} \coloneqq (!a \oplus !b); \mathcal{C}_{\mathsf{sink}}$). Let us assume without loss of generality a synched contract $\mathcal{C}$ with a single final state and reduce the problem to the reachability of its final state in a stable configuration in the monologue semantics (which, based on standard results [7], is undecidable). Let FW denote a forwarder communicator that stops when receiving a special message $stop$, *i.e*: $\mathsf{FW} \coloneqq \big(\&_{a \in \Sigma} ?a; !a; \mathsf{FW}\big) \& ?stop; \mathsf{end}$. We then consider the following communicators, respectively deterministic for (1), polarised for (2) and both for (3), where $+$ denotes either a non-deterministic or a non-polarised choice:

$$\mathcal{C}_1 \coloneqq (!a; ?b; \mathcal{C}; !stop; ?sync; !leak) + (?b; !a; \overline{\mathsf{FW}}; ?sync)$$
$$\mathcal{C}_2 \coloneqq (!a; \mathcal{C}; !stop; ?sync; !leak) + (!a; \overline{\mathsf{FW}}; ?sync)$$
$$\mathcal{C}_3 \coloneqq (!lost; \mathcal{C}; !stop; ?sync; !leak) \oplus (!lost'; \overline{\mathsf{FW}}; ?sync)$$

Then $\mathcal{C}_1$ and $\mathcal{C}_2$ (resp. $\mathcal{C}_3$) leak the message $leak$ in the dualised FIFO (resp. lossy) semantics if and only if $\mathcal{C}$ reaches a stable final state in the FIFO monologue semantics. □

3 Half-Duplex Dialogues

Half-duplex dialogues were introduced by Cécé and Finkel in the context of FIFO communications [9]; this notion captures an idea we informally mentioned for contract communications, namely that two communication buffers are never used at the same time. In other words, a dialogue is half-duplex if, at every moment, at most one of the communicators is allowed to send messages, like in a walkie-talkie conversation.

Definition 5 (Half-duplex property).

- *A configuration* (q_0, q_1, w_0, w_1) *is* half-duplex *if either* $w_0 = \epsilon$ *or* $w_1 = \epsilon$.
- *A dialogue* $\mathcal{D}$ *is* half-duplex *if all its reachable configurations are half-duplex.*
- *A communicator* $\mathcal{C}$ *is a* half-duplex contract *if* $\mathcal{C} \parallel \overline{\mathcal{C}}$ *is half-duplex.*

Example 2. The synched contract $\mathcal{C} \coloneqq !a; ?b; \mathcal{C}$ presented in Ex. 1 is half-duplex for the FIFO, lossy, out-of-order and corruption interferences, and any combination thereof. It is not half-duplex if a or b have duplication errors. Similarly, $\mathcal{U} \coloneqq (!a; !b; !c) \oplus (!b; ?c)$ is a synched contract, but it is not half-duplex in the lossy semantics as $\mathcal{U} \parallel \overline{\mathcal{U}}$ can reduce to $!c \parallel !c$.

The half-duplex property may be imposed by the communication medium (for instance, a bus or radio communications), or it may be a design choice for optimising the implementation of a communication channel. It applies a priori to any communication model, but it should be stressed that it only makes sense for those satisfying the integrity condition of interferences: if $\succeq$ does not satisfy the integrity axiom, half-duplex dialogues are communication-free dialogues. Similarly, the class of half-duplex dialogues is the one of unidirectional communications for the stuttering model if none of the letters is ensured to be duplication-free.

Any synched contract is a half-duplex contract for the FIFO semantics. However, as observed in Ex. 2, this is not true for unreliable communications. Moreover, even for FIFO communications, the converse is false: half-duplex contracts are not necessarily synched contracts. Despite these differences, half-duplex contracts are not very different in nature from synched contracts. For a communicator $\mathcal{C}$, we write $\det(\mathcal{C})$ for the communicator obtained by determinization of $\mathcal{C}$ as a finite state automaton.

Proposition 2. *A connected communicator* $\mathcal{C}$ *is a half-duplex contract in the FIFO semantics if and only if* $\det(\mathcal{C})$ *is a synched contract.*

If we move now to unreliable communications, the connection with the synched contracts becomes a bit looser, but still exhibit some remarkable similarities:

Proposition 3. *Let* $\mathcal{C}$ *be a half-duplex contract. Then the following holds:*

1. $\det(\mathcal{C})$ *is polarised;*
2. *if* $\succeq$ *is non-expanding, then* $\mathcal{C} \parallel \overline{\mathcal{C}}$ *is* k*-bounded if and only if all states are* k*-bounded.*

These results might explain and justify some of the aspects of contracts. First, the condition of polarisation seems a rather fundamental one, and is intimately related to the

half-duplex condition. Second, the computation of the bound on the buffer's size is always extremely simple under the half-duplex hypothesis. Third, the synched hypothesis simplifies the check on polarisation, but it does not suffice to guarantee the half-duplex property for unreliable communications. To make a case for the half-duplex property rather than synched contracts, we need to address two issues:

- being half-duplex is a semantic notion hence might be complex to check, particularly in the light of the previous complexity results, whereas the syntactic synch property can be checked linearly in the number of transitions in the contract;
- it does not prevent unspecified receptions or orphan messages, as already observed in the introduction.

Before addressing these concerns, let us first identify which models of interferences support efficient decision procedures. We propose two notions of "reasonably simple" interference models: an interference model $\succeq$ is *regular* if it is axiomatized by the axioms of Def. 3 plus any subset of the lossy, corruption, and stuttering axiom. It is *semi-linear* if it is axiomatized by such a set of axioms and the out-of-order axiom. The crucial property of these interference models is that deciding the emptiness of $\uparrow L \cap \downarrow L'$ for regular languages L is in $\mathbf{P}$ for regular interferences, and in $\mathbf{NP}$ for semi-linear ones.

Theorem 2. *Let $\succeq$ be any fixed regular (resp. semi-linear) interference model. Then the following decision problem is in $\mathbf{P}$ (resp. $\mathbf{NP}$):*

Input. *A dialogue $\mathcal{D}$.*
Problem. *Is $\mathcal{D}$ half-duplex?*

Despite their semantic definition, half-duplex contracts are thus an effective subclass of communicators.

Theorem 3. *Let $\mathcal{D}$ be a half-duplex dialogue, and $\succeq$ a regular (resp. semi-linear) interference model. Then Post^* is regular (resp. semi-linear), and a representation of it is computable in polynomial-time (resp. in non-deterministic polynomial time).*

This result addresses the second concern: if $\succeq$ is a regular (resp. semi-linear) interference model then the problem of deciding whether a half-duplex dialogue satisfies a given regular safety property (*e.g.* safe receptions, or orphan-freedom) is in $\mathbf{P}$ (resp. $\mathbf{NP}$). Boundedness is moreover in $\mathbf{P}$ if either the communication model is non-expanding (by Prop. 3), or regular (by Thm. 3), and in $\mathbf{NP}$ for expanding, semi-linear interferences.

We only sketch the proof of these two results. First observe that the proof argument of Cécé and Finkel [9] does not scale to unreliable communications: it is not the case that all reachable stable configurations can be computed by considering synchronous executions. For instance, in the out-of-order semantics,

$$!a; !b; end \parallel ?b; ?a; end \rightarrow^* end \parallel end$$

but such a reduction is not possible in the synchronous semantics. The proofs of Thm. 2 and 3 are indeed based on a different observation: the set $\mathsf{Post}^*_{\mathsf{HD}}(\{\dot{\gamma}\})$ of configurations reached from $\dot{\gamma}$ by visiting half-duplex configurations corresponds to the disjunct

$$\bigcup_{\gamma \in \mathsf{Post}^*_{\mathsf{HD}}(\{\dot{\gamma}\}),\gamma \text{ stable}} \mathsf{Post}^*_{\mathsf{uni}}(\{\gamma\})$$

where $\mathsf{Post}^*_{\mathsf{uni}}(\{\gamma\})$ denotes the set of configurations that are reachable from γ by forbidding communications over one of the two queues. It can then be observed that the set of reachable stable configurations γ appearing in the disjunct is finite (since there are only finitely many stable configurations), but moreover computable in polynomial-time for order-preserving unreliable communications, and in non-deterministic polynomial-time for out-of-order unreliable communications. Finally, $\mathsf{Post}^*_{\mathsf{uni}}(\{\gamma\})$ is regular for order-preserving unreliable communications, and semi-linear for out-of-order unreliable communications.

4 LTL Model-Checking

It is sometimes desirable to express temporal properties about communications[3]; for instance, some works on subtyping introduce a liveness condition that is not automatically satisfied by contracts [19]. In this section, we introduce two notions of LTL model-checking against half-duplex dialogues, one based on traces of configurations, and the other on traces of actions. We show that the former is undecidable, even for contracts, whereas the later is decidable if only one kind of actions (either send or receive) is taken into consideration.

4.1 Traces of Configurations

We consider formulas ϕ of LTL as those given by the following grammar:

$$P ::= \langle q, q' \rangle \mid \langle \rightarrow \rangle \mid \langle \leftarrow \rangle \qquad \phi ::= P \mid \phi \wedge \phi \mid \neg\phi \mid \mathsf{X}\,\phi \mid \phi\,\mathsf{U}\,\phi$$

where $q, q' \in Q$. LTL formulas express special properties on the runs of a dialogue: $\langle q, q' \rangle$ asserts that the first configuration of a trace is in control states (q, q'); $\langle \rightarrow \rangle$ (resp. $\langle \leftarrow \rangle$) asserts that the first (resp. the second) queue is empty, $\mathsf{X}\,\phi$ asserts that ϕ is true at the next step of the trace; $\phi'\,\mathsf{U}\,\phi$ that ϕ is true after some time, and meanwhile ϕ' holds. We say that $\mathcal{D}$ satisfies ϕ if for all traces $(\gamma_i)_{i\geq 0}$ of $\mathcal{D}$, $(\gamma_i)_{i\geq 0}$ satisfies ϕ. For instance, $\langle \leftarrow \rangle\,\mathsf{U}\,(\langle \rightarrow \rangle \wedge X(\mathsf{end}, \mathsf{end}))$ asserts that the first party always closes the conversation after replying once to the messages of the second party.

Theorem 4. *The decision problem:*

Input. *A synched, half-duplex contract $\mathcal{C}$ with safe receptions.*
Input. *A formula ϕ of LTL.*
Problem. *Does $\mathcal{C} \parallel \overline{\mathcal{C}}$ satisfy ϕ?*

is undecidable if $\succeq$ is regular (resp. semi-linear).

[3] Note however that temporal properties of contracts do not necessarily translate into the same properties for the programs they type, as the set of (projections of) runs of a program is in general a subset of the runs of its contract.

The proof is by reduction of the model-checking problem for monologues, which is undecidable by standard results (for the lossy and lossy out-of-order semantics, by undecidability of visiting a control state infinitely often in lossy FIFO and lossy counter machines [20], and for out-of-order semantics by undecidability of reachability in Minsky machines –note that $\langle\leftarrow\rangle \wedge \langle\rightarrow\rangle$ encodes zero tests). The reduction of an instance $(\mathcal{M}, \phi)$ of the monologue model-checking problem to an instance $(\mathcal{C}, \phi')$ of the contract model-checking problem uses a very simple contract $\mathcal{C}$ that allows to send any message at any time, whereas the formula ϕ' is a conjunct $\phi_{\mathsf{sched}} \wedge \phi_0$ with ϕ_{sched} forcing a scheduling between the sender and the receiver that simulates the monologue $\mathcal{M}$, and ϕ_0 replicates ϕ up to this simulation. Note that atomic predicates in ϕ_{sched} do not talk about queue contents, thus the model-checking problem remains undecidable in the FIFO and lossy case if predicates P are restricted to control state observations $\langle q, q'\rangle$.

4.2 Traces of Actions

We now consider LTL formulas where the atomic predicates P in the previous grammar range over $\langle pid, \lambda\rangle \in \{0,1\} \times \mathsf{Act}_\Sigma$ and interpret formulas over traces of actions. We say that a dialogue $\mathcal{D}$ satisfies a formula ϕ in the send (resp. receive, resp. send/receive) semantics if all traces of send actions (resp. receive actions, resp. all actions) satisfy ϕ. For instance, $\langle 0, !a\rangle \wedge \mathsf{X}\langle 0, !b\rangle$ asserts that in all executions, only 0 sends messages, which are one a followed by one b; in the send/receive semantics, it moreover asserts that 1 is not receiving these messages.

Theorem 5. *Let $\succeq$ be any fixed regular (resp. semi-linear) interference model. The decision problem:*

Input. *A half-duplex dialogue $\mathcal{D}$, a formula ϕ.*
Problem. *Does $\mathcal{D}$ satisfy ϕ in the send (resp. receive) semantics?*

are decidable. However, this problem is undecidable under the FIFO or lossy send/receive semantics, even for synched half-duplex contract dialogues with safe receptions.

The undecidability results come from a slight adaptation of the proof of Thm. 4. The decidability result for the send semantics is based on the following property:

Proposition 4. *Let $\mathcal{D}$ be a half-duplex dialogue. Then the set of traces of send actions is regular. It is moreover effective for a regular (resp. semi-linear) interference model.*

On the other hand, the set of traces of receive actions is not always regular. For instance, the set of traces of receive actions of $(!a; !b)^* \parallel ?a^*; ?b^*$ for an out-of-order semantics is $\{(1, ?a)^n.(1, ?b)^n : n \geq 0\}$. For the communication models of Thm. 5, the set of traces of receive actions is however recognisable by Parikh automata, which keeps the model-checking problem decidable.

5 Conclusion

Related Work Channel contracts have been popularised by web services programming languages, and influenced *e.g.* the Sing# programming language. The first work formalising the semantics of Sing# contracts as communicating finite state machines, and

the deterministic and polarised (aka autonomous) conditions they should satisfy was by Stengel and Bultan [21]. They focus on realizability of which they show that contracts are a special case in the FIFO semantics. Stengel and Bultan designed the TUNE model-checker for contracts conversations in Sing# (hence with bounded buffers) against LTL formulas, using the SPIN model-checker as a back-end.

Some of the properties we formalised in Prop. 1 are fairly old, and to the best of our knowledge can be traced back to the work of Gouda, Manning and Yu [15]. Cécé and Finkel first proved the theorems of Sec. 2 in the restricted case of FIFO communications; their proofs rely on the observation that all stable reachable configurations are reached in the synchronous semantics as well. As we have shown, the same argument does not hold for any interference model. Cécé and Finkel also showed the undecidability of LTL model-checking over traces of configurations for the FIFO semantics. Their proof technique is significantly more specialised than ours, and could not be used in our setting (the communicators they consider are not contracts, and the proof strongly relies on the FIFO semantics), and they do not consider traces of actions.

Some of the kinds of unreliable communications we consider, like lossiness, stuttering, or message corruption, have been introduced and studied by many authors (*e.g.* Abdullah and Johnson [1], Purushothaman et al. [10], Mayr [18]). In these works, no axiomatization of the interference model is proposed, as the decidability of the problems they consider depend on the particular choice of the communication model; in our case, on the contrary, we are able to provide a generic axiomatization that makes the proofs rather uniform.

It is worth noting that our framework is orthogonal to that of well-structured transition systems [13]: some of the preorders $\succeq$ that we consider are not well-quasi-orders, and our proofs do not rely on these techniques. Our method is rather based on the idea of regular model-checking [6,25].

Perspectives. Our main goal was to recast the theory of contracts in the half-duplex framework, a novel contribution missed by the literature, and to emphasise the problems of effectivity while proposing a new notion of reliable contracts. We have argued that contract communications should be seen as exactly the dualised half-duplex communications; we observed that duality does not drastically reduce the complexity of the communications, which propose an interesting alternative to subtyping. A more refined analysis of the complexity of fixed properties over fixed communication models could however show some advantages of contracts over non-dual half-duplex communications.

Beside complexity issues, the practical relevance of the half-duplex hypothesis would probably merit a more experimental study. On the one hand, scaling this hypothesis to multipartite sessions, if possible, is not obvious, and our results do not cover interesting and useful parts of MSCs and multipartite session types. On the other hand, the half-duplex property could be a quite frequent one in message-passing programming, first of all in MPI, where non half-duplex communications are often avoided as paving the way to head-to-head deadlocks.

References

1. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Inf. Comput. (1996)
2. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. In: Software Concepts and Tools (2003)
3. Barbanera, F., de'Liguoro, U.: Two notions of sub-behaviour for session-based client/server systems. In: PPDP (2010)
4. Basu, S., Bultan, T.: Choreography conformance via synchronizability. In: WWW (2011)
5. Bono, V., Messa, C., Padovani, L.: Typing Copyless Message Passing. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 57–76. Springer, Heidelberg (2011)
6. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular Model Checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
7. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM (1983)
8. Bravetti, M., Zavattaro, G.: Contract Compliance and Choreography Conformance in the Presence of Message Queues. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 37–54. Springer, Heidelberg (2009)
9. Cécé, G., Finkel, A.: Verification of programs with half-duplex communication. Inf. Comput. (2005)
10. Cécé, G., Finkel, A., Purushothaman Iyer, S.: Unreliable channels are easier to verify than perfect channels. Inf. Comput (January 1996)
11. Esparza, J.: Decidability and complexity of Petri net problems - an introduction. In: Petri Nets (1996)
12. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G.C., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in Singularity OS. In: EuroSys (2006)
13. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. (April 2001)
14. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program (2010)
15. Gouda, M.G., Manning, E.G., Yu, Y.-T.: On the progress of communications between two finite state machines. Information and Control (1984)
16. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
17. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL (2008)
18. Mayr, R.: Undecidable problems in unreliable computations. Theor. Comput. Sci. (2003)
19. Padovani, L.: Fair Subtyping for Multi-party Session Types. In: De Meuter, W., Roman, G.-C. (eds.) COORDINATION 2011. LNCS, vol. 6721, pp. 127–141. Springer, Heidelberg (2011)
20. Schnoebelen, P.: Lossy Counter Machines Decidability Cheat Sheet. In: Kučera, A., Potapov, I. (eds.) RP 2010. LNCS, vol. 6227, pp. 51–75. Springer, Heidelberg (2010)
21. Stengel, Z., Bultan, T.: Analyzing singularity channel contracts. In: ISSTA (2009)
22. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and Its Typing System. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
23. Villard, J.: Heaps and Hops. PhD Thesis, LSV, ENS Cachan (February 2011)
24. Villard, J., Lozes, É., Calcagno, C.: Proving Copyless Message Passing. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 194–209. Springer, Heidelberg (2009)
25. Wolper, P., Boigelot, B.: Verifying Systems with Infinite but Regular State Spaces. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 88–97. Springer, Heidelberg (1998)

Behavior Based Service Composition

Fangzhe Chang[1], Pavithra Prabhakar[2], and Ramesh Viswanathan[1]

[1] Bell Laboratories, Alcatel-Lucent
{fangzhe,rv}@bell-labs.com
[2] University of Illinois at Urbana-Champaign
pprabha2@illinois.edu

Abstract. To enable flexible leveraging of the growing plethora of available web services, service clients should be automatically composed based on required behavior. In this paper, we present a foundational framework for behavior based composition. Services advertise their behavior as labeled transition systems with the action labels corresponding to their externally invocable operations. Query logics are defined in a simple extension of μ-calculus with modalities in which variables are allowed to occur. Query logics specify the desired behavior of the composition with the variables standing for the programs that need to be synthesized. We define a special subclass of programs, called deterministic and crash-free, which behave deterministically (even if the services used are nondeterministic) with all program steps successfully executing in whichever state the services may be in during entire execution. We present an algorithm that synthesises deterministic and crash-free programs whenever there exists such a solution. Since the μ-calculus is the most expressive logic for regular properties, our results yield a complete solution to the automatic composition problem for regular behavioral properties.

1 Introduction

The growing plethora of web services promises to turn the Web from an information repository for human consumption into a world-wide system for distributed computing. A number of communication mechanisms (SOAP [27], REST [11,12]) for accessing a service's operations over web protocols, and associated formats for exchanging data (WSDL [9], WADL [13]) have been proposed and standardized. Potentially, then, the web can be seen as providing a huge library of components that can be leveraged in building new applications. In the traditional paradigm for programming applications, the developer is required to have a knowledge of and understand the semantics of any components used. Since services are similar to objects in an object-oriented model exporting multiple operations with an expected protocol for the order in which the operations can be invoked, the developer's understanding has to include this service protocol. This form of *manual composition*, in current software development practice, works well when the number of dependent components is small and change infrequently. However, it is less effective in the context of leveraging web services. First, it makes application development more costly and cumbersome by requiring a manual search for

M. Carbone and J.-M. Petit (Eds.): WS-FM 2011, LNCS 7176, pp. 17–31, 2012.

any relevant services and an understanding of the behavior of their operations and expected protocol. Second, taking advantage of any new and better services requires an extensive manual effort of performing this search again and recoding the application. Third, the existing implementation of the application may suddenly fail if one of the services used changes even in any small way such as changing the name of one of its operations or its service protocol.

To leverage web services that are numerous and continuously being deployed, merged, and deprecated, one therefore needs a form of *automatic composition*. In this paradigm, client applications are described in terms of their desired behavioral properties that does not require awareness of existing services. The program for the application is then automatically synthesized as a composition of operations from existing services. To support such automatic synthesis, services are required to publish more than their static interfaces — service advertisements specify an abstraction of the behavior of their operations and the protocols to be followed for invoking them.

In this paper, we present a framework and solution towards enabling such automatic behavior based composition. We consider service advertisements expressed as labelled transition systems. The action names in such an advertisement correspond to the exported operations and the propositional constants correspond to externally observable tests on the service state or its outputs. To specify the synthesis requirements of the application, we propose a new logic that is a natural extension of μ-calculus L_μ [7]. Based notably on logics such as the Propositional Dynamic Logic (PDL) [15], it is well understood that modalities can be associated with the programs in terms of the transition relations they induce. The logic we propose includes "unknown" modalities with the unknowns being represented by variables that stand for the programs to be synthesized.

The programs we consider for synthesis can invoke service operations (action names) and test for externally observable outputs (propositional constants) in combination with control constructs such as branching and looping. However, we restrict them to have two properties that we identify as being executionally desirable — we term such programs as being *crash-free* and *deterministic*. The execution steps of deterministic crash-free programs are guaranteed to successfully execute and be deterministic in whichever state the service is driven to. In particular, they are automatically guaranteed to respect the service protocol for invoking operations as manifested in its advertisement.

We present a synthesis algorithm that proceeds in two steps. In the first step, we construct the set of all transition relations that are satisfying solutions for the unknown modalities in a formula. A salient aspect of this step, illustrating an appealing feature of the logic, is that this construction can be defined compositionally in the formula. In the second step, we show how to synthesize a program for any given transition relation. The requirement of being crash-free and deterministic makes this step non-trivial. The resulting synthesis algorithm is complete in that it returns a satisfying solution whenever there exists one. This completeness result holds without any restrictions on the service advertisement — the transition system can be non-deterministic and, furthermore, may

not be fully externally observable in that its states do not have to be distinguishable through propositional tests. The synthesis algorithm always yields regular programs — besides establishing decidability, it therefore shows that synthesis requirements expressed in the μ-calculus can be met by finite-state programs even when they are required to be crash-free and deterministic.

2 Formal Framework

Section 2.1 defines the form of behavior advertisements considered in this paper. Section 2.2 defines the class of programs considered for service clients.

2.1 Service Advertisements

The service advertisements we consider are finite labeled transition systems. Let $\mathcal{P}$ be a set of propositional constants. A valuation over $\mathcal{P}$ is a set $\mathbf{v} \subseteq \mathcal{P}$ denoting a truth assignment that assigns true to all propositions $p \in \mathbf{v}$ and false to all $p \notin \mathbf{v}$. We use variables $\mathbf{v}, \mathbf{v}_0, \mathbf{v}_1, \ldots$ to range over propositional valuations.

Definition 1 (Labeled Transition Systems (*LTS*)). *A labeled transition system $\mathcal{T} = (\mathcal{P}, \mathcal{A}, Q, \mathcal{V}, \{\xrightarrow{a} \mid a \in \mathcal{A}\}, I)$ where $\mathcal{P}$ is a set of propositional constants, $\mathcal{A}$ a set of action names, Q a finite set of states, $\xrightarrow{a} \subseteq Q \times Q$ is the transition relation for the action name $a \in \mathcal{A}$, $\mathcal{V} : Q \to 2^{\mathcal{P}}$ is the state valuation function with $\mathcal{V}(q)$ giving the valuation for the propositional constants in any state $q \in Q$, and $I \subseteq Q$ is the set of possible initial states.*

Intuitively, the actions in $\mathcal{A}$ correspond to the operation names that the service makes available. The propositional constants in $\mathcal{P}$ serve as an abstraction for the externally observable properties of operation outputs and the service state. Figure 1 presents an example of an advertisement for a simplified version of an "Amazon"-like shopping service. The operations made available are search s, login l, add-to-cart a, and checkout c. The propositional constants are P_s denoting a successful search result, P_l denoting a successful login, P_a denoting a non-empty cart, and P_c denoting checkout completion. The initial state is q_1. The example illustrates why advertisements are naturally non-deterministic, *e.g.*, a search may or may not result in success. Thus in state q_1, the action s can transition to either state q_3 (when the search is successful) or q_1 (when the search does not yield a result). A similar explanation accounts for the transitions from q_1 on the action l to either q_1 or q_2. After a search yields a result, the item can be added to the cart and checkouts can only be done when the cart is non-empty. While a search can be performed with or without logging in, the advertisement requires that before an item can be added to the cart, one must have logged in. Having logged in at any time, further additions to the cart do not require one to login again.

We will use infix notation for relations writing, e.g., $q \xrightarrow{a} q'$ for $(q, q') \in \xrightarrow{a}$. We write $q \xrightarrow{a}\!\!\!\!\!\!/\;$ to denote that $\nexists q'.\ q \xrightarrow{a} q'$ (and $q \xrightarrow{a}$ otherwise). A (labeled) path

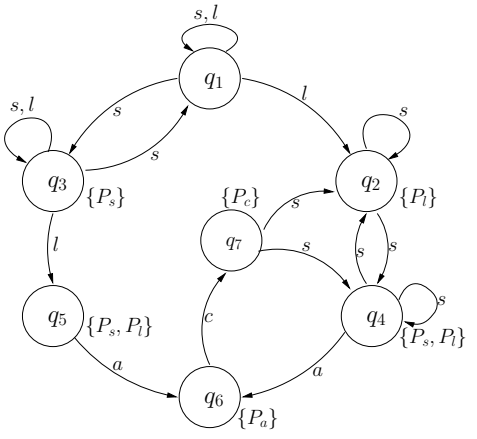


Fig. 1. Service Advertisement Example

π in a transition system is a sequence of the form $q_0 \xrightarrow{a_1} q_1 \cdots \xrightarrow{a_i} q_i \cdots \xrightarrow{a_n} q_n$ with $n \geq 0$ such that $q_i \xrightarrow{a_{i+1}} q_{i+1}$ for $0 \leq i < n$. The trace of the path π is the word $\mathcal{V}(q_0)a_1\mathcal{V}(q_2)a_2 \cdots a_n\mathcal{V}(q_n)$; we use $Trace_{\mathcal{T}}(q_1, q_2)$ to denote the set of words which are traces of paths starting in q_1 and ending in q_2. We will drop the subscript $\mathcal{T}$ when the transition system is clear from the context.

2.2 Programs

The basic operations of client programs are tests of the propositional constants and invocation of action names. Let $\mathcal{P}$ be a set of propositional constants and $\mathcal{A}$ be a set of action names. We define a *program trace* over $(\mathcal{P}, \mathcal{A})$ to be a word in $2^{\mathcal{P}}(\mathcal{A}\, 2^{\mathcal{P}})^*$, *i.e.*, a sequence of the form $\mathbf{v}_0 a_1 \mathbf{v}_1 \ldots a_i \mathbf{v}_i \ldots a_n \mathbf{v}_n$ with $n \geq 0$, $\mathbf{v}_i \subseteq \mathcal{P}$ for $0 \leq i \leq n$, and $a_i \in \mathcal{A}$ for $1 \leq i \leq n$. The operational intuition is as follows: an element a_i corresponds to invoking the action a_i yielding the next state(s), an element $\mathbf{v}_i$ tests whether the propositional valuation in the current state matches $\mathbf{v}_i$, and a program trace is a sequence of such operations. A program is then taken to be a set of such execution traces, *i.e.*, a language $L \subseteq 2^{\mathcal{P}}(\mathcal{A}\, 2^{\mathcal{P}})^*$. Following these intuitions, we define programs and the execution of a program with respect to a service as given by its induced transition relation.

Definition 2. *Let $\mathcal{T} = (\mathcal{P}, \mathcal{A}, Q, \mathcal{V}, \{\xrightarrow{a} \mid a \in \mathcal{A}\}, I)$ be a transition system. The set of programs over $\mathcal{P}$ and $\mathcal{A}$ is defined to be the languages of traces $\mathcal{LT}(\mathcal{P}, \mathcal{A}) = \{L \mid L \subseteq 2^{\mathcal{P}}(\mathcal{A}\, 2^{\mathcal{P}})^*\}$.*

- *For any finite program trace $w \in 2^{\mathcal{P}}(\mathcal{A}\, 2^{\mathcal{P}})^*$, the transition relation induced by w in $\mathcal{T}$, denoted $\mathcal{T}(w)$, is defined by $(q, q') \in \mathcal{T}(w)$ iff there exists a path $q_0 \xrightarrow{a_1} q_1 \cdots \xrightarrow{a_i} q_i \cdots \xrightarrow{a_n} q_n$ such that $q = q_0$, $q' = q_n$ and*

$$w = \mathcal{V}(q_0)\ a_1\ \mathcal{V}(q_1)\ \ldots\ a_i\ \mathcal{V}(q_i)\ \ldots\ a_n\ \mathcal{V}(q_n)$$

– *For any language* $L \subseteq 2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^*$*, the transition relation induced by* L *in* $\mathcal{T}$ *is defined as* $\mathcal{T}(L) = \bigcup_{w \in L} \mathcal{T}(w)$.

The above definition of programs encompasses the ability (e.g., PDL [15]) to use control constructs such as test-based branching and loops in composing service operations. It is important to note that while we have defined programs and their execution with respect to a single labeled transition system, this is sufficient to describe programs using operations from multiple services. This is because multiple service advertisements can be combined into a single labeled transition system using variants of the asynchronous product construction.

In principle a program $L \in \mathcal{LT}(\mathcal{P}, \mathcal{A})$ can be executed on a transition system $\mathcal{T}$ by non-deterministically selecting and then following the proper trace in L which happens to be an execution trace of $\mathcal{T}$. For practical reasons, we identify a special class of programs that are more desirable to execute, which enjoy two special properties: that of determinism and crash-free execution.

For words x, w, we write $x \preceq w$ to denote that x is a prefix of w, *i.e.*, if there exists a word w' with $w = xw'$. The longest common prefix of two words w_1 and w_2, which we denote by $w_1 \curlywedge w_2$, is the longest word that is a prefix of both w_1 and w_2. We then have the following definition of determinism.

Definition 3 (Determinism). *Two program traces* $w_1, w_2 \in 2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^*$ *are said to be* consistent*, written* $w_1 \sim w_2$*, iff* $w_1 \neq w_2 \;\Rightarrow\; w_1 \curlywedge w_2 \in (2^{\mathcal{P}}\,\mathcal{A})^*$. *A language* $L \subseteq 2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^*$ *is* deterministic *iff* $\forall w_1, w_2 \in L.\ w_1 \sim w_2$.

Essentially, two distinct program traces are consistent if, starting from the left, the first position at which they are different is at a propositional valuation. Two consistent program traces are therefore either identical or of the form $w\mathbf{v}_1 w_1$ and $w\mathbf{v}_2 w_2$ with $\mathbf{v}_1 \neq \mathbf{v}_2$.

Let $\mathcal{DLT}(\mathcal{P}, \mathcal{A})$ denote the set of all deterministic languages of program traces over $(\mathcal{P}, \mathcal{A})$. A language $L \in \mathcal{DLT}(\mathcal{P}, \mathcal{A})$ defines executions that are deterministic in a certain sense with respect to an arbitrary (not necessarily deterministic) transition system $\mathcal{T}$, which can be understood informally as follows. If $\mathcal{T}$ is in a state q, we consider all traces of the form $\mathbf{v}w \in L$ whose initial valuation $\mathbf{v}$ matches the valuation in state q. As a consequence of Definition 3, either w is empty or for all such traces the first action a in w is the same. If w is empty, then the execution halts; otherwise, it invokes the action a and continues by executing the program $L' = \{w' \mid \mathbf{v}aw' \in L\}$ (in whatever state $\mathcal{T}$ goes to after the invocation of a). In either case, the next step of either halting or the action a to be invoked and the ensuing program L' to be executed is completely determined by the current state of the transition system. Furthermore, it can be seen that the next program executed $L' \in \mathcal{DLT}(\mathcal{P}, \mathcal{A})$ and therefore the rest of the execution continues to be deterministic.

It is easy to see that the language $L = 2^{\mathcal{P}} a 2^{\mathcal{P}} \cup 2^{\mathcal{P}} b 2^{\mathcal{P}}$ which corresponds to the regular program $a \cup b$ (for some $a, b \in \mathcal{A}$) does not satisfy Definition 3. Note that if program trace w is a proper prefix of another trace w' then w and w' are not consistent, because after the trace corresponding to w has been followed, w indicates a "halt" while w' indicates an action to invoke. Thus, the language

$L = (2^{\mathcal{P}}a)^* 2^{\mathcal{P}}$ (for some $a \in \mathcal{A}$), which corresponds to the regular program a^*, does not satisfy Definition 3. These two examples illustrate that the general notion of determinism, given by Definition 3, suitably weeds out the intuitively non-deterministic operations of choice and unbounded iteration.

The notion of crash-free execution is most easily motivated through the service advertisement $\mathcal{T}$ given by Figure 1 and the PDL program $l\,; P_l?$ represented by the language $L = 2^{\mathcal{P}}\ l\ \{\mathbf{v} \subseteq 2^{\mathcal{P}} \mid P_l \in \mathbf{v}\}$. In executing L (over $\mathcal{T}$), starting in the initial state q_1, the invocation of the operation l could cause $\mathcal{T}$ to move to the state q_1 where L does not prescribe the next action to perform. At that point, the execution then hangs or crashes. Therefore, although L is deterministic, it is still not a particularly satisfactory program (when executed over $\mathcal{T}$).

More generally, although for a deterministic program $L \in \mathcal{DLT}(\mathcal{P}, \mathcal{A})$, any execution step is always uniquely determined in any state q of a service $\mathcal{T}$, the execution may get "stuck" for one of two reasons: (a) there is no trace whose initial valuation matches the current state q (*i.e.*, there is no next execution step identified), or (b) an action a is invoked but is not enabled in state q (*i.e.*, the next execution step identified cannot be performed). We will filter out such programs by refining the induced transition relation of Definition 2 to identify executions that fail to progress. Define prefix closure $L_{\preceq} \subseteq (2^{\mathcal{P}} \cup \mathcal{A})^*$ as the set of all prefixes of traces in L, *i.e.*, $L_{\preceq} = \{w \mid \exists w' \in L.\ w \preceq w'\}$. The following defines an induced relation with failure $\mathcal{T}^{\pitchfork}(L) \subseteq Q \times (Q \cup \{\pitchfork\})$ where the special symbol $\pitchfork \notin Q$ identifies a failed execution.

Definition 4 (Induced Transition Relation with Failure). *For any $L \in \mathcal{DLT}(\mathcal{P}, \mathcal{A})$ and any transition system $\mathcal{T} = (\mathcal{P}, \mathcal{A}, Q, \mathcal{V}, \{\xrightarrow{a} \mid a \in \mathcal{A}\}, I)$, the relation $\mathcal{T}^{\pitchfork}(L) \subseteq Q \times (Q \cup \{\pitchfork\})$ is defined as follows. For any state $q \in Q$,*

1. *for any $q' \in Q$, $(q, q') \in \mathcal{T}^{\pitchfork}(L)$ iff $(q, q') \in \mathcal{T}(L)$*
2. *$(q, \pitchfork) \in \mathcal{T}^{\pitchfork}(L)$ iff there exists a labeled path $q_0 \xrightarrow{a_1} q_1 \cdots \xrightarrow{a_i} q_i \cdots \xrightarrow{a_n} q_n$ with $n \geq 0$ and $q = q_0$ such that for the word $w = \mathcal{V}(q_0)\ a_1\ \mathcal{V}(q_1)\ \ldots\ a_i\ \mathcal{V}(q_i)\ \ldots\ a_n$ (with w defined to be ϵ if $n = 0$) we have that $w \in L_{\preceq}$ and one of the following conditions holds:*
   - *$w\ \mathcal{V}(q_n) \notin L_{\preceq}$, or*
   - *$w\ \mathcal{V}(q_n)\ a \in L_{\preceq}$ for some $a \in \mathcal{A}$ and $q_n \stackrel{a}{\not\to}$.*

A deterministic language L is *crash-free* in a state q of a transition system $\mathcal{T}$ if $(q, \pitchfork) \notin \mathcal{T}^{\pitchfork}(L)$, and L is crash-free for a transition system $\mathcal{T}$ if L is crash-free in every state $q \in I$ where I is the set of initial states of $\mathcal{T}$.

3 Query Logic for Specifying Composed Behavior

We now define our logic for specifying the required behavior of the synthesized compositions. Classical modal logics include modalities of the form $\langle a \rangle$ or $[a]$ for action names (that are constants in any model). We generalize these modalities to allow variables so that formulas can include modalities of the form $\langle x \rangle$ or $[x]$. Such variables serve as placeholders for and to identify the different operations of

the composed service for which programs need to be synthesized with the overall formula expressing the behavioral property of the execution of these operations, potentially in combination with each other. In this paper, we consider the query logic defined by such an extension to the propositional μ-calculus L_μ [7] since it is among the most expressive specification logics. The solutions to the synthesis problem presented in this paper are applicable to queries specified in any logic (e.g., PDL) translatable to L_μ since the standard translations between modal logics will also apply to their extensions with variable modalities.

The query logic, which we call $L_{\mu,\langle x\rangle}$, is defined as follows.

$$\varphi := p \mid X \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle a\rangle\varphi \mid \langle x\rangle\varphi \mid \mu X\varphi$$

where $p \in \mathcal{P}$ is a propositional constant, $a \in \mathcal{A}$ an action name, $x \in \mathcal{AV}$ is a program variable and $X \in \mathcal{PV}$ is a propositional variable (assuming disjoint sets of variable names $\mathcal{PV}$ and $\mathcal{AV}$). We use lower case letters x, y, x_1 to range over program variables $\mathcal{AV}$ and upper case letters X, Y, X_1 to range over propositional variables $\mathcal{PV}$. The only additional clause in the above grammar to that of L_μ is the formula $\langle x\rangle\varphi$. Similar to L_μ, in the formula $\mu X\varphi$, X is required to appear only positively in φ. Using negation, we can define the propositional constants *true* and *false*, propositional conjunction $\wedge$, propositional implication $\rightarrow$, box modalities $[a]\varphi$ and $[x]\varphi$, as well as greatest fixed point formulas $\nu X\varphi$. Although the syntax of formulas explicitly only allows pure variables to appear within modalities, using the standard translation of PDL [15] into L_μ, we can express formulas of the form $\langle\alpha\rangle\varphi$ and $[\alpha]\varphi$ where α is any regular combination of variables, action names and tests of propositional formulas.

The semantics of a formula is defined over a *LTS* $\mathcal{T}$ to yield a set of states just as in L_μ. However, in addition to an environment η mapping free propositional variables to sets of states, the semantics of $L_{\mu,\langle x\rangle}$ formulas additionally has an environment Θ mapping free program variables to programs in $\mathcal{LT}(\mathcal{P}, \mathcal{A})$, so that $\mathcal{T}[\![\varphi]\!]\eta\Theta$ is defined by induction on the structure of φ to return a set of states of $\mathcal{T}$. The inductive case for the variable modality formula is given by

$$\mathcal{T}[\![\langle x\rangle\varphi]\!]\eta\Theta \quad = \quad \{q \mid \exists q'.(q, q') \in \mathcal{T}(\Theta)(x) \text{ and } q' \in \mathcal{T}[\![\varphi]\!]\eta\Theta\}$$

where $\mathcal{T}(\Theta)$ is the map $x \mapsto \mathcal{T}(\Theta(x))$ (see Definition 2 for $\mathcal{T}(\Theta(x))$). For all other forms of formulas, the inductive definition is exactly similar to L_μ [7]. We define satisfaction of a closed formula (i.e., with no free propositional variables) φ as $\mathcal{T} \models \varphi, \Theta$ iff $I \subseteq \mathcal{T}[\![\varphi]\!]\emptyset\Theta$, where I is the set of initial states of $\mathcal{T}$ and $\emptyset$ is the empty environment for propositional variables.

Define an environment Θ to be deterministic and crash-free with respect to a transition system $\mathcal{T}$, if $\Theta(x) \in \mathcal{DLT}(\mathcal{P}, \mathcal{A})$ and $\Theta(x)$ is crash-free for $\mathcal{T}$, for every variable x in the domain of Θ. We consider the following synthesis problem.

Problem 1 (Synthesis of Deterministic Crash-Free Programs). Given a labeled transition system $\mathcal{T}$ and a closed $L_{\mu,\langle x\rangle}$ formula φ, does there exist a Θ that is crash-free and deterministic with respect to $\mathcal{T}$ such that $\mathcal{T} \models \varphi, \Theta$? If so, return a Θ that is crash-free and deterministic with respect to $\mathcal{T}$.

Note that the problem only restricts the synthesized programs to be deterministic (in the sense of Definition 3); the transition system $\mathcal{T}$ is allowed to be non-deterministic. As an example, consider the synthesis of a service consisting of two operations: a search function, which will be denoted by the program variable x in the query formula, and a "one-click" checkout function, denoted by the program variable y. Assume that the common ontology includes the propositional variables P_s denoting a successful search and P_c denoting checkout completion as in Figure 1. The property of the search function is that it can always be performed either after a failed search or performing the "one-click" checkout, expressed by the formula $\varphi_{search} \triangleq (\neg P_s \rightarrow \langle x \rangle true) \wedge [y]\langle x \rangle true$. Note that even this simple property for x utilizes the ability to use the other program variable y in the formula. The property required of the "one-click" checkout is that it can be performed after a successful search after which checkout completion should always hold, expressed by the formula $\varphi_{checkout} \triangleq P_s \rightarrow (\langle y \rangle true \wedge [y]P_c)$. (It is a "one-click" checkout in that its client doesn't have to explicitly do any other operation first such as logging in or adding to the cart.) Finally, we require these properties to hold recursively after any invocation sequence of the synthesized operations (x and y), expressed by the following greatest fixed point formula

$$\varphi \triangleq \nu X \varphi_{search} \wedge \varphi_{checkout} \wedge [x]X \wedge [y]X$$

We can also express a richer property requiring the use of nested alternate fixed points. Suppose that the synthesized service receives commercial proceeds only from checkouts and being somewhat mercenary, it doesn't want its clients to purely conduct searches without doing checkouts. A more precise formulation capturing this intuitive requirement is that no execution sequence is allowed to contain a subsequence that includes infinitely many successful searches (a search x followed by P_s holding) but without including any checkouts (the operation y). This can be expressed by the formula

$$\varphi_{merc} \triangleq \mu Y \nu Z [x](P_s \rightarrow Y) \wedge [x]Z$$

In understanding the above formula, the $[x]$ in the subformula $[x]Z$ should be read as $[\neg y]$ (*i.e.*, any action other than y which in this example is only the action x). The subformula $[x](P_s \rightarrow Y)$ requires that any successful search leads to a state where Y holds; the variable Z then requires this property to hold of the current state and any state reached by a sequence of non-y actions ($[x]Z$). Since the variable Y is bound as a least fixed point, it can only be unwound finitely often. Thus, the formula φ_{merc} holds in states from which every sequence of non-y actions can only include a finite number of x followed by P_s. Following our previous idiom for recursively requiring the property to hold after any execution sequence, the query corresponding to this additional requirement is the formula

$$\psi \triangleq \nu X \varphi_{search} \wedge \varphi_{checkout} \wedge \varphi_{merc} \wedge [x]X \wedge [y]X$$

4 Solution to the Synthesis Problem

Our solution to the synthesis problem is based on the observation that $\mathcal{T} \models \varphi, \Theta$ depends only on the transition relation induced by Θ, i.e., $\mathcal{T}(\Theta)$. The synthesis

algorithm, then, consists of two steps. The first step, presented in Section 4.1, computes constraints on the transition relations induced by the program variable environments which are necessary and sufficient to satisfy a given formula in $L_{\mu,\langle x\rangle}$. The second step, presented in Section 4.2, synthesizes a deterministic crash-free program variable environment whose induced transition relation meets the computed constraints.

4.1 Computing Constraints on Transition Relation

In the first step, we compute necessary and sufficient constraints on the satisfying transition relation assignments for the program variables. Given a *LTS* $\mathcal{T} = (\mathcal{P}, \mathcal{A}, Q, \mathcal{V}, \{\xrightarrow{a} \mid a \in \mathcal{A}\}, I)$ and a closed $L_{\mu,\langle x\rangle}$ formula φ, we are interested in computing the constraint set C such that for any $\hat{\Theta} \in \mathit{Trans}$ we have that $\hat{\Theta} \in C$ iff $\mathcal{T} \models \varphi, \hat{\Theta}$, where *Trans* denotes the function space $\mathcal{AV} \to 2^{Q\times Q}$, *i.e.*, the set of all assignments of program variables to transition relations, and C is a subset of *Trans*. This set C itself cannot be computed compositionally in the formula φ. To permit an inductive definition, we generalize to compute for each state q the constraint set $C(q)$ such that $\hat{\Theta} \in C(q)$ iff $q \in \mathcal{T}[\![\varphi]\!]\hat{\Theta}$.

Following this idea, let *CMap* denote the set of functions $Q \to 2^{\mathit{Trans}}$. Under the pointwise ordering on functions, the set *CMap* is a lattice inheriting the lattice structure of 2^{Trans}. We use $\sqsubseteq$ to denote this ordering and $\bigwedge$ to denote the greatest lower bound. We define a *constraint semantics* $[\![\cdot]\!]^C$ on formulas φ that returns objects in *CMap* with the free variables of φ interpreted similarly, *i.e.*, mapped to objects in *CMap*. Given a $L_{\mu,\langle x\rangle}$ formula φ and an environment $\zeta \in \mathcal{PV} \to \mathit{CMap}$, $\mathcal{T}[\![\varphi]\!]^C\zeta$ as an element of *CMap* is defined inductively:

- $(\mathcal{T}[\![p]\!]^C\zeta)(q) = \mathit{Trans}$ if $p \in \mathcal{V}(q)$ and $\emptyset$ otherwise
- $\mathcal{T}[\![X]\!]^C\zeta = \zeta(X)$
- $(\mathcal{T}[\![\neg\varphi]\!]^C\zeta)(q) = \mathit{Trans}\backslash(\mathcal{T}[\![\varphi]\!]^C\zeta)(q)$
- $(\mathcal{T}[\![\varphi_1 \vee \varphi_2]\!]^C\zeta)(q) = (\mathcal{T}[\![\varphi_1]\!]^C\zeta)(q) \cup (\mathcal{T}[\![\varphi_2]\!]^C\zeta)(q)$
- $(\mathcal{T}[\![\langle a\rangle\varphi]\!]^C\zeta)(q) = \bigcup_{q\xrightarrow{a}q'}(\mathcal{T}[\![\varphi]\!]^C\zeta)(q')$
- $(\mathcal{T}[\![\langle x\rangle\varphi]\!]^C\zeta)(q) = \{\hat{\Theta} \in \mathit{Trans} \mid \exists q' \in Q : (q, q') \in \hat{\Theta}(x), \hat{\Theta} \in (\mathcal{T}[\![\varphi]\!]^C\zeta)(q')\}$
- $\mathcal{T}[\![\mu X\varphi]\!]^C\zeta = \bigwedge\{C \in \mathit{CMap} \mid \mathcal{T}[\![\varphi]\!]^C\zeta[X \mapsto C] \sqsubseteq C\}$, where the environment $\zeta[X \mapsto C]$ maps X to C and otherwise agrees with ζ

Having chosen the appropriate semantic space *CMap*, the inductive semantics is fairly straightforward. Note that the correctness of the semantics of $\neg\varphi$ depends on computing the set of satisfying transition relations *exactly*, and the modal cases highlight the necessity of generalizing to functions on states. The definition of the constraint semantics of $\mu X\varphi$ as the least prefixed point over the space *CMap* (instead of 2^Q as in the standard semantics) is natural but its correctness is somewhat serendipitious. It relies on the following adjunctive structure that can be defined over *CMap*.

Given a $C \in \mathit{CMap}$ and $\hat{\Theta} \in \mathit{Trans}$, define $C_{\restriction\hat{\Theta}} \subseteq Q$ to be the set $\{q \mid \hat{\Theta} \in C(q)\}$, and consider the map $\restriction : \mathit{CMap} \to (\mathit{Trans} \to 2^Q)$ given by $\restriction(C)(\hat{\Theta}) = C_{\restriction\hat{\Theta}}$.

It is easy to see that $\restriction$ preserves least upper bounds and is therefore a left adjoint. Using the commutativity properties of least fixed points with left adjoints (see, *e.g.*, [4]), we can then establish the following lemma, by induction on the structure of the formula, stating that the constraint semantics and standard semantics of a formula are related via this adjunction.

Lemma 1. *For any formula φ, $\zeta \in \mathcal{PV} \to \mathit{CMap}$, and $\hat{\Theta} \in \mathit{Trans}$,*

$$(\mathcal{T}[\![\varphi]\!]^C \zeta)_{\restriction \hat{\Theta}} \;=\; \mathcal{T}[\![\varphi]\!] \zeta_{\restriction \hat{\Theta}} \hat{\Theta}$$

From a computational standpoint, a key consequence of Lemma 1 and the adjunction given by $\restriction$ is that the maximum number of iterations required to compute the least fixed point is the height of the lattice 2^Q (*i.e.*, $|Q|$) rather than the height of the lattice *CMap*. We can therefore establish the following complexity for computing the inductive constraint semantics.

Lemma 2. *For any closed formula φ, the time complexity of computing $\mathcal{T}[\![\varphi]\!]^C$ is $O(|\varphi|\,|Q|^{ad(\varphi)+1}\, 2^{|Q|^2|\mathcal{AV}|})$, and the space complexity is $O(|Q|^2|\mathcal{AV}|\log(|\varphi|))$, where $ad(\varphi)$ denotes the alternation depth of the formula φ.*

As a corollary of Lemma 1, we can establish the correctness of the constraint semantics for obtaining satisfying transition relations for the program variables.

Corollary 1. *Given φ in $L_{\mu,\langle x\rangle}$ and a substitution Θ, $\mathcal{T} \models \varphi, \Theta$ iff $\mathcal{T}(\Theta)$ is in $(\mathcal{T}[\![\varphi]\!]^C)(q)$, for all $q \in I$.*

4.2 Synthesizing Deterministic Crash-Free Programs

Corollary 1 yields the constraint $C = \bigcap_{q\in I}(\mathcal{T}[\![\varphi]\!]^C)(q)$ as being exactly the set of all satisfying transition relation maps for the program variables, with a program environment Θ being a satisfying solution iff $\mathcal{T}(\Theta) = \hat{\Theta}$ for some $\hat{\Theta} \in C$. The size of C is bounded by $2^{|Q|^2|\mathcal{AV}|}$ and each $\hat{\Theta} \in C$ can be considered in turn. Since $\mathcal{T}(\Theta)$ is given pointwise on each program variable, for any such transition relation map $\hat{\Theta} \in \mathit{Trans}$, a deterministic crash-free program environment Θ (such that $\mathcal{T}(\Theta) = \hat{\Theta}$) can be synthesized for each variable x independently by taking $\Theta(x)$ to be a deterministic crash-free program L such that $\mathcal{T}(L) = \hat{\Theta}(x)$. One is thus reduced to solving the problem of given any transition relation $R \subseteq Q \times Q$, synthesize (if there exists) a deterministic crash-free program X such that $\mathcal{T}(X) = R$. In this section, we present our solution to this problem. Due to space considerations, the presentation is limited to an informal sketch of the main ideas.

Consider any transition relation $R \subseteq Q \times Q$. Define L' to be the language $\bigcap_{(q_1,q_2)\notin R}(2^{\mathcal{P}}(\mathcal{A}\,2^{\mathcal{P}})^* \setminus \mathit{Trace}_T(q_1,q_2))$, and $L'_1,\ldots,L'_{n-1}$ to be an enumeration of the languages $\mathit{Trace}_T(q_1,q_2)$, for every $(q_1,q_2) \in R$. It is easy to show that for any program $X \in \mathcal{LT}(\mathcal{P},\mathcal{A})$ we have that $\mathcal{T}(X) = R$ iff $X \subseteq L'$ and $X \cap L'_i \neq \emptyset$ for $1 \leq i < n$. Let $L_i = L'_i \cap L'$ for $1 \leq i < n$ and $L_n = L'$. We then have that for any program $X \in \mathcal{LT}(\mathcal{P},\mathcal{A})$, $X \subseteq L'$ and $X \cap L'_i \neq \emptyset$ for $1 \leq i < n$ iff

$X \subseteq \cup_i L_i$ and $X \cap L_i \neq \emptyset$ for $1 \leq i \leq n$. Note that the languages $L_1, \ldots, L_n$ are regular. Our algorithm to compute a crash-free deterministic program $X \subseteq \cup_i L_i$ which has a non-empty intersection with L_i for $1 \leq i \leq n$, for any given regular languages $L_1, \ldots, L_n$, is as follows. It searches for a subautomaton of a certain form in a finite state automaton belonging to a finite set S constructed using the transition system $\mathcal{T}$ and automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ accepting $L_1, \ldots, L_n$ respectively. We first present the construction of the set S of finite state automata.

We consider bipartite automata in which the outgoing transitions from one partition have labels in $2^{\mathcal{P}}$ and from the other in $\mathcal{A}$; it is complete if each state has outgoing transitions for all labels in the set corresponding to its partition. The transition system $\mathcal{T}$ can be naturally transformed into a complete bipartite automaton B accepting the trace language of $\mathcal{T}$. Since the languages $L_1, \ldots, L_n$ are in $\mathcal{LT}(\mathcal{P}, \mathcal{A})$, there exist deterministic complete bipartite automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ accepting the languages $L_1, \ldots, L_n$, respectively. Let $\mathcal{A}'$ be an automaton for $L' = \cup_{1 \leq i \leq n} L_i$ obtained by taking the product of the automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ and choosing appropriate final states. The automata in S are constructed by taking the product of B with an automaton $\mathcal{A}$ constructed by using $\mathcal{A}'$ and trees T, as follows. We consider directed rooted trees T with m nodes where $1 \leq m \leq 2n-1$. Let V be the set of nodes and E the set of edges. Each node is labelled by a distinct number in $\{1, \ldots, m\}$ with the root labelled by 1. With each edge (u, v) we associate an edge of $\mathcal{A}'$. Given a tree T as above, we define $Aut(\mathcal{A}', T)$ to be the automaton consisting of a copy of $\mathcal{A}'$ for each node of T, with the following changes in the transitions. If an edge (u, v) of T is labelled by a transition $q \xrightarrow{a} q'$, then instead of the transition $q \xrightarrow{a} q'$ in the i-th copy of A' where i is the label of u, there is a transition from q of the i-th copy on a to q' of the j-th copy where j is the label of v. The set S is the set of all automata obtained by taking the product of B with $Aut(\mathcal{A}', T)$ for some tree T of the above form. Observe that there are only finitely many trees with at most $2n-1$ nodes of the required form, and hence S is finite.

We search for a subautomaton such that the resulting program, taken to be the language of the subautomaton, is deterministic, crash-free and has a non-empty intersection with each L_i. Each of these three requirements translate to the following verifiable conditions on the subautomaton. Assume, without loss of generality, that every state in the subautomaton B' is reachable and can reach a final state of B'. To ensure a non-empty intersection with each L_i, the subautomaton should have for each i, a final state which corresponds to a final state of $\mathcal{A}_i$, and for it to be subset of $\cup_i L_i$, each of its final states should correspond to a final state of $\mathcal{A}_i$ for some i. Similarly, determinism translates to the requirement that from each reachable state in the second partition, there is atmost one action from $\mathcal{A}$ which is enabled. And, for the program to be crash-free with respect to $\mathcal{T}$, for every state in the first partition reachable in the subautomaton which corresponds to a non-dead state q of B, there is a transition out of the state corresponding to the proposition labelling the state q (this ensures that the next execution step is always identified); and for every state in the second partition which corresponds to a non-dead state of B, every

transition on an action a out of it leads to a state corresponding to a non-dead state of B (this ensures that the next execution step identified can be performed).

A technical challenge is the proof of correctness, in particular, the proof of the fact that if there exists a deterministic crash-free program P which is a subset of $\cup_i L_i$ and has a non-empty intersection with each L_i, then there exists one which corresponds to the language of a subautomaton of an automaton in S. The proof proceeds by choosing a word from $P \cap L_i$ for each i to form a set W, and using the words in W and the automaton $\mathcal{A}'$ to construct a tree T which is such that there exists a subautomaton of the automaton in S corresponding to this tree whose language is a program of the required form.

The following lemma summarizes the correctness and running time of the synthesis algorithm.

Lemma 3. *Let $\mathcal{T}$ be an LTS over $(\mathcal{P}, \mathcal{A})$ and $L_1, \ldots, L_n$ be regular languages in $\mathcal{LT}(\mathcal{P}, \mathcal{A})$. There is a synthesis algorithm that determines exactly whether there exists a deterministic crash-free program X with respect to $\mathcal{T}$ such that $X \subseteq \bigcup_{1 \leq i \leq n} L_i$ and $X \cap L_i \neq \emptyset$ for $1 \leq i \leq n$ and constructs such an X if there exists one. The time complexity of the algorithm is given by $O(n^n(|\mathcal{T}|(\Pi_i|\mathcal{A}_i|))^n 2^{n|\mathcal{T}|(\Pi_i|\mathcal{A}_i|)})$, where $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are complete bipartite automata for $L_1, \ldots, L_n$ respectively.*

4.3 Example

Consider the service advertisment $\mathcal{T}$ in Figure 1 and the example formula φ from Section 3. The constraint set $((\mathcal{T}[\![\varphi]\!]^C))(q_1)$ includes the following transition relation maps as satisfying solutions:

(1) $x \mapsto \{(q_1, q_1), (q_1, q_3), (q_7, q_2), (q_7, q_4), (q_2, q_2), (q_2, q_4)\}$, $y \mapsto \{(q_3, q_7), (q_4, q_7)\}$
(2) $x \mapsto \{(q_1, q_2), (q_1, q_4), (q_2, q_2), (q_2, q_4), (q_7, q_2), (q_7, q_4)\}$, $y \mapsto \{(q_4, q_7)\}$

The two solutions highlight the interdependence of the transition relations assigned to x and y. In solution (1), x includes the transition (q_1, q_3) (corresponding to not logging in) which forces y to include the transition (q_3, q_7) of checking out from the non logged in state q_3. In solution (2), y does not have to include this transition because x never transitions to the state q_3. Deterministic crash-free programs for the transition relation solutions (1) and (2), respectively, are: (1) $x \mapsto \emptyset s(\emptyset + P_s) + \{P_c\}s(\{P_l\} + \{P_s, P_l\}) + \{P_l\}s(\{P_l\} + \{P_s, P_l\})$, $y \mapsto (\{P_s\}l)^*\{P_s, P_l\}a\{P_a\}c\{P_c\}$ and (2) $x \mapsto (\emptyset l)^+\{P_l\} + \{P_l\}s(\{P_l\} + \{P_s, P_l\}) + \{P_c\}s(\{P_l\} + \{P_s, P_l\})$, $y \mapsto \{P_s, P_l\}a\{P_a\}c\{P_c\}$. In addition to y performing a followed by c, note that the requirement of being crash-free forces any necessary logging in to be performed (by y in Solution (1) and x in Solution(2)) to be realized as, *e.g.*, the language $(\emptyset l)^+\{P_l\}$ rather than the simpler language $\emptyset l\{P_l\}$. The execution of the action login could non-deterministically lead either to successful login or failure, and in either case, the program should identify the next action to be taken. The above two transition relation maps are also satisfying solutions for the example formula ψ since there are no loops in the transition relation for x between states in which search is successful, *i.e.*, containing P_s.

The following transition relation map is also a satisfying solution for φ, but no deterministic crash-free programs can realize the transition relation: $x \mapsto \{(q_1, q_1), (q_1, q_3), (q_1, q_2), (q_2, q_2), (q_2, q_4), (q_7, q_2), (q_7, q_4)\}, y \mapsto \{(q_3, q_7), (q_4, q_7)\}$. No deterministic crash-free program can have both (q_1, q_2) and (q_1, q_3) in its transition relation. Any deterministic program, in state q_1, can chose only one of the two actions s and l, and can consequently stop at at either q_3 or q_4 but not both and cannot include both pairs in its transition relation.

5 Related Work

Semantic Web approaches (e.g., [24], [26]) propose extending ontologies (commonly agreed vocabularies) and using first-order logic to specify behavioral properties of services. Other formalisms have been proposed for specifying the goal application's logic [5,21,8,1,20]; however, composition is primarily intended to be a manual process.

Most of the work on automatic composition uses AI planning [28,3] or planning extended to other logics (e.g., Situation Calculus [18,19], Linear Logic [23]). The main difference from the logic $L_{\mu,\langle x\rangle}$ is that these logics permit the specification of a single synthesized program (as opposed to multiple inter-dependent programs) and specifications are limited to using propositional properties of the state (as opposed to properties of the full execution path).

A closely related theoretical line of work follows what is dubbed the "Roman" model and is represented in [5,6,10,25]. A significant difference from our work is that in the Roman model each of the operations of the synthesized service is mapped to a single action and the synthesized programs therefore correspond to a single invocation of a service operation. On the other hand, the programs considered in this paper are arbitrary languages/automata that can invoke multiple operations sequentially in conjunction with tests, branching, and loops. A second important difference is that in the Roman model, the composed service is explicitly given as a finite state machine whereas we consider its requirements to be specified as a formula in a logic. Apart from our queries therefore having a "declarative" rather than "operational" flavor, a more fundamental consequence is that our queries are inherently more expressive because formulas permit the specification of a set of automata as opposed to a single automaton. As an example, the finiteness requirement property (specified by φ_{merc} or ψ in Section 3) corresponds to an infinite set of automata that cannot be specified as any single finite state machine. On the other hand, requiring trace containment with respect to a specified state machine (the problem addressed in the Roman model) can be encoded in our logic, $L_{\mu,\langle x\rangle}$, as a formula using only one ν and $\langle x\rangle$ modality. A final technical difference is that our synthesis algorithm is more general in that we do not assume that nondeterministic choices in the service advertisements are observationally distinguished in their resulting states.

The problem solved in the second step of our algorithm (Section 4.2) is closely related to the problem of controller synthesis for discrete systems first formulated in [22]. Several technical variants of the problem have been studied;

the deterministic requirement on the programs synthesized most closely corresponds to controllers for open systems [17] as opposed to closed systems [14] under a maximal environment. The initial class of synthesis requirements considered [16] were containment in a class of admissible behaviors which corresponds in our setting to requiring certain specified states to be unreachable from the initial states. These results are inapplicable to our problem because the constraints we need to consider additionally specify: (a) states that have to be guaranteed to be reached, and (b) reachability/unreachability conditions starting from arbitrary states (not necessarily initial states). Our problem is most directly reducible to the controller synthesis problem for μ-calculus [2] (CTL^* does not suffice since it cannot express all regular languages). Nevertheless, we chose to directly formulate a new synthesis algorithm because the controller problem for the weaker logic CTL^* is already $3EXPTIME$-complete [17], and a reduction to μ-calculus controller synthesis problem would result in a time-complexity at least one exponential factor worse than given by Lemma 3. The synthesis requirements resulting from $L_{\mu,\langle x\rangle}$, therefore, has an intermediate level of expressiveness that, to the best of our knowledge, has not been addressed previously in the controller synthesis literature.

References

1. OASIS Standard Web Services Business Process Execution Language Version 2.0 (April 2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
2. Arnold, A., Vincent, A., Walukiewicz, I.: Games for synthesis of controllers with partial observation. Theoretical Computer Science 303(1), 7–34 (2003)
3. Aydin, O., Cicekli, N.K., Cicekli, I.: Automated Web Services Composition with the Event Calculus. In: Artikis, A., O'Hare, G.M.P., Stathis, K., Vouros, G.A. (eds.) ESAW 2007. LNCS (LNAI), vol. 4995, pp. 142–157. Springer, Heidelberg (2008)
4. Backhouse, R.: Galois Connections and Fixed Point Calculus. In: Blackhouse, R., Crole, R., Gibbons, J. (eds.) Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. LNCS, vol. 2297, pp. 89–150. Springer, Heidelberg (2002)
5. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic Composition of *E*-services That Export Their Behavior. In: Orlowska, M.E., Weerawarana, S., Papazoglou, M.P., Yang, J. (eds.) ICSOC 2003. LNCS, vol. 2910, pp. 43–58. Springer, Heidelberg (2003)
6. Berardi, D., Mecella, M., Calvanese, D.: Composing web services with nondeterministic behavior. In: IEEE International Conference on Web Services, ICWS 2006 (2006)
7. Bradfield, J., Stirling, C.: Modal mu-calculi. In: Handbook of Modal Logic, pp. 721–756. Elsevier (2007)
8. Casati, F., Ilnicki, S., Jin, L.-J., Krishnamoorthy, V., Shan, M.-C.: eFlow: A platform for developing and managing composite e-services. In: Proc. of the Academia/Industry Working Conference on Research Challenges (AIWORC 2000), Washington, DC, USA (2000)

9. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL) 1.1 (2001), http://www.w3.org/TR/wsdl
10. De Giacomo, G., Sardina, S.: Automatic synthesis of new behaviors from a library of available behaviors. In: Proc. of IJCAI 2007, pp. 1866–1871 (2007)
11. Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000), http://www.ics.uci.edu/~fielding/pubs/dissertation/top.html
12. Fielding, R., Taylor, R.N.: Principled design of the modern web architecture. In: Proc. of 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland (2000)
13. Hadley, M.: Web application description language (WADL). TR-2006-153 (April 2006), https://wadl.dev.java.net/wadl20061109.pdf
14. Jiang, S., Kumar, R.: Supervisory control of discrete event systems with CTL* temporal logic specifications. SIAM J. Control Optim. 44(6), 2079–2103 (2006)
15. Kozen, D., Tiuryn, J.: Logics of programs. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 789–840 (1990)
16. Kumar, R., Garg, V.K.: Modeling and Control of Logical Discrete Event Systems. Kluwer Academic Publishers, Norwell (1999)
17. Kupferman, O., Madhusudan, P., Thiagarajan, P.S., Vardi, M.Y.: Open Systems in Reactive Environments: Control and Synthesis. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 92–107. Springer, Heidelberg (2000)
18. Levesque, H.J., Reiter, R., Lesperance, Y., Lin, F., Scherl, R.B.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming 31(1-3) (1997)
19. Mcilraith, S., Son, T.C.: Adapting Golog for composition of semantic Web services. In: Proc. of International Conference on Principles of Knowledge Representation and Reasoning, KR 2002 (2002)
20. Nanz, S., Tolstrup, T.K.: Goal-Oriented Composition of Services. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 109–124. Springer, Heidelberg (2008)
21. Pathak, J., Basu, S., Lutz, R., Honavar, V.: Selecting and composing web services through iterative reformulation of functional specifications. In: Proc. of 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2006), Washington, DC, USA (2006)
22. Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proceedings of the IEEE 77 (1989)
23. Rao, J., Küngas, P., Matskin, M.: Composition of semantic web services using linear logic theorem proving. Information Systems 31(4-5), 340–360 (2006)
24. Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web service modeling ontology. Applied Ontology 1(1), 77–106 (2005)
25. Sardina, S., Patrizi, F., De Giacomo, G.: Automatic synthesis of a global behavior from multiple distributed behaviors. In: AAAI 2007: Proceedings of the 22nd National Conference on Artificial Intelligence, pp. 1063–1069. AAAI Press (2007)
26. Sycara, K., Paolucci, M., Ankolekar, A., Srinivasan, N.: Automated discovery, interaction and composition of semantic web services. Journal of Web Semantics 1(1), 27–46 (2003)
27. W3C Recommendation. SOAP version 1.2 (April 2007), http://www.w3.org/TR/soap12
28. Wu, D., Sirin, E., Hendler, J.A., Nau, D.S., Parsia, B.: Automatic web services composition using shop2. In: Proc. of World Wide Web Conference (2003)

Compatibility of Data-Centric Web Services

Benoît Masson[1], Loïc Hélouët[2], and Albert Benveniste[2]

[1] Epitech Rennes, 12 square Vercingétorix, 35000 Rennes, France
benoit.masson@epitech.eu
[2] INRIA Rennes, campus de Beaulieu, 35042 Rennes Cedex, France
{loic.helouet,albert.benveniste}@inria.fr

Abstract. Before using a service in a composite framework, designers must ensure that it is compatible with the needs of the application. The inputs and outputs must comply with the intended ranges of data in the composite framework, and the service must eventually return a value. This paper addresses compatibility for modules described with document-based workflow nets, that can depict the semantics of active XML (AXML) systems, a language for Web Services design. The behavior of non-recursive AXML specifications with finite data can be represented as Docnets, i.e., finite labeled Petri nets carrying information on document types they transform. Compatibility of docnet modules is characterized in terms of a decidable reachability property in the underlying net. Finally, we show the distributivity of compatibility over composition, which allows a faster semi-decision algorithm to verify compatibility between sets of modules.

Keywords: Data-Centric Web Services, Composition, Petri Nets.

1 Introduction

E-business and supply chain management involve a combination of widely distributed workflow systems and data/information management. According to [13], these systems can be viewed as workflows, or as information systems. In the workflow-based perspective, process is emphasized. Web services and their orchestrations are now considered an infrastructure of choice for managing business processes and workflow activities over the Web [12]. BPEL has become the industrial standard for specifying orchestrations, and formalisms such as Orc [10] have also been proposed. In the information-based perspective, processes are considered as operations that are triggered as a result of information changes. Information-centric systems typically rely on database-oriented technologies and the notion of (semi-structured) *document*.

Today, technologies in use for these two aspects are mostly separated. The WIDE approach [5] was a first attempt to combine them, and was further developed in [13] to consider process, information, and organization. The notion of "business artifact" has been proposed at IBM as a framework combining workflow and data management [11,8]. Active XML (AXML) [1,2] was proposed as a framework for *document-based workflows*. It consists of XML documents with embedded guarded service calls and offers mechanisms to store and query semi-structured data distributed over entities called *peers*. In [3,7] distribution was

M. Carbone and J.-M. Petit (Eds.): WS-FM 2011, LNCS 7176, pp. 32–47, 2012.
Springer-Verlag Berlin Heidelberg 2012

explicitly introduced in AXML, thus giving rise to the model *Distributed AXML* (DAXML). In DAXML, guards are local to a peer, services can be local or distant, and in the latter case specified through an *interface*. This allows to reason about a DAXML system, either globally, or locally by representing distant service calls by their interfaces. However, the notion of interface proposed in [7] only specifies data exchanged during distant calls. Implementing an interface then means accepting and returning correct data, but does not guarantee that a distant call eventually returns a value.

This paper proposes a framework for document-based workflows, i.e., workflows that circulate documents and whose transitions are instantiated and guarded by documents. Transitions of the workflow are guarded service calls and returns (the pair (call; return) is not atomic). Documents are business processes comprising both data and references to the services needed to process the data. Typically, documents are meant to be semi-structured data of XML type obeying DTDs. However, unlike in the above-mentioned works where the mechanisms of querying documents were explicitly considered, we will abstract away from any XML-related pattern matching mechanism. We first propose a generic model called *Docnets*. These nets are finite Petri nets whose places are typed by document types, and whose transitions transform documents, or model calls to distant websites. Docnets can be equipped with a composition operator, which allows designing the evolution of documents owned by an agent of a system that offers services to its environment. Such pair of document processing plus environment will be called a *module*. Modules can then be assembled to model the fact that an agent provides services to another agent. However, such composition can only occur if the required services (interfaces) and the provided services are compatible. We show that assuming that the data exchanged between docnet modules can be typed by a finite set of documents, and that services are not recursive, compatibility of services and termination of distant calls is decidable, even for Docnets which may treat an unbounded number of documents.

The paper is organized as follows: Section 2 describes the concepts of active documents. Then Section 3 introduces Docnets, their properties and two composition operators. Sections 4 and 5 propose two notions of compatibility for docnet modules, and shows that they are decidable. Due to lack of space, proofs are omitted but can be found in an extended version of this work [9].

2 Distributed Active Documents

Web Services architectures based on active documents were introduced by [1,2]. The concepts of this framework can be illustrated by the following example. Consider a website that provides information on current weather in a panel of cities in the world. This site returns XML-like documents that carry the name of a city, current temperature and weather, plus a forecast for tomorrow. The main function of this site is to return current data for a city, so for many clients of the service, the forecast is useless. *Active documents* embed references to services that can be called to extend the data in the document. In our weather website,

```
<weather>
  <city> Paris </city>
  <current>
      <sky> sunny</sky>
      <temp>  24 </temp>
  </current>
  <forecast>
      <sky> sunny </sky>
      <temp>  20 </temp>
  </forecast>
</weather>
```

```
<weather>
  <city> Paris </city>
  <current>
      <sky> sunny</sky>
      <temp>  24 </temp>
  </current>
  <forecast>
      <service> www.meteofrance.fr/
                  getForecast?query=Paris
      </service>
  </forecast>
</weather>
```

Fig. 1. A document and an active document returned by a website

the returned value can embed a reference to a new service that can be called by a client to get the forecast, or simply ignored (this is illustrated in Figure 1).

A system composed of active documents is distributed over a set of agents. Each agent possesses local services, and references to services provided by other agents, called external services. Local services are functions that transform documents in two phases: first the service is called, and then the service returns. Calls and returns are guarded — for example, in AXML the guards are expressed using a fragment of Xpath [2]. Calls and returns modify the document owned by an agent in a deterministic way. Due to the mechanism of guards, services do not return immediately after a call, thus, complex workflows can be designed. In addition to local services, agents possess references to services provided by other agents, and offer some of their services to the external world. When seen from a given agent, an external service is simply perceived as a way of mapping a possible set of input parameters to a possible set of output parameters. Calling an external service consists in sending data to another agent and waiting for its answer. Offering a service consists in receiving a new document, transforming it using some local services, and returning the transformed document to the caller.

In this work, we will abstract away from concrete documents and query languages and will only assume that checking that a guard is satisfied is an effective procedure. Our model of active documents is introduced next.

3 Modeling Active Documents Workflows with Petri Nets

We begin with background material on Petri nets. A *labeled net* (LPN) is a tuple (P, T, F, L, ℓ) where P is the set of *places*, T is the set of *transitions*, $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation* seen as a set of (directed) *arcs* between places and transitions, L is the set of *labels*, and $\ell : T \to L$ is the *labeling function*. For any node $n \in P \cup T$, its *preset* and *postset* are defined as ${}^\bullet n = \{x \mid (x, n) \in F\}$ and $n^\bullet = \{x \mid (n, x) \in F\}$, respectively. The "state" of a labeled net is represented by a *marking*, i.e., a mapping $m : P \to \mathbb{N}$,

where $\mathbb{N}$ is the set of non-negative integers. For $p \in P$, the value $m(p)$ is the number of *tokens* in place p. For m, m' two markings and t a transition, say that firing t from marking m is possible and produces marking m', denoted $m[t\rangle m'$ if $\forall p \in {}^\bullet t, m(p) \geq 1$ and if $m'(p) = m(p)-1$ when $p \in {}^\bullet t \backslash t^\bullet$, $m'(p) = m(p)+1$ when $p \in t^\bullet \setminus {}^\bullet t$, and $m'(p) = m(p)$ otherwise. A *labeled Petri net (LPN)* is a tuple (P, T, F, L, ℓ, m_0) where (P, T, F, L, ℓ) is a labeled net and $m_0 : P \to \mathbb{N}$ is the *initial marking*. A *firing sequence* is a bipartite sequence $m_0, t_1, m_1, t_2, m_2, \ldots$ such that $m_{i-1}[t_i\rangle m_i$. Say that m' is reachable from m, if there exists a firing sequence starting from marking m and ending in marking m'.

3.1 Documents and Services

Petri nets alone are not sufficient to describe document manipulation. They just consume and produce untyped tokens. A solution to increase the expressive power of Petri nets is to add colors to tokens and control flows, but this usually results in undecidability of many problems. In this paper, we adopt a different technique. We add types to places (that is places will represent types of documents), and consider transitions as actions that transform documents. Tokens in a place represent documents of a type defined by the place.

We assume a (non necessarily finite) set $\mathcal{S}$ of *services* and *interfaces* names, and a set of *service marks* of the form $f!$ or $f?$ for all $f \in \mathcal{S}$. In a document, a service mark $f!$ means that service f can be called. Similarly, a service mark $f?$ means that service f was called and a return is awaited. To simplify the model, we will assume that every document that our systems manipulate contains at most one occurrence of each service mark. Note however that the model can be easily extended to documents carrying a bounded number of service marks.

Let us denote by $\mathcal{D}$ all types of documents (this set is not necessarily finite). We denote by $\sigma : \mathcal{D} \mapsto 2^{\mathcal{S}\times\{!,?\}}$ the labeling map that associates to every document type $d \in \mathcal{D}$ the service marks that appear in this document, i.e., the service calls and returns that may occur from this document provided guards of the considered services hold. Additional (unspecified) marked services or information may also occur in this document. We assume that $\mathcal{D}$ is equipped with a partial order relation $\leq$ such that for all $d, d' \in \mathcal{D}$, $d \leq d'$ implies $\sigma(d) \subseteq \sigma(d')$. Two document types d and d' are called *comparable* if either $d \leq d'$ or $d' \leq d$ holds. We furthermore suppose that it is decidable whether $d \leq d'$. For two sets of document types D, D', we will say that $D \leq D'$ if for every $d \in D$, there exists $d' \in D'$ such that $d \leq d'$.

A *service* is a tuple (f^c, G_f^c, f^r, G_f^r), where $f \in \mathcal{S}$ is the name of the service, f^c and f^r are the service *call* and *return* functions, with G_f^c and G_f^r the corresponding *guards*. Mappings $f^c, f^r : \mathcal{D} \mapsto \mathcal{D}$ are partial and map document types to document types, and $G_f^c, G_f^r : 2^{\mathcal{D}} \mapsto \{\mathrm{T}, \mathrm{F}\}$ are boolean predicates over sets of document types. Write $D \models G_f^c$ (resp. $D \models G_f^r$) if any set of documents with types $D = \{d_1, \ldots, d_k\}$ satisfies G_f^c (resp. G_f^r). We also assume that guard satisfaction is monotonous w.r.t $\leq$, i.e., if $D \models G_f^c$ and $D \leq D'$, then $D' \models G_f^c$.

3.2 Docnets

Definition 1. *A* docnet *is a tuple* $\mathcal{N} = (P, T, F, L, \ell, m_0, S, D, \ell')$ *where* $S \subseteq \mathcal{S}$ *is a finite set of service and interface names,* $D \subseteq \mathcal{D}$ *,* $L = \big(S \times \{c, r\}\big)$*,* $\ell' : P \mapsto D$ *associates a type to each place of* P *and* (P, T, F, L, ℓ, m_0) *is a LPN.*

Intuitively, places in docnets represent *types of documents* involving a finite set of service marks. This is captured by a chain of labeling maps $P \xrightarrow{\ell'} \mathcal{D} \xrightarrow{\sigma} 2^{S \times \{!,?\}}$ A token in a place represents an instance of a document of the given type. In particular, this document must contain the service marks of the document type associated to the place (plus possibly some additional information). A place can hold any non-negative number of tokens of the referred type. Figure 2 represents a partial docnet. Places are represented by circles, and labeled by the service marks provided by the σ map. Transitions are represented by rectangles with labels at their side. The dashed arrows indicate that the net contains more places and transitions that are not represented. The initial marking is symbolized by dark circles located in the initially marked places. For the sake of readability, we will not discuss the role of interfaces immediately and introduce them only at the end of this section.

As we want docnets to model guarded transformations of documents, we need to impose some consistency between transitions labeling, places labeling, and the transformations attached to transitions.

Definition 2 (well-formedness). *A docnet* $\mathcal{N}$ *is* well-formed *iff:*

1. *Every transition labeled* f^c *(resp.* f^r*) possesses in its preset a place having* $f!$ *(resp.* $f?$*) as part of its* σ*-label and in its postset a place whose* σ*-label has* $f?$ *substituted for* $f!$ *(resp. has* $f?$ *removed from it).*
2. *Any subset of places* P' *such that:*
   - *there exists a place* $p \in P'$ *with* $f!$ *(resp.* $f?$*) as part of its* σ*-label*
   - $\ell'(P')$ *satisfies the guard* G^c_f *(resp.*G^r_f*), and no subset* $P'' \subset P'$ *satisfies it.*

   possesses in its postset a transition t *labeled* f^c *(resp.* f^r*). In addition, places in* $P' \setminus \{p\}$ *have* t *in their preset.*
3. *Every transition labeled by* f^c *(resp.* f^r*) possesses a places* p *in its preset and* p' *in its postset such that* $\ell'(p') = f^c(\ell'(p))$ *(resp.* $\ell'(p') = f^r(\ell'(p))$ *).*

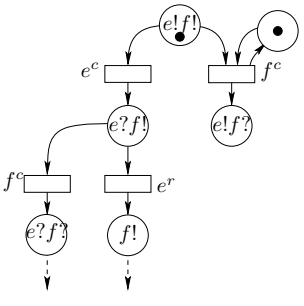


Fig. 2. A well-formed docnet

Condition 1 means that a service can be activated only if it was mentioned in some document in the preset of a transition; activating this service results in a modification of the document. Condition 2 defines guards for docnets. Observe that well-formed docnets may specify services that return documents containing additional service marks of the form $f!$, thus allowing new service calls. Condition 3 indicates that a transition labeled by a service call or return transforms a document into another one according to the function labeling the transition. A consequence of these well-formedness rules is that transitions always have at least one place in their postset, and that subsets of places from which a service can be called (or can return) have a common transition in their postset. A well-formed docnet can thus be infinite. However, if no recursion among services occur, the docnet associated to the evaluation of a given document with local services (i.e., the replacement of all services by the actual data they represent in documents) is necessarily finite. It can be represented as the smallest well-formed docnet containing the initially marked places depicting the document under evaluation.

When a docnet is not well-formed, one can nevertheless define a closure operation to make it a well-formed docnet. For $\mathcal{N}$ a finite docnet defined over a set S of services, the closure operation $close(\mathcal{N})$ is defined as follows: for every service $f \in S$, for every place p such that $f! \in \sigma(\ell'(p))$ and every minimal subset P' of places of $\mathcal{N}$ such that $\ell'(P') \models G_f^c$, add a new transition t labeled by f^c, and a new place p' in the postset of t such that $\ell'(p') = f^c(\ell'(p))$ (if such transition/place does not already exist). Add t to the postset of each place in $\{p\} \cup P'$ and in the preset of each place in $\{p'\} \cup P'$. For every service $f \in S$, for every place p and minimal subset P' of places of $\mathcal{N}$ such that $f? \in \sigma(\ell'(p))$ and $\ell'(P') \models G_f^r$, add a new transition t labeled by f^r, and a new place p' in the postset of t such that $\ell'(p') = f^r(\ell'(p))$ (if such transition/place does not already exist). Add t to the postset of each place in $\{p\} \cup P'$ and in the preset of each place in $\{p'\} \cup P'$. We can now define the well-formed closure *wf-closure*$(\mathcal{N})$ as the limit *wf-closure*$(\mathcal{N}) = \lim_{n \mapsto \infty}$ *wf-closure*$^n(\mathcal{N})$, with:

$$\textit{wf-closure}^0(\mathcal{N}) = \mathcal{N} \quad \text{and} \quad \textit{wf-closure}^n(\mathcal{N}) = \textit{close}(\textit{wf-closure}^{n-1}(\mathcal{N}))$$

The *wf-closure* of a docnet $\mathcal{N}$ may not be finite if some services of S are recursively called. For N a finite docnet with finite set of places P, set of document types $D = \ell'(P)$ and set S of services, if there exists no $k \leq |S|$ and no f such that $f! \in (\sigma(D) \cap \sigma(D^k \setminus D))$ where D^k is the set of document types occurring in *wf-closure*$^k(N)$, then the *wf-closure* of $\mathcal{N}$ is finite. Note that this is only a sufficient condition, and that due to markings, recursion might never occur. The closure operation for a net without transitions is a well-formed net.

Docnets describe the evolution of documents through embedded services evaluation. Of course an agent can only use local services that it implements. This is captured by the notion of *peer*: a peer is a pair $(\mathcal{N}, S)$, where S is the set of services that are accessible by the agent, and $\mathcal{N}$ is a well-formed docnet w.r.t. services of S, which services calls and returns all belong to S. Adding new documents to a system may allow new service calls or returns by making their guards true. Composing workflows of disjoint documents is then more than the simple union of their nets, and is defined by the parallel composition operation below.

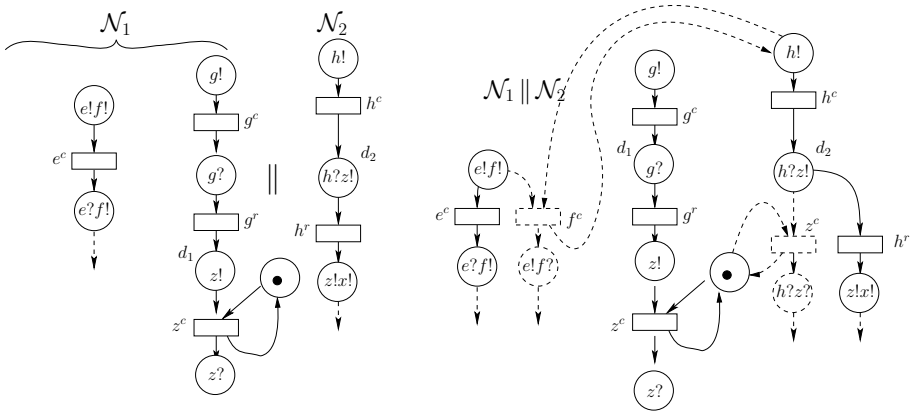


Fig. 3. Parallel composition of docnets

Definition 3 (Parallel composition). *Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two well-formed docnets. Their* parallel composition $\mathcal{N}_1 \,\|\, \mathcal{N}_2$ *is obtained as follows:*

1. *Compute the disjoint union of the underlying nets, seen as graphs, i.e., compute the disjoint union of places, transition, initial markings and flows.*
2. *Make the resulting docnet well-formed by wf-closure.*

The parallel composition preserves the behaviors of both nets. It also allows more behaviors, by synchronizing the filling of places, and then by performing the well-formed closure of the so obtained system. The main intuition behind parallel composition is that when a place with document type d_2 is filled in net $\mathcal{N}_2$, and when there exists a place with type $d_1 \leq d_2$ in net $\mathcal{N}_1$, then anything allowed due to the presence of a token of type d_1 in $\mathcal{N}_1$ should be allowed as soon as a new token arrives in place of type d_2. Figure 3 is an illustration of parallel composition. Note that in $\mathcal{N}_2$ alone, firing z^c is not possible, as no place in $\mathcal{N}_2$ satisfies the guard for z^c. Composing with $\mathcal{N}_1$ provides the needed guard. This hence results in appending a transition labeled by z^c. A similar situation holds for transition labeled by f^c. The places, transitions and flows added at step 2 are represented with dashed lines. Parallel composition is commutative, associative, but not idempotent (composing a docnet with itself results in a larger net).

3.3 Modules and Interactions with the Environment

A site in web service architectures is an open system: it accepts incoming service calls from its environment, and also expects the environment to provide services, known only through a web address (URI), accepted inputs and returned outputs. Even if a site never produces documents of some type, external calls or returns from distant services may involve documents of this type. Hence interacting with an environment may validate guards of services that would not

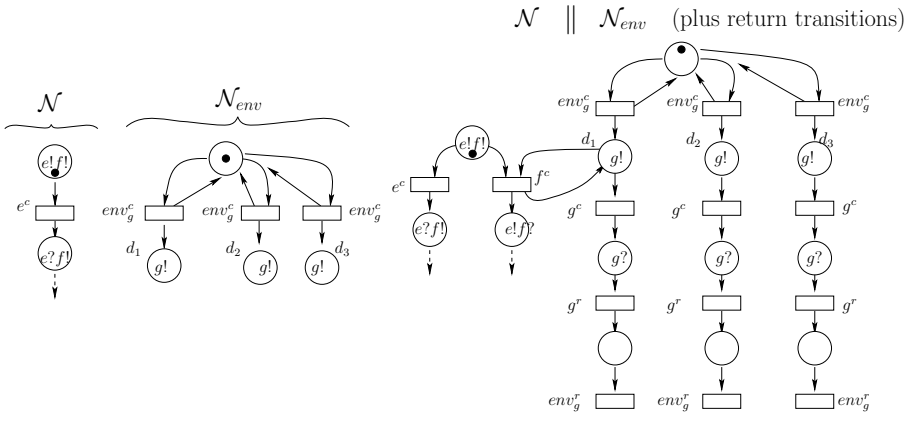


Fig. 4. A docnet providing service g with $D_g = \{d_1, d_2, d_3\}$ to its environment

be enabled otherwise. It is thus worth augmenting a docnet by a model of all demands coming from its environment, and all interactions it may have with distant services. This is achieved by adding a model for *interactions with the environment* to the considered docnet.

Consider a docnet $\mathcal{N}$ and a service g it provides to its environment. Let $D_g = \{d_1, \ldots, d_n\}$ be the set of (valid) document types that are allowed as parameters for a call to g. We assume D_g to be finite, and that each d_i embeds a call to g. Exposing the pair (g, D_g) to the environment is described as the parallel composition of $\mathcal{N}$ with the well-formed closure of a docnet $\mathcal{N}_{\text{env}}$ that contains n places $P^g_{\text{env}} = p^g_1, \ldots, p^g_n$ with respective types $d_1, \ldots, d_n$, plus a transition $t^{d_i}_{g,env}$ labeled by env^c_g for each allowed parameter d_i, with p^g_i in its postset. We also add to $\mathcal{N}_{\text{env}}$ a place p_{env} of type $init_{p_{env}}$, which has each $t^{d_i}_{g,env}$ in its postset and preset. Though this place does not change fundamentally the behavior of the environment net, it allows to control the environment if needed. As a result of step 2 of definition 3, constructing $\mathcal{N} \,\|\, \mathcal{N}_{\text{env}}$ unfolds both $\mathcal{N}$ and $\mathcal{N}_{\text{env}}$.

Arrival of new calls is symbolized by the set of transitions $t^{d_1}_{g,env}, \ldots, t^{d_n}_{g,env}$, with respective postsets $p^g_1, \ldots, p^g_n$. However, termination of calls is not yet modeled in $\mathcal{N} \,\|\, \mathcal{N}_{\text{env}}$. We symbolize this termination by adding a transition t_p labeled by env^r_g from every place p accessible from some p^g_i in P^g_{env} such that $\{g?, g!\} \cap \sigma(\ell'(p)) = \emptyset$ (which means that the call to service g was completed). This construction is illustrated in Figure 4, for a net $\mathcal{N}$ allowing environment calls to service g with parameters $\{d_1, d_2, d_3\}$. This modeling of environment can be extended to an arbitrary number of services in S with their call parameters.

Interfaces and their implementation. Web services are often orchestrations of local services, or services provided by other sites. At design time, these latter are usually known only as interfaces, that depict the parameters sent to a service that implements this interface, and the expected possible values returned by an implementation. In addition to usual service transitions, we allow for interface

transitions. We will differentiate interfaces from services by writing their name in capital. Interface transitions will be simply labeled by I^c and I^r, denoting a call to an external service and a return. Like services, a call to an interface updates a document. It also sends parameters to the called site. The return from a distant call may be of several types, and hence distant call returns can not be modeled as for services. An *interface* is a pair $I = (params_I, D_I)$, where $params_I : \mathcal{D} \mapsto \mathcal{D}$ is a function that extracts parameters of a call from a document, and D_I is a *finite* set of document types depicting the expected returned values after a call to a distant service implementing interface I. At design stage, interfaces need not be implemented, and calls and returns can be represented as transition labeled by I^c and I^r that are fireable from any place with $I!$ in $\sigma(\ell'(p))$ (resp. $I?$ in $\sigma(\ell'(p))$). When no implementation is known, firing an interface changes a tag in a document from $I!$ to $I?$ indicating that an external service is being processed. Unlike services, when a return from an interface call occurs at a place with type d such that $I? \in \sigma(d)$, the effect of receiving an answer on d depends on the received value. Hence for every place with type d such that $I? \in \sigma(d)$, we will create one transition labeled I_i^r per document type d_i in D_I. As for services, we will assume that the effect of receiving an answer of type d_i from a document d is computable and deterministic, and results in a new document type $d +_I d_i$ (operation $+_I$ usually inserts document d_i into d at correct place). Hence, a transition labeled by I_i^r takes a token in a place of type d, and necessarily outputs a token in a place p' of type $d +_I d_i$. I^c and I^r transitions are not guarded: external service can always be called to enrich a document, and the answer can be returned at any moment after a distant call.

Figure 5 shows a docnet with a non-implemented interface I that accepts two return types. Non-implemented interfaces refer to functionalities provided by the environment. Defining the kind of document that can be returned is sometimes sufficient to study properties of a docnet in *any possible environment*: if a document type d is not reachable from an initial marking of a net with non-implemented interfaces, then this document type is not reachable either when interfaces are implemented by services returning only expected values. To keep well-formedness in presence of interfaces, we add a rule to definition 2:
(4) for every place p such that there exists an interface I with $I? \in \ell'(p)$ then there exists a set $t_1, ...t_{|D_I|}$ of transitions labeled by $I_i^r, i \in \{1, .., |D_I|\}$ in the postset of p, and each $t_i, i \in \{1, .., |D_I|\}$ has a place of type $\ell'(p) +_I d_i$ in its postset.

A peer whose docnet models interactions with the environment and contains interface calls and returns can be seen as a module. Once modules are defined, the remaining task is to compose them, that is connect interfaces with services of other modules that implement them.

Definition 4. *A* docnet module *is a triple $M = (\mathcal{N}, S, \mathcal{F}, \mathcal{I})$, where $(\mathcal{N}, S)$ is a peer, $\mathcal{F} \subseteq S$ is a set of pairs of the form (f, D_f) depicting services proposed to the environment and their call parameters, $\mathcal{I}$ is a set of interfaces of the form $(params_I, D_I)$. We require that for every $(f, D_f) \in \mathcal{F}$, the environment part of $\mathcal{N}$ models external calls to every service f with parameters D_f.*

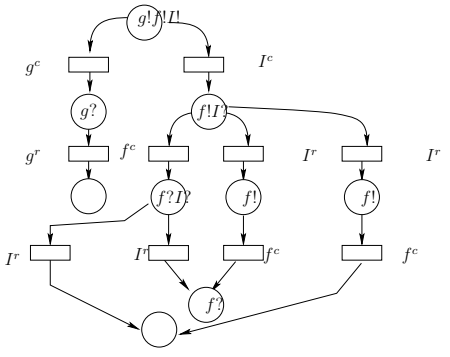


Fig. 5. A docnet with a non-implemented interface

4 Composability

Composability of a service and an interface ensures that modules agree on the exchanged data during a call. Algorithms to check such property are important for distributed systems, since it allows component-based design and orchestration of services. A notion of composability for document-based workflow systems was already proposed in [3,7].

Definition 5. *Let $\mathcal{N}$ be a docnet, and p be one of its places. The* subnet *of $\mathcal{N}$ connected to p is the restriction of $\mathcal{N}$ to set of places P and transitions T, where P,T are the solutions of fixed point equation $(P^n, T^n) = (P^{n-1}, T^{n-1})$ where:*

$$P^0 = \{p\} \qquad P^i = \{p \in P \mid \exists t \in T^{i-1}, p \in t^\bullet \text{ and } p \notin {}^\bullet t\}$$
$$T^0 = \{t \in p^\bullet \mid p \in P^0\} \quad T^i = \{t \in p^\bullet \mid p \in P^{i-1}\}$$

The subnet connected to a place depicts the successive evolutions of a document, forgetting details about guards needed to launch service calls or returns. We will use subnets to characterize control flows associated to external calls.

Definition 6. *Let $\mathcal{N}$ be a docnet, f be a service, and d be a document type such that $f! \in \sigma(d)$. Let $\mathcal{N}' = \mathcal{N}||\mathcal{N}_d$, where $\mathcal{N}_d$ is a net comprising a single place p_d of type d. The* execution *of f at $\mathcal{N}$ from d is the subnet connected to p_d in $\mathcal{N}'$. The* return places *of this execution are places p such that $f?, f! \notin \sigma(\ell'(p))$.*

Intuitively, the execution of f at $\mathcal{N}$ from d depicts the control flow followed by an external call to service f with parameters d arriving at a site that provides service f. Return places are the result of the execution of a request to f starting from parameter of type d.

Definition 7. *Let M_1, M_2 be docnet modules, and $I = (params_I, D_I)$ be an interface of M_1. We will say that service f of M_2 is* composable with interface I at place p *in M_1 if and only if:*

- *$I! \in \sigma(\ell'(p))$: an external call to a service depicted by interface I in $\mathcal{N}_1$ can be performed from place p.*

- *M_2 accepts call to service f with parameter d=$params_I(\ell'(p))$, i.e., $(f, D_f) \in \mathcal{F}_2$, with $d \in D_f$.*
- *Every return place connected to place p_d in the net $\mathcal{N}_2||\mathcal{N}_d$ has a type in D_I.*

We will say that service f of M_2 is composable *with interface I of M_1, and write $f \Vdash I$, if it is composable with I at all places p of $\mathcal{N}_1$ such that $I! \in \sigma(\ell'(p))$.*

Note that composability of a service and its interface does not mean that the called service always terminates. It ensures that parameters of calls are accepted, that returned values are defined in the interface, but not that a return place is eventually filled, nor that such place exists (this might for instance be the case if executing f^c needs a guard that is never true at the called site). Section 5 addresses termination of external service calls. Composability of an interface $I = (params_I, D_I)$ and of a service f that accepts input parameters D_f is decidable when the type of all places in the preset of transitions labeled by I^c is finite (this ensures finiteness of call possibilities as $params_I(\ell'(^\bullet(l^{-1}(I^c))))$ is finite) and the environment part of $\mathcal{N}_2$ is finite. In particular, if $\mathcal{N}_1$ and $\mathcal{N}_2$ are finite, composability of I and f is decidable. See [3] for a full proof in the DAXML context.

All AXML concepts can be mapped to Docnets. An AXML document is a finite XML document, depicted by a document type in a docnet. AXML works by calling services, and returning separately the results. Calls and returns are guarded queries, that is deterministic computations of a set of values from a finite XML database. Guards are defined with a fragment of Xpath, and their evaluation is also an effective procedure. However, negation of Xpath expressions to test absence of some data breaks monotony (i.e., we do no necessarily have $d \models g$ and $d \leq d' \Rightarrow d' \models g$). To preserve monotony, which is essential for parallel composition and modeling interactions with the environment, we have to restrict to *positive* guards that can only test the presence of some data or pattern on documents. Last, an AXML service can return references to other services to be called, hence yielding recursion. However, one can easily avoid recursion by forbidding cyclic dependencies among calls and returned values embedding services. Hence as far as positively guarded document transformations are concerned, docnets and AXML systems are equivalent models. The remaining question is then the modeling of an environment. In docnets, we assume that every service proposed to the environment can be called with a finite set of parameters, described as a finite set of document types. Such a restriction does not exist in AXML, but could be enforced using a DTD to filter external calls. We refer to [3] for a complete semantic mapping between AXML and Docnets.

5 Compatibility between Modules

In this section, we go beyond composability and address the termination of distant calls. We study two different notions of "behavioral" compatibility between modules, namely weak and strong compatibility. The weak notion allows the reception of particular environment calls (i.e., firing of transitions labeled env_f^c for some $f \in \mathcal{F}$) that may unblock the treatment of a distant call, while the

strong one should complete requests in any environment. We then show that for finite modules, these properties are decidable (Theorem 2). We also show that compatibility "distributes" over the composition of modules (Theorem 4), which leads to a faster semi-algorithm to decide compatibility. The work in [4] considers a close notion for session types. Starting from a global specification, the problem is to ensure that a distribution on distant sites allows termination and correct typing of returned values. We emphasize that in our model, services distribution is already performed, which leverages a part of the problem addressed in [4].

Definition 8 (compatibility). *Consider a module M_1 accepting environment calls to $f \in \mathcal{F}$, and a module M_2 owning an interface I such that $f \Vdash I$. M_1 and M_2 are* weakly (I, f)-compatible *(denoted $M_1 \ {}_I\!\lhd_f \ M_2$) if and only if after **some** environment calls to $\mathcal{F}$, any firing of a transition env_f^c with parameters allowed by I is eventually followed by a corresponding response env_f^r.*

M_1 and M_2 are (strongly) (I, f)-compatible *(denoted $M_1 \ {}_I\!\lhd_f \ M_2$) if and only if after **any** environment calls to $\mathcal{F}$, any firing of a transition env_f^c with parameters allowed by I is eventually followed by a corresponding response env_f^r.*

Clearly, $M_1 \ {}_I\!\lhd_f \ M_2$ implies $M_1 \ {}_I\!\lhd_f \ M_2$. Let us now prove that these notions of compatibility are decidable. The two notions mean that an external call to a service can terminate with or without the help of its environment. If we consider a marking of the docnet depicting behaviors of module M_1, and if we isolate a token in a parameter place p_d filled by a transition labeled by env_f^c, we should be able to find a reachable marking in which **this** token can be consumed by a transition labeled with env_f^r. "Isolating a token" can be modeled by adding to $\mathcal{N}_1$ (with the parallel composition operator) a copy of the net $\mathcal{N}_{params_I}$ associated to the processing of a the call parameters of I, which minimal place can be fed only once. We can then connect $\mathcal{N}_1$ to $\mathcal{N}_{params_I}$ in two different ways: in the first way, the first transitions (transitions labeled by env_f^c in $\mathcal{N}_{params_I}$) consume the token of place p_{env} in the environment part of $\mathcal{N}_1$. This modeling prevents any incoming call from the environment once a document is being processed in $\mathcal{N}_{params_I}$. Call this net $\mathcal{N}_{strong}$. The second solution is to let the $\mathcal{N}_1$ and $\mathcal{N}_{params_I}$ run in parallel, hence allowing environment calls while a document is being processed in $\mathcal{N}_{params_I}$. Call this net $\mathcal{N}_{weak}$.

Then, we can show that weak and strong (I, f)-compatiblity amounts to verifying some home-space property [6] respectively in $\mathcal{N}_{weak}$ and $\mathcal{N}_{strong}$ (complete proof is detailed in [9]). Note that an environment call that does not terminate in general, may terminate when restricted to parameters D_I. So, the fact that a particular environment call does not return in general does not imply that two modules are not (I, f)-compatible. Note also that deciding a home-space property relies on reachability in Petri nets, and can hence be a costly operation.

Theorem 1. *Consider two modules M_1 and M_2. Let I be an interface of M_2 and f a service of M_1 such that $f \Vdash I$. If $\mathcal{N}_1$ and $\mathcal{N}_2$ are finite, strong (I, f)-compatibility and weak (I, f)-compatibility are decidable.*

Usually, module composition involves more than one pair service/interface. A module M_1 can provide some services to M_2, but at the same time expect some

functionalities (expressed as interfaces) that are implemented in M_2. Interfaces and services are paired via explicit mappings specified by the designer. A *pairing map* ξ is a mapping from some interfaces of a module to services provided by another module, with the constraint that $\xi(I) \Vdash I$ for all $I \in \text{dom}(\xi)$.

We can now define compatibility of two modules with respect to a composition schema defined by a pairing map. The modules M_1 and M_2, with respective sets of external services $\mathcal{I}_1$ and $\mathcal{I}_2$, are *strongly* [resp., *weakly*] *compatible* with respect to a pairing map ξ if and only if for all $I \in \text{dom}(\xi) \cap \mathcal{I}_1$, $M_1 \ I \lhd \xi(I) \ M_2$ [resp., $M_1 \ I \lesssim \xi(I) \ M_2$], and for all $I \in \text{dom}(\xi) \cap \mathcal{I}_2$, $M_2 \ I \lhd \xi(I) \ M_1$ [resp., $M_2 \ I \lesssim \xi(I) \ M_1$]. Strong and weak compatibility are denoted respectively $M_1 \overset{\xi}{\bowtie} M_2$ and $M_1 \overset{\xi}{\approx} M_2$ (the symbol ξ may be omitted when it is clear from the context). The decidability result of compatibility of services and interfaces can be easily extended to modules and pairing maps (complete proof in [9]).

Theorem 2. *Let M_1, M_2 be two docnet modules, with finite docnets. Then, compatibility and weak compatibility of modules are decidable.*

5.1 Connecting Interfaces and Their Implementations

Let us consider two docnet modules M_1 and M_2 such that M_1 comprises a non-implemented interface I and M_2 a service f with $f \Vdash I$. As M_1 and M_2 represent distinct sites that communicate through invocations, the document types manipulated in a module should not be used to satisfy guards in the other module. We will hence consider that distinct modules are defined over distinct and incomparable document types. The *composition* of M_1 and M_2 under mapping (f, I) is denoted by $M_1 \bigotimes_{I,f} M_2$ and consists in a new module $M = (\mathcal{N}', S_1 \cup S_2, \mathcal{F}_1 \cup \mathcal{F}_2, \mathcal{I}_1 \setminus \{(params_I, D_I)\} \cup \mathcal{I}_2)$, where $\mathcal{N}'$ is the docnet computed as follows:

- Compute $\mathcal{N} = \mathcal{N}_1 || \mathcal{N}_2$
- Compute $\mathcal{N} || \mathcal{N}_{d_1} || \dots || \mathcal{N}_{d_k}$ for every $d_i, i \in \{1, .., k\}$ such that there exists a transition t_i in $\mathcal{N}$ with label I^c, and a place $p \in {}^\bullet t_i$ with type d such that $params_I(d) = d_i$.
- Connect every transition t_i in $\mathcal{N}_1$ to place p_{d_i} in $\mathcal{N}_{d_i}$ (i.e., set $t^\bullet = t^\bullet \cup p_{d_i}$).
- Connect every return place p_j of type d_j in a net $\mathcal{N}_{d_i}$ to every transition t'_i labeled by I^r_j in the subnet connected to the transition t_i that feeds place p_{d_i} in $\mathcal{N}_1$ (i.e., set $t'_i \in p_j^\bullet$).
- Remove from $\mathcal{N}_1$ all transitions t'_i labeled by I^r_j in the subnet connected to the transition t_i that feeds place p_{d_i} such that no return place carries type d_j in the subnet connected to p_{d_i}.

Note that as $\mathcal{N}_2$ already accepts calls from the environment to service f, adding a request from $\mathcal{N}_1$ to execute service f does not add new document types to $\mathcal{N}_2$. Figure 6 below illustrates an implementation of an interface I by a service h. The interface can call h with a single parameter type, and accordingly, h can return two results. In this drawing, the added arcs are represented by dashed

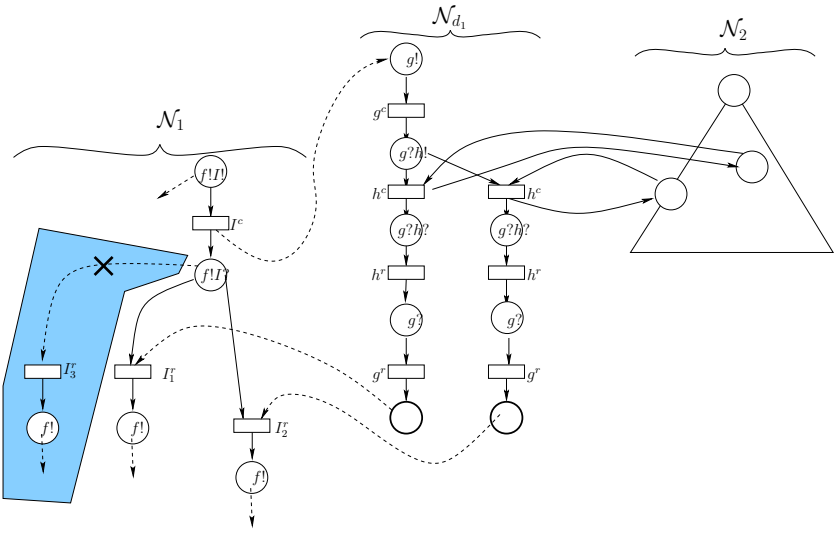


Fig. 6. Composition of modules

lines, return places by thick circles, and the removed part of the net by a gray zone. Transition I_3^r cannot be fired anymore in the composition, as return type d_3 is never produced when executing h. This construction extends in an obvious way to an arbitrary number of modules and arbitrary pairing maps, and we will denote by $M_1 \bigotimes_\xi M_2$ the composition of two modules under pairing map ξ (the ξ symbol may be omitted when the map is clear from context). Slightly abusing the notation , we will also write $\bigotimes_{i \in K} M_i$ to denote the composition of a set of modules $\{M_i \mid i \in K\}$ with appropriate pairing maps.

Theorem 3. *Let $(M_i)_{i \in K}$ be a finite family of modules, and let $M' = (M_1 \bigotimes_{\xi_1} M_2) \bigotimes_{\xi_2} \cdots \bigotimes_{\xi_{k-1}} M_k))$. Let m be a non-reachable marking of $\mathcal{N}_1$. Then for every reachable marking m' in the docnet of M', the restriction of m' to places of $\mathcal{N}_1$ differs from m.*

This property can be used to check for local safety properties of modules. The proof of this theorem is rather straightforward, as assembling modules does not create documents that were not already considered in the environment.

5.2 Distributivity of Compatibility

Compatibility has an interesting property: if several modules are pairwise-compatible, then any of their compositions are also compatible. This is useful because it allows a faster semi-algorithm to decide whether a large set of modules is compatible, by checking compatibility between pairs of modules only.

Theorem 4. *Let $(M_i)_{i \in K}$ be a finite family of modules. For any disjoint sets $K_1, K_2 \subseteq K$ and any pairing maps defined over disjoint domains, if all modules M_i are pairwise-compatible, then $(\bigotimes_{i \in K_1} M_i) \bowtie (\bigotimes_{j \in K_2} M_j)$.*

Note that this theorem also holds for weak compatibility (a complete proof can be found in [9]). Also observe that the converse implication of Theorem 4 is not always true, i.e., $(\bigotimes_{i\in K_1} M_i) \bowtie (\bigotimes_{j\in K_2} M_j)$ does not imply that $M_i \bowtie M_j$ for all $i \in K_1$ and $j \in K_2$. Considering three modules M_i, M_j, M_k, with $i \in K_1$ and $j, k \in K_2$, the composition of M_j with M_k can restrict the possible behaviors of M_j. Hence, a value returned by a service of M_j that was not allowed by interface I of M_i may never be returned by the composition of M_j and M_k.

Theorem 4 provides a semi-algorithm to check compatibility of a set of modules without building a docnet involving all modules. The semi-algorithm checks compatibility for every pair of modules, and returns `true` when all the checks are positive, thus proving global compatibility. It returns `false` otherwise: it does not necessarily mean that the modules are not compatible, and finer checks can then be performed. Again, complete details and algorithms can be found in [9].

6 Conclusion and Perspectives

We have proposed a Petri net model for document-based workflows called *docnets*. It encodes the semantics of a subset of Distributed Active XML. Composability and compatibility between modules with finite docnets are decidable, and semi-algorithms can be used for faster decision. A first extension of this work is to refine compatibility to cases where some environment calls needed to ensure progress of a service are guaranteed to occur by a contract. Even if recursion leads to undecidability [3,7], we also think that our results still hold if recursion does not create an unbounded number of service references in document types.

Compatibility is brought back to home-space problems, which use reachability checks (an $EXPSPACE$-hard problem). This may mean that our compatibility notion is not practical. However, Docnets have well-structured sets of configurations, and the markings considered in our compatibility definition contain only one token in return places. This may allow solving compatibility with efficient backward analysis techniques. We also need to improve drastically the size of the considered docnets, which grow rapidly during composition. A key issue is to avoid enumerating data values (for instance in calls parameters). Finally, we think that working with infinite but well-structured sets of document types, still allows decidability of compatibility.

References

1. Abiteboul, S., Benjelloun, O., Manolescu, I., Milo, T., Weber, R.: Active XML: A data-centric perspective on web services. In: BDA 2002 (2002)
2. Abiteboul, S., Segoufin, L., Vianu, V.: Static analysis of Active XML systems. In: PODS 2008, pp. 221–230 (2008)
3. Benveniste, A., Hélouët, L.: Distributed Active XML and service interfaces. Technical Report 7082, INRIA (2009)
4. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)

5. Ceri, S., Grefen, P., Sánchez, G.: WIDE: A distributed architecture for workflow management. In: RIDE 1997, pp. 76–79 (1997)
6. de Frutos Escrig, D., Johnen, C.: Decidability of home space property. Technical Report 503, LRI (1989)
7. Hélouët, L., Benveniste, A.: Document based modeling of web services choreographies using Active XML. In: ICWS 2010, pp. 291–298 (2010)
8. Hull, R.: Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges. In: Meersman, R., Tari, Z. (eds.) OTM 2008. LNCS, vol. 5332, pp. 1152–1163. Springer, Heidelberg (2008)
9. Masson, B., Hélouët, L., Benveniste, A.: Compatibility between DAXML schemas. Technical Report 7559, INRIA (2011)
10. Misra, J., Cook, W.R.: Computation orchestration. Software and Systems Modeling 6(1), 83–110 (2007)
11. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal 42(3), 428–445 (2003)
12. van der Aalst, W.M.P., van Hee, K.: Workflow management: Models, Methods, and Systems. MIT Press (2002)
13. Wang, J., Kumar, A.: A Framework for Document-Driven Workflow Systems. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 285–301. Springer, Heidelberg (2005)

Time and Exceptional Behavior in Multiparty Structured Interactions

Hugo A. López[1] and Jorge A. Pérez[2]

[1] PLS, IT University of Copenhagen
[2] CITI - Departamento de Informática, FCT - New University of Lisbon

Abstract. The Conversation Calculus (CC) is a model of multiparty interactions which extends the π-calculus with the notion of *conversation*—a possibly distributed medium in which participants may communicate. Here we study the interplay of time and exceptional behavior for models of structured communications based on conversations. We propose C3, a *timed* variant of the CC in which conversations feature both standard and exceptional behavior. The exceptional behavior may be triggered either by the passing of time (a timeout) or by an explicit signal for conversation abortion. By presenting a compelling example from a healthcare scenario, we argue that the combination of time and exceptional behavior leads to more meaningful models of structured communications.

1 Introduction

This paper is an initial step in understanding how *time* and forms of *exceptional behavior* can be jointly captured in models of multiparty structured communications.

Time usually plays a crucial rôle in practical scenarios of structured communication. Consider, for instance, a web banking application: interaction between a user and her bank generally takes place by means of a secure session, which is meant to expire after a certain period of inactivity of the user. When that occurs, she must exhibit again her authentication credentials, so as to initiate another session or to continue with the expired session. In some cases, the session (or parts of it) has a *predetermined duration* and so interactions may also be bounded in time. The user may need to reinitiate the session if, for instance, her network connection is too slow. Crucially, the different incarnations of time in interactions (session durations, timeouts, delays) can be seen to be closely related to the behavior of the system in exceptional circumstances. A specification of the web banking application above would appear incomplete unless one specifies how the system should behave when, e.g., the session has expired and the user attempts to reinitiate it, or when interaction is taking longer than anticipated.

In real scenarios of structured communications, time then appears to go hand in hand with *exceptional behavior*. This observation is particularly evident in *healthcare scenarios* [20]—a central source of motivation for our work. In healthcare scenarios, structured communications often involve *strict time bounds*, as in, e.g., "monitor the patient every two hours, for the next 48 hours". They may also include interaction patterns defined as both a *default behavior* and an *alternative behavior* to be executed in case of unexpected conditions, as in, e.g., "contact a substitute doctor if the designated

M. Carbone and J.-M. Petit (Eds.): WS-FM 2011, LNCS 7176, pp. 48–63, 2012.

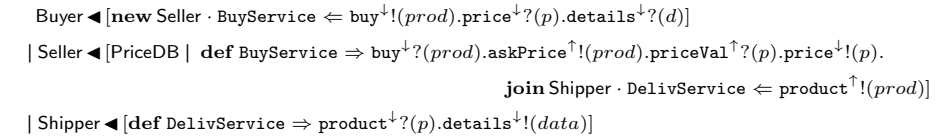


Fig. 1. The purchasing scenario in CC

doctor cannot be reached within 15 minutes". Also, scenarios may involve tasks that may be *suspended* or *aborted*, as in, e.g.,"stop administering the medicine if the patient reacts badly to it".

Unfortunately, expressing appropriately the interplay of time and exceptional behavior in known formalisms for structured communications turns out to be hard. In fact, although some of such formalisms have been extended with constructs for exceptional behavior (see, e.g., [10,4,7]), to the best of our knowledge none of these works considers the crucial interplay with timed behavior. To overcome this lack, here we introduce C3, a model of structured communications that integrates time and exceptional behavior in the context of multiparty interactions. C3 arises as an extension of the Conversation Calculus (CC) [24,23], a variant of the π-calculus [21] enhanced with the notion of *conversation*—a possibly distributed medium in which participants may communicate. In C3, conversations have durations and are sensible to compensations. Below, we first present the CC by means of a running example; then, we introduce C3 by enhancing the example with time and exceptional behavior.

The CC is an interesting base language for our study. First, being an extension of the π-calculus, the CC is a fairly *simple* and *elegant* model. As such, the definition of C3 can take advantage of previous works on extensions of the π-calculus with time and forms of exceptional behavior (see, e.g., [1,11]). Second, the CC counts with a number of *reasoning techniques* to build upon, in particular so-called *conversation types* [5]. Third, and most importantly, the CC allows for the specification of *multiparty interactions*, which are ubiquitous in many practical settings.

Fig. 1 gives a CC specification of the well-known *purchasing scenario* [9,23], which describes the interaction of a buyer and a seller for buying a given product; the seller later involves a shipper who is in charge of delivering the product. In the CC, a *conversation context* represents a distributed communication medium where two or more partners may interact. Process $n \blacktriangleleft [P]$ is the conversation context with behavior P and identity n; process P may seamlessly interact with processes contained in other conversation contexts named n. The model in Fig. 1 thus involves three participants: Buyer, Seller, and Shipper. Buyer invokes a new instance of the `BuyService` service, defined by Seller. As a result, a *conversation* on a fresh name is established between them; this name can then be used to exchange information on the product and its price (the latter is retrieved by Seller from the database PriceDB). When the transaction has been agreed, Shipper joins in the conversation, and receives product information from Seller and delivery details from Buyer. The model in Fig. 1 relies on the following *service idioms* which, interestingly, can be derived from the basic syntax of the CC:

$$
\begin{array}{lcll}
\mathbf{def}\, s \Rightarrow P & \triangleq & \mathbf{s}^{\downarrow}?(x).x \blacktriangleleft [P] & \text{define a service } s \text{ with behavior } P \\
\mathbf{new}\, n \cdot s \Leftarrow Q & \triangleq & (\nu c)(n \blacktriangleleft [\mathbf{s}^{\downarrow}!(c)] \mid c \blacktriangleleft [Q]) & \text{create instance of a service } s \text{ located at } n \\
\mathbf{join}\, n \cdot s \Leftarrow Q & \triangleq & \mathbf{this}(x).(n \blacktriangleleft [s^{\downarrow}!(x)] \mid Q) & \text{join instance of service } s \text{ located at } n
\end{array}
$$

Above, $\mathbf{s}^{\downarrow}?(x)$ and $\mathbf{s}^{\downarrow}!(c)$ are *directed* input and output prefixes (as in the π-calculus), and the $\mathbf{this}(x)$ prefix binds the enclosing conversation context to name x. The *message direction* $\downarrow$ (resp. $\uparrow$) decrees that the action should take place in the *current* (resp. *enclosing*) conversation context. We use $\star\,\mathbf{def}\; s \Rightarrow P$ to the denote an idiom for *persistent* service definition, which can be defined using recursion.

The main design decision in defining C3 is considering time and exceptional behavior *directly* into conversations: C3 features *timed*, *compensable* conversation contexts, denoted as $n \blacktriangleleft [P\,;\, Q]^t_\kappa$. As before, n is the identity of the conversation context. Process P describes the *default* behavior for n, which is performed while the *duration* t is greater than 0. Observable actions from P witness the time passage in n; as soon as $t = 0$, the default behavior is dynamically replaced by Q, the *compensating* behavior. Name κ represents an explicit *abort* mechanism: the interaction of $n \blacktriangleleft [P\,;\, Q]^t_\kappa$ with a *kill process* on κ, written $\kappa^\dagger$, immediately sets t to 0.

An immediate and pleasant consequence of our extended conversation contexts is that the signature of the service idioms given above can be extended too. Hence, C3 specifications can express rich information on timeouts and exceptional behavior. It suffices to extend the idioms representing *timed* service definition and instantiation:

$$
\begin{array}{lcll}
\mathbf{def}\, s\, \mathbf{with}\, (\kappa, t) \Rightarrow \{P; Q\} & \triangleq & \mathbf{s}^{\downarrow}?(y).y \blacktriangleleft [P\,;\, Q]^t_\kappa & \text{service definition} \\
\mathbf{new}\, n \cdot s\, \mathbf{with}\, (\kappa, t) \Leftarrow \{P; Q\} & \triangleq & (\nu c)(n \blacktriangleleft [\mathbf{s}^{\downarrow}!(c)] \mid c \blacktriangleleft [P\,;\, Q]^t_\kappa) & \text{service instantiation}
\end{array}
$$

In the former we assume y, and c are fresh in P and Q, and different from κ, t, n, while in the latter $n \blacktriangleleft [\mathbf{s}^{\downarrow}!(c)]$ stands for $n \blacktriangleleft [\mathbf{s}^{\downarrow}!(c)\,;\, \mathbf{0}]^\infty_\emptyset$. This way, we are able to define timed, compensable extensions for service definition and instantiation idioms; they rely on a compensation signal κ, a timeout t, and a compensating protocol definition Q. As a simple example, the C3 processes

$$
\begin{array}{l}
\mathsf{Provider} \blacktriangleleft [\mathbf{def}\; \mathtt{MyService}\; \mathbf{with}\; (\kappa_p, t_p) \Rightarrow \{R; T\}] \\
\mathsf{Client} \blacktriangleleft [\mathbf{new}\; \mathsf{Provider} \cdot \mathtt{MyService}\; \mathbf{with}\; (\kappa_c, t_c) \Leftarrow \{P; Q\}]
\end{array}
$$

may interact and evolve into $(\nu s)(\mathsf{Provider} \blacktriangleleft [s \blacktriangleleft [R\,;\, T]^{t_p}_{\kappa_p}] \mid \mathsf{Client} \blacktriangleleft [s \blacktriangleleft [P\,;\, Q]^{t_c}_{\kappa_c}])$.

Some related approaches (e.g. [10]) distinguish the behavior originated in the standard definition of a service from the behavior associated to related compensating activities. In those works, the objective is to return to the standard control flow by orderly escaping from compensating activities; handling nested compensations thus becomes a delicate issue. In contrast, we do not enforce such a distinction: we believe that in many realistic scenarios the main goal is timely availability of services; hence, the actual origin of the offered services should be transparent to the users. This way, e.g., for the users of a web banking application, interacting with the main server or with one of its mirrors is unimportant as long as users are provided with the required services.

$$\mathsf{NewBuyer} \blacktriangleleft [\textstyle\prod_{i\in[1..3]} \mathbf{new}\ \mathsf{Seller}_i \cdot \mathtt{BuyService}\ \mathbf{with}\ (c_i, v_i) \Leftarrow \{P_i; Q_i\} \mid \mathsf{Control}\,;\ \mathsf{CancelOrder}]_x^{tmax}$$

$$\mid \textstyle\prod_{i\in[1..3]} \mathsf{Seller}_i \blacktriangleleft [\mathsf{PriceDB} \mid \mathbf{def}\ \mathtt{BuyService}\ \mathbf{with}\ (b_i, w_i) \Rightarrow \{\mathtt{offer}^{\downarrow}?(prod).\ \mathtt{askPrice}^{\uparrow}!(prod).\mathtt{priceVal}^{\uparrow}!(p).\mathtt{price}^{\downarrow}!(p).\ \mathbf{join}\ \mathsf{Shipper} \cdot \mathtt{DelivService} \Leftarrow \mathtt{product}^{\uparrow}!(prod); R_i\}\,;\ \mathsf{CancelSell}_i]_{x_i}^{t_i}$$

$$\mid \mathsf{Shipper} \blacktriangleleft [\mathbf{def}\ \mathsf{DelivService}\ \mathbf{with}\ (d, t) \Rightarrow \{\mathtt{product}^{\downarrow}?(p).\mathtt{details}^{\downarrow}!(data); T\}\,;\ \mathbf{0}]_z^{t_3}$$

$$\text{where}\quad P_i \triangleq \mathtt{offer}^{\downarrow}!(prod).\mathtt{price}^{\downarrow}?(p).\mathtt{com}_i{}^{\uparrow}!(p).\mathtt{details}^{\downarrow}?(d)$$

$$\mathsf{Control} \triangleq \star(\mathtt{com}_1{}^{\uparrow}?(p).(c_2^{\dagger} \mid c_3^{\dagger}) + \mathtt{com}_2{}^{\uparrow}?(p).(c_1^{\dagger} \mid c_3^{\dagger}) + \mathtt{com}_3{}^{\uparrow}?(p).(c_1^{\dagger} \mid c_2^{\dagger}))$$

Fig. 2. The purchasing scenario in C3

We illustrate these ideas by considering an extended purchase scenario in C3; see Fig. 2. Suppose a buyer who is willing to interact with a specific provider only for a finite amount of time. She first engages in conversations with several providers at the same time; then, she picks the provider with the best offer, abandoning the conversations with the other providers. In the model, $\star\, P$ denotes the replicated version of process P, with the expected semantics. We consider one buyer and three sellers. NewBuyer creates three instances of the `BuyService` service, one from each seller. The part of each such instances residing at NewBuyer can be aborted by suitable messages on c_i. The part of the protocol for `BuyService` that resides at NewBuyer is similar as before, and is extended with an output signal $\mathtt{com}_i$ which allows to commit the selection of seller i. The commitment to one particular seller (and the discard of the rest) is implemented by process Control. The duration of NewBuyer is given by $tmax$; its compensation activity (CancelOrder) is left unspecified. Seller_i follows the lines of the basic scenario, extended with compensation signals y_i which trigger the compensation process $\mathsf{CancelSell}_i$. Notice that while Q_i controls the failure of the i-th service invoked by NewBuyer, CancelOrder is meant to control the failure of NewBuyer as a whole.

This extended example illustrates two of the features of C3: explicit conversation abortion and conversations bounded in time. The first one can be appreciated in the selection implemented by Control, which ensures that only one provider will be able to interact with NewBuyer, by explicitly aborting the conversations at NewBuyer with the other two providers. However, Control only takes care of the interactions at the buyer side; there are also conversation pieces at each Seller_i, which are not under the influence of Control (we assume $c_i \neq w_i$). The "garbage-collection" of such pieces is captured by the second feature: since such conversations are explicitly defined with the time bound w_i, they will be automatically collected (i.e. aborted) after w_i time units. That is, the passing of time avoids "dangling" conversation pieces. This example reveals the complementarity between the explicit conversation abortion (achieved via abortion signals) and the more implicit conversation abortion associated to the passing of time.

The rest of the paper is organized as follows. In Section 2 we summarize the main definitions of the CC. Section 3 introduces the syntax and semantics of C3. Section 4 discusses the expressiveness of C3, by comparing it to some other related languages. Then, in Section 5, we present a compelling example of C3 in a healthcare scenario. We review related work in Section 6; some concluding remarks are given in Section 7.

2 The Conversation Calculus (CC)

Here we briefly introduce the Conversation Calculus (CC, in the following). The interested reader is referred to [24,23] for further details.

The CC corresponds to a π-calculus with *labeled* communication and extended with *conversation contexts*. A conversation context can be seen as a medium in which interactions take place. It is similar to sessions in service-oriented calculi (see [12]) in the sense that every conversation context has an unique identifier (e.g.: an URI). Interactions in CC may be intuitively seen as communications in a pool of messages, where the pool is divided in areas identified by conversation contexts. Multiple participants can access many conversation contexts concurrently, provided they can get hold of the name identifying the context. Moreover, conversations can be nested multiple times (as in, e.g., a private chat room within a multi-user chat application).

Definition 1 (CC **Syntax).** *Let $\mathcal{N}$ be an infinite set of names. Also, let $\mathcal{L}$, $\mathcal{V}$, and χ be infinite sets of labels, variables, and recursion variables, respectively. Using d to range over $\uparrow$ and $\downarrow$, the set of actions α and processes P is given below:*

$$\alpha ::= \mathtt{l}^d!(\vec{n}) \mid \mathtt{l}^d?(\vec{x}) \mid \mathbf{this}(x) \quad P, Q ::= n \blacktriangleleft [P] \mid \sum_{i \in I} \alpha_i.P_i \mid P \mid Q \mid (\nu n)P \mid \mu X.P \mid X$$

Above, $\vec{n}$ and $\vec{x}$ denote tuples of names and variables in $\mathcal{N}$ and $\mathcal{V}$, respectively. Actions can be an output $\mathtt{l}^d!(\vec{n})$ or an input $\mathtt{l}^d?(\vec{x})$, as in the π-calculus, with $\mathtt{l} \in \mathcal{L}$ in both cases. The *message direction* $\downarrow$ (read "here") decrees that the action it is associated to should take place in the *current* conversation context, while $\uparrow$ (read "up") decrees that the action should take place in the *enclosing* one. We often omit the "here" direction, and write $\mathtt{l}?(y).P$ and $\mathtt{l}!(\vec{n}).P$ rather than $\mathtt{l}^{\downarrow}?(y).P$ and $\mathtt{l}^{\downarrow}!(\vec{n}).P$. The context-aware prefix $\mathbf{this}(x)$ binds the name of the enclosing conversation context to x. The syntax of processes includes the conversation context $n \blacktriangleleft [P]$, where $n \in \mathcal{N}$. We follow the standard π-calculus interpretation for guarded choice, parallelism, restriction, and recursion (for which we assume $X \in \chi$). As usual, given $\sum_{i \in I} \alpha_i.P_i$, we write $\mathbf{0}$ when $|I| = 0$, and $\alpha_1.P_1 + \alpha_2.P_2$ when $|I| = 2$. We assume the usual definitions of free/bound variables and free/bound names for a process P, noted $fv(P), bv(P)$ and $fn(P), bn(P)$, respectively. The set of names of a process is defined as $n(P) = fn(P) \cup bn(P)$. Finally, notice that labels in $\mathcal{L}$ are not subject to restriction or binding.

The semantics of the CC is given as a labeled transition system (LTS). As customary, a transition $P \xrightarrow{\lambda} P'$ represents the evolution from P to P' through action λ. We write $P \xrightarrow{\lambda}$ if $P \xrightarrow{\lambda} P'$, for some P'. We define $P \rightarrow P'$ as $P \xrightarrow{\tau} P'$. We use $P \rightarrow^* P'$ to denote the transitive closure of $P \rightarrow P'$, and write $P \stackrel{\lambda}{\Rightarrow} P'$ when $P \rightarrow^* \xrightarrow{\lambda} \rightarrow^* P'$.

Definition 2. *Transition* labels λ *are defined in terms of* actions σ*, as defined by the following grammar:* $\sigma ::= \tau \mid \mathtt{l}^d?(\vec{x}) \mid \mathtt{l}^d!(\vec{n}) \mid \mathtt{this} \qquad \lambda ::= \sigma \mid c\,\sigma \mid (\nu n)\lambda.$

Action τ denotes internal communication, while $\mathtt{l}^d?(\vec{x})$ and $\mathtt{l}^d!(\vec{n})$ represent an input and output to the environment, respectively. Action `this` represents a conversation identity access. A transition label λ can be either the (unlocated) action σ, an action σ *located at* conversation c (written $c\,\sigma$), or a transition label in which n is bound with

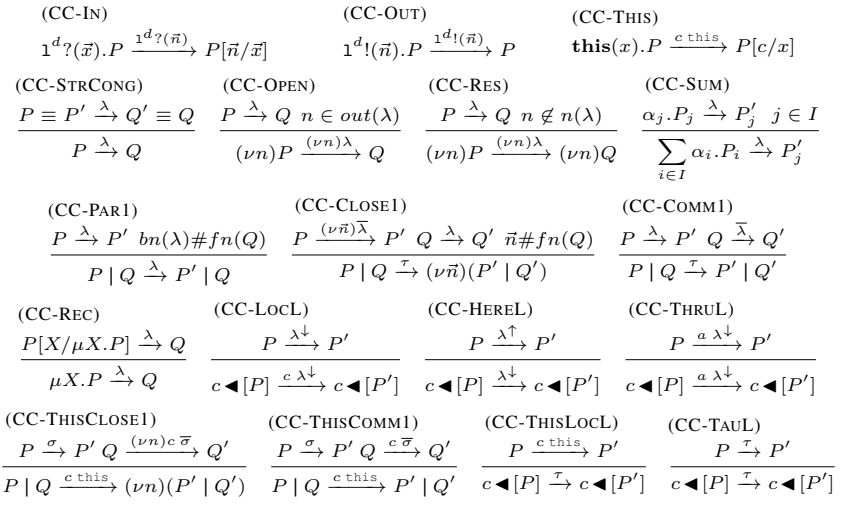


Fig. 3. An LTS for CC. Rules with labels ending with "1" have a symmetric counterpart (with label ending with "2") which is elided

scope λ. This is the case of bounded output actions. $out(\lambda)$ denotes the names produced by a transition, so $out(\lambda) = a$ if $\lambda = 1^d!(a)$ or $\lambda = c1^d!(a)$ and $c \neq a$. A transition label λ denoting input/output communication, such as $1^d?(\vec{x})$ or $1^d!(\vec{n})$, is subject to *duality*, noted $\overline{\lambda}$. We write $\overline{1^d?(\vec{x})} = 1^d!(\vec{n})$ and $\overline{1^d!(\vec{n})} = \{1^d?(\vec{x}) \mid \vec{x} \in \mathcal{V}\}$.

Fig. 3 presents the LTS. There, $\equiv$ stands for a structural congruence relation on processes; see [23] for details. The rules in the upper part of Fig. 3 follow the transition rules for a π-calculus with recursion. For instance, rule (CC-Open) corresponds to the usual scope extrusion rule. The rest of the rules are specific to the CC. Rule (CC-This) captures the name of an enclosing conversation context. Rule (CC-LocL) locates an action to a particular conversation context, and rule (CC-HereL) changes the direction of an action occurring inside a context. Rules (CC-ThisClose1) and (CC-ThisComm1) are located versions of (CC-Close) and (CC-Comm), respectively. Rule (CC-ThisLocL) hides an action occurring inside a conversation context. Rules (CC-ThruL) and (CC-TauL) formalize how actions change when they "cross" a conversation context.

3 C3: CC with Time and Compensations

As anticipated in the Introduction, the syntax of C3 extends that of the CC with timed, compensable conversation contexts and a process for aborting running conversations:

Definition 3 (C3 Syntax). *The syntax of* C3 *processes is obtained from that given in Definition 1 by replacing the conversation contexts* $n \blacktriangleleft [P]$ *with* $n \blacktriangleleft [P\,;\,Q]^t_\kappa$ *(with* $n, \kappa \in \mathcal{N}$ *and* $t \in \mathbb{N}_0 \cup \{\infty\}$*) and by adding* $\kappa^\dagger$ *to the grammar of processes.*

Every notational convention introduced for CC processes carries over to C3 processes. In particular, as in the CC, notice that labels in $\mathcal{L}$ are not subject to restriction or binding. Unlike the LTS of CC, however, we assume a relation of structural congruence as in the π-calculus only (i.e., a congruence on processes with the usual axioms for α-conversion, parallel composition, restriction, and $\mathbf{0}$). In particular, because of the timed nature of conversation contexts in C3 (on which we comment below), we refrain from adopting the axioms for manipulation of conversation contexts given in [23].

In C3, time is relative to each conversation context: it serves as a bound on the duration of the interactions *contained* in it. The time signature $t+1$ in a conversation context $c \blacktriangleleft [P\,;\,Q]^{t+1}_\kappa$ can evolve into t if the enclosing process P executes a "standard" action (i.e. any action except a compensation), or to 0 in case P fires a compensation. Hence, time in C3 is inherently *local* to each conversation context, rather than *global* to the whole system. We find this treatment of time in accordance with the intention of conversation contexts—distributed pieces of behavior in which a communication is organized. Put differently, since conversation contexts are essentially distributed abstractions of the participants of the multiparty interaction, considering a time signature local to each of them is a way of enforcing distribution. Also, as shown by our examples, this notion of time is convenient for the interplay with exceptional behavior.

The LTS for C3 is defined by the rules in Fig. 4; transition labels are obtained by extending the set of *actions* σ of the LTS of CC with a new action $\kappa^\dagger$, for an abortion (kill) process on κ. The convention on rule names for symmetric counterparts given in the LTS of the CC carries over to the LTS of C3. Moreover, for each of the left rules in Fig. 4—which describe evolution in the default behavior and have rule names ending in "L"—, there is an elided right rule characterizing evolution in the compensation behavior.

The passage of time in C3 is governed by the time elapsing function below. Intuitively, one time unit passes by as a consequence of the action. (Actions with durations different from one can be easily accommodated.)

Definition 4 (Time-elapsing function). *Given a* C3 *process* P*, we use* $\phi(P)$ *to denote the function that decreases the time bounds in* P*, inductively defined as:*

$$\phi(n \blacktriangleleft [Q\,;\,R]^{t+1}_\kappa) = n \blacktriangleleft [\phi(Q)\,;\,R]^{t}_\kappa \quad \phi(P \mid Q) = \phi(P) \mid \phi(Q) \quad \phi((\nu n)P) = (\nu n)\phi(P)$$
$$\phi(n \blacktriangleleft [Q\,;\,R]^{0}_\kappa) = n \blacktriangleleft [Q\,;\,\phi(R)]^{0}_\kappa \qquad \phi(P) = P \quad \textit{Otherwise.}$$

Given $k > 0$*, we define* $\phi^k(P) = \phi(P)$ *if* $k = 1$ *and* $\phi^k(P) = \phi(\phi^{k-1}(P))$*, otherwise.*

We describe some representative rules in Fig. 4. Rule (PAR1) decrees that executing an action in P decreases in one the time signatures of the conversation contexts in Q. This is the only rule that appeals to the time-elapsing function. For example, assuming a process P such that $P \xrightarrow{\lambda} P'$, using rules (LOCL), (RES), and (PAR1), we can infer that process $(\nu n)(c \blacktriangleleft [P\,;\,Q]^{t_1}_{\kappa_1}) \mid c \blacktriangleleft [R\,;\,S]^{t_2}_{\kappa_2}$ performs λ and evolves into $(\nu n)(c \blacktriangleleft [P'\,;\,Q]^{t_1-1}_{\kappa_1}) \mid \phi(c \blacktriangleleft [R\,;\,S]^{t_2}_{\kappa_2})$. Rules formalizing communication and closing of scope extrusion do not affect the passage of time. In process

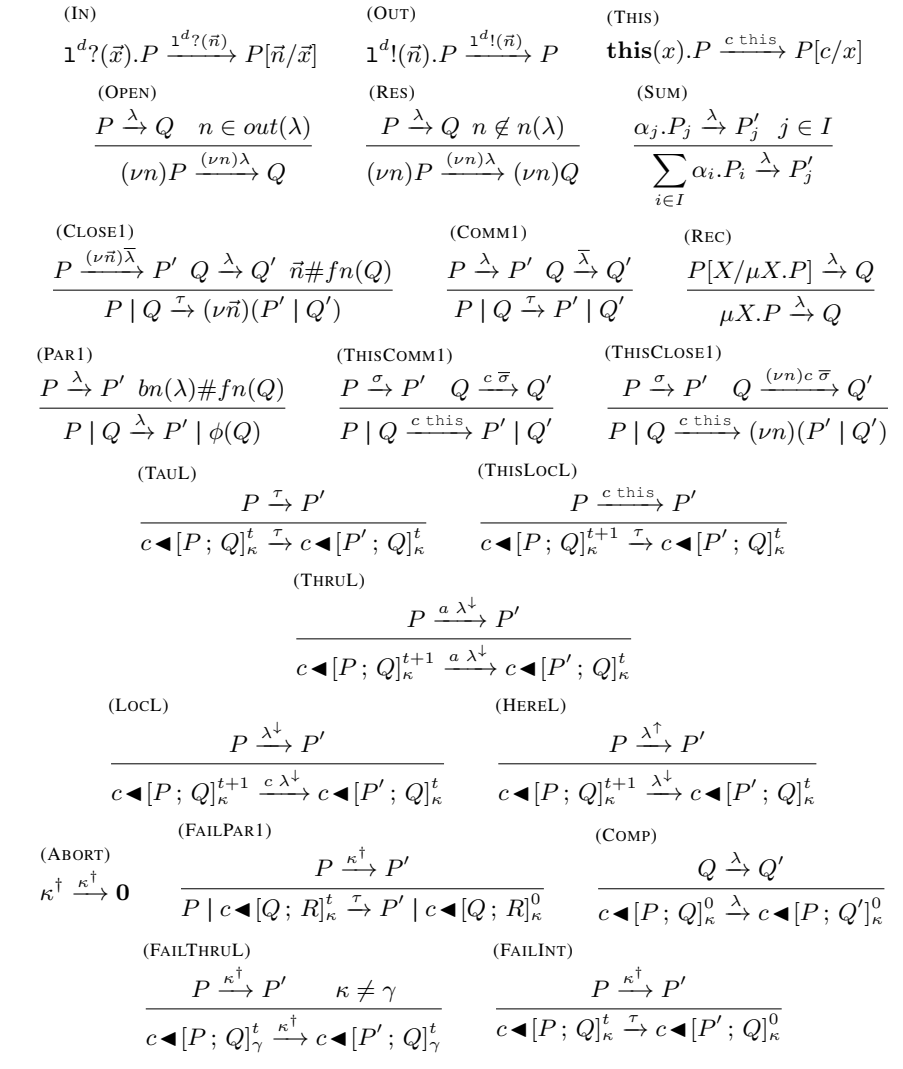


Fig. 4. Rules for the LTS of C3

$e \blacktriangleleft [d \blacktriangleleft [(\nu n)(c \blacktriangleleft [P\,;\,Q]^{t_1}_{\kappa_1}) \mid c \blacktriangleleft [R\,;\,S]^{t_2}_{\kappa_2}\,;\,U]^{t_3}_{\kappa_3}\,;\,V]^{t_4}_{\kappa_4}$, timer t_4 will be updated after applications of (THRUL), unless a transition $c \blacktriangleleft [R\,;\,S]^{t_2}_{\kappa_2} \xrightarrow{\overline{\lambda}}$ can be inferred, in which case rules (THISCLOSE1) and (THISLOCL) are applied and timers t_1, t_2, t_3 are updated. This means that only the actions leading to a synchronization—but not the synchronization itself—contribute to the passage of time. Visible actions are privileged in the sense that they affect the time bound of the enclosing conversation context—compare rules (THRUL) and (TAUL).

$$\text{(TC1)}\ \ \texttt{throw}.P \xrightarrow{\texttt{throw}} P \qquad \text{(TC2)}\ \ \frac{P \xrightarrow{\texttt{throw}} R}{P \mid Q \xrightarrow{\texttt{throw}} R} \qquad \text{(TC3)}\ \ \frac{P \xrightarrow{\texttt{throw}} R}{n \blacktriangleleft [P] \xrightarrow{\texttt{throw}} R}$$

$$\text{(TC4)}\ \ \frac{P \xrightarrow{\lambda} P' \quad \lambda \neq \texttt{throw}}{\texttt{try}\, P\, \texttt{catch}\, Q \xrightarrow{\lambda} \texttt{try}\, P'\, \texttt{catch}\, Q} \qquad \text{(TC5)}\ \ \frac{P \xrightarrow{\texttt{throw}} R}{\texttt{try}\, P\, \texttt{catch}\, Q \xrightarrow{\tau} Q} \qquad \text{(TC6)}\ \ \frac{P \xrightarrow{\texttt{throw}} R}{\texttt{try}\, P\, \texttt{catch}\, Q \xrightarrow{\tau} Q \mid R}$$

Fig. 5. LTS rules for an extension of the CC with try-catch

Rules (ABORT), (FAILPAR1), and (COMP) handle abortion signals and exceptional behavior in C3: the first formalize such signals; the second represents the abortion of a conversation context; the third formalizes the behavior of an aborted conversation context. Informally, compensation signals travel vertically upwards over levels of nested conversation contexts: following (FAILTHRUL) in process $c \blacktriangleleft [d \blacktriangleleft [\kappa_c^\dagger\,;\, Q]^{t_d}_{\kappa_d}\,;\, R]^{t_c}_{\kappa_c}$, action $\kappa_c^\dagger$ will cross d so as to affect c, its the outermost conversation context. Compensation signals can also affect surrounding conversation contexts: using (FAILPAR1) in process $c \blacktriangleleft [\kappa_d^\dagger\,;\, Q]^{t_c}_{\kappa_c} \mid d \blacktriangleleft [R\,;\, S]^{t_d}_{\kappa_d}$, will activate the alternative behavior in conversation context d (assuming $\kappa_c \neq \kappa_d$). Finally, compensation signals become unobservable if they lead to abortion (cf. rule (FAILINT)).

4 The Expressiveness of C3, Informally

With the purpose of illustrating some of its features, here we compare C3 with two of its sublanguages. The first sublanguage, noted $\mathsf{C3}^{-k}$, is C3 without compensation signals, while the second, noted $\mathsf{C3}^{-t}$, corresponds to C3 without time. Hence, while in $\mathsf{C3}^{-k}$ there are no explicit abortion signals and conversation contexts have time bounds, in $\mathsf{C3}^{-t}$ there are abortion signals, but all conversation contexts have ∞ as time bound. The semantics of each fragment can be obtained from that for C3 with the expected modifications. Our treatment is largely informal, as our objective is to shed light on the nature of C3. Formal comparisons of relative expressiveness are left for future work.

C3 and Constructs for Exception Handling. We consider the extension of the CC given in Section 2 with a try-catch operator. So we extend the syntax in Definition 1 with a new process construct $\texttt{try}\, P\, \texttt{catch}\, Q$ and a new action `throw`. The LTS of CC is extended with rules TC1-TC5 in Fig. 5. These rules give the semantics of exception handling considered in works such as, e.g., [3]. Let us refer to this extension as $\mathsf{CC}^{\mathsf{tc1}}$.

In $\mathsf{C3}^{-t}$ we can model a try-catch construct with such a semantics. Consider an encoding (language translation) $[\![\cdot]\!]^{tc}$, an homomorphism for every operator except for:

$$[\![\texttt{try}\, P\, \texttt{catch}\, Q]\!]^{tc} = n \blacktriangleleft [[\![P]\!]^{tc}\,;\, [\![Q]\!]^{tc}]^{\infty}_{\kappa_n} \qquad [\![\texttt{throw}]\!]^{tc} = \kappa_n^\dagger$$

where κ_n and n are distinguished fresh names. The encoding captures the semantics of the try-catch operator thanks to the fact that in C3 a compensation signal (a process $\kappa_n^\dagger$) can have effect on the conversation context enclosing it (cf. rule FAILINT in Fig. 4).

A compositional encoding in the opposite direction, i.e., an encoding of $\mathsf{C3}^{-t}$ into $\mathsf{CC}^{\mathsf{tc1}}$, in which try-catch blocks are used to model conversation contexts, appears difficult. This is because of the nature of compensation in C3 (and $\mathsf{C3}^{-t}$): extended conversation contexts can be aborted by both *internal* and *external* signals (cf. rules (FAILINT) and (FAILPAR1) in Fig. 4), and not only by signals inside the try block. A possibility is to define an encoding that, using the number of conversation contexts and abortion signals in the process, creates a try-catch for every possible combination. This is a rather unsatisfactory solution, as interactions may give rise to new conversation contexts, and so the number of "global" try-catch constructs required for the encoding may not be predictable in advance. Based on these observations, one could conjecture that $\mathsf{C3}^{-t}$ is strictly more expressive than $\mathsf{CC}^{\mathsf{tc1}}$.

Notice that an encoding such as the above would not work with a CC with try-catch with a different semantics. For instance, semantics enforcing advanced treatment of nested try blocks [26] may be difficult to handle. Let us consider $\mathsf{CC}^{\mathsf{tc2}}$, the extension of CC with the semantics given by rules TC1-TC4 and TC6 in Fig. 5. This is the semantics used in, e.g., [4]. The crucial difference between rules TC5 and TC6 is that in the latter the state of the try-block just after the exception has been raised (i.e., R in both rules) is preserved when the exception block (i.e., Q) is called for execution, while in the former such a state is discarded. It is not obvious how to represent such a preservation of state in C3, as the standard part of the conversation context is completely discarded when the context is aborted, and there is no way of accessing it afterwards. We thus conjecture the non existence of an encoding of $\mathsf{CC}^{\mathsf{tc2}}$ into $\mathsf{C3}^{-t}$.

Time and Interruptions in C3. From the point of view of exceptional behavior, time in C3 can be assimilated to an interruption mechanism over the standard part of a conversation context. We now informally claim that this character of timed behavior represents a difference in expressive power between $\mathsf{C3}^{-k}$ and the extensions of CC with try-catch described above.

Let us first consider $\mathsf{CC}^{\mathsf{tc1}}$. An encoding of $\mathsf{C3}^{-k}$ into $\mathsf{CC}^{\mathsf{tc1}}$ could exploit the fact that the semantics of try-catch allows to interrupt the behavior of a process placed in the try block. This is true even for persistent processes, such as $P = \mu X.\mathtt{l}_1!(n).X$. However, it is not obvious at all where to place the `throw` prefixes so as to properly model the passage of time. That is, process interruption could be encoded but not at the *right time*: hence, it is not possible to guarantee that the behavior of the $\mathsf{C3}^{-k}$ process is faithfully captured. Therefore, we conjecture that there is no encoding of $\mathsf{C3}^{-k}$ into $\mathsf{CC}^{\mathsf{tc1}}$, up to some notion of operational correspondence sensible to timed behavior. When considering $\mathsf{CC}^{\mathsf{tc2}}$, the conjecture appears somewhat more certain since, as discussed before, the semantics of $\mathsf{CC}^{\mathsf{tc2}}$ does preserve the last state of the try block before the exception is raised. That is, such a semantics does not implement interruption of the try block. This way, using P defined as above, process $S = \mathtt{try}\, P \parallel \mathtt{throw}.\mathbf{0}\, \mathtt{catch}\, \mathtt{l}_2!(n)$ would exhibit persistent behavior, even after the exception has been raised.

5 A Healthcare Compelling Example

Here we present a compelling example for C3: a medicine delivery scenario, adapted from [18,6], which features time and exceptional behavior. After presenting the

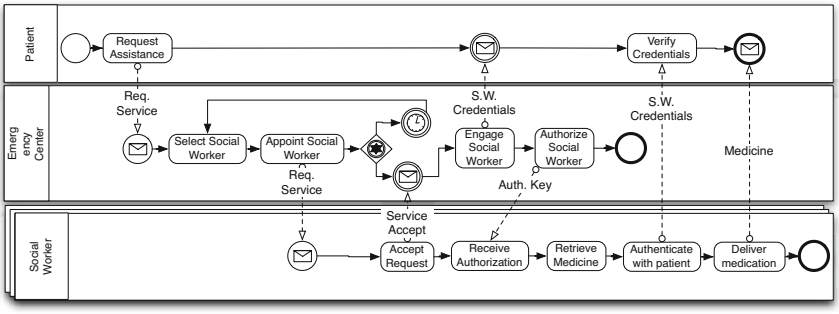


Fig. 6. BPMN diagram for the medicine delivery scenario

scenario, we present two models: while the first one is a basic model in CC, the second one is a model in C3 which allows a more comprehensive specification of the scenario.

The Medicine Delivery Scenario. Alice is a patient recently discharged from a hospital after a cardiac arrest. Sensors attached to Alice monitor her health conditions 24 hours a day; data is controlled by an Emergency Response Center (EC). The medicine delivery scenario takes place when Alice feels weak and, instead of driving to the pharmacy to get the medicine, asks to be supported by the EC. To this end, the EC requests a Social Worker (SW) to bring the medicine to Alice. There are both *mobile* SWs (dealing with requests outside the EC) and *in-house* SWs (the rest). If none of the mobile SWs can attend the request then an in-house SW is contacted. The selected SW gets appointed by the EC by sending him authorization keys for receiving the medicine and communicating with Alice. The SW can now acknowledge the request and go to the pharmacy. After a successful message exchange between Alice's terminal and the EC, the SW is authenticated and entitled to receive the medicine. Finally, the SW must authenticate to Alice in order to deliver the medicine. Fig. 6 gives a specification of the scenario in the Business Process Modeling Notation (BPMN 2.0), the de-facto notation for the specification of executable business processes.

In addition to the expected functional requirements, non-functional requirements related to the *availability* of the system are also important in the medicine delivery scenario. For instance, it is necessary to ensure that in the exceptional circumstance in which there are no available social workers nearby the patient, there is always someone who can complete the process and provide the medication. These are local policies (as they rule the execution of an agent in the system) but have effects at a global level (as they may refer to other agents apart from the ones currently executing). Timed aspects intrinsically related to the scenario are also inherently global. For instance, studies in [22] show that patients with out-of-hospital cardiac arrests who are not provided with medicine within 17 minutes have a higher chance of death. This kind of policies do not pertain the behavior of a single agent, but they involve global conditions applied to the whole interaction of roles involved in the process. Thus, they are global policies which have specific consequences in the involved principals.

The Scenario in CC and C3. Our first model of the scenario is a CC specification, given as process T in Fig. 7. The model features three interacting agents, defined as

$$T \triangleq Patient \mid EC \mid SW \quad \text{where:}$$
$$Patient \triangleq n \blacktriangleleft [\mathbf{new}\, e \cdot AB \Leftarrow \mathtt{b}!(l).\mathtt{a}?(z,m).\mathtt{a2}?(z,m).\mathtt{a3}?(p)]$$
$$EC \triangleq e \blacktriangleleft [\star\mathbf{def}\, AB \Rightarrow \mathtt{b}?(y).\mathbf{join}\, s \cdot BC \Leftarrow \mathtt{c}!(y).\mathtt{acc}?(v).(\nu pk)\mathtt{a}!(v,pk).\mathtt{c}!(pk)]$$
$$SW \triangleq s \blacktriangleleft [\mathbf{def}\, BC \Rightarrow \mathtt{c}?(z).((\nu v)\mathtt{acc}!(v).\mathtt{c}?(m_n).\mathtt{a2}!(n,m_n).\mathtt{a3}!(p))]$$

Fig. 7. Medicine Delivery Scenario: Basic Model in CC

processes $Patient$, EC, and SW. Each one defines a conversation context (noted n, e, and s, resp.). Interactions occur between $Patient$ and EC first, then between EC and SW, and finally between SW and $Patient$. More precisely, $Patient$ starts the protocol by invoking service AB, located at e. The body of $\mathbf{new}\, e \cdot AB \Leftarrow \cdots$ first receives a request from $Patient$, and then extends the established conversation so as to include SW, using the idiom $\mathbf{join}\, s \cdot BC \Leftarrow \cdots$. Once $Patient$, EC, and SW share the conversation, they are able to interact. This is evident in, e.g., interactions on c and acc between EC and SW and interactions on a between $Patient$ and EC.

While the CC specification is useful to describe the basic interacting behavior in the medicine delivery scenario, additional elements are required in order to completely faithful with the functional and non-functional requirements of the scenario. Some of such elements are difficult to express in CC. The specification in C3, given as process S in Fig. 8, allows to give a more comprehensive account of the scenario.

In the extended model, we consider a new agent, $Nurse$, representing a nurse and defining a conversation context u. The interaction patterns follow the lines of the previous model, with a few extensions. For instance, the body of the service AB located at e contains calls to DB, a database containing the contacts of available SWs. The direction of the messages between DB and the service is $\uparrow$, as communications have to cross the boundary defined by the service definition. AB describes the process that first selects one of the SWs and then forwards the request for attention started by the patient. Process SW_i represents the behavior of the i-th available mobile SW. Following the scenario (and unlike the model in CC), in the C3 specification the SW can either accept or ignore the request; in the latter case the process iterates until some SW is appointed. Upon acceptance, EC will generate new credentials identifying the SW (represented in the model with a fresh name m); in turn, these will be transmitted to the patient for further checking. Besides this extended behavior with respect to SWs, two important exceptional behaviors can be observed in this example. The first one concerns the prompt response required by the patient. In case the request for attention is not delivered on due time and the patient starts feeling dizzy, sensors attached to patient's body can detect problems in her health conditions (say, low blood pressure) and restart automatically the medicine delivery process in order to ensure the request call is answered. The behavior of the sensors here is abstracted by a non-deterministic choice; more detailed specifications are of course possible. This behavior is present in the alternative behavior for conversation context n, and it will be fired either when the expected time t_A has passed by, or when EC reports unavailability (represented by process $\kappa_A^\dagger$). The second timeout refers to internal process requirements from EC, which stipulate that

$$S \triangleq Patient \mid EC \mid SW_i \mid Nurse \quad \text{where}$$

$$Patient \triangleq n \blacktriangleleft [\mathbf{new}\, e \cdot AB \Leftarrow \mathtt{b}!(l).(\mathtt{a}?(z,m).\mathtt{a2}?(z,m).\mathtt{a3}?(p) + \kappa_A^\dagger)\,; \\ \mathbf{new}\, e \cdot AB \Leftarrow \mathtt{b}!(l).\mathtt{a}?(z,m).\mathtt{a2}?(z,m).\mathtt{a3}?(p)]_{\kappa_A}^{t_A}$$

$$EC \triangleq e \blacktriangleleft [DB \mid \star\, \mathbf{def}\, AB \Rightarrow \mathtt{b}?(y).\mu X.\mathtt{req}^\uparrow!(y).\mathtt{rep}^\uparrow?(n). \\ \mathbf{join}\, s_i \cdot BC \Leftarrow \mathtt{c}!(y).(\mathtt{dny}?(v).X + \mathtt{acc}?(v).(\nu m)\mathtt{a}!(v,m).\mathtt{c}!(m))\,; \\ \kappa_A^\dagger \mid DB \mid \star\, \mathbf{def}\, AB \Rightarrow \mathtt{b}?(y).\mathbf{join}\, u \cdot BD \Leftarrow \mathtt{d}!(y).\mathtt{acc}?(v).(\nu m)\mathtt{a}!(v,m).\mathtt{d}!(m)]_{\kappa_B}^{t_B}$$

$$SW_i \triangleq s_i \blacktriangleleft [\mathbf{def}\, BC \Rightarrow \mathtt{c}?(z).((\nu v)(\mathtt{dny}!(v) + \mathtt{acc}!(v)).\mathtt{c}?(m_n).\mathtt{a2}!(n,m_n).\mathtt{a3}!(p))\,;\, \mathbf{0}]_{\kappa_C}^{t_C}$$

$$Nurse \triangleq u \blacktriangleleft [\mathbf{def}\, BD \Rightarrow \mathtt{d}?(z).((\nu v)\mathtt{acc}!(v).\mathtt{d}?(m).\mathtt{a2}!(k,m).\mathtt{a3}!(p))\,;\, \mathbf{0}]_{\kappa_D}^{\infty}$$

Fig. 8. The medicine delivery scenario in C3

each request for attention has to be attended within t_B time units. This is irrespective from any further communication done elsewhere. The specification for the EC is thus fault tolerant, and on unavailability of a mobile SW, it will rely on service BD offered by $Nurse$.

6 Related Work

Although there is a long history of timed extensions for process calculi (see, e.g., [1]) and the study of constructs for exceptional behavior has received significant attention (see [11] for an overview), time and its interplay with forms of exceptional behavior do not seem to have been jointly studied in languages for structured communication. Our previous work [19] reported an LTL interpretation of the session language in [12] and proposed extensions with time, declarative information, and a construct for session abortion. The language in [12], however, does not support multiparty interactions. The differences in expressiveness between C3 and a variant of the CC featuring try-catch constructs [24] have been already discussed in Section 4.

Time and exceptional behavior have been considered only separately in orchestrations and choreographies. As for time, Timed Orc [25] introduced real-time observations for orchestrations by introducing a delay operator, while Timed COWS [16] extends COWS (the Calculus for Orchestration of Web Services [15]) with operators of delimitation, abortion, and delays for orchestrations. As for exceptional behavior, [10,7] propose languages for *interactional exceptions*, in which exceptions in a protocol generate coordinated actions between all peers involved. Associated type systems ensure communication safety and termination among protocols with normal and compensating executions. In [7], the language is enriched further with multiparty session and global escape primitives, allowing nested exceptions to occur at any point in an orchestration. As for choreographies, [8] introduced an extension of a language of choreographies with try/catch blocks, guaranteeing that embedded compensating parts in a choreography are not arbitrarily killed as a result of an abortion signal. Similarly, [27] presents a semantics for language for choreographies with exception handling and finalization

constructs, which allows a projection from exceptional behavior of a choreography to its endpoints. In comparison to C3, such a semantics enforces a different (centralized) treatment of choice, defines compensation blocks as exceptions in sequential languages, and does not provide support for nested compensating blocks.

Our work has been influenced by extensions to the (asynchronous) π-calculus, notably [14,1]. In particular, the rôle of the time-elapsing behavior for conversation contexts used in C3 draws inspiration from the behavior of long transactions in webπ [14], and from the π-calculus with timers in [1]. The nature of the languages in [14,1] and C3 is very different. In fact, while C3 is a synchronous language, the calculi in [14,1] are asynchronous. Also, webπ is a language for study long-running transactions; hence, exceptions in webπ and compensations in C3 have different meanings, even the constructs appear syntactically similar.

7 Concluding Remarks

We have reported initial steps towards a joint study of time and exceptional behavior in the context of multiparty structured communications. We have presented C3, a variant of the CC in which conversation contexts have an explicit duration, a compensation activity, and can be explicitly aborted. The expressiveness and relevance of its two main features (explicit abortion signals and timed behavior) have been illustrated in a compelling example extracted from a healthcare scenario of structured communications.

There are a number of directions which are worth pursuing based on the developments presented here; we briefly mention some of them. The most pressing issue concerns analysis techniques for C3 specifications. We would like to develop type disciplines for ensuring communication correctness in models featuring time and exceptional behavior. For this purpose, conversation types [5] and the linear/affine type system proposed in [2] might provide a reasonable starting point. We are also interested in a notion of *refinement* between a model in CC and an associated model in C3. In [17, Chapter 7] we report some preliminary ideas in this direction: intuitively, the objective is to decree that a C3 model is a refinement of a related CC model if they pass a set of *tests* present in both models; the challenge is to obtain suitable characterizations for such tests, considering time and the execution of compensating behavior. Finally, we would like to obtain separation results for the expressiveness conjectures stated in Section 4. As we have pointed out, different models for exception handling induce different semantics for treating exceptional behavior; it would be interesting to understand their precise relation in terms of expressiveness. While some previous work has addressed similar questions [13], we think it would be relevant to carry out similar studies in the context of languages for structured communications, such as CC and C3.

Acknowledgments. This research has been supported by the Danish Research Agency through the Trustworthy Pervasive Healthcare Services project (grant #2106-07-0019, www.TrustCare.eu) and by the Portuguese Foundation for Science and Technology (FCT/MCTES) through the Carnegie Mellon Portugal Program, grant INTERFACES NGN-44 / 2009. We thank the anonymous reviewers for their useful comments.

References

1. Berger, M., Honda, K.: The two-phase commitment protocol in an extended pi-calculus. Electr. Notes Theor. Comput. Sci. 39(1) (2000)
2. Berger, M., Yoshida, N.: Timed, Distributed, Probabilistic, Typed Processes. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 158–174. Springer, Heidelberg (2007)
3. Bravetti, M., Zavattaro, G.: On the expressive power of process interruption and compensation. Mathematical Structures in Computer Science 19(3), 565–599 (2009)
4. Caires, L., Ferreira, C., Vieira, H.: A Process Calculus Analysis of Compensations. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 87–103. Springer, Heidelberg (2009)
5. Caires, L., Vieira, H.T.: Conversation types. Theor. Comput. Sci. 411(51-52), 4399–4440 (2010)
6. Campadello, S., Compagna, L., Gidoin, D., Holtmanns, S., Meduri, V., Pazzaglia, J., Seguran, M., Thomas, R.: Scenario Selection and Definition. Research report A7.D1.1, SERENITY consortium (2006)
7. Capecchi, S., Giachino, E., Yoshida, N.: Global Escape in Multiparty Sessions. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2010. LIPIcs, vol. 8, pp. 338–351. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl (2010)
8. Carbone, M.: Session-based choreography with exceptions. In: PLACES 2008. ENTCS, vol. 241, pp. 35–55 (2008)
9. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
10. Carbone, M., Honda, K., Yoshida, N.: Structured Interactional Exceptions in Session Types. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer, Heidelberg (2008)
11. Ferreira, C., Lanese, I., Ravara, A., Vieira, H.T., Zavattaro, G.: Advanced Mechanisms for Service Combination and Transactions. In: Wirsing, M., Hölzl, M. (eds.) SENSORIA. LNCS, vol. 6582, pp. 302–325. Springer, Heidelberg (2011)
12. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
13. Lanese, I., Vaz, C., Ferreira, C.: On the Expressive Power of Primitives for Compensation Handling. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 366–386. Springer, Heidelberg (2010)
14. Laneve, C., Zavattaro, G.: Foundations of Web Transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
15. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
16. Lapadula, A., Pugliese, R., Tiezzi, F.: C-clock-WS: A Timed Service-Oriented Calculus. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 275–290. Springer, Heidelberg (2007)
17. López, H.A.: Foundations of Communication-Centred Programming. PhD thesis, IT University of Copenhagen (2012)
18. López, H.A., Massacci, F., Zannone, N.: Goal-Equivalent Secure Business Process Reengineering. In: Di Nitto, E., Ripeanu, M. (eds.) ICSOC 2007. LNCS, vol. 4907, pp. 212–223. Springer, Heidelberg (2009)
19. López, H.A., Olarte, C., Pérez, J.A.: Towards a unified framework for declarative structured communications. In: PLACES. EPTCS, vol. 17, pp. 1–15 (2009)

20. Lyng, K.M., Hildebrandt, T., Mukkamala, R.R.: From Paper Based Clinical Practice Guidelines to Declarative Workflow Management. In: Ardagna, D., Mecella, M., Yang, J. (eds.) BPM Workshops. LNBIP, vol. 17, pp. 336–347. Springer, Heidelberg (2009)
21. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts I and II. Journal of Information and Computation 100, 1–77 (1992)
22. Rittenberger, J.C., Bost, J.E., Menegazzi, J.J.: Time to give the first medication during resuscitation in out-of-hospital cardiac arrest. Resuscitation 70(2), 201–206 (2006)
23. Vieira, H.T.: A Calculus for Modeling and Analyzing Conversations in Service-Oriented Computing. PhD thesis, Universidade Nova de Lisboa (2010)
24. Vieira, H.T., Caires, L., Seco, J.C.: The Conversation Calculus: A Model of Service-Oriented Computation. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 269–283. Springer, Heidelberg (2008)
25. Wehrman, I., Kitchin, D., Cook, W.R., Misra, J.: A timed semantics of orc. Theor. Comput. Sci. 402(2-3), 234–248 (2008)
26. Xu, J., Romanovsky, A.B., Randell, B.: Coordinated exception handling in distributed object systems: From model to system implementation. In: ICDCS, pp. 12–21 (1998)
27. Hongli, Y., Xiangpeng, Z., Chao, C., Zongyan, Q.: Exploring the Connection of Choreography and Orchestration with Exception Handling and Finalization/Compensation. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 81–96. Springer, Heidelberg (2007)

Toward Design, Modelling and Analysis of Dynamic Workflow Reconfigurations

A Process Algebra Perspective

Manuel Mazzara[1], Faisal Abouzaid[2], Nicola Dragoni[3], and Anirban Bhattacharyya[1]

[1] Newcastle University, Newcastle upon Tyne, UK
{Manuel.Mazzara,Anirban.Bhattacharyya}@ncl.ac.uk
[2] École Polytechnique de Montréal, Canada
m.abouzaid@polymtl.ca
[3] Technical University of Denmark (DTU), Copenhagen
ndra@imm.dtu.dk

Abstract. This paper describes a case study involving the dynamic reconfiguration of an office workflow. We state the requirements on a system implementing the workflow and its reconfiguration, and describe the system's design in BPMN. We then use an asynchronous π-calculus and $Web\pi_\infty$ to model the design and to verify whether or not it will meet the requirements. In the process, we evaluate the formalisms for their suitability for the modelling and analysis of dynamic reconfiguration of dependable systems. The ultimate aim of this research is to identify strengths and weaknesses of formalisms for modelling dynamic reconfiguration and verifying requirements.

1 Introduction

Competition drives technological development, and the development of dependable systems is no exception. Thus, modern dependable systems are required to be more flexible, available and dependable than their predecessors, and dynamic reconfiguration is one way of achieving these requirements.

A significant amount of research has been performed on hardware reconfiguration (see [5] and [9]), but little has been done for reconfiguration of services, especially regarding computational models, formalisms and methods appropriate to the service domain. Furthermore, much of the current research assumes that reconfiguration can be instantaneous, or that the environment can wait during reconfiguration for a service to become available (see [14] and [13]). These assumptions are unrealistic in the service domain. For example, instantaneous mode change in a distributed system is generally not possible, because the system usually has no well-defined global state at a specific instant (due to significant communication delays). Also, waiting for the reconfiguration to complete is not acceptable if (as a result) the environment becomes dangerously unstable or the service provider loses revenue by the environment aborting the service request.

M. Carbone and J.-M. Petit (Eds.): WS-FM 2011, LNCS 7176, pp. 64–78, 2012.

These observations lead to the conclusion that further research is required on dynamic reconfiguration of dependable services, and especially on its formal foundations, modelling and verification. In a preliminary paper [16], we examined a number of well-known formalisms for their suitability for reconfigurable dependable systems. In this paper, we focus on one of the formalisms ($Web\pi_\infty$) and compare it to a π-calculus in order to perform a deeper analysis than was possible in [16]. We use a more complex case study involving the reconfiguration of an office workflow for order processing, define the requirements on a system implementing the workflow and its reconfiguration, and describe the design of a system in BPMN (see section 2). We then use an asynchronous π-calculus with summation (in section 3) and $Web\pi_\infty$ [19] (in section 4) to model the design and to verify whether or not the design will meet the reconfiguration requirements. We chose process algebras because they are designed to model interaction between concurrent activities. An asynchronous π-calculus was selected because π-calculi are designed to model link reconfiguration, and asynchrony is suitable for modelling communication in distributed systems. $Web\pi_\infty$ was selected because it is designed to model composition of web services.

Thus, the contribution of this paper is to identify strengths and weaknesses of an asynchronous π-calculus with summation and $Web\pi_\infty$ for modelling dynamic reconfiguration and verifying requirements (discussed in section 5). This evaluation may be useful to system designers intending to use formalisms to design dynamically reconfigurable systems, and also to researchers intending to design better formalisms for the design of dynamically reconfigurable systems.

2 Office Workflow: Requirements and Design

This case study describes dynamic reconfiguration of an office workflow for order processing that is commonly found in large and medium-sized organizations [7]. These workflows typically handle large numbers of orders. Furthermore, the organizational environment of a workflow can change in structure, procedures, policies and legal obligations in a manner unforseen by the original designers of the workflow. Therefore, it is necessary to support the unplanned change of these workflows. Furthermore, the state of an order in the old configuration may not correspond to any state of the order in the new configuration. These factors, taken in combination, imply that instantaneous reconfiguration of a workflow is not always possible; neither is it practical to delay or abort large numbers of orders because the workflow is being reconfigured. The only other possibility is to allow overlapping modes for the workflow during its reconfiguration.

2.1 Requirements

A given organization handles its orders from existing customers using a number of activities arranged according to the following procedure:

1. `Order Receipt`: an order for a product is received from a customer. The order includes customer identity and product identity information.

2. `Evaluation`: the product identity is used to perform an inventory check on the availability of the product. The customer identity is used to perform a credit check on the customer using an external service. If both the checks are positive, the order is accepted for processing; otherwise the order is rejected.
3. `Rejection`: if the order is rejected, a notification of rejection is sent to the customer and the workflow terminates.
4. If the order is to be processed, the following two activities are performed concurrently:
   (a) `Billing`: the customer is billed for the total cost of the goods ordered plus shipping costs.
   (b) `Shipping`: the goods are shipped to the customer.
5. `Archiving`: the order is archived for future reference.
6. `Confirmation`: a notification of successful completion of the order is sent to the customer.

In addition, for any given order, `Order Receipt` must precede `Evaluation`, which must precede `Rejection` or `Billing` and `Shipping`.

After some time, managers notice that lack of synchronisation between the Billing and Shipping activities is causing delays between the receipt of bills and the receipt of goods that are unacceptable to customers. Therefore, the managers decide to change the order processing procedure, so that `Billing` is performed before `Shipping` (instead of performing the two activities concurrently). During the transition interval from one procedure to the other, the following requirements must be met:

1. The result of the `Evaluation` activity for any given order should not be affected by the change in procedure.
2. All accepted orders must be billed and shipped exactly once, then archived, then confirmed.
3. All orders accepted after the change in procedure must be processed according to the new procedure.

2.2 Design

We designed the system implementing the office workflow using the Business Process Modeling Notation (BPMN) [4]. We chose BPMN because it is a widely used graphical tool for designing business processes. In fact, BPMN is a standard for business process modelling, and is maintained by the Object Management Group (see http://www.omg.org/).

The system is designed as a collection of eight pools: Office Workflow, Order Generator, Credit Check, Inventory Check, Reconf. Region, Bill&Ship1, Bill&Ship2 and Archive. The different pools represent different functional entities, and each pool can be implemented as a separate concurrent task (see Figure 1). Office Workflow coordinates the entire workflow: it receives a request from a customer, and makes a synchronous call to Order Generator to create an order. It then calls Credit Check (with the order) to check the creditworthiness of the customer, and tests the returned value using an Exclusive Data-Based

Gateway. If the test is positive, Office Workflow calls Inventory Check (with the order) to check the availability of the ordered item, and tests the returned value. If either of the two tests is negative, the customer is notified of the rejected order and the workflow terminates. If both tests are positive, Office Workflow calls Reconf. Region, which acts as a switch between configuration 1 and configuration 2 of the workflow, and thereby handles the reconfiguration of the workflow.

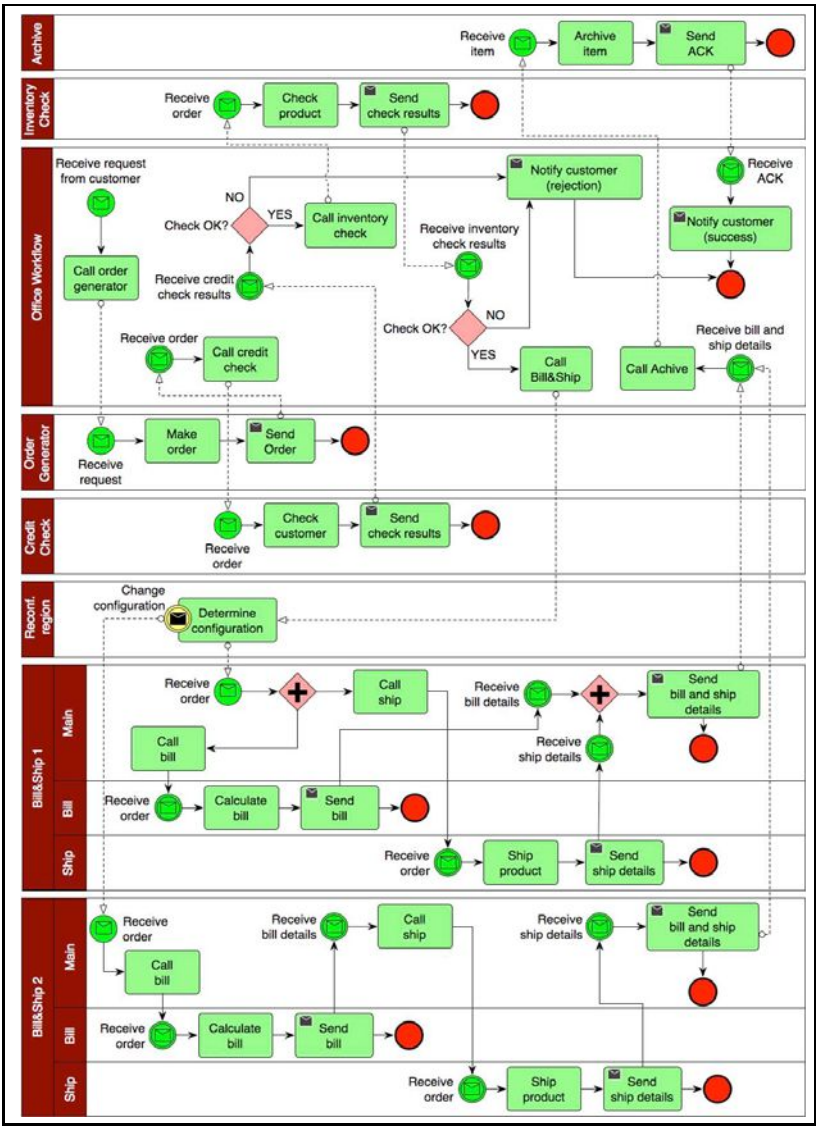


Fig. 1. Office workflow - BPMN diagram of the reconfiguration

Reconf. Region calls Bill&Ship1 by default: it makes an asynchronous call to the Main pool within Bill&Ship1, which uses a Parallel Gateway to call Bill and Ship concurrently and merge their respective results, and then returns these results to Office Workflow. The Office Workflow then calls Archive to store the order, then notifies the customer of the successful completion of the order, and then terminates the workflow. However, if Reconf. Region receives a change configuration message, it calls the Main pool within Bill&Ship2 instead, which makes sequential a call to Bill and then to Ship, and then returns the results to Office Workflow.

Notice that for the sake of simplicity, we assume neither Bill nor Ship produces a negative result. Furthermore, the Bill and Ship pools are identical in both configurations, which suggests their code is replicated (rather than shared) in the two configurations. Finally, we assume the reconfiguration is planned rather than unplanned.

3 Asynchronous π-Calculus

The asynchronous π-calculus ([10], [3]) is a subset of Milner's π-calculus [21], and it is known to be more suitable for distributed implementation. It is considered a rich paradigm for asynchronous communication, although it is not as expressive as Milner's π-calculus in representing mixed-choice constructs, such as $\overline{a}.P + b.P'$ (see [23]).

We recall the (monadic) asynchronous π-calculus. Let $\mathcal{N}$ be a set of names (e.g. $a, b, c, ...$) and $\mathcal{V}$ be a set of variables (e.g. $x, y, z, ...$). The set of the asynchronous π-calculus processes is generated by the following grammar:

$$P ::= \bar{x}z \mid G \mid P|P \mid [a = b]P \mid (\nu x)P \mid A(x_1, ..., x_n)$$

where guards G are defined as follows:

$$G ::= 0 \mid x(y).P \mid \tau.P \mid G + G$$

Intuitively, an output $\bar{x}z$ represents a message z tagged with a name x indicating that it can be received (or consumed) by an input process $x(y).P$ which behaves as $P\{z/y\}$ upon receiving z. Furthermore, $x(y).P$ binds the name y in P and the restriction $(\nu x)P$ declares a name x private to P and thus binds x. Outputs are non-blocking.

The parallel composition $P|Q$ means P and Q running in parallel. $G + G$ is the non-deterministic choice that is restricted to τ and input prefixes.

$[a = b]P$ behaves like P if a and b are identical.

$A(y_1, ..., y_n)$ is an *identifier* (also *call*, or *invocation*) of arity n. It represents the instantiation of a defined agent. We assume that every such identifier has a unique, possibly recursive, definition $A(x_1, ..., x_n) \stackrel{def}{=} P$ where the x_is are pairwise distinct, and the intuition is that $A(y_1, ..., y_n)$ behaves like P with each y_i replacing x_i.

Furthermore, for each $A(x_1, ..., x_n) \stackrel{def}{=} P$ we require: $fn(P) \subseteq \{x_1, ..., x_n\}$, where $fn(P)$ stands for the set of free names in P, and $bn(P)$ for the set of bound names in P. The *input prefix* and the ν *operator* bind the names. For example, in a process $x(y).P$, the name y is bound. In $(\nu x)P$, x is considered to be bound. Every other occurrences of a name like x in $x(y).P$ and x, y in $\bar{x}\langle y\rangle.P$ are free.

Due to lack of space we omit to give details on structural congruence and operational semantics for the asynchronous π-calculus. They can be found in [1] for the version of the calculus we use in this paper.

The Model in Asynchronous π-Calculus. The model in asynchronous π-calculus needs to keep the synchronization between actions in sequence coherent with the workflow definition. So sequence is implemented by using parallel composition with prefix and postfix on the same channel. Channel names are not restricted since the full system is not described here and has to be put in parallel with the detailed implementation of the environment process described (that will be omitted here).

The entire model is expressed in asynchronous π-calculus as follows:

Entire Model

Let $params =$
$\{customer, item, Archive, ArchiveReply, Bill, BillReply, BillShip, Confirm,$
$CreditCheck, CreditOk, CreditReject, InventoryCheck,$
$InventoryOk, InventoryReject, OrderGenerator,$
$OrderGeneratorReply, OrderReceipt, Reject, Ship, ShipReply, reco, recn\}$

We can define the $Workflow$ process as follows:

$Workflow(params) \triangleq$
$(\nu\, order)\ (OrderReceipt(customer, item).\overline{OrderGenerator}\, customer, item$
$| OrderGeneratorReply(order).\overline{CreditCheck}\, customer$
$| (creditOk().\overline{InventoryCheck}\, item + CreditReject().\overline{Reject}\, order)$
$| (InventoryOk().\overline{BillShip} + InventoryReject().\overline{Reject}\, order)$
$| rec_o().BillShip().(\overline{Bill}\, customer, item, order \mid \overline{Ship}\, customer, item, order)$
$| BillReply(order).ShipReply(order).\overline{Archive}\, order$
$+rec_n().BillShip().(\overline{Bill}\, customer, item, order$
$| BillReply(order).\overline{Ship}\, customer, item, order)\ | ShipReply(order).\overline{Archive}\, order$
$| ArchiveReply(order).\overline{Confirm}\, order)\ | Workflow(params)$

In the model, the old region is identified as follows:

$rec_o().BillShip().(\overline{Bill}\, customer, item, order \mid \overline{Ship}\, customer, item, order)$
$| BillReply(order).ShipReply(order).\overline{Archive}\, order$

And the new region is:

$rec_n().BillShip().(\overline{Bill}\, customer, item, order$
$| BillReply(order).\overline{Ship}\, customer, item, order)\ | ShipReply(order).\overline{Archive}\, order$

In the asynchronous π-calculus, two outputs cannot be in sequence. In order to impose ordering between $\overline{Bill}$ and $\overline{Ship}$, in the new region, it is necessary to put a guard on *Ship*, which requires enlarging the boundary of the old region to include the processes in the environment of the workflow that synchronize with $\overline{Bill}$ and $\overline{Ship}$. We did not model these processes because they are outside the system being designed, but the limitations of the asynchronous π-calculus imply that we must be able to access the logic of external services for which we know only the interfaces. For a more detailed description of this problem, please see [12].

The entire model represents a specific instance of the workflow that spawn concurrently another instance with fresh customer and item which here are assumed to be fresh names but in reality will be user entered (but it is not relevant to our purposes). We have to assume the existence of a "higher level" process (at the level of the BPEL engine) that activates the entire workflow and bounds the names that are free in the above π-calculus process. In this model channels $creditOK, creditReject, InventoryOK$ and $InventoryReject$ are used to receive the result of the credit check and inventory check, respectively. The old/new region is externally triggered using specific channels rec_o and rec_n chosen according to the value x received on channel *region*:

$$(\nu\, x)Workflow(param) \;\mid\; region(x).([x = new]\overline{rec_n} \;\mid\; [x = old]\overline{rec_o})$$

In section 4 we show a more efficient solution using Webπ_∞.

Analysis in π-Logic. Logics have long been used to reason about complex systems, because they provide abstract specifications that can be used to describe system properties of concurrent and distributed systems. Verification frameworks can support checking of functional properties of such systems by abstracting away from the computational contexts in which they are operating.

In the context of π-calculi, one can use the π-logic with the HAL Toolkit model-checker [8]. The π-logic has been introduced in [8] to specify the behavior of systems in a formal and unambiguous manner by expressing temporal properties of π-processes.

Syntax of the π-logic. The logic integrates modalities defined by Milner ([22]) with $EF\phi$ and $EF\{\chi\}\phi$ modalities on possible future. The π-logic syntax is:

$$\phi ::= true \mid \sim\phi \mid \phi \wedge \phi' \mid EX\{\mu\}\phi \mid EF\phi \mid EF\{\chi\}\phi$$

where μ is a π-calculus action and χ could be μ, $\sim\mu$, or $\bigvee_{i\in I}\mu_i$ and where I is a finite set.

Semantics of π-formulae is given below:

- $P \models true$ for any process P;
- $P \models \sim\phi$ iff $P \not\models \phi$;
- $P \models \phi \wedge \phi'$ iff $P \models \phi$ and $P \models \phi'$;

- $P \models EX\{\mu\}\phi$ iff there exists P' such as $P \xrightarrow{\mu} P'$ and $P' \models \phi$ (**strong next**);
- $P \models EF\phi$ iff there exists $P_0, ..., P_n$ and $\mu_1, ..., \mu_n$, with $n \geq 0$, such as $P = P_0 \xrightarrow{\mu_1} P_1 ... \xrightarrow{\mu_n} P_n$ and $P_n \models \phi$. The meaning of $EF\phi$ is that ϕ must be true sometimes in a possible future.
- $P \models EF\{\chi\}\phi$ if and only if there exists $P_0, ..., P_n$ and $\nu_1, ..., \nu_n$, with $n \geq 0$, such that $P = P_0 \xrightarrow{\nu_1} P_1 ... \xrightarrow{\nu_n} P_n$ and $P_n \models \phi$ with:
  - $\chi = \mu$ for all $1 \leq j \leq n$, $\nu_j = \mu$ *or* $\nu_j = \tau$;
  - $\chi = \sim \mu$ for all $1 \leq j \leq n$, $\nu_j \neq \mu$ *or* $\nu_j = \tau$;
  - $\chi = \bigvee_{i \in I} \mu_i$: for all $1 \leq j \leq n, \nu_j = \mu_i$ for some $i \in I$ *or* $\nu_j = \tau$.

  The meaning of $EF\{\chi\}\phi$ is that the truth of ϕ must be preceded by the occurrence of a sequence of actions χ.

Some useful dual operators are defined as usual:
$false$, $\phi \vee \phi$, $AX\{\mu\}\phi$ ($\sim EX\{\mu\} \sim \phi$), $< \mu > \phi$ (weak next), $[\mu]\phi$ (Dual of weak next), $AG\phi$ ($AG\{\chi\}$) (always).

Properties of the dynamic reconfiguration model
We need to verify that during the reconfiguration interval the requirements given in section 2.1 hold. For this purpose, we need to express the requirements formally, if possible, using the π-logic.

The Result of the `Evaluation` Activity for Any Given Order Should Not Be Affected by the Change in Procedure. The following formula means whatever the chosen path (old or new region), an order will be billed, shipped and archived or refused:

$AG\{EF\{OrderReceipt()\}true\}$
$AG\{(EF\{\overline{Bill}\, customer, item, order\}true \wedge EF\{\overline{Ship}\, customer, item, order\}true \wedge$
$EF\{\overline{Archive}\, order\}true) \vee EF\{\overline{Reject}\,\}true\}$

All Accepted Orders Must Be Billed and Shipped Exactly Once, Then Archived, Then Confirmed. The following formula means that after an order is billed and shipped, it is archived and confirmed, and cannot be billed nor shipped again:

$AG\{EF\{BillShip()\}true\}$
$AG\{EF\{\overline{Bill}\, customer, item, order\}true \wedge EF\{\overline{Ship}\, customer, item, order\}true \wedge$
$EF\{\overline{Archive}\, order\}true\} \wedge EF\{\overline{Confirm}\, order\}true\}$
$AG\{\{\overline{Bill}\, customer, item, order\}false \wedge \{\overline{Ship}\, customer, item, order\}false\}$

All Orders Accepted after the Change in Procedure Must Be Processed According to the New Procedure. We can express in the π-logic the following requirement: "after a reception on the channel rec_n, no other reception on channel rec_0 will be accepted". This meets the desired requirement since it is obvious from the model that, if a signal is received on channel rec_n, the order will be processed according to the new procedure.

$AG\{\{rec_n()\}true\ AG\{rec_0()\}false\}$

However, since the choice between the old procedure and the new one is non-deterministic, this formula will not be true, although it is an essential requirement for the model. This result illustrates the difficulty of the asynchronous π-calculus to model the dynamic reconfiguration properly. A first attempt to answer this problem is presented in the next section.

4 Webπ_∞

Webπ_∞ is a conservative extension of the π-calculus developed for modelling and analysis of Web services and Service Oriented Architectures. The basic theory has been developed in [19] and [15], whilst its applicability has been shown in other work: [12] gives a BPEL semantics in term of Webπ_∞, [6] clarifies some aspects of the Recovery Framework of BPEL, and [18] exploits a web transaction case study (a toy example has also been discussed in [16]).

Syntax and Semantics. The syntax of webπ_∞ *processes* relies on a countable set of *names*, ranged over by $x, y, z, u, \cdots$. Tuples of names are written $\widetilde{u}$. We intend $i \in I$ with I a finite non-empty set of indexes.

$$P \ ::= \ 0 \ \mid \ \overline{x}\,\widetilde{u} \ \mid \ \sum_{i\in I} x_i(\widetilde{u_i}).P_i \ \mid \ (x)P \ \mid \ P|P \ \mid \ !x(\widetilde{u}).P \ \mid \ \langle\!|P \ ; \ P|\!\rangle_x$$

It is worth noting that the syntax of webπ_∞ simply augments the asynchronous π-calculus with a workunit process. A workunit $\langle\!|P \ ; \ Q|\!\rangle_x$ behaves as the *body* P until an abort $\overline{x}$ is received, and then it behaves as the *event handler* Q.

We give the semantics of webπ_∞ in two steps, following the approach of Milner [20], separating the laws that govern the static relations between processes from the laws that rule their interactions. The static relations between processes are governed by the *structural congruence* $\equiv$, the least congruence satisfying the Abelian monoid laws for parallel and summation (associativity, commutativity and $\mathbf{0}$ as identity) and closed with respect to α-renaming and the axioms shown in table 1.

Table 1. webπ_∞ Structural Congruence

Scope laws	$(u)\mathbf{0} \equiv \mathbf{0}, \qquad (u)(v)P \equiv (v)(u)P$
	$P \mid (u)Q \equiv (u)(P \mid Q)\ , \quad if\ u \notin \mathrm{fn}(P)$
	$\langle\!|(z)P \ ; \ Q|\!\rangle_x \equiv (z)\langle\!|P \ ; \ Q|\!\rangle_x\ , \quad if\ z \notin \{x\} \cup \mathrm{fn}(Q)$
Workunit laws	$\langle\!|\mathbf{0} \ ; \ Q|\!\rangle_x \equiv \mathbf{0}$
	$\langle\!|\langle\!|P \ ; \ Q|\!\rangle_y \mid R \ ; \ R'|\!\rangle_x \equiv \langle\!|P \ ; \ Q|\!\rangle_y \mid \langle\!|R \ ; \ R'|\!\rangle_x$
	$\langle\!|(z)P \ ; \ Q|\!\rangle_x \equiv (z)\langle\!|P \ ; \ Q|\!\rangle_x\ , \quad if\ z \notin \{x\} \cup \mathrm{fn}(Q)$
Floating law	$\langle\!|\overline{z}\,\widetilde{u} \mid P \ ; \ Q|\!\rangle_x \equiv \overline{z}\,\widetilde{u} \mid \langle\!|P \ ; \ Q|\!\rangle_x$

The scope laws are standard while novelties regard workunit and floating laws. The law $\langle\!|\mathbf{0}\ ;\ Q|\!\rangle_x \equiv \mathbf{0}$ defines committed workunit, namely workunit with $\mathbf{0}$ as body. These ones, being committed, are equivalent to $\mathbf{0}$ and, therefore, cannot fail anymore. The law $\langle\!|\langle\!|P\ ;\ Q|\!\rangle_y \mid R\ ;\ R'|\!\rangle_x \equiv \langle\!|P\ ;\ Q|\!\rangle_y \mid \langle\!|R\ ;\ R'|\!\rangle_x$ moves workunit outside parents, thus flattening the nesting. Notwithstanding this flattening, parent workunits may still affect the children by means of names. The law $\langle\!|\overline{z}\,\widetilde{u} \mid P\ ;\ Q|\!\rangle_x \equiv \overline{z}\,\widetilde{u} \mid \langle\!|P\ ;\ Q|\!\rangle_x$ floats messages outside workunit boundaries. By this law, messages are particles that independently move towards their inputs. The intended semantics is the following: if a process emits a message, this message traverses the surrounding workunit boundaries until it reaches the corresponding input. In case an outer workunit fails, recoveries for this message may be detailed inside the handler processes.

The dynamic behavior of processes is instead defined by the reduction relation $\rightarrow$ which is the least relation satisfying the axioms and rules shown in table 2 and closed with respect to $\equiv$, $(x)\_$, $\_ \mid \_$, and $\langle\!|\ \_\ ;\ Q|\!\rangle_z$. In the table we use the shortcut: $\langle\!|P\ ;\ Q|\!\rangle \stackrel{\text{def}}{=} (z)\langle\!|P\ ;\ Q|\!\rangle_z$ where $z \notin \text{fn}(P) \cup \text{fn}(Q)$

Table 2. $\texttt{web}\pi_\infty$ Reduction Semantics

COM	$\overline{x_i}\,\widetilde{v} \mid \sum_{i\in I} x_i(\widetilde{u_i}).P_i \ \rightarrow\ P_i\{\widetilde{v}/\widetilde{u_i}\}$
REP	$\overline{x}\,\widetilde{v} \mid !x(\widetilde{u}).P \ \rightarrow\ P\{\widetilde{v}/\widetilde{u}\} \mid !x(\widetilde{u}).P$
FAIL	$\overline{x} \mid \langle\!|\prod_{i\in I}\sum_{s\in S} x_{is}(\widetilde{u_{is}}).P_{is} \mid \prod_{j\in J}!x_j(\widetilde{u_j}).P_j\ ;\ Q|\!\rangle_x \ \rightarrow\ \langle\!|Q\ ;\ \mathbf{0}|\!\rangle$ *where* $J \neq \emptyset \vee (I \neq \emptyset \wedge S \neq \emptyset)$

Rules (COM) and (REP) are standard in process calculi and model input-output interaction and lazy replication. Rule (FAIL) models workunit failures: when a unit abort (a message on a unit name) is emitted, the corresponding body is terminated and the handler activated. On the contrary, aborts are not possible if the transaction is already terminated (namely every thread in the body has completed its own work), thus we close the workunit restricting its name.

The Model in Webπ_∞. For the modelling purposes of this work, the idea of workunit and event handler turn out to be particularly useful. Webπ_∞ uses the mechanism of workunit to bound the identified regions, and event raising is exploited to operate the non immediate change (reconfiguration). The model can be expressed as follows (as a shortcut we will use here process invocation):

$Workflow(customer, item) \triangleq$
$(\nu\, order)\ OrderReceipt(customer, item).\overline{OrderGenerator}\, customer, item$
$\mid OrderGeneratorReply(order).\overline{CreditCheck}\, customer$
$\mid (CreditCheckReply_t(order).\overline{InventoryCheck}\, item$
$+ CreditCheckReply_f(order).\overline{Reject}\, order)$
$\mid (InventoryCheckReply_t(order).\overline{BillShip}$
$+ InventoryCheckReply_f(order).\overline{Reject}\, order)$
$\mid \langle\!|BillShip().(\overline{Bill}\, customer, item, order \mid \overline{Ship}\, customer, item, order$
$\mid (\nu\, customer)(\nu\, item)\, Workflow(customer, item))$

$; \ (\nu\, customer)(\nu\, item)\, Workflow_n(customer, item)\}_{rec}$
$| \, BillReply(order).ShipReply(order).\overline{Archive}\, order$
$| \, ArchiveReply(order).\overline{Confirm}\, order$

Webπ_∞ shows here a subtle feature which is important for modelling reconfigurable systems. Since the floating laws of structural congruence allow the asynchronous outputs in a workunit to freely escape, once the region to reconfigure has been entered and the $\overline{BillShip}$ has been triggered, $\overline{Bill}\, customer, item, order$ and $\overline{Ship}\, customer, item, order$ will not be killed by any incoming rec signal. This means that, once the region has been entered by an order, that order will go through without being interrupted by reconfiguration events and the old order will be processed according to the old procedure, not the new one. Future orders will find instead only the new procedure $Workflow_n$ waiting for orders:

$Workflow_n(customer, item) \triangleq$
$(\nu\, order)\ OrderReceipt(customer, item).\overline{OrderGenerator}\, customer, item$
$| \, OrderGeneratorReply(order).\overline{CreditCheck}\, customer$
$| \, (CreditCheckReply_t(order).\overline{InventoryCheck}\, item +$
$CreditCheckReply_f(order).\overline{Reject}\, order)$
$| \, (InventoryCheckReply_t(order).\overline{BillShip}\ +$
$InventoryCheckReply_f(order).\overline{Reject}\, order)$
$| \, BillShip().(\overline{Bill}\, customer, item, order \, | \, BillReply(order).\overline{Ship}\, customer, item, order)$
$| \, ShipReply(order).\overline{Archive}\, order \ | \, ArchiveReply(order).\overline{Confirm}\, order$
$| \, (\nu\, customer)(\nu\, item)\, Workflow_n(customer, item)$

As in the π-calculus model, we have to assume the existence of a top level process activating the entire workflow and bounding all the names appearing free in the above π-calculus process. The change in procedure will be activated when the channel t is triggered.

$$(\nu\, customer)(\nu\, item)(\nu\, rec)\, Workflow(customer, item) \, | \, t().\overline{rec}$$

This process is also responsible for triggering the reconfiguration.

Analysis in Webπ_∞. Analysis in Webπ_∞ is intended as equational reasoning. At the moment, one severe weakness of Webπ_∞ is its lack of tool support, i.e. automatic system verification. However, it is clearly possible to encode Webπ_∞ into the π-calculus, being the only technical complication the encoding of the workunit and its asynchronous interrupt. Once the compilation into the π-calculus has been done, we can proceed using HAL. From one side, Webπ_∞ simplifies the modelling of dependable systems expressing with its workunit the recovery behavior. On the other side, it makes the verification more difficult. Luckily, there is an optimal solution using Webπ_∞ as *modelling language* and the π-calculus as *intermediate language*, i.e. a *verification bytecode*. We can then offer a practical modelling suite to the designer and still use the tool support for the π-calculus. At the moment our research has not gone so far, so we will just discuss the three requirements here. We will analyse the requirements in terms of equational reasoning (see [19] and [15]). The case study of this paper is interesting at showing both the modelling power of Webπ_∞ and the weaknesses of its reasoning system.

The Result of the `Evaluation` Activity for Any Given Order Should Not Be Affected by the Change in Procedure. The acceptability of an order (`Evaluation` activity) is computed *outside* the region to be reconfigured, and there is no interaction between `Evaluation` and the region. That means that the `Evaluation` in the old procedure $workflow$ is exactly the same as in the new procedure $workflow_n$, i.e. the checks are performed in the same exact order. We can formally express it, in term of equational reasoning, stating that the `Evaluation` activity in the old procedure $workflow$ is bisimilar to the `Evaluation` activity in the new procedure $workflow_n$ which is trivially true.

All Accepted Orders Must Be Billed and Shipped Exactly Once, Then Archived, Then Confirmed. The presence of a workunit does not affect how the order itself is processed. The workflow of actions described by the requirement can be formally expressed as follows:

$$(\nu\, x)(\nu\, y)\ (\overline{Bill}\, customer, item, order \mid \overline{Ship}\, customer, item, order \\ \mid BillReply(order).\overline{x} \mid ShipReply(order).\overline{y}\ \mid x().y().\overline{Archive}\, order \\ \mid ArchiveReply(order).\overline{Confirm}\, order)$$

In plain words this process describes billing and shipping happening in any order but both before archiving and confirming. The channels x and y are there precisely to work as a joint for billing and shipping. If we want to express the requirements in term of equational reasoning, we can require that both the old and the new regions have to be bisimilar with the above process. However, this is too strict since the above process allows a set of traces which is a superset of both the set of traces of the old configuration and the new one. In this case similarity could be considered instead of bisimilarity.

All Orders Accepted after the Change in Procedure Must Be Processed According to the New Procedure. To show this requirements has been implemented in the model semantic reasoning is not necessary, structural congruence is sufficient. The change in procedure is here modelled by triggering the rec channel and spawning the workunit handler. The handler then activates a new instance of the workflow based on the new procedure scheme called $workflow_n$. The floating laws of structural congruence of Webπ_∞ (definition 1) allow the asynchronous outputs in a workunit to freely escape the workunit itself. Thus, once the region to reconfigure has been already entered and the $\overline{BillShip}$ has been triggered, $\overline{Bill}\, customer, item, order$ and $\overline{Ship}\, customer, item, order$ will not be killed by any incoming rec signal. Thus, once the region has been entered by an order, that order will be not interrupted by reconfiguration events so that old order will be processed according to the old procedure and not the new one.

5 Discussion

In this section, we discuss three issues which arose during design and modelling: how the modelling influenced our design, how the π-calculus and Webπ_∞

compare with respect to modelling, and correctness criteria for verification of the workflow reconfiguration.

Modelling and Design. Different formalisms have different biases on design because of their different perspectives. In one of the alternative designs we considered, the Bill and Ship pools were outside the reconfiguration region, so that their code was shared between the two configurations. Thus, the boundary of the reconfiguration region was different. We chose the design in section 2.2 because it is easier to model. It is the job of a formalist to model what the system designers produce, and ask them to change the design if it cannot be modelled or is unverifiable. Our experience with asynchronous π-calculi and Webπ_∞ suggested that extending the boundary of the reconfiguration region to include billing and shipping was a practical choice. This is because in the asynchronous π-calculus (and consequently in Webπ_∞), two outputs cannot be in sequence. So, in order to impose ordering between `Bill` and `Ship`, we had to enlarge the boundary of the reconfiguration region to include the processes in the environment of the workflow that synchronize with them. The negative side of this solution is that we have been forced to include in the region parts of the system that were not intended to be changed. Here the asynchronous π-calculus shows its weakness in terms of reconfiguring processes dynamically.

Comparison of π-calculus and Webπ_∞. This paper has shown the Webπ_∞ workunit as being able to offer a more efficient solution to the problem of modelling the case study. In particular, by means of the Webπ_∞ floating laws, reconfiguration activities can be better handled. However, at the moment, one weakness of Webπ_∞ is its lack of tool support, whereas the π-calculus is supported by verification tools (e.g. TyPiCal [11] and HAL [8]). Therefore, Webπ_∞ has to be intended as a a front end for modelling with the the π-calculus as the *verification bytecode*. As mentioned above, neither the asynchronous π-calculus nor Webπ_∞ can have two outputs in sequence, and this leads to the specific design choice.

Correctness Criteria. The standard notion of correctness used in process algebras is congruence based on bisimulation. However, our requirements are not all expressible as congruences between processes. The first and third requirements can be expressed as congruences, and so bisimulation can be used in the reasoning. The second requirement cannot be expressed as a congruence because the old and new configurations are not behaviourally congruent. So, we have used reasoning based on simulation instead. Thus, we found that congruence as it has been used in section 4 is not always applicable for verifying the correctness of our models. Therefore, in section 3 we have investigated model checking.

The discussion leads us to the following:

1. It is easier to model workflow reconfiguration in Webπ_∞ than in the asynchronous π-calculus. However, modelling would be even easier in a synchronous version of Webπ_∞.

2. Model checking is more widely applicable than equational reasoning based on congruences for verifying workflow reconfiguration.

These two conclusions seem to have wider applicability than just reconfiguration of workflows; but this needs to be verified.

Future Work. We intend to proceed with a deeper analysis of alternative designs for this case study, and evaluate other formalisms, such as VDM [2] and Petri nets [24]. In order to show that the system is actually executable, a first prototype has been implemented in BPEL [17] The positive outcome of this research leads us to look for larger industrial case studies that can help us to design, implement and evaluate formalisms for the modelling and analysis of dynamic reconfiguration.

Acknowledgments. This work is partly funded by the EPSRC under the terms of a graduate studentship. The paper has been improved by conversations with John Fitzgerald, Cliff Jones, Alexander Romanovsky, Jeremy Bryans, Gudmund Grov, Mario Bravetti, Massimo Strano, Michele Mazzucco, Paolo Missier and Mu Zhou. We also want to thank members of the Reconfiguration Interest Group (in particular, Kamarul Abdul Basit, Carl Gamble and Richard Payne), the Dependability Group (at Newcastle University) and the EU FP7 DEPLOY Project (Industrial deployment of system engineering methods providing high dependability and productivity).

References

1. Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous π-calculus. Theoretical Computer Science 195(2), 291–324 (1998)
2. Bjorner, D., Jones, C.B. (eds.): The Vienna Development Method: The Meta-Language. LNCS, vol. 61. Springer, Heidelberg (1978)
3. Boudol, G.: Asynchrony and the π-calculus. Rapport de recherche 1702. Technical report, INRIA, Sophia-Antipolis (1992)
4. BPMN. Bpmn - business process modeling notation, `http://www.bpmn.org/`
5. Carter, A.: Using dynamically reconfigurable hardware in real-time communications systems: Literature survey. Technical report, Computer Laboratory, University of Cambridge (November 2001)
6. Dragoni, N., Mazzara, M.: A Formal Semantics for the WS-BPEL Recovery Framework - The π-Calculus Way. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 92–109. Springer, Heidelberg (2010)
7. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proceedings of the Conference on Organizational Computing Systems (COOCS 1995). ACM (1995)
8. Ferrari, G.L., Gnesi, S., Montanari, U., Pistore, M.: A model-checking verification environment for mobile processes. ACM Transactions on Software Engineering and Methodology 12(4), 440–473 (2003)
9. Garcia, P., Compton, K., Schulte, M., Blem, E., Fu, W.: An overview of reconfigurable hardware in embedded systems. EURASIP J. Embedded Syst. (January 2006)

10. Honda, K., Tokoro, M.: An Object Calculus for Asynchronous Communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)
11. Kobayashi, N.: Typical: Type-based static analyzer for the pi-calculus, http://www.kb.ecei.tohoku.ac.jp/~koba/typical/
12. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for ws-bpel. Journal of Logic and Algebraic Programming 70(1), 96–118 (2007)
13. Magee, J., Dulay, N., Kramer, J.: Structuring parallel and distributed programs. Software Engineering Journal (Special Issue) 8(2), 73–82 (1993)
14. Magee, J., Kramer, J., Sloman, M.: Constructing distributed systems in conic. IEEE Transactions on Software Engineering 15(6), 663–675 (1989)
15. Mazzara, M.: Towards Abstractions for Web Services Composition. PhD thesis, Department of Computer Science, University of Bologna (2006)
16. Mazzara, M., Bhattacharyya, A.: On modelling and analysis of dynamic reconfiguration of dependable real-time systems. In: International Conference on Dependability, DEPEND (2010)
17. Mazzara, M., Dragoni, N., Zhou, M.: Dependable workflow reconfiguration in ws-bpel. In: To appear in Proc. of NODES 2011, Copenhagen, Denmark (2011)
18. Mazzara, M., Govoni, S.: A Case Study of Web Services Orchestration. In: Jacquet, J.-M., Picco, G.P. (eds.) COORDINATION 2005. LNCS, vol. 3454, pp. 1–16. Springer, Heidelberg (2005)
19. Mazzara, M., Lanese, I.: Towards a Unifying Theory for Web Services Composition. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 257–272. Springer, Heidelberg (2006)
20. Milner, R.: Functions as processes. Mathematical Structures in Computer Science 2(2), 119–141 (1992)
21. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press (1999)
22. Milner, R., Parrow, J., Walker, D.: Modal logics for mobile processes. Theoretical Computer Science (1993)
23. Palamidessi, C.: Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In: Mathematical Structures in Computer Science, pp. 256–265. ACM (1997)
24. Petri, C.A.: Kommunikation mit Automaten. PhD thesis, Fakultaet Mathematik und Physik, Technische Universitaet Darmstadt (1962)

An Operational Semantics of BPEL Orchestrations Integrating Web Services Resource Framework

José Antonio Mateo, Valentín Valero, and Gregorio Díaz

Informatics Research Institute of Albacete,
University of Castilla-La Mancha, Campus Universitario s/n,
02071. Albacete, Spain
{jmateo,valentin,gregorio}@dsi.uclm.es

Abstract. Web service compositions are gaining increasingly attention for the development of complex web systems by combination of existing ones. In this paper, we present a formal framework that integrates a well-known business process language (BPEL) with a recent technology for describing distributed resources throughout the Internet (WSRF). We define an operational semantics for a language that integrates both approaches taking into account the main features of them, such as notifications, event handling, fault handling and timed constraints.

1 Introduction

Nowadays, the development of software systems is becoming more complex with the appearance of new computational paradigms such as Service-Oriented Computing (SOC), Grid Computing and Cloud Computing. In these systems, the service provider needs to ensure some levels of quality and privacy to the final user in a way that had never been raised. It is therefore necessary to develop new techniques to benefit from the advantages of recent approaches, as web service compositions. Formal models of concurrency have been widely used for the description and analysis of concurrent and distributed systems. Thus, it is common to apply these techniques to these new fields of computing. In this paper, we propose an operational semantics for a language that allows us to manage web services with associated resources by using the existing machinery in distributed systems, namely web service orchestrations.

Web services architecture has been widely accepted as a means of structuring interactions among services. The definition of a web service-oriented system involves two complementary views: Choreography and Orchestration. On the one hand, the choreography concerns the observable interactions among services and can be defined by using specific languages, for example, Web Services Choreography Description Language (WS-CDL) [22] or by using more general languages like UML Messages Sequence Charts (MSC). On the other hand, the orchestration concerns the internal behavior of a web service in terms of invocations to other services. Web Services Business Process Execution Language (WS-BPEL) [1] is normally used to describe web service orchestrations, so this

M. Carbone and J.-M. Petit (Eds.): WS-FM 2011, LNCS 7176, pp. 79–94, 2012.

is the considered de facto standard language for describing web services workflow in terms of web service compositions.

To facilitate additional interoperability among services, more standardization is required to deal with distributed resources. In January of 2004, several members of the *Globus Alliance* organization and the computer multinational *IBM* with the help of experts from companies such as *HP, SAP, Akamai, etc.* defined the basis architecture and the initial specification documents of a new standard for that purpose, Web Services Resource Framework (WSRF) [11]. Although the web service definition does not consider the notion of state, interfaces frequently provide the user with the ability to access and manipulate states, that is, data values that persist across, and evolve as a result of web service interactions. The messages that the services send and receive imply (or encourage programmers to infer) the existence of an associated stateful resource. It is then desirable to define web service conventions to enable the discovery of, introspection on, and interaction with stateful resources in standard and interoperable ways [6].

We then define an integrated language, which is based on BPEL, but including many aspects of WSRF. There are many different works that formalize the main constructions of WS-BPEL. In the related section work we show a brief description of some of these works. However, we have considered that the integration of WSRF, and specifically the time constraints that are inherent in this service language, would be captured in a better way in a specific language, in which the main activity constructions of WS-BPEL are considered, as well as a model for event and fault handling.

We can see a WS-Resource as a collection of properties P identified by an address *EPR* with an associated *timeout*. This timeout represents the WS-Resource lifetime. Without loss of generality, we have reduced the resource properties set to only one allowing us to use the resource identifier *EPR* as the representative of this property. As well, in BPEL, we have taken into consideration the root scope only, avoiding thus any class of nesting among scopes, and we have considered the event and fault handling, leaving the other handling types as future work. The rest of the paper is organized as follows. In Section 2, we present the basic concepts to understand this paper and some related works. In Section 3, we define the model and its operation and semantics, whereas, in Section 4, we describe a case study to illustrate how it works. Finally, Section 5 finishes the paper giving some conclusions and possible future works.

2 Background and Related Work

2.1 Overview of BPEL/WSRF

WSRF [2] is a resource specification language developed by OASIS and some of the most pioneering computer companies, whose purpose is to define a generic framework for modeling web services with stateful resources, as well as the relationships among these services in a Grid/Cloud environment. This approach consists of a set of specifications that define the representation of the WS-Resource

Table 1. WSRF main elements

Name	Describes
WS-ResourceProperties	WSRF uses a precise specification to define the properties of the WS-Resources.
WS-Basefaults	To standardize the format for reporting error messages.
WS-ServiceGroup	This specification allows the programmer to create groups that share a common set of properties.
WS-ResourceLifetime	The mission of this specification is to standardize the process of destroying a resource and identify mechanisms to monitor its lifetime.
WS-Notification	This specification allows a *NotificationProducer* to send *notifications* to a *NotificationConsumer* in two ways: without following any formalism or with a predefined formalism.

in the terms that specify the messages exchanged and the related XML documents. These specifications allow the programmer to declare and implement the association between a service and one or more resources. It also includes mechanisms to describe the means to check the status and the service description of a resource, which together form the definition of a WS-Resource. In Table 1 we show the main WSRF elements:

On the other hand, web services are becoming more and more important as a platform for Business-to-Business integration. Web service compositions have appeared as a natural and elegant way to provide new value-added services as a combination of several established web services. Services provided by different suppliers can act together to provide another service; in fact, they can be written in different languages and can be executed on different platforms. As we noticed in the introduction, we can use web service compositions as a way to construct web service systems where each service is an autonomous entity which can offer a series of operations to the other services conforming a whole system. In this way, it is fairly necessary to establish a consistent manner to coordinate the system participants such that each of them may have a different approach, so it is common to use specific languages such as WS-BPEL to manage the system workflow. WS-BPEL, for short BPEL, is an OASIS orchestration language for specifying actions within web service business processes. These actions are represented by the execution of two types of activities (*basic* and *structured*) that perform the process logic. *Basic activities* are those which describe elemental steps of the process behavior and *structured activities* encode control-flow logic, and can therefore contain other basic and/or structured activities recursively [1].

2.2 Related Work

WS-BPEL has been extensively studied with many formalisms, such as Petri nets, Finite State Machines and process algebras, but there are only a few works considering WS-BPEL enriched with WSRF, and they only show a description of

this union, without a formalization of the model. In [20] Slomiski uses BPEL4WS in Grid environments and discusses the benefits and challenges of extensibility in the particular case of OGSI workflows combined with WSRF-based Grids. Other two works centered around Grid environments are [15] and [9]. The first justifies the use of BPEL extensibility to allow the combination of different GRIDs, whereas Ezenwoye et al. [9] share their experience on BPEL to integrate, create and manage WS-Resources that implement the factory/instance pattern.

On the other hand, Ouyang et al. [18] define the necessary elements for translating BPEL processes into Petri nets. Thus, they cover all the important aspects in the standard such as exception handling, dead path elimination and so on. The model they consider differs from ours in that we formalize the whole system as a composition of orchestrators with resources associated, whereas they describe the system as a general scope with nested sub-scopes leaving aside the possibility of administering resources. Furthermore, we have also formalized the event handling and notification mechanisms. Another extensive semantics for BPEL 2.0 is presented in [8] by Dumas et al, which introduces two new interesting improvements. They define several patterns to simplify some huge nets and introduce the semantics for the WS-BPEL 2.0 new patterns. Related to π-calculus semantics, Dragoni and Mazzara [7] propose a theoretical scheme focused on dependable composition for the WS-BPEL recovery framework. In this approach, the recovery framework is simplified and analyzed via a conservative extension of π-calculus. The aim of this approach clearly differs from ours, but it helps us to have a better understanding of the WS-BPEL recovery framework. Other work focused on the BPEL recovery framework is [19]. Although this is more focused in the compensation handler, they describe the corresponding rules that manage a web service composition. Our work is therefore quite complete as we define rules for nearly all possible activities. In addition, we also consider time constraints. Finally, we would like to highlight the works of Farahbod et al. [10] and Busi et al. [5]. In the first one, the authors extract an abstract operational semantics for BPEL based on abstract state machines (ASM) defining the framework BPEL_{AM} to manage the agents who perform the workflow activities. In this approach time constraints are considered, but they do not formalize the timed model. On the other hand, the goal of the latter one is fairly similar to ours. They also define a π-calculus operational semantics for BPEL and describe a conformance notion. They present all the machinery to model web service compositions (choreographies and orchestrations). The main difference with our work is that we deal with distributed resources. In a similar fashion Luchi and Mazzara in [17] presents other π-calculus operational semantic, $web\pi_\infty$, wich is centered on the idea of event notification as the unique error handling mechanism. It is clear that this proposal differs from ours since they focus their attention in the error handling mechanism, however their claiming of simplifying the error handling using only the notification mechanism can be performed in our proposal since this is the mechanism used in the resource framework and therefore a technique shared by WS-BPEL and WS-RF.

For further details about the formalization of service oriented languages we would like to encourage the reader to review the works presented at the SENSORIA project in [21]. Here, an extensive work is presented from different international research groups aimed by the common goal of providing a rigorous software engineering view point for service-oriented system using as a cornerstone the formal specification of Web Services and WS-BPEL in particular. Works such as SOCK [21], CaSPiS [3], COWS [14], B-lite [13] or Orc [12] are either presented or reviewed. The first one, SOCK (Service Oriented Computing Kernel [21]), is a formal calculus which aims at characterizing the basic features of Service Oriented Computing and takes its inspiration from WS-BPEL, considered by the authors as the "de facto" standard for Web Service technology. The second one, CaSPiS (Calculus of Services with Pipelines and Sessions [3]) uses the Java framework IMC. Authors take advantage of the already built-in IMC features such us session oriented and pattern matching communication mechanisms easing the task of implementing in Java all CaSPiS abstractions. Other one, COWS (Calculus for Orchestration of Web Services [14]), is a new foundational language for SOC whose design has been influenced by WS-BPEL. COWS combines a number of elements from process calculi, e.g. asynchronous communication, polyadic synchronization, pattern matching, protection, delimited receiving and killing activities. Other one reviewed also in this book is B lite [13]. This is a lightweight language for web services orchestration designed around some of WS-BPEL peculiar features like partner links, process termination, message correlation, long-running business transactions and compensation handlers aiding to clarify some ambiguous aspects of the WS-BPEL specification. The last one, Orc [12], is not influenced by any other language used for orchestration purposes like WS-BPEL. The authors define the language as a novel language for distributed and concurrent programming which provides uniform access to computational services, including distributed communication and data manipulation, through sites. Using four simple concurrency primitives, the programmer orchestrates the invocation of sites to achieve a goal, while managing timeouts, priorities, and failures. This language uses as a basic activity the information published by a site and therefore each site invocation always finishes with one or more either nested or parallel publications.

3 Syntax and Semantics of BPEL+RF

We use the following notation: *ORCH* is the set of orchestrators in the system, *Var* is the set of integer variable names, *PL* is the set of necessary partnerlinks, *OPS* is the set of operations names that can be performed, *EPRS* is the set of resource identifiers, and *A* is the set of basic or structured activities that can form the body of a process.

The specific algebraic language, then, that we use for the activities is defined by the following BNF-notation:

$$A ::= throw \mid \overline{receive}(pl, op, v) \mid invoke(pl, op, v_1) \mid exit \mid$$
$$reply(pl, v) \mid \overline{reply}(pl, op, v_2) \mid assign(expr, v_1) \mid empty \mid$$
$$A\,;A \mid A \parallel A \mid while(cond, A) \mid wait(timeout) \mid$$
$$pick(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^{n}, A, timeout) \mid getProp(EPR, v) \mid$$
$$createResource(EPR, val, timeout, O, A) \mid$$
$$setProp(EPR, expr) \mid setTimeout(EPR, timeout) \mid$$
$$subscribe(O, EPR, cond', A)$$

where $O \in ORCH, EPR \in EPRS, pl, pl_i \in PL, op, op_i \in OPS, timeout \in \mathbb{N}$, *expr* is an arithmetic expression constructed by using the variables in *Var* and integers; v, v_1, v_2, v_i range over *Var*, and $val \in \mathbb{Z}$. A condition *cond* is a predicate constructed by using conjunctions, disjunctions, and negations over the set of variables *Var* and integers, whereas *cond′* is a predicate constructed by using the corresponding *EPR* (as the resource value) and integers. Notice that *setProp* and *getProp* do not contain the property name since, for simplicity, we are only considering a single property for each resource. We therefore use its EPR as representative of this property, as we already observed in the introduction.

BPEL basic activities used in our model are: *invoke* to request services offered by service providers, *receive* and *reply* to provide services to partners, *throw* to signal an internal fault explicitly, *wait* to specify a delay, *empty* to do nothing, *exit* to end the business process and *assign*, which is used to copy data from a variable to another. And the *structured activities* used are: *sequence*, which contains two activities that are performed sequentially, *while* to provide a repeated execution of one activity, *pick* that waits for the occurrence of exactly one event from a set of events (including an alarm event), and then executes the activity associated with that event, and, finally, *flow* to express concurrency. Another family of control flow constructs in BPEL includes event, fault and compensation handlers. An event handler is enabled when its associated event occurs, being executed concurrently with the main orchestrator activity. Unlike event handlers, fault handlers do not execute concurrently with the orchestrator main activity [18]. We only cover in this work the fault and event handling. The correspondence among the syntax of WS-BPEL, WSRF and our model is shown in Table 2. Note that we do not take into consideration correlation sets, dynamic partnerlinks or instance creation, since we only deal with the static aspects of WS-BPEL. We plan as part of our future work an extension of this operational semantics enriched with these additional constructions, as well as with the inclusion of structured variables, instead of just considering all variables as integers.

An orchestration is now defined as a tuple $O = (PL, Var, A, A_f, \mathcal{A}_e)$, where A and A_f are activities defined by the previous syntax and $\mathcal{A}_e$ is a set of activities. Specifically, A represents the normal workflow, A_f is the orchestrator fault handling activity and $\mathcal{A}_e = \{A_{e_i}\}_{i=0}^{m}$ are the event handling activities.

The operational semantics is defined at three levels, the internal one corresponds to the evolution of one activity without notifications. In the second one, we define the transition rules which establish the orchestrator semantics in the event that some notifications are triggered, whereas the third level corresponds to the composition of the different orchestrators and resources to conform a choreography.

Table 2. Conversion table

WS-BPEL/WSRF Syntax	Model
<process ...> <partnerLinks> ... </partnerLinks>? <Variables> ... </Variables>? <faultHandlers> ... </faultHandlers>? <eventHandlers> ... </eventHandlers>? (activities)* </process>	(PL,Var,A,A_f,$\mathcal{A}_e$)
<throw/>/any fault	throw
<receive partnerLink="pl" operation="op" variable="v" createInstance="no"> </receive>	receive(pl,op,v)
<reply partnerLink="pl" variable="v"> </reply>	reply(pl,v)
<invoke partnerLink="pl" operation="op"inputVariable="v_1" outputVariable="v_2"?> </invoke>	invoke(pl,op,v_1); [$\overline{reply}$(pl,op,v_2)]
<empty> ... </empty>	empty
<exit> ... </exit>	exit
<assign><copy><from>expr</from><to>v_1</to></copy></assign>	assign(expr,v_1)
<wait><for>timeout</for> </wait>	wait(timeout)
<sequence> $activity_1$ $activity_2$ </sequence> <flow> $activity_1$ $activity_2$ </flow>	$\frac{A_1 \; ; \; A_2}{A_1 \parallel A_2}$
<while><condition>cond</condition>$activity_1$</while>	while(cond,A)
<pick createInstance="no"> <onMessage partnerLink="pl" operation="op"variable="v"> $activity_1$ </onMessage> <onAlarm><for>timeout</for>$activity_1$</onAlarm> </pick>	pick($\{(pl_i, op_i, v_i, A_i)\}_{i=1}^{n}$, A,timeout)
<invoke partnerLink="Factory"operation="CreateResource" inputVariable="val,timeout" outputVariable="EPR"> </invoke><assign><copy><from variable="EPR">part="ref" query="/test:CreateOut/wsa:endpointreference"</from> <to> partnerlink="Factory"</to></copy></assign>	createResource(EPR,val,timeout,O,A)
<wsrp:GetResourceProperty>$property_1$ </wsrp:GetResourceProperty>	getProp(EPR,v)
<wsrp:SetResourceProperties> <wsrp:Update> $property_1$ </wsrp:Update> < /wsrp:SetResourceProperties>	setProp(EPR,val)
<wsrl:SetTerminationTime> <wsrl:RequestedTerminationTime> timeout </wsrl:RequestedTerminationTime> </wsrl:SetTerminationTime>	setTimeout(EPR,timeout)
<wsnt:Subscribe> <wsnt:ConsumerReference>O</wsnt: ConsumerReference> <wsnt:ProducerReference>EPR</wsnt: ProducerReference> <wsnt:Precondition>cond'</Precondition> </wsnt:Subscribe>	subscribe(O,EPR,cond',A)
<wsnt:Notify> <wsnt:NotificationMessage> <wsnt:SubscriptionReference>O</wsnt:SubscriptionReference> <wsnt:ProducerReference>EPR</wsnt:ProducerReference> <wsnt:Message> ... </wsnt:Message> </wsnt:NotificationMessage> </wsnt:Notify>	Spawn the associated event handler activity

Table 3. Action transition rules without notifications

(Throw) $(throw, s) \xrightarrow{throw} (empty, s)$ **(Exit)** $(exit, s) \xrightarrow{exit} (empty, s)$

(Receive) $(receive(pl, op, v), s) \xrightarrow{receive(pl,op,v')} (empty, s')$
where $v \in Var, v' \in \mathbb{Z}, op \in OPS, pl \in PL$, and $s' = (\sigma[v'/v], \rho)$.

(Invoke) $(invoke(pl, op, v_1), s) \xrightarrow{invoke(pl,op,\sigma(v_1))} (empty, s)$

(**Reply**) $(\overline{reply}(pl, op, v_2), s) \xrightarrow{\overline{reply}(pl,op,v_2')} (empty, s')$
where $v_2 \in Var, v_2' \in \mathbb{Z}, pl \in PL, op \in OPS$, and $s' = (\sigma[v_2'/v_2], \rho)$.

(Reply) $(reply(pl, v), s) \xrightarrow{reply(pl,v)} (empty, s)$

(Assign) $(assign(expr, v_1), s) \xrightarrow{assign(expr,v_1)} (empty, s')$
where $v_1 \in Var$, $expr$ is an arithmetic expression, and $s' = (\sigma[expr/v_1], \rho)$.

(Seq1) $$\frac{(A_1, s) \xrightarrow{a} (A_1', s'), a \neq exit, a \neq throw}{(A_1; A_2, s) \xrightarrow{a} (A_1'; A_2, s')}$$

(Seq2) $$\frac{(A_1, s) \xrightarrow{a} (empty, s'), a \neq exit, a \neq throw}{(A_1; A_2, s) \xrightarrow{a} (A_2, s')}$$

(Seq3) $$\frac{(A_1, s) \xrightarrow{a} (empty, s), (a = throw \vee a = exit)}{(A_1; A_2, s) \xrightarrow{a} (empty, s)}$$

(Par1) $$\frac{(A_1, s) \xrightarrow{a} (A_1', s'), a \neq exit, a \neq throw}{(A_1||A_2, s) \xrightarrow{a} (A_1'||A_2, s')}$$

(Par2) $$\frac{(A_2, s) \xrightarrow{a} (A_2', s'), a \neq exit, a \neq throw}{(A_1||A_2, s) \xrightarrow{a} (A_1||A_2', s')}$$

(Par3) $$\frac{(A_1, s) \xrightarrow{a} (empty, s), (a = throw \vee a = exit)}{(A_1||A_2, s) \xrightarrow{a} (empty, s)}$$

(Par4) $$\frac{(A_2, s) \xrightarrow{a} (empty, s), (a = throw \vee a = exit)}{(A_1||A_2, s) \xrightarrow{a} (empty, s)}$$

(Par5) $(empty||empty, s) \xrightarrow{\tau} (empty, s)$

(While1) $$\frac{cond(s)}{(while(cond, A), s) \xrightarrow{\tau} (A; (while(cond, A), s))}$$

(While2) $$\frac{\neg cond(s)}{(while(cond, A), s) \xrightarrow{\tau} (empty, s)}$$

(Pick) $(pick(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^{n}, A, t), s) \xrightarrow{pick(pl_i,op_i,v_i',A_i)} (A_i, s')$
where $t \geq 1, v_i \in Var, v_i' \in \mathbb{Z}, pl_i \in PL$, and $s' = (\sigma[v_i'/v_i], \rho)$.

(CR) $(createResource(EPR, val, t, O, A), s) \xrightarrow{createResource(EPR,val,t,O,A)} (empty, s')$
where $t \geq 1, val \in \mathbb{Z}$ and $s' = (\sigma, \rho \cup \{EPR, val, \emptyset, O, t, A\})$, if $EPR \notin \rho$. Otherwise, $\rho' = \rho$.

(GetProp) $$\frac{s = (\sigma, \rho), EPR \in \rho}{(getProp(EPR, v), s) \xrightarrow{getProp(EPR,v')} (empty, s')}$$
where $v \in Var, v' \in \mathbb{Z}$ and $s' = (\sigma[v'/v], \rho)$.

(GetProp2) $$\frac{s = (\sigma, \rho), EPR \notin \rho}{(getProp(EPR, v), s) \xrightarrow{throw} (empty, s)}$$

(SetTime) $$\frac{s = (\sigma, \rho), EPR \in \rho}{(setTimeout(EPR, t), s) \xrightarrow{setTimeout(EPR,t)} (empty, s')}$$
where $t \geq 1$, $s' = (\sigma, \rho[t/EPR]_2)$.

(SetTime2) $$\frac{s = (\sigma, \rho), EPR \notin \rho}{(setTimeout(EPR, t), s) \xrightarrow{throw} (empty, s)}$$

(SetTime3) $$\frac{s = (\sigma, \rho), EPR \in \rho}{(setTimeout(EPR, 0), s) \xrightarrow{throw} (empty, s)}$$

(Subs) $$\frac{}{(subscribe(O, EPR, cond', A), s) \xrightarrow{subscribe(O,EPR,cond',A)} (empty, s')}$$
where $s' = (\sigma, Add\_subs(\rho, EPR, O, cond', A))$

(Subs2) $$\frac{s = (\sigma, \rho), EPR \notin \rho}{(subscribe(O, EPR, cond'), s) \xrightarrow{throw} (empty, s)}$$

We first introduce some definitions that are needed to define the operational semantics.

Definition 1 (States). We define a state as a pair s=(σ, ρ), where σ represents the variable values and ρ captures the resource state. Each orchestrator will have its own local variables, whereas the resource information (ρ) will be shared by all the orchestrators, i.e., any change introduced in ρ by any orchestrator will be visible to others. Thus, $\sigma : Var \rightarrow \mathbb{Z}$, and $\rho = \{(EPR_i, v_i, Subs_i, t_i, O_i, A_{e_i})\}_{i=1}^{r}$, where r is the number of resources in the system. Each resource has its own identifier, EPR_i, and, at each state, has a particular value, v_i, and a lifetime,

Table 4. Delay transition rules without notifications

(Wait1D) $\dfrac{t > 1}{(wait(t), s) \rightarrow_1 (wait(t-1), s^+)}$	**(Wait2D)** $(wait(1), s) \rightarrow_1 (empty, s^+)$
(SequenceD) $\dfrac{(A_1, s) \rightarrow_1 (A_1', s^+)}{(A_1; A_2, s) \rightarrow_1 (A_1'; A_2, s^+)}$	**(EmptyD)** $(empty, s) \rightarrow_1 (empty, s^+)$
(ParallelD) $\dfrac{(A_1, s) \rightarrow_1 (A_1', s^+) \wedge (A_2, s) \rightarrow_1 (A_2', s^+)}{(A_1 \| A_2, s) \rightarrow_1 (A_1' \| A_2', s^+)}$	
(Pick1D) $(pick(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^n, A, 1), s) \rightarrow_1 (A, s^+)$	
(ReceiveD) $(receive(pl, op, v), s) \rightarrow_1 (receive(pl, op, v), s^+)$	
(InvokeD) $(invoke(pl, op, v_1, v_2), s) \rightarrow_1 (invoke(pl, op, v_1, v_2), s^+)$	
(Pick2D) $\dfrac{t > 1}{(pick(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^n, A, t), s) \rightarrow_1 (pick(\{pl_i, op_i, v_i, A_i\}_{i=1}^n, A, t-1), s^+)}$	

t_i, initialized with the *createResource* function, which can be changed by using the function *setTimeout*. Moreover, $Subs_i = \{(O_{i_j}, cond'_{i_j}, A_{e_{i_j}})\}_{j=1}^{s_i}$, $i \in [1...r]$, is the set of resource notification subscribers, their associated delivery conditions and the event handling activity $A_{e_{i_j}}$ that must be thrown in the case that $cond'_{i_j}$ holds; s_i is the number of orchestrators currently subscribed to this resource and $O_{i_j} \in ORCH$ are the subscriber identifiers. Given a state $s = (\sigma, \rho)$, a variable v and an expression e, we denote by $s' = (\sigma[e/v], \rho)$ the state obtained from s by changing the value of v for the evaluation of e and $s^+ = (\sigma, \rho')$, where $\rho' = \{(EPR_i, v_i, Subs_i, t_i - 1, O_i, A_{e_i}) | t_i > 1\}_{i=1}^r$ being $\rho = \{(EPR_i, v_i, Subs_i, t_i, O_i, A_{e_i})\}_{i=1}^r$. □

Next we define some notation that we use in the operational semantics. $EPR_i \in \rho$ will denote that there is a tuple $(EPR_i, v_i, Subs_i, t_i, O_i, A_{e_i}) \in \rho$. Given a predicate *cond*, we use the function $cond(s)$ to mean the resulting value (true or false) of this predicate at the state s. Besides, $\rho[w/EPR]_1$ is used to denote that the new value in ρ of the resource *EPR* is w, $\rho[t/EPR]_2$ denotes a change in the *timeout* of the resource in ρ and $Add\_subs(\rho, EPR_i, O_{i_j}, cond'_{i_j}, A_{e_{i_j}})$ denotes that $(O_{i_j}, cond'_{i_j}, A_{e_{i_j}})$ is added to the subscribers of the resource $EPR_i \in \rho$ or $cond' = cond'_{i_j}$ in the case that O_{i_j} was already in $Subs_i$. We need two additional functions to extract the event handling activities that will be launched when the subscriber condition holds at the current state s: Given s=(σ, ρ) with $\rho = \{(EPR_i, v_i, Subs_i, t_i, A_{e_i})\}_{i=1}^r$, we define $N(O, s) = \{A_{e_{i_j}} | (O_{i_j}, cond'_{i_j}, A_{e_{i_j}}) \in Subs_i, O_{i_j} = O, cond'_{i_j} = true\}_{i=1}^r$, $j \in [1...s_i]$.

The other function is used to launch the event handling activities when the resource lifetime expires: $T(O, s) = \{A_{e_i} | (EPR_i, v_i, Subs_i, 1, O_i, A_{e_i}) \in \rho, O = O_i\}_{i=1}^r$. Now, a partnerlink is a pair (O_i, O_j) representing the two roles in communication: sender and receiver.

Definition 2 (Activity Operational semantics). We specify the activity operational semantics by using two types of transition:

a. $(\mathrm{A,s}) \xrightarrow{a} (A', s')$, a $\in$ Act (Action transitions).
b. $(\mathrm{A,s}) \rightarrow_1 (A', s^+)$ (Delay transitions).

where Act is the set of actions that can be performed, namely:
$Act = \{\tau$, throw, receive(pl,op,v), reply(pl,v), invoke(pl,op,v_1), $\overline{reply}$(pl,op,v_2), assign(e,v_1), exit, setProp(EPR,e), pick($\{(\mathrm{pl}_i,\mathrm{op}_i,\mathrm{v}_i,\mathrm{A}_i)\}_{i=1}^{n}$,A,timeout), createResource(EPR,val,timeout,O,A_{e_i}), setTimeout(EPR,timeout), getProp(EPR,v) and subscribe(O,EPR,cond$'$,A_{e_i})$\}$.

□

Notice that we have included a τ-action that represents an empty movement. *Action transitions* capture a state change by the execution of an action $a \in Act$, which can be empty (τ). *Delay transitions* capture how the system state changes when one time unit has elapsed. In Tables 3,4, we show the rules of these transitions.

Due to the lack of space, we only introduce a short explanation of some of the rules in Table 3. As can be observed from the rules, for the basic activities (*throw, exit, invoke, receive, reply, ...*), when the corresponding action is performed we reach the empty activity. Rules for the sequence and parallel operators are straightforward, but notice that when one of the arguments performs either the *throw* action or the *exit* action, the composite activity also performs this action conducting us to the empty activity. As regards the delay transition rules in Table 4, notice that the activities for which time elapsing is allowed are *wait, empty, receive, invoke* and *pick*.

When a resource has used up its lifetime or when a subscription condition holds, a specific activity is initiated in the corresponding resource subscribers, which is captured by the rules in Table 5. In these rules, the parallel operator has been extended to spawn some event handling activities, which run in parallel with the normal activity of an orchestrator. We therefore introduce the rules by using the following syntax for the activities in execution: $(A, \mathcal{A}_e)$, where A represents the normal orchestrator workflow, and $\mathcal{A}_e = \{A_{e_i}\}_{i=0}^{m}$ are the handling activities in execution. Given any activity A, we write for short $A||\mathcal{A}_e$ to denote $(A||(A_{e_1}||(\ldots||A_{e_m})))$. We assume in this operator that those event handling activities that were already in $\mathcal{A}_e$ will not be spawned twice.

Definition 3 (Operational semantics with notifications). We extend both types of transition to act on pairs $(A, \mathcal{A}_e)$. The transitions have now the following form:

a. $(O : (A, \mathcal{A}_e), s) \xrightarrow{a} (O : (A', \mathcal{A}'_e), s')$, a $\in$ Act.
b. $(O : (A, \mathcal{A}_e), s) \rightarrow_1 (O : (A', \mathcal{A}'_e), s^+)$.

where $O = (PL, Var, A, A_f, \mathcal{A}_e)$.

□

Finally, the outermost semantic level corresponds to the choreographic level, which is defined upon the two previously levels. In Table 6, we define the transition rules related to the evolution of the choreography as a whole. Observe that

Table 5. Action and delay transition rules with notifications

$$\textbf{(Notif1)}\ \frac{(A,s) \xrightarrow{a} (A',s'), a \neq exit, a \neq throw}{(O:(A,\mathcal{A}_e),s) \xrightarrow{a} (O:(A',\mathcal{A}_e||N(O,s')),s')}$$

$$\textbf{(Notif2)}\ \frac{(A_{e_i},s) \xrightarrow{a} (A'_{e_i},s'), a \neq exit, a \neq throw}{(O:(A,\mathcal{A}_e),s) \xrightarrow{a} (O:(A,\mathcal{A}'_e||N(O,s')),s')}$$

where $\mathcal{A}'_e = \{A'_{e_i}\},\ A'_{e_i} = A'_{e_j}, j \neq i.$

$$\textbf{(Notif3)}\ \frac{(A,s) \xrightarrow{throw} (empty,s)}{(O:(A,\mathcal{A}_e),s) \xrightarrow{throw} (O:(A_f,\mathcal{A}_e)),s)}$$

$$\textbf{(Notif4)}\ \frac{(A_e,s) \xrightarrow{throw} (empty,s)}{(O:(A,\mathcal{A}_e),s) \xrightarrow{throw} (O:(A_f,\mathcal{A}_e)),s)}$$

$$\textbf{(Notif5)}\ \frac{(A,s) \xrightarrow{exit} (empty,s)}{(O:(A,\mathcal{A}_e),s) \xrightarrow{exit} (O:(empty,empty)),s)}$$

$$\textbf{(Notif6)}\ \frac{(A_e,s) \xrightarrow{exit} (empty,s)}{(O:(A,\mathcal{A}_e),s) \xrightarrow{exit} (O:(empty,empty)),s)}$$

$$\textbf{(NotifD)}\ \frac{(A,s) \rightarrow_1 (A',s^+), (A_{e_i},s) \rightarrow_1 (A'_{e_i},s^+), \forall i}{(O:(A,\mathcal{A}_e),s) \rightarrow_1 (O:(A',\mathcal{A}'_e||T(O,s)),s^+)}$$

Table 6 includes rules with negatives premises, which, in principle, could pose decidability problems. Nevertheless, these premises are all observable from the syntax of the involved terms.

Definition 4 (Choreography operational semantics). A choreography is defined as a set of orchestrators that run in parallel exchanging messages: $C = \{O_i\}_{i=1}^{c}$, where c is the number of orchestrators presented in the choreography. A *choreography state* is then defined as follows: $S_c = \{(O_i : (A_i, \mathcal{A}_e{}^i), s_i)\}_{i=1}^{c}$, where A_i is the activity being performed by O_i at this state, $\mathcal{A}_e{}^i$ are the event handling activities that are currently being performed by O_i, and s_i its current state. □

Definition 5 (Labeled transition system).
For a choreography C, we define the semantics of C as the labeled transition system obtained by the application of rules in Table 6, starting at the state s_{0_C}:

$$lts(C) = (\mathcal{Q}, s_{0_C}, \rightarrow)$$

where $\mathcal{Q}$ is the set of reachable choreography states, and $\rightarrow = \rightarrow_1 \cup \{\xrightarrow{a} \mid$ for all basic activity a, or $a = \tau\}$. □

4 Case Study: Online Auction Service

The case study concerns a typical online auction process, which consists of three participants: the online auction system and two buyers, A_1 and A_2. A seller

Table 6. Choreography transition rules

(Chor1)
$$\frac{(O_i : (A_i, {\mathcal{A}_e}^i), s_i) \xrightarrow{exit} (O_i : (empty, empty), s_i)}{\{(O_k : (A_k, {\mathcal{A}_e}^k)), s_k)\}_{k=1}^{c} \xrightarrow{exit} \{(O_k : (A'_k, {\mathcal{A}''_e}^k), s'_k)\}_{k=1}^{c}}$$
where $A'_k = A_k,\ {\mathcal{A}''_e}^k = {\mathcal{A}_e}^k || N(O_k, s'_k)$ *if* $k \neq i$. *If* $k = i,\ A'_k = empty, {\mathcal{A}''_e}^k = empty$ *and* $s'_i = s_i$

(Chor2)
$$\frac{(O_i : (A_i, {\mathcal{A}_e}^i), s_i) \xrightarrow{a} (O_i : (A'_i, {\mathcal{A}'_e}^i), s'_i),\ a \neq exit,\ a \neq receive,\ a \neq invoke,\ a \neq reply,\ a \neq \overline{reply},\ a \neq pick}{\{(O_j : (A_j, {\mathcal{A}_e}^j), s_j)\}_{j=1}^{c} \xrightarrow{a} \{(O_j : (A'_j, {\mathcal{A}''_e}^j), s'_j)\}_{j=1}^{c}}$$
such that $A'_j = A_j,\ {\mathcal{A}''_e}^j = {\mathcal{A}_e}^j || N(O_j, s'_j), \forall j \neq i, j \in \{1, \ldots, c\}$.

(Chor3)
$$\frac{(O_i : (A_i, {\mathcal{A}_e}^i), s_i) \rightarrow_1 (O_i : (A'_i, {\mathcal{A}'_e}^i), {s_i}^+),\ \forall i \in \{1 \ldots c\}, \text{and rules chor4, chor5, chor6 are not applicable}}{\{(O_i : (A_i, {\mathcal{A}_e}^i), s_i)\}_{i=1}^{c} \rightarrow_1 \{(O_i : (A'_i, {\mathcal{A}''_e}^i), {s_i}^+)\}_{i=1}^{c}}$$
such that $A'_i = A_i,\ {\mathcal{A}''_e}^i = {\mathcal{A}_e}^i || T(O_i, {s_i}^+)$.

(Chor4)
$$\frac{(O_i : (A_i, {\mathcal{A}_e}^i), s_i) \xrightarrow{invoke(pl,op,\sigma_i(v_1))} (O_i : (A'_i, {\mathcal{A}'_e}^i), s_i),\ pl = (O_i, O_j),\ s_i = (\sigma_i, \rho_i),\ (O_j : (A_j, {\mathcal{A}_e}^j), s_j) \xrightarrow{receive(pl,op,\sigma_i(v_1))} (O_j : (A'_j, {\mathcal{A}'_e}^j), s'_j)}{\{(O_k : (A_k, {\mathcal{A}_e}^k)), s_k)\}_{k=1}^{c} \xrightarrow{invoke(pl,op,\sigma_i(v_1))} \{(O_k : (A'_k, {\mathcal{A}''_e}^k), s'_k)\}_{k=1}^{c}}$$
where $A'_k = A_k,\ {\mathcal{A}''_e}^k = {\mathcal{A}_e}^k || N(O_k, s'_k)$ *if* $k \neq i, k \neq j$.

(Chor5)
$$\frac{(O_i : (A_i, {\mathcal{A}_e}^i), s_i) \xrightarrow{reply(pl,v)} (O_i : (A'_i, {\mathcal{A}'_e}^i), s_i),\ pl = (O_i, O_j),\ s_i = (\sigma_i, \rho_i),\ (O_j : (A_j, {\mathcal{A}_e}^j), s_j) \xrightarrow{\overline{reply}(pl,op,\sigma_i(v))} (O_j : (A'_j, {\mathcal{A}'_e}^j), s'_j)}{\{(O_k : (A_k, {\mathcal{A}_e}^k), s_k)\}_{k=1}^{c} \xrightarrow{reply(pl,v)} \{(O_k : (A'_k, {\mathcal{A}''_e}^k), s'_k)\}_{k=1}^{c}}$$
where $A'_k = A_k,\ {\mathcal{A}''_e}^k = {\mathcal{A}_e}^k || N(O_k, s'_k)$ *if* $k \neq i, k \neq j$.

(Chor6)
$$\frac{(O_i : (A_i, {\mathcal{A}_e}^i), s_i) \xrightarrow{invoke(pl,op,v_1)} (O_i : (A'_i, {\mathcal{A}'_e}^i), s_i),\ pl = (O_i, O_j),\ s_i = (\sigma_i, \rho_i),\ (O_j : (A_j, {\mathcal{A}_e}^j), s_j) \xrightarrow{pick(pl,op,\sigma_i(v_1),A)} (O_j : (A'_j, {\mathcal{A}'_e}^j), s'_j)}{\{(O_k : (A_k, {\mathcal{A}_e}^k), s_k)\}_{k=1}^{c} \xrightarrow{invoke(pl,op,\sigma_i(v_1))} \{(O_k : (A'_k, {\mathcal{A}''_e}^k), s'_k)\}_{k=1}^{c}}$$
where $A'_k = A_k,\ {\mathcal{A}''_e}^k = {\mathcal{A}_e}^k || N(O_k, s'_k)$ *if* $k \neq i, k \neq j$.

owes a good that wants to sell to the highest possible price. Therefore, he introduces the product in an auction system for a certain time. Then, buyers (or bidders) may place bids for the product and, when time runs out, the highest bid wins. In our case, we suppose the resource is the product for auction, the value of the resource property is the current price (only the auction system can modify it), the resource subscribers will be the buyers, their subscription conditions hold when the current product value is higher than their bid, and the resource lifetime will be the time in which the auction is active. Finally, when the lifetime has expired, the auction system sends a notification to the buyers with the result of the process (the identifier of the winner, v_w) and, after that, all the processes finish. Let us consider the choreography $C = (O_{sys}, O_1, O_2)$, where $O_i = (PL_i, Var_i, A_i, A_{f_i}, \mathcal{A}_{e_i})$, i=1,2, $Var_{sys} = \{v_w, v_{EPR}, end\_bid\}$, $Var_1 = \{v_1, v_{w_1}\}$, $Var_2 = \{v_2, v_{w_2}\}$, $A_{f_1} = exit$, and $A_{f_2} = exit$. Variable v_{EPR} serves to temporarily store the value of the resource property before being sent; v_1, v_2, v_w, v_{w_1}, v_{w_2} are variables used for the interaction among participants, and, finally, *end_bid* is used to terminate the auction. Suppose $s_{0_{sys}}$, s_{0_1} and s_{0_2} are the initial states of O_{sys}, O_1 and O_2, respectively, and all the variables are initially 0:

$A_{sys} = assign(1, end\_bid); createResource(EPR, 25, 48, O_{sys}, A_{not});$
$while(end\_bid > 0, A_{bid})$
$A_1 = subscribe(O_1, EPR, EPR >= 0, A_{cond_1}); while(v_{w_1} == 0, A_{pick_1})$
$A_2 = subscribe(O_2, EPR, EPR >= 0, A_{cond_2}); while(v_{w_2} == 0, A_{pick_2})$, being:
$A_{not} = assign(0, end\_bid); getProp(EPR, v_w);$
$((invoke(pl_3, bid\_finish_1, v_w)||invoke(pl_4, bid\_finish_2, v_w))$
$A_{bid} = getProp(EPR, v_{EPR});$
$pick((pl_1, cmp, v_1, while(v_1 > v_{EPR}, assign(v1, v_{EPR}); setProp(EPR, v_{EPR}))),$
$(pl_2, cmp, v_2, while(v_2 > v_{EPR}, assign(v2, v_{EPR}), setProp(EPR, v_{EPR}))), empty, 48)$
$A_{cond_1} = getProp(EPR, v_{EPR}); invoke(pl_1, bid\_up_1, v_{EPR})$
$A_{cond_2} = getProp(EPR, v_{EPR}); invoke(pl_2, bid\_up_2, v_{EPR})$
$A_{pick_1} = pick((pl_1, bid\_up_1, v_1, assign(v_1, v_1 + random()); invoke(pl_1, cmp, v_1);$
$subscribe(O_1, EPR, EPR >= v_1, A_{cond_1})), (pl_3, bid\_finish_1, v_{w_1}, empty), empty, 48)$
$A_{pick_2} = pick((pl_2, bid\_up_2, v_2, assign(v_2, v_2 + random()); invoke(pl_2, cmp, v_2);$
$subscribe(O_2, EPR, EPR >= v_2, A_{cond_2})), (pl_4, bid\_finish_2, v_{w_2}, empty), empty, 48)$

Let us note that the operations bid_up_1 and bid_up_2 are used to increase the current bid by means of a random function, the operations bid_finish_1, bid_finish_2 update the value of v_w to finish both buyers. Finally, *cmp* is an auction system operation that receives the bids, v_i, and if the variable value is greater than the current value of v_{EPR}, then v_{EPR} is updated. As well, by means of the activity $setProp(EPR, v_{EPR})$, we update the resource property.

In the appendix Fig. 1 shows a part of the labeled transition system of C. This figure shows a trace sample where after creating the resource EPR both bidders O_1 and O_2 subscribe to this resource. Afterwards, both bidders are able to increase theirs bids by means of bid_up_i operations and therefore the auction starts. The auction process is depicted here by the states $Chor_5^{+n}$, $Chor_6^{+n}$, $Chor_7^{+n}$, $Chor_8$, $Chor_9$ and $Chor_{10}$. In these states, there are several actions running in parallel: A_{bid}, A_{pick1} and A_{pick2} representing the communication over the partnerlinks, A_{cond1} and A_{cond2} representing the bidding and A_{not} to finalize the auction. The states indexed by $+n$ have a self-loop to symbolize the passage of time, whereas the transitions labeled with condition expressions control whether the timeout has expired finishing the auction with A_{not}.

5 Conclusions and Future Work

We have presented in this paper a formal model for the description of composite web services with resources associated, and orchestrated by a well-know business process language (BPEL). The main contribution has therefore been the integration of WSRF, a resource management language, with BPEL, taking into account the main structural elements of BPEL, as its basic and structured activities, notifications, event handling and fault handling. Furthermore, special attention has been given to timed constraints, as WSRF consider that resources can only exist for a certain time (lifetime). Thus, resource leasing is considered in this work, which is a concept that has become increasingly popular in the field of distributed systems. To deal with notifications, event handling and fault

handling, the operational semantics has been defined at three levels, the outermost one corresponding to the choreographic view of the composite web services.

As future work, we plan to extend the language with some additional elements of BPEL, such as termination and compensation handling. Compensation is an important topic in web services due to the possibility of faults. We are currently working on an alternative semantics for BPEL+RF based on timed colored Petri nets. We will then define an equivalence between both semantics, and then, once correctness has been proved, we will be able to use some of the existing Petri net tools to accomplish the validation and verification phases. Thus, we are planning to develop a tool to implement the translation from a BPEL+RF specification into timed colored Petri nets.

Acknowledgement. We would like to thank to the reviewers the ideas and modifications that they have kindly suggested us. Their work has contributed to improve this paper significantly. This work is partially supported by the Spanish Government (co-financed by FEDER funds) with the project TIN2009-14312-C02-02 and the JCCLM regional project PEII09-0232-7745.

References

1. Andrews, T., et al.: BPEL4WS – Business Process Execution Language for Web Services, Version 1.1 (2003), `http://www.ibm.com/developerworks/library/specification/ws-bpel/`
2. Banks, T.: Web Services Resource Framework (WSRF) - Primer. OASIS (2006)
3. Bettini, L., De Nicola, R., Loreti, M.: Implementing Session Centered Calculi. In: Wang, A.H., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 17–32. Springer, Heidelberg (2008)
4. Bruni, R., Foster, H., Lluch Lafuente, A., Montanari, U., Tuosto, E.: A Formal Support to Business and Architectural Design for Service-Oriented Systems. In: Wirsing, M., Hölzl, M. (eds.) SENSORIA. LNCS, vol. 6582, pp. 133–152. Springer, Heidelberg (2011)
5. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and Orchestration: A Synergic Approach for System Design. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 228–240. Springer, Heidelberg (2005)
6. Czajkowski, K., Ferguson, D., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W.: The WS-Resource Framework Version 1.0 (2004), `http://www.globus.org/wsrf/specs/ws-wsrf.pdf`
7. Dragoni, N., Mazzara, M.: A Formal Semantics for the WS-BPEL Recovery Framework - The π-Calculus Way. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 92–109. Springer, Heidelberg (2010)
8. Lohmann, N.: A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)
9. Ezenwoye, O., Sadjadi, S.M., Cary, A., Robinson, M.: Orchestrating WSRF-based GridServices. Technical Report FIU-SCIS-2007-04-01 (2007)

10. Farahbod, R., Glässer, U., Vajihollahi, M.: A Formal Semantics for the Business Process Execution Language for Web Services. In: Joint Workshop on Web Services and Model-Driven Enterprise Information Services (WSMDEIS), pp. 122–133 (2005)
11. Foster, I., Frey, J., Graham, S., Tuecke, S., Czajkowski, K., Ferguson, D., Leymann, F., Nally, M., Storey, T., Weerawaranna, S.: Modeling Stateful Resources with Web Services, Globus Alliance (2004)
12. Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc Programming Language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS/FORTE 2009. LNCS, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)
13. Lapadula, A., Pugliese, R., Tiezzi, F.: A Formal Account of WS-BPEL. In: Wang, A.H., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 199–215. Springer, Heidelberg (2008)
14. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
15. Leyman, F.: Choreography for the Grid: towards fitting BPEL to the resource framework. Journal of Concurrency and Computation: Practice & Experience 18(10), 1201–1217 (2006)
16. Lohmann, N., Verbeek, E., Ouyang, C., Stahl, C.: Comparing and Evaluating Petri Net Semantics for BPEL. Journal of Business Process Integration and Management 4(1), 60–73 (2009)
17. Lucchi, R., Mazzara, M.: A Pi-calculus Based Semantics for WS-BPEL. Journal of Logic and Algebraic Programming 70(1), 96–118 (2007)
18. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Science of Computing Programming 67(2-3), 162–198 (2007)
19. Qiu, Z., Wang, S., Pu, G., Zhao, X.: Semantics of BPEL4WS-Like Fault and Compensation Handling. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 350–365. Springer, Heidelberg (2005)
20. Slomiski, A.: On using BPEL extensibility to implement OGSI and WSRF Grid workflows. Journal of Concurrency and Computation: Practice & Experience 18, 1229–1241 (2006)
21. Wirsing, M., Hölzl, M. (eds.): SENSORIA. LNCS, vol. 6582. Springer, Heidelberg (2011)
22. Web Services Choreography Description Language Version 1.0 (WS-CDL), `http://www.w3.org/TR/ws-cdl-10/`

Appendix: Case Study Figure

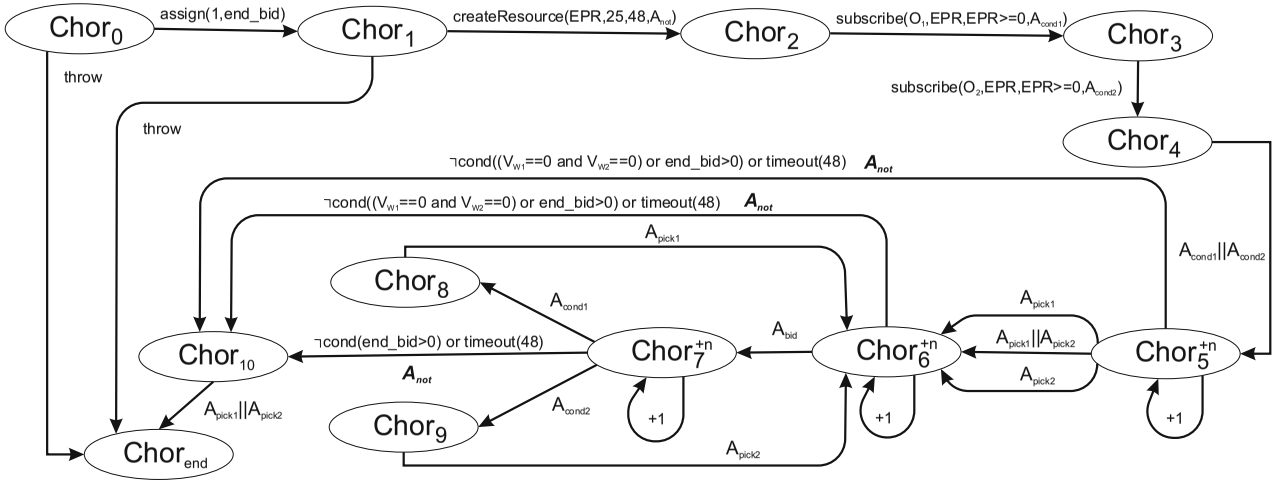


Fig. 1. A piece of $lts(C)$ for the online auction service

Design of a BPEL Verification Tool*

Elie Fares, Jean-Paul Bodeveix, and Mamoun Filali

IRIT, Université de Toulouse

Abstract. The objective of this paper is to define a formal framework for expressing a BPEL transformation based semantics of BPEL constructs. Our main contribution is twofold. First, the transformation patterns are specified in a language close to the target's realtime verification language FIACRE. Since they are expressed at the level of the concrete syntax, with respect to the tool designer, they are formal and better readable than if they were expressed at the abstract syntax level. Second, the transformation automatically produces a structured model in the form of an abstract syntax tree. This is achieved by using the language Camlp4 that allows us to define meta extensions to the targeted specification language which in turn supports the expression of transformation patterns. Furthermore, thanks to the use of FIACRE, we get a model checking-based verification support.

1 Introduction

Web services are distributed applications designed for achieving a specific business task over the web. In order to carry out the business goals, these web services should be composed in a way so that they could interact. WS-BPEL [4] is a well known service composition language. It defines business processes through the orchestration of different partners interactions.

Several works have addressed mainly the semantics issue (see section 6). Most of these works are based on defining - mathematically or graphically - transformation patterns for the BPEL constructs. The patterns produce expressions of the target language whether in concrete syntax or in abstract syntax. The generation in concrete syntax is usually more readable but does not guarantee the correction by construction of the produced code. As for the patterns generated in abstract syntax, they tend to be hardly readable.

In this paper we will study transformation patterns from BPEL to a real time verification language, FIACRE. A lot has been done on the verification of BPEL by transformations to formal modeling languages (see section 6). The aim of this work is to give insights for the design of a verification tool dedicated to BPEL. This tool is based on transformation patterns. We adopt a genuine way of representation of these patterns that could help in narrowing the gap between the definition of the patterns and their implementations and hence easing the validation of the tool. The followed method offers the readability of these patterns

* This work has been partially sponsored by the ANR project ITEMIS and Aerospace Valley project TOPCASED.

M. Carbone and J.-M. Petit (Eds.): WS-FM 2011, LNCS 7176, pp. 95–110, 2012.

in a concrete syntax close to the target's concrete syntax (FIACRE) at one hand and the reliability of the code generators in abstract syntax at the other. This is quite helpful since the transformation patterns would be isolated from the generation tools, which leads to a twofold advantage. First, a clear specification of the patterns which would ease the human intervention and the validation of the patterns if needed. Second, the transformation patterns would be ready to be directly executed. This would allow the generation of a FIACRE model as an abstract syntax tree (AST), in turn the generated code would be syntactically correct by construction and also suited for further transformations.

The rest of the paper is organized as follows. In the next section, we will give an overview of the BPEL and FIACRE languages. Section 3 presents the transformation framework and some meta extensions of FIACRE which support it. In Section 4 we begin by introducing the idea of the transformation and the technique for specifying the transformation patterns. Afterwards, we bring examples of the FIACRE patterns. Moving to Section 5, we illustrate the instantiation of our transformation patterns with an example. In Section 6, we present the related works and a comparison with other potential techniques before drawing a conclusion and an overview of the future works.

2 A Brief Overview of BPEL and FIACRE

2.1 BPEL

BPEL [4] is a language that describes the behavior of a business process by interacting with a group of partner web services. This interaction, which is done for web services orchestration purposes, is possible thanks to `Partnerlinks`. They represent the static part of a BPEL process and are described in a WSDL document in which the operations offered by a web service are also given. The dynamic part of BPEL is described by means of activities.

Basic Activities define the elementary operations of the business process. The basic activities are the usual `Receive`, `Reply`, `Invoke`, `Assign` activities and the BPEL specific ones : `Validate` to type check the value of a variable, `Wait` to delay the execution, `Throw or Rethrow` faults, `Exit`, `Empty`, `Compensate and CompensateScope` to trigger the nested compensation handlers.

Structured Activities define the order in which nested activities are executed. These are the `Sequence` and the `Flow` for the sequential and parallel execution respectively, the `Pick` for a choice over message or alarm events, the `While`, the `RepeatUntil`, the `If`, the `ForEach`. Finally, the `Scope` allows to decompose the process into subprocesses and may be associated to other BPEL handlers.

Moreover, links may be used to provide additional control over the execution's order of parallel activities. Each activity may have multiple outgoing links as well as multiple incoming links. Every outgoing link has a transition condition associated with it. The transition conditions are boolean expressions written in other languages as XPath [5]. These transition conditions are evaluated by the targeted activities in the form of a JoinCondion.

2.2 FIACRE

FIACRE [6] is a formal intermediate language dedicated to the modeling of both the behavioral and timing aspects of systems used for formal verification purposes. Basically it is a process algebra where hierarchical components communicate either through synchronous messages by means of ports or through shared memory by means of variables. These ports and variables define the interface of a component. The components may be leaves(**process**) or nodes (**component**).

- **process:** describes a sequential behavior using symbolic transition systems based on [10]. Thus, a process is defined by a set of states and transitions. Each transition has a guard and a non deterministic action that may synchronize on a timed port (port attached to a real time constraint) and update local or shared variables.
- **component:** describes the parallel composition of subcomponents. Moreover, the components may introduce variables and ports shared by its subcomponents. Real time constraints and priorities can be attached to ports. The real time constraint attached to a port `p` is a time interval `I`. This means that once the event `p` is enabled, the execution should wait at least the minimum bound of `I` and its maximum bound at most.

In Fig. 1, two process instances of `P` are composed together by synchronizing on the ports `ping, pong` and sharing the variable `w`. We show the textual syntax and the graphical syntax where processes are illustrated as automatons and components as boxes communicating through ports (•) or shared variables (■).

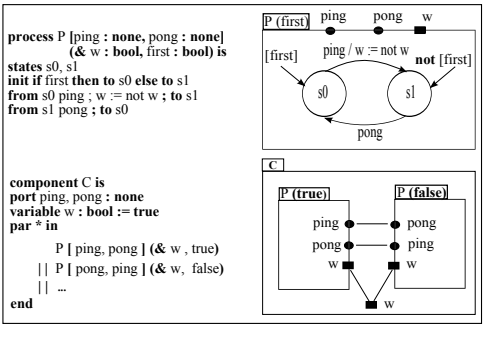


Fig. 1. Graphical notations

3 Transformation Framework

We start by presenting the basis of our transformation. Subsequently, we introduce the meta-extensions of FIACRE's concrete syntax needed for handling most of the BPEL constructs.

3.1 Transformation Basis

Our goal is to transform a source language into an abstract syntax of the target's language and not just a textual form. Moreover, we are interested in using transformation patterns that are expressed in a formalism close to the concrete syntax of the target language. Hence, these patterns are expected to be both readable by

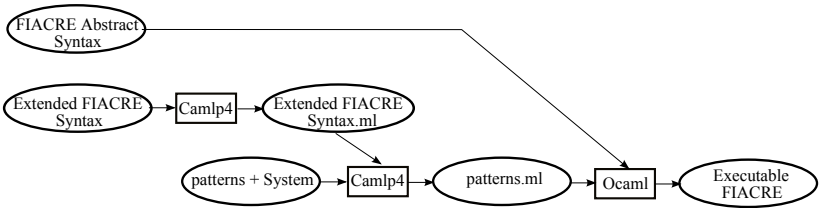


Fig. 2. Transformation Chart

the tool designer and executable. This would guarantee an easy understanding of the transformation code. In this matter, Camlp4 [13] fits fully with our requirements. The chain of integration of Camlp4 in our transformation is depicted in Fig 2. We start by defining the abstract syntax of the target language FIACRE in OCAML [14]. The concrete syntax of FIACRE is written in Camlp4 and extended by the meta-level constructs. The meta-level constructs are associated to iterators. They allow injecting Ocaml code into extended FIACRE code. This is later transformed into standard Ocaml code defining executable functions that generate FIACRE abstract syntax elements.

3.2 Camlp4 and FIACRE Meta-extensions

Camlp**4** is a **P**re-**P**rocessor-**P**retty-**P**rinter for Objective Caml. An interesting feature of Camlp4 is that it allows defining and thus extending the syntax of Ocaml by introducing new syntactic categories (FIACRE program, FIACRE declaration, ...). It makes it possible for the insertion of foreign language text within quotations inside Ocaml source text. Camlp4 transforms these quotations to pure Ocaml code. For example, the following Ocaml code assigns to the `f` variable the abstract syntax of the FIACRE code written inside the quotation `<:fiacre<...>>`. Thus, we can define Ocaml functions which manipulate FIACRE abstract syntax using concrete syntax.

```
let f = <:fiacre< type t_err is union no | exit | other end  ... >>
```

The corresponding generated Ocaml code is the following : [1]

```
let f = [ Fiacre.TypeDecl ("t_err", Fiacre.Union
[ ("no", None) ; ("exit", None) ; ("other", None) ]) ]
```

Another interesting feature of Camlp4 is the ability of evaluating Ocaml expressions inside foreign language text through the anti-quotation mechanism which allows to insert computed target language elements into static source text.

[1] It is interesting to remark here what should have been written if our transformation had to deal with abstract syntax.

```
let input_type c t = <:texp<
    union $"resultat_"^c$ of $t$
        | $"error_"^c$ of fault
    end
```

We have defined in Camlp4 extensions of the FIACRE language which introduce indexed constructs (indexed Parallel operator, indexed Choice operator ...)[2].

Operator		Notation ASCII	Use
folded	Parallel	$\|\|\_\{i : l\}.p_i$	parallel composition of the p_i components (for i in the list l)
	Choice	$[\![\_\{i : l\}.a_i$	choice between the actions a_i
	Sequence	$;\_\{i : l\}.a_i$	sequence of actions a_i
	Union	$\|\_\{i : l\}.c_i[\textbf{of } t_i]$	declaration of variants
Concatenation		$\{\%\ Elem_1 ... Elem_n\ \%\}$	builds an identifier as a string concatenation
Meta evaluation		$ Meta expression $	evaluates the Meta expression
Parameters value		$'parameter$	return the parameter's value

Fig. 3. Meta extension of FIACRE constructs

For instance, the folded paralled operator (Fig. 3) over a list allows us to compose the nested acivities of a BPEL flow. Thanks to this notation, activities are given inline and their packaging as FIACRE components is done automatically.

4 From BPEL to FIACRE

4.1 Transformation Principles

The structure of the transformation is based on the structure of the BPEL process. Each BPEL construct is mapped to a FIACRE pattern. By the use of Camlp4, each pattern is defined through a function generating FIACRE code.

Transformation Structure: The static part of BPEL (WSDL) is modeled in FIACRE as global types. As for the dynamic part consisting of the primary activity of the BPEL process and its associated handlers, it is modeled as a toplevel FIACRE component containing the FIACRE pattern of the composition of the primary activity with these handlers (Fig. 4). This outermost component builds a parallel composition of component instances of the BPEL nested activities. Moreover, BPEL basic activities are transformed to FIACRE processes while BPEL structured activities and handlers – being able to contain other activities – are transformed to FIACRE components.

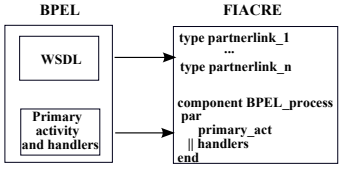


Fig. 4. Transformation structure

[2] p_i,a_i,c_i denote respectively process compositions, actions and type constructors depending on i.

Transformation Kernel: the transformation from BPEL to FIACRE consists of a set of *transformation patterns*. These patterns are defined as FIACRE-code generator functions written in the Camlp4 preprocessor language (see section 3). They are called when a transformation of a BPEL construct is needed which leads to the generation of the corresponding FIACRE pattern. Every function consists of two parts, an interface and a body.

1. The interface describes the parameters required by the function. These may be universal ones which are shared by all the activities, such as the name and the links conditions. They may also be activity-dependent such as the communication endpoints of BPEL communicating activities. Examples of such parameters are discussed later in this paper (see invoke and flow).
2. In case of a BPEL basic activity, a BPEL structured activity or handler, or a WSDL modeling, the body is represented by a FIACRE process, a FIACRE component, and a FIACRE type declarations respectively.

4.2 Modeling the WSDL

The interaction with the environment is supported by Partnerlinks. In BPEL, each Partnerlink may contain two roles (`myRole` and `partnerRole`) typed with `Porttype`. Each of the `Porttype` declares several operations used to receive (Input) or send (Output) messages. Consequently, this structure is modeled in FIACRE by creating two different enumerated types named `inputs` and `outputs` used to model respectively the inputs and the outputs of each operation. The type `inputs` (resp. `outputs`) will be the union of the type of :

- the `input` (resp. `output`) arguments of operations of the `myRole` of every Partnerlink.
- the `output` (resp. `input`) arguments of operations of the `partnerRole` of every PartnerLink.

The name of the constructor is built from the name of the partnerlink and a suffix (`mR_input` or `pR_output`) used for input (resp. `outputs`). As for the name of the constructor's type argument, it is built using the suffix `mR_input_type` or `pR_output_type`.

```
type inputs is union
  |_{pl:pls}.{% $pl_name pl$ _mR_input %} of {% $pl_name pl$ _mR_input_type %}

|
  |_{pl:pls}.{% $pl_name pl$ _pR_input %} of {% $pl_name pl$ _pR_output_type %}
end
```

Similarly, we define the `mR_input_type` and `pR_output_type` family of types by considering the porttypes and eventually their operations. Note that for abstraction purposes, only boolean and enumerated types are preserved during this

transformation. Actually, with respect to model checking purposes, considering the actual values is not realistic because of state explosion.

4.3 Behavioral Aspects in FIACRE

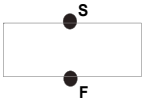


Fig. 5. Common interface

Each BPEL activity is mapped to a FIACRE component sharing a common interface. It will consist of the set of FIACRE ports and shared variables that each pattern should have in order to enable their composition. We start by introducing a basic interface (Fig. 5) containing 2 ports : S and F (start, finish) used to connect the activities in the composition. At the end of every activity, the finish port is synchronized with the start port of the subsequent activity. We will extend this interface progressively by what is discussed in this paper.

Common Behavior of Activities in FIACRE: All of the activities in FIACRE share a common behavior. This consists of modeling the outgoing and incoming links (with `suppJoinFailure = yes` [4]). Furthermore, each activity will include additional control events and shared variables used in FIACRE. These are used for modeling the forced termination of activities or for reinitializing the activity in case it is nested inside of a repeatable construct. In Fig. 6, we show the expected behavior of every BPEL activity. An activity is able to execute when its link counter reaches 0. A source activity signals its links by decrementing the counter associated to its targeted activities and by updating their respective transition conditions. This can happen at two occasions which are the case of the activity's normal execution or the case of the Dead Path Elimination (DPE) [3].

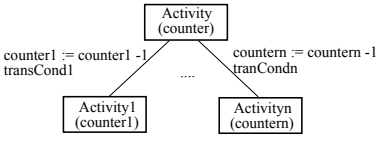


Fig. 6. Activities behavior

This common behavior is implemented by the `prologue` and `epilogue` functions. These functions define a set of FIACRE transitions using the quotation `<:ltrans<` and are parameterized with the activity `a` which is a meta-level function parameter. We give in the following the prologue function which is written in FIACRE code extended with indexed or meta constructs used to generate meta-parameters dependent code.

[3] The dead path elimination is the result of two cases : (i) The join condition of an activity evaluates to False. (ii) If during the execution of a structured activity A, its specification says that an activity B nested within A will not be executed.

```
let prologue a = let oa = a.outActs and ol = a.outLinks in
<:ltrans<
from {%init_ $a.atype$ %}
        select
            on $fexp:stopped a.sc$ ; stpd ;to {% init_ $a.atype$ %}
        □
            on not $fexp:stopped a.sc$ and reinit;
            $act: reinitialize a$; to {%init_ $a.atype$ %} (* call exernal function *)
        □
            on not $fexp:stopped a.sc$ and info.act_info[ $int:a.num$ ].counter=0
            and not info.act_info[$int:a.num$].skip and $a.joinCond$;
            start; to {%start_ $a.atype$ %}
        □
            on not $fexp:stopped a.sc$ and info.act_info[$int:a.num$ ].counter=0
            and not info.act_info[$int:a.num$].skip and not $a.joinCond$;
            start; to join_fail
        □
            on not $fexp:stopped a.sc$ and info.act_info[ $int:a.num$ ].counter=0
            and info.act_info[$int:a.num$].skip;
            ;_{i:oa}.info.act_info[$int:i$].counter:=info.act_info[$int:i$].counter-1;
            ;_{i:ol}. info.link_info[$int:i$] := false;
            info.act_info[$int:a.num$].finished := true;
            to {%init_ $a.atype$ %}
    end
from join_fail
        select
            on $fexp:stopped a.sc$; stpd; to {%init_ $a.atype$ %}
        □
            on not $fexp:stopped a.sc$;
            $act: links_cond_fail a$; (* call exernal function *)
            to {%init_ $a.atype$ %}
        end
>>
```

- The reinitialization request is modeled by a boolean variable `reinit`. In case of reinitialization request, the activity resets the values of all its variables by calling the function `reinitialize` (code line 8).
- The stopping demand is also modeled by a boolean variable. As a result of a stopping demand, the activity can no longer proceed and responds by communicating its `stpd` port.
- Normal execution : the targeted activities are able to execute once their counter evaluates to 0 and their join conditions evaluates to *True* (code lines 10-11). After their execution, they set their outgoing links. This is handled in the `epilogue` function. We do not show it here since its modeling is a special case of the `prologue` function (see `links_cond_fail` function).

```
let links_cond_fail a = let oa = a.outActs and ol = a.outLinks in
<:act<
  finish;
  ;_{i:oa}. info.act_info[$int:i$].counter := info.act_info[$int:i$].counter - 1;
  ;_{i:ol}. info.link_info[$int:i$] := false;
  info.act_info[ $int:a.num$ ].finished := true
>>
```

In this function, we make use of the indexed sequence operator. We first iterate over the list `oa` containing a variable number of outgoing activities `;_f{i : oa}` so their respective `counter` variables are decremented by 1. Same thing goes for the list of outgoing links so they are set to False.

- First case of the DPE: If the join condition evaluates to *False* (code line 14-15), the activty sets its outgoing links to False before terminating. This is done by calling the `links_cond_fail` function.
- Second case of the DPE : this is modeled by valuating the `skip` variable to *True* which is evaluated before the start of each activity (code lines 18-10). In this case, the same behavior of the `links_cond_fail` function is made.

Basic Activities. All of the BPEL basic activities are modeled in FIACRE by means of synchronizing ports or shared variables. For instance, the `receive`, `reply` and `invoke` activities are modeled with the two synchronizing ports (`c_msg, r_msg`) used to communicate with the environment. These ports are respectively typed with the `inputs` and `outputs` types. The BPEL variables are modeled with a shared variable `vars`. As for the BPEL faults, they are modeled by the synchronizing port `Fa`.

Due to the lack of space, we only give the asynchronous invoke pattern (Fig. 7) with which we illustrate our technique. More on the modeling of these patterns is given in [7]. Each pattern is implemented as a Camlp4 function containing FIACRE code. They all respect the common behavior of activities by making use of the `prologue` and `epilogue` functions.

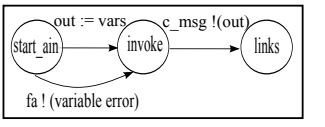


Fig. 7. Asynchronous Invoke

Asynchronous Invoke Pattern: in the following `invoke_async` function:

```
let invoke_async a pl pt op v t =
<:proc<
process $a.name$
    [start: none, finish: none, c_msg: inputs, r_msg: outputs,
    f: fault,stpd:none,reinited:none]
    ( &info: read  write t_info,&err_var: read  write t_scope_err,
      &rec_var: read  write t_var_record,& reinit : read bool) is
    states init_invoke, start_invoke,links_cond,join_fail,send_invoke
    var input: 't
    init to init_invoke
    $trans:prologue_bas a$  (* insert prologue transitions *)
    from start_invoke
        select
            on $fexp:stopped a.sc$; stpd; to init_invoke
        □
            on not $fexp:stopped a.sc$;
              select
                  input := rec_var.$v$;to send_invoke
              □
                  f ! F_variableError; to send_invoke
              end
        end
    from send_invoke
        select
            on $fexp:stopped a.sc$; stpd; to init_invoke
        □
            on not $fexp:stopped a.sc$;
            c_msg! {% 'pl _input%} ({% 'pt "_" 'pl _input%}
            ({% 'op "_" 'pt "_" 'pl _input%} (input)));
            to links_cond
        end
    $trans: epilogue a$  (* insert epilogue transitions *)
```

The interface of this function consists of the activity `a` and of the communication endpoint used by the invoke activity. That is the partnerlink `pl`, the portType `pt`, the operation `op`, the manipulated variable `v` and its associated type `t`. Concerning the `invoke` FIACRE process, it starts at the `start_invoke` state (code line 12) where the value of the input variable is copied into a local variable (code line 17) or a `F_variableError` (designates the BPEL default faults) is thrown (code line 19). By the use of the n-ary concatenation operator `% ...%`, a message is built from the `pl`, `pt` and `op` parameters and the contents of the variable (code lines 28-29). It is then sent through the `c_msg` port.

Structured Activities. They specify the order in which their nested activities are executed. The modeling in FIACRE consists in adding a Controller process based on the type of the BPEL structured activity. This controller specifies the way the nested components of structured activities are executed (sequential, parallel,...). The body of the structured activities functions consists in the composition of a controller process with a list of nested activities.

Flow Pattern In Fig. 8, the nested activities are composed with a flow controller that controls their concurrent execution. The controller starts by signaling simultaneously the start of all the nested activities (`SA`). Then, it does a blocking wait until the finish signals of all the activities are received asynchronously.

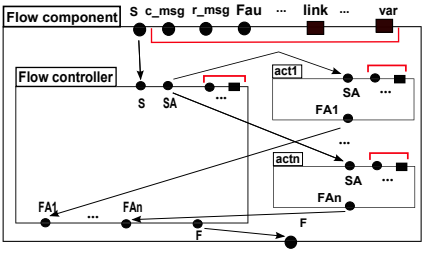


Fig. 8. Flow Component

Flow Function In the following, the interface of the function associated with the flow pattern consists of the common parameter `a` and a list of its nested activities. The flow component calls the `flow controller` function. This call generates the flow controller component. Afterwards, the flow controller is composed with the nested activities of the flow component using the indexed parallel construct $||\_i : l.p_i$ to produce the composition of a variable number of nested activities.

```
let flow a lc =
<:decls<
 $decl:flow_controller a (List.map (fun c → c.num) lc)$

component $a.name$
    [startflow: none,finishflow: none,c_msg: inputs,r_msg: outputs,f: fault,
stpd :none,reinited : none]
    ( &info: read write t_info,&err_var: read write t_scope_err,
      &rec_var: read write t_var_record,& reinit: read bool) is
  port start:none, {c:lc}. {%finish_ $string_of_int c.num$%}:none
  par * in
      {%flow_controller_ $a.name$ %}
        [startflow, finishflow, start,{c:lc}.
          {%finish_ $string_of_int c.num$ %}, stpd, reinited]
        ( &info,&err_var,& reinit)
  || ||_{c:lc}. $c.name$
        [start,{%finish_ $string_of_int c.num$ %},
         c_msg,r_msg,f,stpd,reinited]
        ( &info,&err_var,&rec_var,& reinit)
  end
```

4.4 Verification

Our verification framework supports three kinds of properties :

1. Structural properties : express the "well-formedness" of the BPEL code such as conflicting `receive` activities. They are verified by the Tina toolbox [22].
2. Temporal properties : typical properties written in LTL such as the safety, liveness and response properties. They are verified by the Tina toolbox.
3. Timed properties : written in MITL [3] which is a timed variant of LTL and are handled here using timed observers. They are verified via timed observers [7].

5 Case Study

We illustrate the use of the patterns by building the FIACRE code corresponding to the purchase order example Fig.9 [4].

Upon the reception of the purchase order, the process initiates concurrently two sequences of activities for calculating the final price and for finalizing the shipment. However the final price cannot be calculated before the reception of the shipping price. This is represented by a link from the `Invoke Shipping` to the `Invoke Complete Price Calculation`. Once the two tasks are complete, an invoice is sent to the customer.

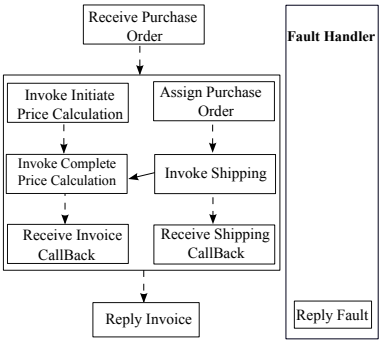


Fig. 9. Purchase example

Use Case Modeling and Verification

In the following, we give excerpts of the code used to generate the executable FIACRE code of this example. In the first excerpt we show how activities information are declared. The first activity is named `seq_purchase` and is referenced with the number `0`. This activity have neither incoming links (`nbInLinks =0`), nor outgoing links. This is why the list of outgoing links are empty. As for the `invoke_shipping`, it contains an outgoing link towards the activity number `7` which is the reference number of the `invoke initiate price calculation` activity. All the activities are specified accordingly.

```
let seq = {
  name = "seq_purchase";
  num = 0;
  sc = "BPEL_process";
  atype = "sequence";
  joinCond = transfo_jcond link_names "true";
  nbInLinks = 0;
  outLinks = □;
  outActs = □;
}
        ...  (* apply it to all the activities *)

  in let invShip = {
  name = "invoke_shipping";
  num = 10;
  sc = "BPEL_process";
  atype = "invoke";
  joinCond = transfo_jcond link_names "$true";
  nbInLinks = 0;
  outLinks = [0];
  outActs = [7];
              ...
```

Finally, generator functions are called for each activity by providing to each of them the parameters that we discussed earlier. The execution of these functions generates the FIACRE code of the example.

```
let activities = [sequence_purchase; rec_purchase; ...] in
let f = <:fiacre<
    type anytype is union
      |_{t:pls_ftype pls}. 't
    end

    type fault is union
      |_{f:pls_fault pls}. 'f of anytype
      | F_variableError
      | F_expressionError
    end

    type t_var_record is record
      ,_{v:vars}. $fst v$ : $snd v$
    end

    $decl:sequence seq [rec_purchase;flowAct;rep]
    $decl:receive rec_purchase "purchasing" "purchaseorderPT"
      "sendPurchaseOrder" "PO" "POmessage"$
    $decl:flow flowAct [seqPrice; SeqShip]$
      ...
>>
```

6 Related Works

Various works have been pursued in the area of modeling and verification of BPEL processes. Most of this work [21], [8], [11], [18], [16], [2] is based on defining transformation patterns for the BPEL constructs and then applying proving or model checking techniques on the result. Some of the works [21,8] considered a mapping towards process algebra languages like CCS and LOTOS respectively. Some [11,18,16] have also considered transformations to Petri Nets. Others [17] have addressed transformations towards Promela and used the model checker Spin [12]. Approaches based on theorem proving methods have also been used. We quote the work of [2] in which a mapping from BPEL to Event_B has been done. A comparison of our transformation w.r.t the mentioned work is given in [7]. Here we only focus on the transformation's tooling aspects.

6.1 BPEL Verification Tools

We consider the comparison with techniques used for building verification tools based on transformation patterns. For this purpose, we define some useful comparison criteria. The first criteria is that the transformation patterns are isolated, explicitly given and not embedded in the generation tools. This allows the clear understanding of these patterns and hence the validation of their correctness. The second criteria is the implementation language of these patterns. The ideal is that this language is the same as the target language of the transformation. The last criteria is whether the transformation produces text or an abstract syntax tree. The latter case allows the easy writing of post-transformations and the generation of syntactically correct target code by construction.

First, the tool BPEL2PN [11] generates Petri Nets in AST but the patterns are written in Java. In [16] the author presents the `bpel2owfn` tool which generates Open Workflow Nets. In this tool, the Petri Nets patterns of the BPEL constructs are isolated from the code generator module. However, the drawback of their technique is that these patterns are written in C++ where instances of the

Tool	BPEL2PN	bpel2owfn	bpel2b	activeBPEL	WSAT	Our proposal
Explicit patterns	–	C++	–	–	–	$\frac{FIACRE}{Camlp4}$
spec. formalism	Petri Nets	Petri Nets	Event-B	TA	Aut.	Ext. FIACRE
impl. language	Java	C++	Java	Java	Java	Ext. FIACRE
Target language	Petri Nets (AST)	Petri Nets (AST)	B(AST)	TA (UPPAAL)	Promela	FIACRE(AST)

Fig. 10. Transformation Patterns Tools

models are created. Consequently, the representation of these patterns is hard to understand. The transformation towards Event_B is embedded in the `BPEL2B` tool. The patterns in this technique are written in Java and suffer from the same drawback of unreadability. In [20] the authors integrate an algorithm for mapping BPEL constructs to Timed Automata (TA) patterns [19] in the `ActiveBPEL` tool. Yet, these patterns are defined mathematically and are not ready to be directly executed. We finally cite the `WSAT` tool [9] in which the patterns are defined graphically which again can not directly be executed. To the best of our knowledge, the criteria of readability of the transformations remains a research goal that has not been yet considered as such.

In the following table (Fig. 10), we give an overview of the related works concerning the representations of the transformation patterns.

6.2 Comparison of Transformation Environments

We cite other Meta-tools that we may have used in our transformation but did not because they do not meet our criteria needs. For instance, Cpp (C preprocessor) supports the expression of some transformation patterns but is only text based and is not powerful enough to manage the use of indexed constructs. Acceleo [1] is a Model to Text tool (M2T) which could be used to define transformation patterns in textual form but the generated code would also be textual. ATL [15] is a Model to Model (M2M) transformation tool but the patterns need to be given at the abstract level which is usually hard to express and read. To finish, Java can also be used as a Model to Text or as a Model to Model tool. However, in both cases, the readability of the generated patterns is low.

Example: this example shows how the FIACRE union type `inputs` (sect. 4.2) can be generated in other formalisms. This type contains two variants for each partnerlink. They describe the inputs (resp. outputs) of operations of process roles (resp. partner roles) attached to the partnerlink. Even with a simple example, the readability and the compactness of our patterns compared to others can be appreciated.

M2T Tools - Acceleo and Java: this transformation operates on a BPEL process. It generates a string written in FIACRE concrete syntax declaring the `inputs` union type. What makes Acceleo better readable than Java is that in the former, Acceleo code is injected into strings while it is the other way around in the latter.

Furthermore, Acceleo uses OCL and iterators. However in both cases, there is no guarantee of the syntactical correctness of the generated string. Hence, it needs to be checked after each transformation.

Acceleo	Java

[template public generateProcess (p : Process)]
type inputs is union
[for (pl : PartnerLink | p.partnerLinks.children) separator (' \n| ')]
[pl.name/] of [pl.name/]_ty
[/for]
end
[/template]
	private static String genInputs1(
	org.eclipse.bpel.model.Process p) {
	String res = "type inputs is union";
	for (PartnerLink pl : p.getPartnerLinks()) {
	res += " | " + pl.getName() + "_mR_input of "
	+ pl.getName() + "mR_input_type" + "\n";
	res += " | " + pl.getName() + "_pR_output of "
	+ pl.getName() + "pR_output_type" + "\n";
	}
	return res;
	}

M2M Tools - ATL and Java: this transformation operates as well on a BPEL process. However, it does not generate a string as earlier but rather a program which builds an instance of the FIACRE metamodel. In the case of Java, this program can be type checked to guarantee the well-formedness of the instance. However, at the contrary of M2T tools, we reason on the abstract syntax of the target language. The drawback of such tools is that the generated code tend to be unreadable for those who are not familiar with the internal representation of the target language (FIACRE).

ATL	Java

rule Process2Program {
from p: BPEL!Process
to prg: FIACRE!Program (decls <-
p.partnerLinks.children->add(d)),
d: FIACRE!TypeDecl(name <- 'inputs',
is <- tu),
tu: FIACRE!Union(constr <-
p.partnerLinks.children
->collect(pl |
thisModule.makeInputConstr(pl)))
}
 	
lazy rule makeInputConstr {
from pl: BPEL!PartnerLink
to c:FIACRE!Constr(name <- pl.name,
type <- ty),
ty: FIACRE!TypeId(decl <- pl)
}
	private static TypeDecl genInputs2
	(org.eclipse.bpel.model.Process p) {
	TypeDecl res = FiacreFactory.eINSTANCE.
	createTypeDecl();
	Union u = FiacreFactory.eINSTANCE.
	createUnion();
	res.setName("inputs");
	res.setIs(u);
	for (PartnerLink pl : p.getPartnerLinks()){
	Constr ci = FiacreFactory.eINSTANCE.
	createConstr();
	u.getConstr().add(ci);
	ci.setName(pl.getName() + "_mR_input");
	//ci.setType(typeref);
	Constr co = FiacreFactory.eINSTANCE.
	createConstr();
	u.getConstr().add(co);
	co.setName(pl.getName() + "_pR_output");
	//co.setType(typeref);
	}
	return res;
	}

Discussion: The Camlp4 environment allows the generation of metamodel instances - not only strings - while offering a readable syntax close to string generators (M2T). The gained readability makes it easier to informally validate the transformation patterns against their informal specifications. Such a task would be very hard to fulfill in the case the other referenced techniques were used.

7 Conclusion and Future Work

We have presented a transformation and verification framework for BPEL. The framework is based on a set of transformation patterns from BPEL to FIACRE. In order to enhance the readability of the proposed patterns, we have suggested an original idea of how these transformation patterns can be defined in a source code close to the target's language code. This is done using the Camlp4/Ocaml technology. Another main feature is that the generated code is an AST form and hence it is syntactically correct by construction and ready for post manipulations. We are convinced that such a readability of the patterns, offered to the tool designers, is a mandatory first step for the correctness and consequently the safety of these transformations. We have showed on an example how these patterns can be easily instantiated to build the FIACRE model. For verification purposes, temporal and timed properties can be validated on such a model.

For the future works, we are currently working on the addition of annotations supporting assertions about the BPEL execution. The aim is to verify statically these assertions through the FIACRE transformation by model checking.

References

1. Acceleo, `http://www.acceleo.org/doc/obeo/en/acceleo-2.6-reference.pdf`
2. Ait-Sadoune, I., Ait-Ameur, Y.: A proof based approach for modelling and verifying web services compositions. In: Proc. of the 14th Int. Conf. on Engineering of Complex Computer Systems, pp. 1–10. IEEE, Washington, USA (2009)
3. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality (1996)
4. Alves, A., Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., König, D., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language Version 2.0. OASIS Committee Draft (May 2006)
5. Berglund, A., Boag, S., Chamberlin, D., Fernández, M.F., Kay, M., Robie, J., Siméon, J.: XML Path Language (XPath) 2.0 (W3C Recommendation) (January 2007)
6. Farail, P., Gaufillet, P., Peres, F., Bodeveix, J.-P., Filali, M., Berthomieu, B., Rodrigo, S., Vernadat, F., Garavel, H., Lang, F.: FIACRE: an intermediate language for model verification in the TOPCASED environment. In: ERTS, Toulouse, January 29-February 01 (2008), `http://www.see.asso.fr`
7. Fares, E., Bodeveix, J.-P., Filali, M.: Verification of Timed BPEL 2.0 Models. In: Halpin, T., Nurcan, S., Krogstie, J., Soffer, P., Proper, E., Schmidt, R., Bider, I. (eds.) BPMDS 2011 and EMMSAD 2011. LNBIP, vol. 81, pp. 261–275. Springer, Heidelberg (2011)
8. Ferrara, A.: Web services: a process algebra approach. In: ICSOC 2004: Proceedings of the 2nd International Conference on Service Oriented Computing, pp. 242–251. ACM Press, New York (2004)
9. Fu, X., Bultan, T., Su, J.: WSAT: A Tool for Formal Analysis of Web Services. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 510–514. Springer, Heidelberg (2004)
10. Henzinger, T.A., Manna, Z., Pnueli, A.: Temporal proof methodologies for timed transition systems. Inf. Comput. 112(2), 273–337 (1994)

11. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri Nets. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 220–235. Springer, Heidelberg (2005)
12. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (September 2003)
13. INRIA, `http://caml.inria.fr/pub/docs/manual-camlp4`
14. INRIA, `http://caml.inria.fr/pub/docs/manual-ocaml`
15. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
16. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: van Hee, K., Reisig, W., Wolf, K. (eds.) Proceedings of the Workshop on Formal Approaches to Business Processes and Web Services, pp. 21–35 (June 2007)
17. Nakajima, S.: Model-checking behavioral specification of bpel applications. Electron. Notes Theor. Comput. Sci. 151, 89–105 (2006)
18. Ouyang, C., Verbeek, E., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in ws-bpel. Technical report (2005)
19. Pu, G., Zhao, X., Wang, S., Qiu, Z.: Towards the semantics and verification of bpel4ws. Electron. Notes Theor. Comput. Sci. 151, 33–52 (2006)
20. Qian, Y., Xu, Y., Wang, Z., Pu, G., Zhu, H., Cai, C.: Tool Support for BPEL Verification in ActiveBPEL Engine. In: 18th Australian Software Engineering Conference, ASWEC 2007, pp. 90–100 (April 2007)
21. Salaun, G., Bordeaux, L., Schaerf, M.: Describing and reasoning on web services using process algebra. In: IEEE International Conference on Web Services, p. 43 (2004)
22. Tina, `http://homepages.laas.fr/bernard/tina`

Applying Process Analysis to the Italian eGovernment Enterprise Architecture*

Roberto Bruni[1], Andrea Corradini[1], Gianluigi Ferrari[1], Tito Flagella[2], Roberto Guanciale[1], and Giorgio Spagnolo[1]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Link.it, Pisa, Italy

Abstract. We report our experiences gained when integrating process analysis activities into a regional gateway of the Italian eGov platform to promote real-time process monitoring within a Service Oriented Architecture. We exploit ProM, a state-of-the-art suite providing several analysis algorithms for business processes. First, we outline our technological integration efforts, focusing on the architectural changes and implementation strategies to make ProM tools available at runtime for monitoring the gateway. Next we improve an existing performance algorithm with a new approach to deal with invisible transitions when evaluating the synchronization times of complex nets. Finally, we introduce a methodology to transform high level process notations, like BPMN, to Petri Nets in order to enact the analysis techniques and convey back their results.

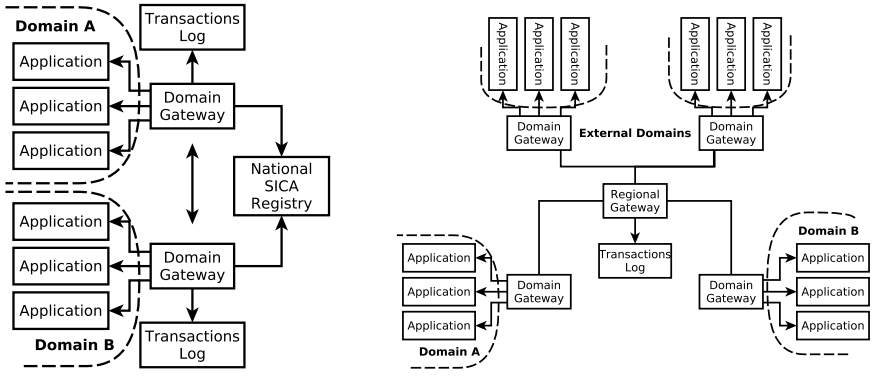
1 Introduction

In 2003 the CNIPA (National Center for IT in Public Administration, now DigitPA) began the specification of an Enterprise Architecture for ensuring interoperability among the software applications of the Italian Public Administrations. The resulting architecture is called SPCoop (*Public Cooperative System*) and it is based on a Service Oriented Architecture model. It was designed for standardizing both the service agreements (*contracts*) between applications of different domains and the communication protocol used for application interoperability: the former contain both formal (XSD, WSDL, ...) and semiformal documentation concerning service interoperability details; the latter is an extension of the SOAP 1.1 envelope, called *e-Gov Envelop* which consists of a custom SOAP header carrying various message addressing information.

According to the SPCoop specification, every Public Administration must offer its application services through a unique entity named *Porta di Dominio* (Domain Gateway), which also acts as a proxy for invocations of remote services. Thus the Domain Gateway interoperates with gateways of other domains for accepting or delivering messages to and from application services. Domain

* Research partially funded by Regione Toscana through project RUPOS *(Ricerca sull'Usabilità delle Piattaforme Orientate ai Servizi)* and by the EU Integrated Project 257414 ASCENS.

M. Carbone and J.-M. Petit (Eds.): WS-FM 2011, LNCS 7176, pp. 111–127, 2012.



(a) Standard Logging (b) Centralized Logging

Fig. 1. Business Transactions Logging in SPCoop

Gateways act according to service agreements recorded in a central registry named SICA (*Interoperability and Application Cooperation Service Registry*).

After some years of experimentation, on February 2005 the Italian Government issued the Law Decree n. 42 [15] which establishes the conceptual architecture and the technical and governance rules of SPCoop, giving in this way legal value to interactions among SOA applications built over the standard. We refer to [5] for a comprehensive description of the SPCoop architecture.

In 2005 Link.it, a spin-off of the Computer Science Department in Pisa, started an open-source project, named *OpenSPCoop* [10], having the goal of implementing all the infrastructural components required by the SPCoop architecture. Among them, the *Domain Gateway* and the *Service Registry*. The software developed within the project quickly became the reference SPCoop implementation and is widely adopted in the main national eGov projects. More details about OpenSPCoop can be found, in Italian, in [7].

As well explained in [5], one of the hardest challenges in the SPCoop architecture is to define metrics for measuring the service level agreements (SLAs) and to design a system for monitoring the SLAs. In fact SPCoop can host potentially thousands of service providers, each having potentially different SLAs for each client, even for the same service. Nevertheless, as a consequence of the availability of a standard product managing all eGov business transactions, a uniform repository of all service transaction logs is available in organizations adopting SPCoop, as shown in Fig. 1a. The case of geographical federations of Public Organizations, e.g. Regional Entities, is particularly interesting. In this case, SPCoop requires to have a centralized regional gateway, logging both intra-regional and inter-regional application communications, as shown in Fig. 1b.

This scenario makes it possible and extremely interesting enabling business process analysis on a Domain Gateway as a mean to monitor SLAs. In this paper we report about the current activities aimed at extending the OpenSPCoop

platform with monitoring functionalities based on the exploitation of system's logs for the performance and conformance analysis of business process models.

After introducing the main concepts concerning Business Process Management in Section 2, in Section 3 we present the Monitoring Framework designed to extend OpenSPCoop. In particular we describe how the ProM framework [9] has been integrated in the platform in order to exploit some of its analysis algorithms. In Section 4 we discuss the analysis algorithms currently available in the platform, and in Section 5 we present the ongoing activity aimed at providing automated support to the translation of BPMN models into Petri nets, and to read back the analysis results on the original BPMN models. We assume the readers have some familiarity with Petri nets' basics and notation.

2 Business Process Management

Business Process Management (BPM) is a young discipline related to the understanding, design, organization, enactment and improvement of the tasks to be performed to carry out some specific goal. BPM calls for a paradigm shift, from the data-centered to the process-centered view. The idea is to develop suitable artifacts, called *process representations*, that can be used to coordinate a generic software system that enacts the business process. As the process representation must be agreed upon by different stakeholders, ranging from the business domain experts and knowledge workers to the system architect and developers, graphical (workflow-like) languages are the best suited candidates. Over the years, several notations have been proposed to the purpose, often pushed by large industrial consortia, supported by various platforms and integrated in mainstream development environments. Some successful examples are Event-driven Process Chains (EPC) [2], the Business Process Execution Language (BPEL) [13] and the Business Process Modeling Notation (BPMN) [14], which has recently become a widely adopted standard. There are three main categories of flow objects in BPMN: 1) events denote something that happens during the course of a business process; 2) activities denote units of work to be accomplished during the course of a business process; 3) gateways denote the splitting and joining of workflow paths. Events are represented as circles, activities as rounded boxes and gateways as diamond shapes. Different kinds of decorations are introduced to clarify the nature of the flow object. For example, there can be start, intermediate and end events and different symbols are used for them.

Albeit the syntax of process notations is always defined very precisely, e.g. as XML schemes, their semantics is often described only verbally. Inevitably, any formal analysis for business processes must go through a rigorous semantics and in the recent literature several models have been used to address the issue (e.g. π-calculus [11], ASM [6], Petri nets [16] and in particular workflow nets [1]).

In this paper, we exploit Petri nets as underlying formal support to analyze process representations. We argue that Petri nets are a convenient solution, because several tools are available for their analysis, and their graphical notation can serve to report the outcome of the analysis in a form that is relatively close

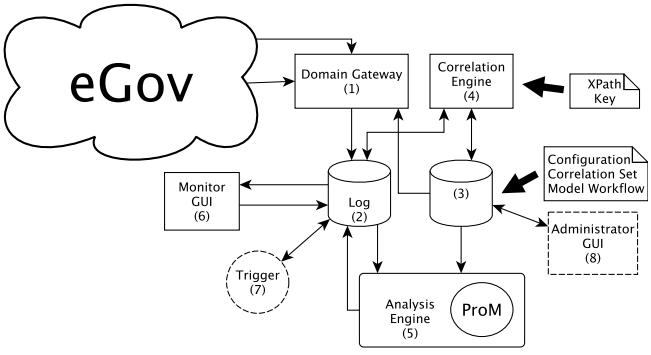


Fig. 2. The OpenSPCoop Business Process Monitoring Framework

to the original artifact. Roughly, BPMN events are encoded as places, tasks as transitions and gateways as net fragments involving both places and transitions.

Hereafter, we will focus the formalization on the so called *evaluation phase*, i.e. the business activity monitoring phase, that can provide relevant feedback about the effectiveness of the business process after deployment. The runtime data to be evaluated are provided by the middleware in terms of system events, that are collected in log files. Typical questions to be addressed are the *conformance checking* and the *performance evaluation*. The conformance checking allows one to spot the discrepancies between the planned process and its realization. The main technique used for conformance checking is the so-called *log replay* (see Section 4). The performance evaluation, instead, computes quantitative values allowing us to measure the deployed process. For example, timestamps can be used to calculate important parameters (latency, synchronization time) for dimensioning e.g. message buffers.

From a methodological perspective, to enable these analysis it must be the case that: 1) a formal model of the process is available; 2) all relevant activities are logged, 3) activities in the log must be correlated if they belong to the same instance of the process, or kept separated if they belong to different instances (or to different processes), 4) activities in the log are temporally ordered, e.g. using timestamps. It is quite common that for each activity, the beginning and the end are registered separately in the log, with different timestamps.

3 The Process Monitoring Platform

In order to equip the Italian eGovernment Enterprise Architecture with a business process monitoring system, we extended the architecture of OpenSPCoop as shown in Fig. 2. The Domain Gateway (1) is the existing software infrastructure intercepting all communications from and to a Public Administration.

In perspective, the idea is to instantiate the framework to a Regional Gateway, in order to exploit the centralized logging at regional level: preliminary experimentation are carried on within a project funded by Regione Toscana and are likely to be extended to other Italian regions.

As mentioned in Section 2, in order to enable process analysis techniques, audit logs must be grouped into *traces* in the log repository (2), each of them representing a unique process instance. To this aim, the Domain Gateway has been enriched with XPath support: a database (3) is used to configure the XPath expressions that have to be evaluated by the Domain Gateway on the intercepted messages in order to extract the information required to identify the proper business process instance. The Correlation Engine (4) is responsible to structure the Domain Gateway logs and to group isolated events into process instances. The configuration database stores the correlation sets required to link the XPath expressions extracted by the domain gateway. Decoupling the Domain Gateway and the Correlation Engine allows us to reconstruct the process instances with minimal effect on the message throughput.

The Analysis Engine (5) is responsible for loading the traces from the log database, applying the analysis algorithms to them, and recording their results. Driven by the application domain, we started focusing on two kinds of analysis, namely *conformance* and *performance analysis* of log traces representing process instances with respect to the process model formalized as a Petri net.

As described below, the Analysis Engine exploits algorithms that are executed in ProM [9]. This is an extensible, integrated framework that supports a wide variety of process mining and analysis techniques in the form of plug-ins. We adopted Version 6 of ProM because its well-structured and modular architecture features a careful separation between analysis algorithms and graphical user interface that simplified the task of integrating it into the Analysis Engine. On the negative side, not all plug-ins of the previous major release, ProM 5.2, have been ported (yet) to ProM 6. For example, a ProM 6 plug-in is already available [12] to handle the conformance analysis, but there was no plug-in to evaluate the performance of Petri nets. Therefore we implemented a new ProM 6 plug-in for performance analysis taking inspiration from the techniques adopted in ProM 5.2 but with a different handling of the "invisible transitions", as explained in the next section.

The log database can be used by external components to retrieve execution traces and the corresponding analysis results, enabling the framework to further extensions including user interface components (6) that graphically represent the results (e.g. a standalone GUI, or a web-based monitoring console) and supervisor components (7) that trigger computations whenever some constraints are violated (e.g. alert the system administrator if a process does not respect a SLA or if it performs unexpected execution). The whole system is equipped with an administration GUI (8) that manages the configuration database including XPath expressions and correlation sets, as well as the formal business process representation in the form of Petri nets or more abstract formalisms like BPMN.

4 Formal Analysis Based on Petri Nets

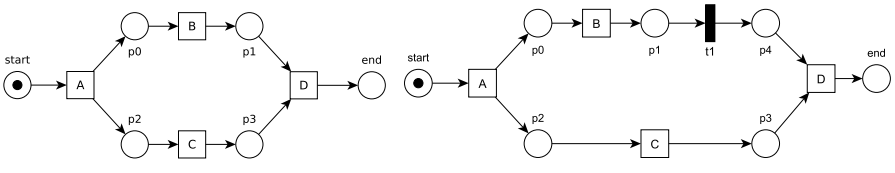
The current version of the Analysis Engine of the OpenSPCoop monitoring platform supports conformance and performance analysis of the logs against a model of the business process represented as a Petri net.

The basic building blocks of logs are events. An event e can be seen abstractly as a pair $e = (a, t)$ representing an action a recorded by the Domain Gateway and the corresponding timestamp t. In the following we denote by $\alpha(e)$ and $\theta(e)$ the action and timestamp of e, respectively. Events that belong to the same process instance are grouped into *traces*. Formally, a trace T is a finite sequence of events $T[1], \ldots, T[n]$ such that $\theta(T[i]) \leq \theta(T[i+1])$ for all $i \in [1, n)$. A *log* L is a set of traces, recording the activity performed by a system during a finite number of process executions. We assume here that all traces are instances of the same process and that for each action there exists a corresponding transition in the net, that for simplicity will be denoted with the same name of the action.

The key algorithm exploited to analyze the Petri net model with respect to the log is the *log replay* algorithm [17]. For each trace in the log, the log replay starts by placing one token in the start place of the net. For each event in the trace the corresponding transition is fired in a *non-blocking way*, and the marking of the net is updated. "Non-blocking replay" means that if the log replay requires to fire a transition that is not enabled, the missing tokens are artificially created.

In general, there could exist transitions of the Petri net that are not associated with any logged action: such transitions are called *invisible*. This can happen in several cases: for example, the transition models an internal choice that is not visible in the system, or it is used to implement a connector of a more abstract modeling language (as we shall see later for BPMN). Usually, invisible transitions are considered to be lazy [4], i.e., when firing a visible transition t corresponding to an event of the trace, only then the invisible transitions enabling t are fired. For example, considering the net in Fig. 3b, if the log does not contain actions corresponding to transition $t1$ then it is handled as invisible, and its firing is delayed until an action corresponding to D is found in the trace. Further details on the log replay algorithm can be found in [17].

The output of the log replay of a trace T can be represented as an ordered list R of pairs (tr, i), representing that the transition tr has been fired to mimic the event $T[i]$. Note that the presence of invisible transitions can result in



(a) Only visible transitions (b) One invisible transition, $t1$

Fig. 3. Two Petri nets with concurrent branches

several transitions fired to mimic a single event. In presence of recursion several occurrences of the same transition are also possible.

The result of the log replay can be used to evaluate conformance and performance of the Petri net model. Conformance problems can be discovered by analyzing the tokens that have been artificially created (the *missing tokens*) and the tokens that were not consumed (the *remaining tokens*). For example, let us consider the net in Fig. 3a and the traces $T = [(A, 1s), (B, 2s), (C, 4s), (D, 8s)]$, $T' = [(A, 1s)]$ and $T'' = [(B, 1s)]$. Trace T is compliant with the net: the log replay does not identify any missing token, terminates with a marking containing one token in the end place, and returns the sequence $[(A, 1), (B, 2), (C, 3), (D, 4)]$. The log replay for T' does not produce any missing token but the remaining tokens are $\{p0 \rightarrow 1, p2 \rightarrow 1\}$: the trace is a partial execution and B and C should be executed to continue the process. The log replay for T'' terminates with a missing token $\{p0 \rightarrow 1\}$ and remaining tokens $\{start \rightarrow 1, p1 \rightarrow 1\}$: the action A was skipped by the execution.

Since the logs contain timestamps, the log replay can be used to compute performance measures of the system [3]. The idea is to calculate the time interval between production and use of tokens in each place. This technique can be applied only to traces that do not require the production of missing tokens, because such tokens cannot have time information. During the log replay the following metrics can be computed for each trace and each place:

- *sojourn time* ($\bowtie$): the time interval between arrival and departure of a token;
- *synchronization time* ($\ltimes$): the time interval between arrival of a token in the place and enabling of a transition in the post-set of the place;
- *waiting time* ($\rtimes$): the time interval between enabling of a transition in the post-set of the place and token departure (thus $\ltimes + \rtimes = \bowtie$).

To clarify the metrics evaluated by this technique we exploit the Petri net depicted in Fig. 3a and the traces in Table 1a. The net starts with the transition A, concurrently fires B and C and terminates after the transition D. When applying the log replay to trace T_1 we cannot associate performance metrics to the starting place, since the trace does not carry information about the start time of the process. The first event in the trace records the firing of transition A at time $1s$, thus tokens arrive at $p0$ and $p2$ at time $1s$. Since both $p0$ and $p2$ have an enabled transition at time $1s$, their synchronization times are $\ltimes(p0) = \ltimes(p2) = 0s$. The next event of the trace is $(B, 2s)$: the token in $p0$ is consumed and a token is produced in $p1$. The sojourn time of $p0$ is then evaluated as $\bowtie(p0) = 2s - 1s = 1s$. The log replay continues with the third event recorded: the firing of C at time $4s$. The token in $p2$ is consumed and a new token is produced in $p3$. The sojourn time of $p2$ is evaluated as $\bowtie(p2) = 4s - 1s = 3s$. Notice that now $(4s)$ transition D is enabled and thus the synchronization times of its pre-set are $\ltimes(p1) = 4s - 2s = 2s$ and $\ltimes(p3) = 4s - 4s = 0s$. While processing the last recorded event $(D, 8)$ the log replay consumes the tokens in $p1$ and $p3$, and their sojourn times are $\bowtie(p1) = 8s - 2s = 6s$ and $\bowtie(p3) = 8s - 4s = 4s$.

Table 1b reports performance details for three traces and their averages. We omit all the columns for $\rtimes$ as we recall that $\rtimes = \bowtie - \ltimes$. In this example $\ltimes(p1)$

Table 1. Traces and performance results with different evaluation methods

(a) System Log

T_1	T_2	T_3
(A, 1s)	(A, 3s)	(A, 9s)
(B, 2s)	(B, 5s)	(C, 10s)
(C, 4s)	(C, 8s)	(B, 11s)
(D, 8s)	(D, 11s)	(D, 15s)

(b) Results for net of Fig. 3a

	T_1		T_2		T_3		Average	
	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$
p0	1	0	2	0	2	0	1.67	0
p1	6	4	6	3	3	0	5	2.33
p2	3	0	5	0	1	0	2	0
p3	4	0	3	0	5	1	4	0.33

(c) Net of Fig. 3b (lazy)

	T_1		T_2		T_3		Average	
	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$
p0	1	0	2	0	2	0	1.67	0
p1	6	0	6	0	4	0	5.33	0
p2	3	0	5	0	1	0	2	0
p3	4	4	3	3	5	5	4	4
p4	0	0	0	0	0	0	0	0

(d) Net of Fig. 3b (ProM 5.2)

	T_1		T_2		T_3		Average	
	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$
p0	1	0	2	0	2	0	1.67	0
p1	2	0	3	0	0	0	2.67	0
p2	3	0	5	0	1	0	2	0
p3	4	0	3	0	5	1	0	0.33
p4	4	0	3	0	4	0	0	0

(e) Net of Fig. 3b (eager)

	T_1		T_2		T_3		Average	
	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$	$\bowtie$	$\ltimes$
p0	1	0	2	0	2	0	1.67	0
p1	0	0	0	0	0	0	0	0
p2	3	0	5	0	1	0	2	0
p3	4	0	3	0	5	1	0	0.33
p4	6	4	6	3	3	0	5	2.33

is usually greater than $\ltimes(p3)$, namely, transition B is fired almost always before C and the transition that waits more time to be fired after its activation is D.

We extend the previous example to stress the role of invisible transitions, referring to the net of Fig. 3b. Handling invisible transitions as lazy, the log replay for trace T_1 yields the sequence $[(A,1),(B,2),(C,3),(t1,4),(D,4)]$. Notice that, even if transition $t1$ is enabled after the event $T_1[2]$, it is delayed until its activation is required by the visible transition D. Evaluating performance of this Petri net using this approach yields the results in Table 1c. Notice that both $p1$ and $p4$ have no synchronization time and that the synchronization time of $p3$ is greater than zero even when C is fired after B: in our opinion these measures do not interpret faithfully the process instance.

A slightly different strategy (used in ProM 5.2) generates the same sequence of transitions, but instead of associating the invisible transition with the triggering event, it uses the last replayed event. Thus for trace T_1 the sequence $[(A,1),(B,2),(C,3),(t1,3),(D,4)]$ is returned, and the corresponding performance measures are reported in Table 1d. This approach allows us to correctly evaluate synchronization times whenever C precedes B, but fails otherwise.

Implementing the new performance plug-in for ProM 6 we exploited a different strategy, motivated not only by the measures just analysed, but also by the need of projecting the performance measures from the net back to the BPMN model, as discussed later. Intuitively, before associating with places the performance measures, we rearrange the sequence of transitions returned by the log replay in order to fire invisible transitions as soon as possible. We call the resulting sequences *eager* and define them formally as follows.

Definition 1 (last visible transition). *Let* $R = [(tr_1, i_1), (tr_2, i_2), ..., (tr_n, i_n)]$ *be a sequence of transitions and corresponding event indexes, and let* $j \leq n$.

```
for j from 2 to size(R) do
  tr, i ← R[j]
  if tr is invisible then
    k, done ← j − 1, false
    while k > 0 ∧ ¬done do
      tr_prev, i_prev ← R[k]
      if •tr ∩ tr•_prev ≠ ∅ then
        done ← true
      else
        R[k + 1] ← (tr_prev, i_prev)
        R[k] ← (tr, max(R ↓ k, 1))
        k ← k − 1
      end if
    end while
  end if
end for
```

Fig. 4. Conversion of a generic sequence R to a corresponding eager sequence

We define the last visible transition $R \downarrow j$ *of* R *before the* j*th transition as*

$$R \downarrow j = \begin{cases} -1 & \text{if } j \leq 0 \\ j-1 & \text{if } j \in [2,n] \text{ and } R[j-1] \text{ is visible} \\ R \downarrow (j-1) & \text{otherwise} \end{cases}$$

Definition 2 (eager sequence). *Let* $R = [(tr_1, i_1), (tr_2, i_2), ..., (tr_n, i_n)]$ *be a sequence as above. Then* R *is* eager *if for all* $j \in [1, n]$ *one of the following holds*[1]

- tr_j *is visible*
- tr_j *is invisible and either* $R \downarrow j = -1$ *or* ${}^\bullet tr_j \cap tr^\bullet_{R \downarrow j} \neq \emptyset$

Intuitively, a sequence is eager if for each invisible transition t the last preceding visible transition enables t, if it exists. Converting a generic sequence R obtained from a log replay to a corresponding eager sequence can be done easily with the algorithm in Fig. 4.

For example, by applying the algorithm to the trace T_1 we obtain the eager sequence $[(A,1),(B,2),(t1,2),(C,3),(D,4)]$, and the corresponding performance measures are reported in Table 1e. Notice that now only places $p3$ and $p4$ can have a synchronization time greater than zero: in our opinion these measures describe more faithfully the evolution of the net. Section 5 further motivates our approach, by describing advantages of interpreting invisible transitions as "eager" when the performance measures of the Petri net must be projected back to business process models at the higher level of abstraction.

4.1 Using Invisible Transitions to Improve Performance Analysis

As described above, performance analysis associates suitable measures with the *places* of a net. But often one is rather interested in measures related with the

[1] As usual, by ${}^\bullet t$ we denote the *pre-set* of transition t, and by $t^\bullet$ its *post-set*.

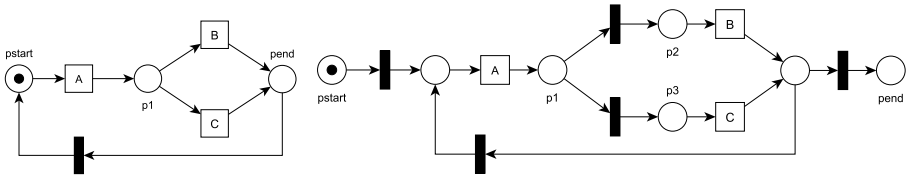


Fig. 5. Improve performance analysis by introducing invisible transitions

visible transitions, which represent system activities, like their *waiting time* or *duration*. In the nets of Fig. 3 the waiting time of each transition can be identified with the largest among the waiting times of the places in their pre-set, but this is not always true. The Petri net in Fig. 5a describes a process where action A precedes the execution of either B or C. In this case the waiting time of $p1$ is determined by the time interval between the timestamp of action A and the timestamp of the subsequent action (either B or C). However, from this measure we cannot determine if B and C have different waiting times, i.e. if the average waiting times of $p1$ is different when either B or C occurs. This issue is even more serious due to presence of recursion. This problem can arise only when a place in the pre-set of a visible transition is also in the pre-set of another transition. We call *isolated* a visible transition such that this does not happen.

Definition 3 (isolated transition). *A visible transition t of a Petri net is called* isolated *if* $\forall p \,.\, p \in {}^\bullet t \Rightarrow p^\bullet = \{t\}$.

By introducing suitable invisible transitions we can transform a net into an equivalent one (i.e., with the same traces of visible transitions) where all visible transitions are isolated. For example, the net of Fig. 5b is obtained through this transformation from that of Fig. 5a: notice that now we can interpret the waiting times of places $p2$ and $p3$ as the waiting times of B and C, respectively.

Another measure that is often useful when analysing a system is the duration of certain activities. Since the events recorded in a log are considered as instantly executed (they have execution timestamps recorded, but not their duration) it is not possible to infer how long the corresponding action ran. Thus in order to evaluate the execution time of business process activities each of them has to be modeled by at least two actions, representing its start and its termination, that have to be recorded by the system in the log with the corresponding timestamps. Clearly, depending on the application, it might be convenient to represent an activity with a more complex Petri net, for example using three transitions to model the activity start, successful completion and failure respectively.

5 Supporting BPMN Modeling

Several standards for business process modeling (e.g. BPMN and JBPM) provide much richer graphical primitives than those available for Petri nets, and

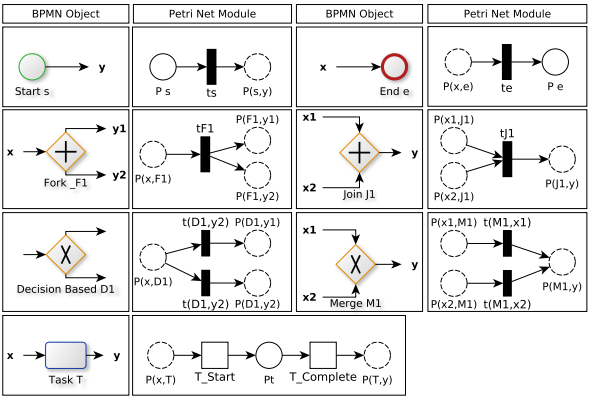


Fig. 6. From BPMN to Petri net

allow to model a system at a higher abstraction level, easier to understand for the stakeholders. In our setting, in order to exploit the algorithms presented in Section 4 to analyze processes expressed in richer process description languages, a straightforward strategy is to define a model transformation that starting from the more abstract specification yields a Petri net. The resulting net can be analyzed with the presented algorithms to verify conformance of executions and to evaluate their performance. The critical issue is then to project back the analysis result to the starting abstract model, as discussed below.

As a business process modeling formalism our platform will support BPMN. We adopt essentially the methodology presented in [8] to transform (a subset of) BPMN models into Petri nets. Even if the subset of BPMN considered here is quite minimal, on one hand we think that it is sufficient to illustrate the advantages of our approach, on the other hand we have concrete evidence that it can be extended seamlessly to other BPMN constructs like events and messages.

The transformation rules for our BPMN subset are presented in Fig. 6 with a slight change in the rule transforming a BPMN task, for which we generate a net with two transitions, as discussed above. It is worth stressing that these transformation rules guarantee that all visible transitions in the resulting net are isolated, which allows us to exploit the performance algorithm to infer the execution time of each BPMN task. Let us discuss now, on the basis of these translation rules, how the analysis results obtained on a net can be projected back to the BPMN model.

Performance Analysis. Recall that in the algorithm proposed in Section 4 invisible transition are treated as eager: if such a transition is fired by the log replay, it is considered to be fired as soon as it is enabled. For this reason, the waiting time for places having no visible transition in their post-sets is always zero. Referring to the rules in Fig. 6, the only places that can have non-zero waiting times are $P(x,T)$ and Pt, i.e. the places that model BPMN activities. Similarly, the synchronization time of a place can be greater than zero only if at least one

transition in its post-set depends on another place. The places in Fig. 6 that can have non-zero synchronization times are just $P(x1, J1)$ and $P(x2, J1)$, i.e. the places involved in modeling join gateways. Based on these considerations we can project the performance measures of a net obtained from the translation to the original BPMN model as follows:

- For each task T the execution time is $\rtimes(Pt)$ and the activation time is $\rtimes(P(x, T))$.
- For each concurrent branch ($i \in [1, 2]$) enclosed by a fork ($F1$) and a join ($J1$) gateway: (i) the synchronization time is $\ltimes(P(xi, J1))$, namely the time spent to wait the termination of the other concurrent activities (ii) the execution time (referred to as $\rtimes(F1i)$) is the sum of $\bowtie$ of all places reachable by traversing the graph starting from $P(F1, yi)$ without visiting $P(xi, J1)$. Notice that $\ltimes(P(xi, J1)) + \rtimes(F1i)$ is constant for each branch of a fork/join pair. We call this time the *fork* execution time ($\bowtie(F1, J1)$).

We stress that the considerations discussed above do not hold for the original ProM 5.2 performance plug-in. In particular, places involved by the join gateways ($P(x1, J1)$ and $P(x2, J1)$) can have synchronization time greater than zero only if their pre-sets contain at least one visible transition. Hence, the algorithm can infer synchronization times of join gateways only if the concurrent branches terminate with a BPMN task.

Conformance Analysis. We exploit a similar reasoning to project back data of conformance results. Log replay artificially produces missing tokens only to fire visible transitions. Hence only places in the pre-set of at least one visible transition can have missing tokens. In Fig. 6 these places are $P(x, T)$ and Pt. Moreover, log replay fires invisible transitions only if their execution is required to activate a visible transition. For example, for any execution of the start transition ts the log replay fires a visible transition that consumes the token in the place $P(s, y)$. The same consideration holds for all invisible transitions that produce only one token. Hence, only places in the post-set of a visible transition or of an invisible transition spawning several tokens can have remaining tokens. In Fig. 6 these places are Pt, $P(T, y)$, $P(F1, y1)$ and $P(F1, y2)$, that are used to model a single BPMN activity and the fork gateway. Thus, we project the conformance metrics of the Petri net to the starting BPMN model as follows:

- for each task T missing tokens in $P(x, T)$ are referred as "unsound executions", missing tokens in Pt are referred as "internal failures", remaining tokens in Pt are referred as "missing completion"
- for each branch ($i \in [1, 2]$) of a fork $F1$, the remaining tokens in $P(F1, yi)$ are referred as "interrupted branches". Notice that for each execution of the transition $tF1$ a token can remain in either $P(F1, y1)$ or $P(F1, y2)$, but not in both places. "Interrupted branches" are also the flows corresponding to places $P(T, y)$ that have remaining tokens.

6 Citizen Migration

In this section we present a real life Italian eGov business process. The BPMN model in Fig. 7 summarizes the main activities required to perform a change of residence of an Italian citizen (the annotations can be ignored for the moment). In order to proceed with the actual task of changing the residence, first the request has to be verified and the necessary sensible data have to be collected by the eGov platform. Note that two different validation tasks can be executed, depending on the citizen being emigrating or immigrating. The Petri net in Fig. 8 has been obtained by applying the model transformation described in Section 5. Note that for each BPMN task (e.g. *ChangeResidence*) the net has two transitions (e.g. CRS and CRC) representing the start and completion of the activity. The trace in Fig. 9a records a possible execution of the Business Process.

An Example of Performance Analysis. In order to evaluate performance measures of the Petri net, our plug-in applies the log replay to obtain the sequence $R1 = [(t1, 1), (t2, 1), (DCS, 1), (t3, 1), (IS, 2), (IC, 3), (DCC, 4), (t5, 4), (t7, 4), (CRS, 5), (CRC, 6)]$

Our implementation considers invisible transitions fired immediately. It transform the transition sequence into the eager sequence:

$R2 = [(t1, 1), (t2, 1), (t3, 1), (DCS, 1), (IS, 2), (IC, 3), (t5, 3), (DCC, 4), (t7, 4), (CRS, 5), (CRC, 6)]$

Figure 8 includes the performance metrics evaluated by our plug-in. All white places have zero synchronization time and zero waiting time. Waiting times of places pi, pdc and pcr represent the *execution times* of the corresponding BPMN activities. Waiting time of place $p3$ is $24s$. In fact, execution of transitions $t1$, $t2$ and $t3$ are associated with the event index 1, while the transition IS is associated with the event index 2. The only place having non-zero synchronization time is $p5$ ($147s$). This time is equal to the interval among execution of the transitions IC and DCC. We stress that the synchronization time of place $p5$ results from the transformation of the sequence $R1$ into the eager sequence $R2$ (migrating transition $t5$ immediately after IC). We already discussed in Section 5 that

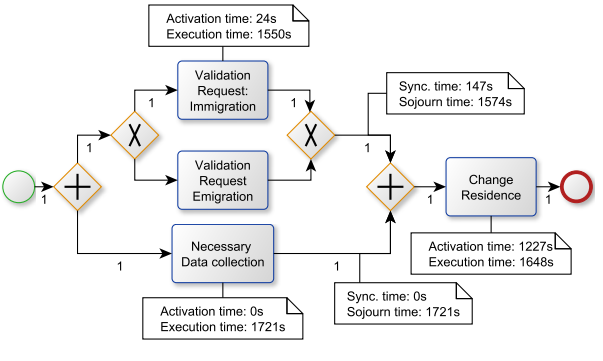


Fig. 7. BPMN model for citizen migration (annotated with performance analysis)

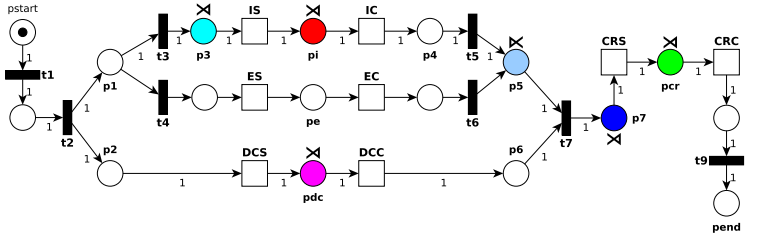
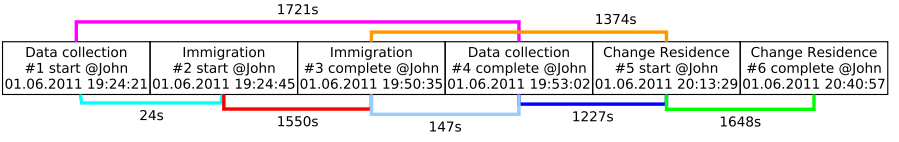


Fig. 8. Petri net for citizen migration (annotated with performance analysis)



(a) Sound execution

Immigration #1 start @John 01.06.2011 17:14:56	Immigration #2 complete @John 01.06.2011 17:31:19	Emigration #3 start @John 01.06.2011 18:24:33	Emigration #4 complete @John 01.06.2011 18:37:10	Data collection #5 start @John 01.06.2011 18:25:27	Change Residence #6 start @John 01.06.2011 18:35:21	Change Residence #7 complete @John 01.06.2011 18:42:41

(b) Unsound execution

Fig. 9. Execution Logs

the ProM 5.2 implementation cannot infer synchronization time for the place $p5$, because its pre-set contains invisible transitions only. ProM 5.2 yields non-zero synchronization time for $p6$ (1227s). Figure 7 shows the projection of the performance analysis back to the BPMN model.

An Example of Conformance Analysis. The trace log depicted in Fig. 9b provides an example of wrong execution. In fact, the trace records the execution of both the exclusive tasks (*Immigration* and *Emigration*) and does not carry events regarding the completion of the activity *DataCollection*.

The conformance measures obtained from log replay for this trace are depicted in Fig. 10. Place $p3$ misses one token: it is required to replay the transition ES that is not enabled because the alternative branch *Immigration* already started. Place *pdc* contains one remaining token: it depends on the missing termination of *Data Collection*. Since termination of this activity is not recorded, place $p7$ does not receive a token during the log replay and the transition $t7$ is never enabled. Hence, the synchronization fails resulting in remaining tokens in both $p4$ and $p5$. The failed synchronization is also the cause of the missing token in place $p8$, required to replay the transition CRS.

Figure 11 projects the conformance data back to the BPMN model. The figure reports: (i) the missing completions of *Data Collection*; (ii) the unsound execution of *Emigration*, because the other branch of the exclusive gateway was already activated; (iii) two interrupted branches, because the join gateway does

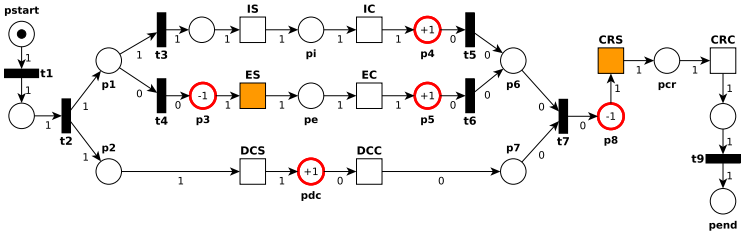


Fig. 10. Petri net (annotated with conformance analysis)

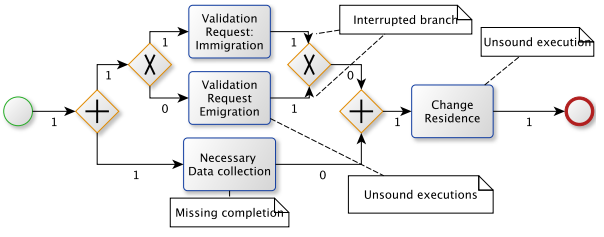


Fig. 11. BPMN (annotated with conformance analysis)

not synchronize; (iv) the unsound execution of *Change Residence*, caused by the failed synchronization.

7 Integrating ProM in the Process Monitoring Platform

ProM 6 exploits the concept of *context* to allow plug-ins to access to the surrounding environment. A context implements the management of resources, e.g. thread management, parameter resolution, progress monitoring and process interruption. In order to start the ProM framework, the *Boot.boot* static method has to be invoked on the chosen context implementation. The *Boot* factory is delegated to create the proper context and to invoke its entry point after the framework initialization. The integration strategy focuses on the development of a specific context (*ServiceContext*) that is Swing-free, but that allows the environment to interact with the ProM plug-ins, for example by triggering the execution of a specific analysis algorithm whenever a process instance terminates. Moreover the context should keep a ProM instance alive to satisfy several requests and to reuse allocated resources (like thread pools). The implementation of the ServiceContext allows us to allocate thread pools directly or to delegate their management to the surrounding environment (e.g. to the application server JBoss). Moreover, the context exposes the ProM algorithms via an API: the analysis results are returned as serializable objects.

In order to fulfill the lack of a performance analysis plug-in for ProM 6, we provided a new implementation exploiting the formal approach described in Section 4. Our implementation exploits some functionalities provided by existing

plug-ins (e.g. ETConformance makes available the log-replay). This way we can benefit from further enhancements of the ProM framework. We developed three other plug-ins: (i) *BpmnToPetri* transforms a BPMN model into a Petri net; the plug-in currently supports the subset of BPMN described in Section 5. (ii) *ConformanceToBPMN* and (iii) *PerformanceToBPMN* annotate the BPMN model with suitable artifacts to represent the conformance and performance measures.

8 Concluding Remarks

We have presented a business process monitoring platform for the Italian eGovernment Enterprise Architecture. The main contributions of this paper are (i) the integration of the ProM framework into a Service Oriented Architecture, allowing the stakeholders to take benefit transparently from several formal methods, (ii) an updated performance evaluation algorithm that manages invisible transaction as eager, and (iii) a methodology to apply the analysis to high level process models. Our monitoring platform can be extended to support several new features. We are developing a web based monitoring interface to represent the analysis results graphically. We are also focusing on the integration of data mining techniques into the analysis engine. The interplay between process analysis and data mining should help discovering, for example, clusters of messages that cause bottlenecks. Finally, we plan to implement a supervisor component to trigger suitable actions whenever a SLA constraint is violated.

References

1. van der Aalst, W.M.P.: The application of Petri nets to workflow management. Journal of Circuits, Systems, and Computers 8(1), 21–66 (1998)
2. van der Aalst, W.M.P.: Formalization and verification of event-driven process chains. Information and Software Technology, 639–650 (1999)
3. van der Aalst, W.M.P., van Dongen, B.F.: Discovering Workflow Performance Models from Timed Logs. In: Han, Y., Tai, S., Wikarski, D. (eds.) EDCIS 2002. LNCS, vol. 2480, pp. 45–63. Springer, Heidelberg (2002)
4. van der Aalst, W.M.P., de Medeiros, A.K.A., Weijters, A.J.M.M.: Genetic Process Mining. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 48–69. Springer, Heidelberg (2005)
5. Baldoni, R., Fuligni, S., Mecella, M., Tortorelli, F.: The Italian *e*-Government Enterprise Architecture: A Comprehensive Introduction with Focus on the SLA Issue. In: Nanya, T., Maruyama, F., Pataricza, A., Malek, M. (eds.) ISAS 2008. LNCS, vol. 5017, pp. 1–12. Springer, Heidelberg (2008)
6. Börger, E., Thalheim, B.: Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 24–38. Springer, Heidelberg (2008)
7. Corradini, A., Flagella, T.: OpenSPCoop: un Progetto Open Source per la Cooperazione Applicativa nella Pubblica Amministrazione. In: AICA 2007 (2007)
8. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of BPMN process models (2007), `http://eprints.qut.edu.au/7115/`

9. Eindhoven Univ. of Technology: ProM, http://www.processmining.org/
10. Link.it: OpenSPCoop, http://www.openspcoop.org
11. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Inf. Comput. 100(1), 1–40 (1992)
12. Muñoz-Gama, J., Carmona, J.: A Fresh Look at Precision in Process Conformance. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 211–226. Springer, Heidelberg (2010)
13. Oasis: Web Services Business Process Execution Language Version 2.0 (2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
14. OMG: Business Process Model and Notation, http://www.omg.org/spec/BPMN/
15. Parlamento Italiano: Istituzione del sistema pubblico di connettività e della rete internazionale della pubblica amministrazione, D.L (42) del (February 28, 2005); G.U. (73) del (March 30, 2005), http://www.parlamento.it/parlam/leggi/deleghe/05042dl.html
16. Petri, C.: Kommunikation mit Automaten. Ph.D. thesis, Institut für Instrumentelle Mathematik, Bonn (1962)
17. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. 33(1), 64–95 (2008)

Domain-Specific Multi-modeling of Security Concerns in Service-Oriented Architectures

Juan Pedro Silva Gallino, Miguel de Miguel, Javier F. Briones, and Alejandro Alonso

Universidad Politécnica de Madrid (UPM)
{psilva,mmiguel,jfbriones,aalonso}@dit.upm.es

Abstract. As a common reference for many in-development standards and execution frameworks, special attention is being paid to Service-Oriented Architectures. SOAs modeling, however, is an area in which a consensus has not being achieved. Currently, standardization organizations are defining proposals to offer a solution to this problem. Nevertheless, until very recently, non-functional aspects of services have not been considered for standardization processes. In particular, there exists a lack of a design solution that permits an independent development of the functional and non-functional concerns of SOAs, allowing that each concern be addressed in a convenient manner in early stages of the development, in a way that could guarantee the quality of this type of systems. This paper, leveraging on previous work, presents an approach to integrate security-related non-functional aspects (such as confidentiality, integrity, and access control) in the development of services.

1 Introduction

Service-Oriented Architectures (SOA), of great popularity nowadays, and Web Services (WS), the technology generally used to implement them, go hand in hand in such a way that they are even referred to as WSOA (Web Service Oriented Architecture). They achieve the integration of heterogeneous technologies, providing interoperability, and yielding the reutilization of pre-existent systems.

At the same time, Model Driven Applications (MDA) [14] arises as a new paradigm that tries to shift models out of a mere documentary role, and turn them into a first class development artifact. This approach provides, among other benefits, a greater understanding of the system as a whole and a platform-independent development, improving the reusability of the design, and simplifying the evolution of the system, thus increasing productivity.

WS-* standards provide a strong foundation for the development of services. WS-Policy and related specifications offer the possibility of considering extra-functional concerns of those services. However, these specifications are XML-based, and its syntax is not designed to be read/written by humans. An abstraction of such languages is desirable.

From the experience obtained during the participation in a research project focused on the service-oriented reformulation of a banking core [3], the need

M. Carbone and J.-M. Petit (Eds.): WS-FM 2011, LNCS 7176, pp. 128–142, 2012.

for means to independently address non-functional properties (NFPs) to foster design (functional and non-functional) reuse along different companies became evident. Different modeling techniques and approaches in use among the different financial institutions made input-models-language independence fundamental. Moreover, the secrecy of security designs is also paramount; security design should only be accessed by trusted number of individuals. Different banks use different infrastructures, and different security solutions and approaches, for instance. It is the authors' belief that this situation repeats itself in other industries.

If the research's results were to be useful throughout the banking industry, the NFPs of the systems had to be supported in different shapes and forms. A clear candidate approach to address this issue, therefore, is that proposed by Multi-Dimensional Separation of Concerns (MDSoC) [21], and seconded in Viewpoints [7], aspect-oriented modeling (AOM) or early aspects (EA) [5,17]. These approaches share common ideas, and allow for the separation of functional and extra-functional characteristics of the systems *at design time*, providing a base for the solution presented in this paper.

Within these different views or concerns of the system, the elements have to be described somehow. In general, aspect-oriented solutions prefer general purpose modeling languages (GPMLs) for this task. GPMLs' approach towards this problem is to provide extension mechanisms to adapt the generic modeling elements to the particular concern or domain that is being addressed by the model. Domain-specific modeling languages (DSMLs), however, takes a different approach, and reduce the available modeling elements to key, familiar problem-domain concepts (and rules), narrowing down the design space [8], to achieve higher levels of productivity. In the approach presented in this paper, DSMLs have been selected to appropriately address the different NFPs.

A final element of concern is the technical expertize needed to express correct (both syntactically and conceptually) security and other NFPs' configurations. One of the ideas in SOA is that business be aligned with the IT infrastructure. Therefore, stakeholders at the business level are candidate to providing input models of the system. Nevertheless, these stakeholders, although capable of expressing security intentions or requirements to these business services, are probably not security savvy enough to express a complete security configuration, nor are aware of the availability of the different security infrastructure elements.

Although multiple implementation technologies exist to facilitate the development of web services and SOA systems, the lack of a sound methodological base for the development of such applications stresses the need for new modeling methods or techniques that could guarantee the quality of the development of this type of systems. In summary, it was identified that there exists a lack of a design solution that:

- Allows an independent development of the functional and non-functional (NF) concerns of service-oriented architectures, fostering reuse.
- Permits that each concern be addressed in a convenient manner, at the appropriate abstraction level.

- Avoids the need for non-functional (in this case, security) expertize at the business/system architect role.
- Is modular enough so that the functional model is not polluted with non-functional information, and secrecy of security design could be maintained.
- Provides a practical way of measuring and analyzing the system for non-functional characteristics in early stages of the development.

Some approaches applying MDA to the development of Web Services already exist (e.g. [16], [10]). However, these approaches do not offer thorough support for access control (AC) descriptors, code generation, Web Services Description Language (WSDL), and WS-(Security)Policy. This paper, leveraging on previous work, introduces the CIM-to-PIM layer of an approach to integrate security-related non-functional aspects (such as confidentiality, integrity, and AC) in the development of services, parting from high-level descriptions of business services compositions, and incorporating technical elements through a chain of transformation and compositions, towards the final implementation of the individual, security-aware services.

The paper is structured as follows. Section 2 describes the general architecture of the approach, while section 3 introduces the Supply Chain Management [25] use case, and describes its implementation along the steps of the proposed solution. Section 4 presents some relevant related work. Finally, section 5 provides some conclusions and possible future lines of work.

2 Integrating the Non-functional Concerns

As first presented in [19], a complete solution addressing non-functional aspects of service architectures was defined. The addressed characteristics are NF properties subject of being described as policies, of which security and access control are prominent examples. This section presents the methodology proposed to integrate these non-functional aspects in the development of web services. A well-known use case, including security concerns, has been selected to illustrate the usefulness of the approach.

2.1 A Framework for the Inclusion of Security Concerns

Figure 1 shows the overall structure of the designed solution, consisting of a chain of model transformations and compositions (indicated by arrows). Every box in the figure indicates a different type of model, including:

- Business model indicating non-functional intentions (CIM Level).
- Functional system model (PIM 1 Level).
- Non-functional properties models (PIM 1 Level).
- Intermediate Meta Model (iMM) model (PIM 2 Level).
- WSDL, WS-Policy, and XML models (PSM Level).

The iMM metamodel was defined with the objective of achieving an abstraction both of input modeling techniques/metamodels, and output target platforms. It is also designed for holding, in one unique model, all information necessary for analysis and to generate the different output artifacts. The different steps' resulting artifacts, including the iMM models, are automatically generated, and the developers need never interact with them if unwanted. However, the modeler has the opportunity to tune/modify the intermediate results to shape the subsequent outcome.

The presented approach proposes the development of each individual concern as an independent model, in a similar fashion to that of Multi-Dimensional Separation of Concerns [21]. Different domain-specific metamodels, each one appropriate to the particular concern in hand, are used to define the non-functional models. The addressed security concerns are individually mapped from the iMM into implementation artifacts in the target platform, independently of the particular services under design.

In that way, the system under implementation provides the particular values for the final implementation elements, but concern models (and their mappings onto the platform) may evolve independently. Such models are composed into a complete model of the system by weaving them together. A possible set of metamodels for the consideration of security concerns in service-oriented architectures is presented in [19] and included in section 2.3.

For the generation, the process model is split into atomic services and their associated NF intents. The different resulting outputs to be obtained, indicated in figure 1, include access-control descriptors (platform dependant), code templates (platform dependant), and WSDL and WS-Policy documents (common to all target platforms). The composition execution should then make use of these artifacts to execute the process.

2.2 Development Process

The sequence of steps to perform in the proposed approach, leveraging on that first introduced in [19], is presented here:

1. A business process model, annotated with abstract security intents (also known as primitives), is presented as input to the process.
2. The intents at CIM level drive the selection of the appropriate security and access-control models. The developer selects, tunes, or even defines new AC and security models, as better fits the system.
3. A functional PIM model is derived from the CIM model or, alternatively, provided by the modeler as an input functional model at PIM level (PIM 1).
4. By means of composition models, the modeler indicates which non-functional properties should be associated to which resources in the functional model. Ideally, pre-filled composition models should be proposed by a tool automating the development process.
5. At this stage, the platform-independent iMM model (PIM 2) contains all (functional and extra-functional) information.

6. Next, the iMM model is transformed into platform dependant (PSM) models. Default values may be used for the generation, unless optional platform information (indicating preferences and/or availability of platform characteristics) is fed to the transformation. In the example presented in this paper, assertions in a WS-SecurityPolicy metamodel, extended with a "preference" attribute, may be used for representing the available/preferred mechanisms to use from the target platform.
7. Finally, service and policy descriptors are derived from the PIM 2 model. Resulting WSDL documents hold references to the appropriate policies defined in the different generated WS-Policy documents. Simultaneously, code templates and non-functional configurations are automatically created.

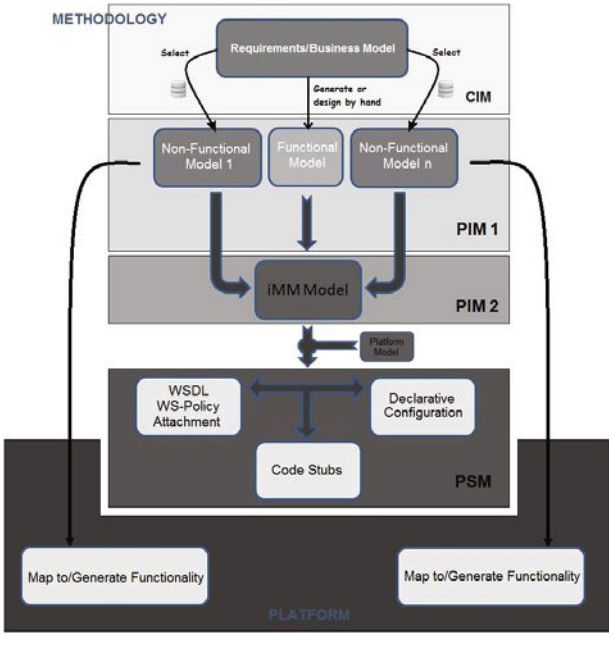


Fig. 1. Structure of the proposed solution

Within the process model, the modeler may express high level security intents (e.g., non-repudiation, confidentiality, integrity), instead of technical mechanisms which are necessary to achieve them (e.g., encryption, signature, time-stamping). The know-how of the mechanisms necessary to achieve the intents is contained in the metadata and the transformations. In this way, the business/system architect is freed from the need of technical security knowledge.

Repositories containing different expert-made AC and security models could provide the non-functional inputs. The idea behind the approach is that different NF models, suitable for different concerns in different domains, would represent

a pool of options from which to select the most appropriate one. Experts in the particular NF domain create such models, and also indicate how the different elements map onto the target platform. Adequate metadata is associated to these models for automatic discovery support. In that way, appropriate models are automatically selected from the repositories and offered to the modeler.

Analyses and metrics, performed and calculated on the iMM model, reassure the modeler confidence in the correctness of the design. Errors detected in this stage are less costly to correct, and the system architect may have the opportunity to consult a domain expert to solve any complicated issue.

The benefits of this approach is twofold: the system's developer need not be security savvy, and allows greater reuse of (functional and non-functional) designs. NF models may be reused among different projects. Also, different NF characteristics may be applied to the same functional design to generate different resulting systems.

2.3 Metamodels

Referring to the metamodels employed in this solution, some pre-existent metamodels have been adopted, some other were merged together, whilst the rest had to be defined for the occasion. The different metamodels presented here, are described in more detail in [19].

Modeling Security Intents at the CIM Level. Business processes diagrams have become the mainstream alternative to model the compositions and interactions of services. Such models provide a great means for describing the interaction of the services at a high level of abstraction. They also represent an outstanding instrument on which to express abstract non-functional intentions that allow stakeholders to express their security concerns without the need for a detailed technical knowledge. Ontologies have risen as useful tools to agree on a common language for the description and discovery of elements (web services, for instance). The use of an ontology at the CIM level may provide not only a means to define a common vocabulary for security intents, but may also represent a tool for the matchmaking [1] of the non-functional concerns models contained in the repositories that may fulfill the expressed requirements.

In the security domain, the NRL ontology [9] represents a very complete option, scoping from high-level security objectives, through security credentials, to technical encryption algorithms, and has been selected to formalize a vocabulary for the security intents.

Functional Input Metamodel: UML. If a PIM level input is to be provided, UML, the modeling standard of choice in most cases nowadays, has been selected as the input's metamodel. UML's SoaML profile [2] was implemented and applied to these input models to guide the UML-to-iMM transformation.

[1] The process of identifying the elements that fulfill the imposed requirements, and its degree of fulfillment.

Security Metamodel: QoS Metamodel. UML's QoS profile [22] defines the "QoS Framework Metamodel" as an extension to UML's metamodel. The defined concepts can be used to model non-functional characteristics as a domain-specific metamodel (as in, for instance, [10]). This is the metamodel selected to illustrate the use of the approach being presented.

Access Control Metamodel: SecureUML. SecureUML [11], a Role Based Access Control (RBAC, one of the currently most used AC techniques) metamodel is used to exemplify possible access-control input models. The modeling and implementation of AC, which follows a similar process to that of the security properties, is not included in this article.

Weaving Metamodels. Weaving metamodels guide the composition mechanisms of the different models. In this case, the composition of functional models with AC models or security models require different weaving associations, described in [19]. The semantics of the associations are implemented as part of the composition rules.

iMM Metamodel. The intermediate metamodel is one of the fundamental parts of this approach. Composed of three differentiated parts (System Design, Access Control, and Policy), this particular implementation, described in [19], was designed to be as general as possible, in order to maximize support for AC techniques and policy standards.

The functional part of the metamodel follows a component-based approach, and is used to describe service components, entities and interfaces, among other metaclasses. The generic metamodel from [15], incorporated in the "Access Control" part, allows for the use of different access-control techniques (and mutation among them). Finally, the "Policy" package of CBDI-SAE Meta Model for SOA [4] was selected for the service policies part. The three selected metamodels were then studied to identify equivalent concepts that provided merging points.

An integrated metamodel, such as iMM, provides a great subject of analysis and metrics' calculations. In this particular case, access control conflicts analysis, or security coverage metrics, for instance, could be defined. Analyses and metrics for iMM models are out of the scope of this paper.

WSDL Metamodel. A WSDL metamodel has been used for the transformations, and in the compositions (in order to visualize the future WSDL document as a model and facilitate the specification of policy application points).

WS-Policy Metamodel. Equivalently, a WS-Policy metamodel has been developed. The transformations and later generation of WS-Policy documents operate on this metamodel, including the case of WS-SecurityPolicy models.

WS-SecurityPolicy Metamodel. A WS-SecurityPolicy metamodel has been defined as an extension to the WS-Policy metamodel, providing a means to edit and validate security policy assertions, and to indicate the platform's security

capabilities and preferences. It is, to the best knowledge of the authors, the first available implementation of a WS-SecurityPolicy metamodel.

3 WS-I's Supply Chain Management Use Case

To illustrate the use of the proposed approach, the Web Services Interoperability Organization's (WS-I) [23] Supply Chain Management (SCM) use case [25] has been selected.

The SCM application [25,24] presents a typical business-to-consumer (B2C) model: a *Retailer* offering electronic goods to consumers. To fulfill orders the *Retailer* has to request products ("shipGoods" operation) and manage stock levels in the three *Warehouses*. When an item in stock falls below a certain threshold, the *Warehouse* must restock the item from the relevant *Manufacturer*'s inventory ("POSubmit" operation, a typical business-to-business (B2B) model). A complete description of the use case architecture's can be found in [25], and in [26] for security requirements. Access-control was originally not addressed in this use case, and will not be included in the example in this paper.

Two frameworks for the Java programming language have been selected to support the execution of the system:

- Spring framework [20], a framework leveraging enterprise-level functionalities for Java POJOs (Plain Old Java Objects).
- Apache CXF [1] framework, supporting policies and Literal and RPC style Web Services (as demanded by the SCM use case specification).

Frameworks provide much functionality which would be otherwise necessary to generate for the execution of the different systems, greatly simplifying the development of generators. Newer frameworks' declarative-based configuration, in particular, support the configuration of non-functional characteristics of the services (such as security or AC, in this case) independently of the functional code. This is of great usefulness in this case, allowing for an independent generation of the different artifacts, and the reuse of generators for different target platforms.

3.1 Creating and Selecting the Input Models

A process model of the system is considered as the primary input. This model, annotated with security and access control intents, indicates the non-functional requirements on the different services. For instance, an interaction marked "non-repudiation" would imply associating to the message the actions of authenticating, signing and time-stamping. The NRL ontology [9], designed to facilitate automatic discovery and invocation, has been selected in this example to formalize the vocabulary for the annotation. Using NRL's matchmaking algorithm, different matching security and AC models are presented to the modeler for selection. In this case, the QoS model would match the different security required characteristics, as it models the security mechanisms to satisfy them.

Table 1. QoS Catalog for the QoS Category "Security"

	Dimension	Unit Type	shipGoodsRequest
Int	IntegrityAlg.	Enum. Literal	SHA1
	TimestampRequired	Boolean	true
	EncryptSignature	Boolean	true
	Signature Encryption Algorithm	Crypto Alg.	Crypto Alg.'s value
Conf	Crypto Alg.	String	AES256
Auth	TokenType	Enum. Literal	X509
	TokenKey Alg.	Enum. Literal	RSA15
	SignToken	Boolean	true
	EncryptToken	Boolean	true

Retailer System's UML Model. Functional models of the system will most likely be derived from the process model, developed from scratch to satisfy it, or reused from a previous system model (perhaps even reverse-engineered from legacy code). The 'de facto' standard for functional systems' modeling is UML, so it was decided to use it to model the Retailer system. As described in section 2.3, the SoaML profile is used to guide the posterior transformation. UML class diagrams for all the services in the SCM example are available in [25].

Modeling of the Security Intents with the QoS Metamodel. The first step towards modeling security concepts under the QoS metamodel is defining a QoS Catalog for the target (one or more) QoS Category. A catalog defines the different characteristics (may be regarded as meta-classes in a metamodel) and dimensions (the variables defining a particular characteristic) that may be present in a model of such category. In this particular case, only one category will be modeled: Security. Within this category, three characteristics are defined: Integrity, Confidentiality, and Authentication. Table 1 presents the different QoS Dimensions defined for these three characteristics, and the values assigned to them in the case of the "shipGoodsRequest" message of the "shipGoods" operation, offered by the *Warehouse* and used by the *Retailer* services.

3.2 Transformation into the iMM Model

Composition of Retailer System's UML Model with Non-Functional Models. Having two different non-functional aspects of the system under consideration, two composition models are necessary. The first one relates the *Retailer* system elements with the security characteristics in the QoS model.

Figure 2 presents the three panel view provided by the AMW tool [6] to create such composition models. In this figure, the different model elements are associated to its security requirements. The figure brings out, for instance, the association of "ShipGoodsResponse" with the security characteristics of signing, with the corresponding *Warehouse* X.509 certificate, the message and timestamp. The composition mechanism for SecureUML input models is equivalent.

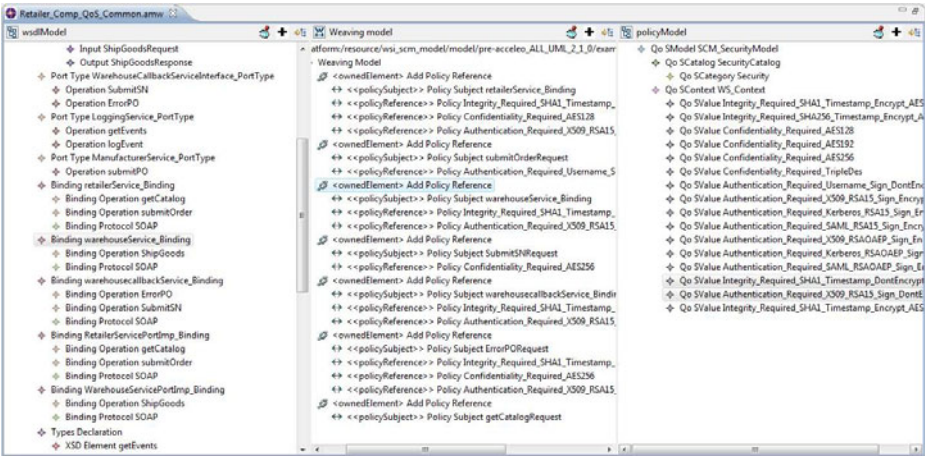


Fig. 2. *Composition of Retailer's QoS and UML Models*

3.3 Generation of Resulting Artifacts

For the generation of both WSDL and WS-Policy documents, the models are first transformed into a XML model and later extracted into the final XML syntax. The resulting WSDL documents (not included, equivalent to those in [25]) reference, within the description of the service elements, the corresponding service policies. The resulting WS-Policy document (figure 3) contains the policies applied to the *Warehouse* service (correspondent to the requirements and the QoS input model) and provide an equivalent expression, in WS-SecurityPolicy assertions, to the security configurations described below. The description of the generation of the functional code is out of the scope of the paper.

Generation of the WS-Security Configuration. The following configuration elements represent the target platform's results for the non-functional concerns requirements. All information necessary to generate the non-functional configuration files is available in the iMM model.

Figure 4 shows an excerpt of the configuration file generated for the Warehouse service. In it, the "WarehouseA" service is defined as a *jaxws:endpoint* and configured correspondingly. The actual bean implementing the service ("warehouseAService") is passed as a reference to the endpoint. The implementor bean is itself configured with the different properties it requires.

Security interceptors are also defined for the service endpoint. Only the output interceptor ("WarehouseEndTimestampSign_Response") is presented in figure 4 for visualization. The input interceptor's configuration is very similar, and therefore excluded. Security interceptors' configuration represent, therefore, the actual piece of the configuration file that depends of the security policy part of the iMM model.

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<wsp:Policy xmlns:sp = 'http://schemas.xmlsoap.org/ws/2002/12/secext'
            ...
            xmlns:wsp = 'http://www.w3.org/ns/ws-policy'>
  <wsp:Policy name = 'warehouseService_Binding_Binding_Policy' wsu:Id = 'warehouseService_Binding_Binding_Policy'>
    <wsp:All>
      <sp:AsymmetricBinding wsp:Ignorable = 'false' wsp:Optional = 'false'>
        <sp:IncludeTimestamp wsp:Ignorable = 'false' wsp:Optional = 'false'/>
        <sp:InitiatorToken wsp:Ignorable = 'false' wsp:Optional = 'false'>
          <wsp:Policy name = 'InitiatorTokenNestedPolicy' wsu:Id = 'InitiatorTokenNestedPolicy'>
            <wsp:All>
              <sp:X509Token wsp:Ignorable = 'false' wsp:Optional = 'false'
                  sp:IncludeToken = 'http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRecipient'>
                <wsp:Policy name = 'InitiatorTokenX509TokenNestedPolicy' wsu:Id = 'InitiatorTokenX509TokenNestedPolicy'>
                  <wsp:All>
                    <sp:WssX509V3Token10 wsp:Ignorable = 'false' wsp:Optional = 'false'/>
                  </wsp:All>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:All>
          </wsp:Policy>
        </sp:InitiatorToken>
        <sp:RecipientToken wsp:Ignorable = 'false' wsp:Optional = 'false'>
          <wsp:Policy name = 'RecipientTokenNestedPolicy' wsu:Id = 'RecipientTokenNestedPolicy'>
            <wsp:All>
              <sp:X509Token wsp:Ignorable = 'false' wsp:Optional = 'false'
                  sp:IncludeToken = 'http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToInitiator'>
                <wsp:Policy name = 'RecipientTokenX509TokenNestedPolicy' wsu:Id = 'RecipientTokenX509TokenNestedPolicy'>
                  <wsp:All>
                    <sp:WssX509V3Token10 wsp:Ignorable = 'false' wsp:Optional = 'false'/>
                  </wsp:All>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:All>
          </wsp:Policy>
        </sp:RecipientToken>
        <sp:AlgorithmSuite wsp:Ignorable = 'false' wsp:Optional = 'false'>
          <wsp:Policy name = 'AlgorithmSuiteNestedPolicy' wsu:Id = 'AlgorithmSuiteNestedPolicy'>
            <wsp:All>
              <sp:Basic128Rsa15 wsp:Ignorable = 'false' wsp:Optional = 'false'/>
            </wsp:All>
          </wsp:Policy>
        </sp:AlgorithmSuite>
      </sp:AsymmetricBinding>
      <sp:SignedParts wsp:Ignorable = 'false' wsp:Optional = 'false'>
        <sp:Header wsp:Ignorable = 'false' wsp:Optional = 'false' Name = 'ConfigurationHeader'
                   Namespace = 'http://www.ws-i.org/SampleApplications/SupplyChainManagement/2002-08/Warehouse'/>
        <sp:Body wsp:Ignorable = 'false' wsp:Optional = 'false'/>
      </sp:SignedParts>
    </wsp:All>
</wsp:Policy>
```

Fig. 3. *Warehouse Service's WS-Policy document*

Within this configuration, and encircled, two entries stand out:

- the different securing actions to be performed (key="action" and, according to the requirements, value="Timestamp Signature"),
- the actual elements to be signed (key="signatureParts" and, again following the requirements, value="..Timestamp; ..Body; ..Header").

The presented use case, although briefly described, illustrates how the proposed approach provides mechanisms to address the objectives placed in section 1. Abstract NF intents are first mapped to NF design models, composed into a complete system model, refined, and finally mapped to implementation artifacts. The security interceptors and the access control context's elements (not presented) represent the mappings from the respective security and access control concerns. The iMM model of the system provides the particular values for elements (e.g., the security actions to perform). In this way, different concerns models, independently managed (maintained, evolved, mapped to target platforms, etc.) may be re-used for the implementation of different systems.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans  xmlns="http://www.springframework.org/schema/beans"
        xmlns:jaxws="http://cxf.apache.org/jaxws" ...>
        <!-- SERVICES - WarehouseA-->
        <jaxws:endpoint id="warehouseA" address="/warehouseA"
                        implementorClass="org.ws_i.sampleapplications....WarehouseAService">
                <jaxws:implementor>
                        <ref bean="warehouseAService"/>
                </jaxws:implementor>
                <jaxws:outInterceptors>
                   <ref bean="WarehouseEndTimestampSign_Response"/>
                </jaxws:outInterceptors>
                <jaxws:inInterceptors>
                   ...
                </jaxws:inInterceptors>
        </jaxws:endpoint>
        ...
        <bean id="warehouseAService" class="org.ws_i.sampleapplications....WarehouseAService">
                <property name="logStub" ref="loggingClient"/>
                <property name="manufacturerAService" ref="manufacturerAClient"/>
                ...
        </bean>
        ...
        <!-- WSS4JOutInterceptor for timestamping and signing the SOAP response. -->
        <bean
             id="WarehouseEndTimestampSign_Response"
             class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor">
             <constructor-arg>
                <map>
                   <entry key="action" value="Timestamp Signature"/>
                   <entry key="user" value="warehouse"/>
                   <entry key="signaturePropFile" value="warehouseKeystore.properties"/>
                   <entry key="passwordCallbackClass" value="org.ws_i.sampleapplications.
                                                          ...WarehousePasswordCallback"/>
                   <entry key="signatureParts" value="
                           {Element}{http://docs.oasis-open.org/wss/2004/01/
                                     oasis-200401-wss-wssecurity-utility-1.0.xsd}Timestamp;

                           {Element}{http://schemas.xmlsoap.org/soap/envelope/}Body"/>
                </map>
             </constructor-arg>
        </bean>
        ...
</beans>
```

Fig. 4. *Warehouse Service's Security Configuration*

4 Related Work

Among the proposals that deal with the different aspects supported by WS-Policy, the approach by Ortiz and Hernández [16] is one of the most representatives. Their solution makes use of aspect-orientation, Service Component Architecture (SCA) modeling and a UML (Unified Modeling Language) profile. Ours, in change, proposes the use of domain specific languages (DSLs), more adapted to each of the extra-functional aspects in hand, to later combine these DSLs into one integrated model. No means to generate the implementation of the security aspects previously mentioned are described in [16].

In [12], the use of a security view (a view where security properties are associated to functional elements in a process) is suggested. A first approximation towards the modeling of security characteristics in event-driven process chains (EPCs) is presented, modeling security at a technical level. Sharing the idea

of the use of views or different models for addressing NF concerns, both approaches differ (among other things) in that [12]'s doesn't make use of MDA's abstract-to-specific chain of transformations, not benefiting from MDA's reutilization capabilities, and that our approach isolates the process modeler from technical security knowledge.

A methodology for an end to end SOA security configuration in proposed by [18]. The approach is model-driven, and makes use of templates to express identified security patterns, initially added as abstract keywords that represent security requirements at business level, and then at a service model level. This mapping from business-level requirements to technical intents is a manual task, performed by a software architect. Security intents are complemented with a Security Infrastructure Model (SIM) to later generate a concrete configuration. The SIM, although closer to topology modeling, fulfills an equivalent functionality to that of the WS-SecurityPolicy platform model in this work. Our approach and the one in [18] differ in that the latter is limited to security, and in, perhaps, the greatest strength in this proposal: the use of separate concern models to address each non-functional characteristic, a feature that aids reuse and modularity of design and implementation.

A security metamodel is presented in [13] to model security considerations in business processes, map them to service-oriented architectures, and generate WS-SecurityPolicy configurations. The focus is on confidentiality, integrity, identification, and authentication using patterns. Security goals (primitives) are expressed in these models, and policy assertions satisfying these goals are specified. In our work, the use of domain specific independent metamodels to describe the individual concerns facilitates the understanding, analysis, and generation of the different aspects of the service individually. The work in [13] uses one unique metamodel to express all characteristics of the services, hindering, therefore, the autonomous evolution of the different concerns. Additionally, our work is not limited to security concerns, being flexible enough to integrate other type of policies. Nonetheless, the work in [13] is sound, and worth of consideration.

5 Conclusions and Future Work

This paper has presented a piece of work focused on the achievement of an integrated, model-driven solution for the development of service-oriented architectures with security and access control support. The different participant metamodels have been presented, and a well-known use case, mapped to an unmodified, state-of-the-practice commercial framework has been included to illustrate the usefulness of the approach.

Multiple domain-specific models independently addressing NF concerns, and being independently mapped into target platforms, foster concern models reuse, and favors that each concern be addressed in a convenient manner. Additionally, the modularity of the approach permits that each concern design, functional or not, is not polluted with extraneous information. Finally, the integrated model offers a practical way of measuring and analyzing the system in early stages of the development.

The proposed solution presents great flexibility in its evolution regarding the appearance of new standards or technologies. Moreover, the WS-Policy framework itself has been defined in a generic fashion, allowing for many standards to be formulated based on it. Consequently, all standards already available, or in process of being defined under its umbrella may easily be supported by this solution, and its assertions can readily be applied. This leverages its use for developing not only security-aware services, but also include reliability, timing constraints, secure exchange, transactions, etc.

With respect to future work, in the short term it will be focused on the completion of a prototype tool for the CIM-PIM-PSM chain. In a longer term, it is planned to consider other policy aspects under standardization process, to be able to use the information intrinsic to the assertions (for shaping of code generation, configuration of target platform, generation/use of an extra-functional aspect, etc.).

On a different token, the shifting from a generation approach based on a monolithic metamodel, towards a composition-based generation approach (in which any metamodel could be used to generate artifacts based on a set of weaving associations) could provide an alternative line of research.

Acknowledgment. This work was supported in part by the Centro Español de Desarrollo Tecnológico (CDTI, Ministry of Industry, Commerce and Turisms), by means of the ITECBAN (Infraestructura Tecnológica y Metodológica de Soporte para un Core Bancario) project, from the INGENIO 2010 program.

References

1. Apache. Apache CXF (2010)
2. Berre, A.: Service oriented architecture Modeling Language (SoaML)-Specification for the UML Profile and Metamodel for Services, UPMS (2008)
3. CDTI. ITECBAN
4. Dodd, J., Allen, P., Butler, J., Olding, S., Veryard, R., Wilkes, L.: CBDI-SAE Meta Model for SOA Version 2. Technical report, Everware-CBDI (2007)
5. Elrad, T., Aldawud, O., Bader, A.: Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 189–201. Springer, Heidelberg (2002)
6. Del Fabro, M.D., Bézivin, J., Jouault, F.: AMW: a generic model weaver. In: Proceedings of the Using Metamodels to Support MDD Workshop, 10th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2005 (2005)
7. Finkelsetin, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering 2 (1992)
8. Kelly, S., Tolvanen, J.-P.: Domain-specific modeling: enabling full code generation. Wiley-IEEE, Hoboken, New Jersey (2008)
9. Kim, A., Luo, J., Kang, M.: Security Ontology to Facilitate Web Service Description and Discovery. In: Spaccapietra, S., Atzeni, P., Fages, F., Hacid, M.-S., Kifer, M., Mylopoulos, J., Pernici, B., Shvaiko, P., Trujillo, J., Zaihrayeu, I. (eds.) Journal on Data Semantics IX. LNCS, vol. 4601, pp. 167–195. Springer, Heidelberg (2007)

10. Larrucea, X., Alonso, R.: Modelling and Deploying Security Policies. In: WEBIST 2009 - Proceedings of the Fifth International Conference on Web Information Systems and Technologies, Lisboa, Portugal, pp. 411–414. INSTICC Press (2009)
11. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-Based Modeling Language for Model-Driven Security. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 426–441. Springer, Heidelberg (2002)
12. Jensen, M., Feja, S.: A Security Modeling Approach for Web-Service-Based Business Processes. In: 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, ECBS 2009, San Francisco, California, USA, pp. 340–347. IEEE Computer Society (2009)
13. Menzel, M., Meinel, C.: A Security Meta-model for Service-Oriented Architectures. In: 2009 IEEE International Conference on Services Computing, Bangalore, India, pp. 251–259. IEEE (September 2009)
14. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1 (2003)
15. Mouelhi, T., Fleurey, F., Baudry, B., Le Traon, Y.: A Model-Based Framework for Security Policy Specification, Deployment and Testing. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 537–552. Springer, Heidelberg (2008)
16. Ortiz, G., Hernández, J.: Service-Oriented Model-Driven Development: Filling the Extra-Functional Property Gap. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 471–476. Springer, Heidelberg (2006)
17. Rashid, A., Sawyer, P., Moreira, A., Araújo, J.: Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In: IEEE International Conference on Requirements Engineering, p. 199 (2002)
18. Satoh, F., Nakamura, Y., Mukhi, N., Tatsubori, M., Ono, K.: Methodology and Tools for End-to-End SOA Security Configurations. In: 2008 IEEE Congress on Services, SERVICES I, Honolulu, Hawaii, USA, pp. 307–314. IEEE Computer Society (2008)
19. Gallino, J.P.S., de Miguel, M.A., Briones, J.F., Alonso, A.: Model-Driven Development of a Web Service-Oriented Architecture and Security Policies. In: 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, Carmona, Spain, pp. 92–96. IEEE Computer Society, Los Alamitos (2010)
20. SpringSource. Spring Framework (2010)
21. Sutton Jr., S.M.: N degrees of separation: multi-dimensional separation of concerns. In: International Conference on Software Engineering, pp. 107–119 (1999)
22. The Object Management Group (OMG). UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms Version 1.1 (2008)
23. Web Services Interoperability Organization, `http://www.ws-i.org`
24. WS-I. Sample Architecture Usage Scenarios (2003)
25. WS-I. Supply Chain Management Sample Architecture (2003)
26. WS-I. Sample Applications Security Architecture Document (2006)

Author Index