Cosimo Laneve
Jianwen Su (Eds.)

LNCS 6194

Web Services and Formal Methods

6th International Workshop, WS-FM 2009
Bologna, Italy, September 2009
Revised Selected Papers

Springer

Lecture Notes in Computer Science 6194

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

Cosimo Laneve   Jianwen Su (Eds.)

Web Services and Formal Methods

6th International Workshop, WS-FM 2009
Bologna, Italy, September 4-5, 2009
Revised Selected Papers

Springer

Volume Editors

Cosimo Laneve
Università di Bologna, Dipartimento de Scienze dell'Informazione
Mura Anteo Zamboni 7, 40127 Bologna, Italia
E-mail: laneve@cs.unibo.it

Jianwen Su
University of California, Department of Computer Science
Santa Barbara, CA 93106-5110, USA
E-mail: su@cs.ucsb.edu

Library of Congress Control Number: 2010930913

CR Subject Classification (1998): H.4, H.3.5, C.2, H.3, D.2, H.5

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-14457-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-14457-8 Springer Berlin Heidelberg New York

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This volume contains the papers presented at WS-FM 2009: The 6th International Workshop on Web Services and Formal Methods held during September 4–5, 2009 in Bologna, Italy.

There were 18 submissions by authors from 12 countries. Each submission was reviewed by at least 3, and on the average 3.9, Program Committee members. The committee decided to accept 10 papers. Most of the selected papers are reports on work in progress on problems related to formal aspects of Web services. This workshop also features three invited talks by Mariangiola Dezani (Sessions and Session Types: An Overview), Robin Milner (Processes, and Categories of Bigraphs) and Maurizio Lenzerini. Dezani's talk is included in this volume.

We thank all authors who submitted papers to this workshop, and the members of the Program Committee for their work in the review process. We are also grateful to the CONCUR 2009 organizers who take care of many organizational details for the workshop and, in particular, to Mario Bravetti and Gianluigi Zavattaro. We also thank EasyChair that helped us in the management of every step of the workshop.

April 2010

Cosimo Laneve
Jianwen Su

Organization

Program Chairs

Cosimo Laneve (Program Co-Chair)	Università di Bologna, Italy
Jianwen Su (Program Co-Chair)	University of Califorina at Santa Barbara, USA

Program Committee

Wil van der Aalst	Eindhoven University of Technology, The Netherlands
Albert Benveniste	IRISA/INRIA, France
Karthikeyan Bhargavan	Microsoft Research Cambridge, UK
Roberto Bruni	Università di Pisa, Italy
Diego Calvanese	Free University of Bolzano, Italy
Alin Deutsch	University of California San Diego, USA
Marlon Dumas	University of Tartu, Estonia
José Luiz Fiadeiro	University of Leicester, UK
Xiang Fu	Georgia Southwestern State University, USA
Philippa Gardner	Imperial College, UK
Kohei Honda	Queen Mary, University of London, UK
Nickolas Kavantzas	Oracle Co., USA
Zongyan Qiu	Peking University, China
Vasco T. Vasconcelos	University of Lisbon, Portugal
Karsten Wolf	University of Rostock, Germany

External Reviewers

Alberto Lluch Lafuente
Fabio Patrizi
Giuseppe De Giacomo
Ivan Lanese
Kathrin Kaschner
Luca Padovani
Lucian Wischik
Luis Cruz Filipe
Manuel Mazzara
Martin Berger
Marzia Buscemi
Niels Lohmann
Olivia Oanea
Simon Gay
Victor Vianu

Table of Contents

Sessions and Session Types: An Overview

Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro

Dipartimento di Informatica, Università di Torino
corso Svizzera 185, 10149 Torino, Italy
{dezani,deliguoro}@di.unito.it

Abstract. We illustrate the concepts of sessions and session types as they have been developed in the setting of the π-calculus. Motivated by the goal of obtaining a formalisation closer to existing standards and aiming at their enhancement and strengthening, several extensions of the original core system have been proposed, which we survey together with the embodying of sessions into functional and object-oriented languages, as well as some implementations.

Keywords: Process calculi, Type Systems, Service Oriented Computing.

1 Introduction

The rapid growth of web technologies and of service oriented programming is promoting a fruitful interaction between research communities and standards organizations, with the aim of designing languages and systems for communication centred computations based on a sound theoretical footing.

Session types are one of the formalisms that have been proposed to structure interaction and reason over communicating processes and their behaviour. They appeared in [THK94] and subsequently in [HVK98], where the issue of formalising in a type system the concept of session was framed in the (polyadic) π-calculus with types. The basic idea is to introduce a new form of polymorphism which allows the typing of channel names by structured sequences of types, abstractly representing the trace of the usage of the channels.

The apparently weak constraint constituted by typing channels with session types, while disregarding the interleaved usage of the channels themselves within the process term, is however sufficient to detect subtle errors in the implementation of communication protocols. In fact it reveals to be the right setting where concepts developed for the π-calculus or in general for process algebras can be combined: we think of error freeness checked via typability, of internal mobility which nicely captures the idea of private conversations, of linearity and type duality which enforce the mirroring of the channel usage into its type, and of channel transmission, at the very basis of the π-calculus, to model service delegation.

C. Laneve and J. Su (Eds.): WS-FM 2009, LNCS 6194, pp. 1–28, 2010.

Since then a substantial body of research has been carried out: to better understand the potentiality of the proposed calculi, as it is the case of the introduction of subtyping polymorphism for session types in [GH05]; to strengthen the expressive power of session type systems with respect to relevant computational properties like progress and deadlock-freedom in [DCdLY08], building over ideas of [Kob06]; to widen the scenarios which can be modelled in the calculus, stepping to multiparty sessions instead of just dyadic ones [HYC08], or to detect realizable choreographies via a system of global session types in [CHY07]; to propagate the session type technology to existing programming languages as in [VGR06] for the functional paradigm or in [CCDC^{+}09] for the object-oriented one, providing implementations and applications.

Session types are by no means the only proposal for a theoretical foundation of communication centred programming which has been based on process algebras. Service calculi as well as protocol descriptions called "contracts" have been devised (for which see the references in Section 6) and in some cases the relations with session types have been investigated, although much remains to be done. The comparisons of the superficially different formalisms enlightening common underlying concepts will hopefully improve the language design and the programming practice for communication based computing.

In the present paper we will survey all these aspects mainly informally, by means of examples or just providing pointers to the literature. We begin in Section 2 with session types in their global versus local formulation, though this is a recent development: this is where the basic concepts and formalisms are presented. Section 3 overviews the numerous extensions for the original system which have been proposed to gain expressivity and to catch stronger computational properties. Section 4 is devoted to the embedding of sessions and their typings into the functional and object-oriented programming paradigms. In Section 5 we report on implementations of sessions and session types which use mainstream programming languages. Finally in Section 6, we quickly review formalisms and calculi which appear to be close to session type systems and to their goals.

2 Basic Concepts and Systems

In networking a *session* is a logic unit of information exchange between two or more communicating agents. The essential concern of a session is to specify the topic of conversation as well as the sequence and direction of the communicated messages. This has been formalized as a type system for a dialect of Milner's π-calculus in a series of papers by Honda and others [THK94, HVK98, YV07], and recently extended to express ideas from W3C-CDL (`http://www.w3.org/TR/ws-cdl-10/`), a language for choreography. To look at sessions and session types in their latest incarnation, we follow [CHY07, HYC08], where sessions are described at different levels. At the global level they are abstract specifications

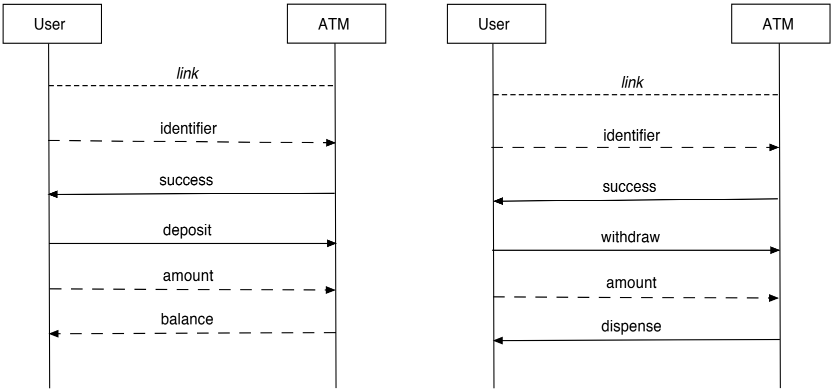


Fig. 1. UML sequence diagrams of some User-ATM interactions

of globally available services (called *interactions* in [CHY07]), whose types are global session types, or simply *global types*. At the local level they are protocols described in the participant perspective: the local session types or just *session types*, which can be assigned to *end-point* processes, the actual participants of the interaction. These two levels are related to each other: the global processes (that can be thought of as choreographies) and the global types should project to the local ones, where end-points play the role of the actual implementations of the specified system.

To illustrate these concepts and their formal representation, let us consider the following protocol which describes a simplified interaction between a customer (User) and an automated teller machine (ATM)[1]:

- First the User communicates her/his identifier to the ATM;
- The ATM answers with either success or failure.
  - In the first case the User can ask either for doing a deposit or a withdraw.
    - For a deposit the User communicates an amount, and waits for a balance.
    - For a withdraw first the User communicates an amount and then the ATM answers with either dispense or overdraft.
  - If the answer is failure, then the interaction terminates.

Two possible interactions are described in the UML sequence diagrams in Figure 1. Note that identifier, amount and balance are row data and have been represented by dashed arrows, while success, failure, deposit, withdraw, dispense and overdraft are labels used to choose between different options, shown in the diagrams by solid arrows.

[1] The example of the interaction among User, ATM and Bank comes from [HVK98], and it has been used by several authors. We adapt this example also to illustrate the subsequent developments and variations of the original system.

Following [CHY07][2] a global description of this interaction is as follows:

$$
\begin{array}{l}
\mathsf{User} \longrightarrow \mathsf{ATM} : \mathsf{identifier}.\\
\mathsf{ATM} \longrightarrow \mathsf{User} :\\
\quad \begin{array}{lll}
\{ & \mathsf{success} : & \mathsf{User} \longrightarrow \mathsf{ATM} : \\
 & & \begin{array}{lll}
 \{ & \mathsf{deposit} : & \begin{array}{l} \mathsf{User} \longrightarrow \mathsf{ATM} : \mathsf{amount}.\\ \mathsf{ATM} \longrightarrow \mathsf{User} : \mathsf{balance}.\\ \mathsf{end} \end{array}\\
 \Box & \mathsf{withdraw} : & \begin{array}{l} \mathsf{User} \longrightarrow \mathsf{ATM} : \mathsf{amont}.\\ \mathsf{ATM} \longrightarrow \mathsf{User} :\\ \quad \begin{array}{ll} \{ & \mathsf{dispense} : \mathsf{end}\\ \Box & \mathsf{overdraft} : \mathsf{end}\\ \} & \end{array} \end{array}\\
 \} & &
 \end{array}\\
\Box & \mathsf{failure} : & \mathsf{end}\\
\} & &
\end{array}
\end{array} \tag{1}
$$

The arrows $\mathsf{User} \longrightarrow \mathsf{ATM}$ and $\mathsf{ATM} \longrightarrow \mathsf{User}$ represent the direction of the message, which in the first line is simply an identifier. The alternatives between the possible answers success or failure by the ATM (and similarly in the subsequent lines, where branching actions are described) are grouped by curly brackets and separated by $\Box$. In the last case the protocol terminates, while in the first one it goes on with nested choices, by choosing among deposit and withdraw. In the first case the User is expected to send the amount and to wait for the balance from the ATM. In the case of withdraw instead, after sending the amount the User will receive either a dispense or an overdraft message from the ATM.

The global type of the current interaction can be simply obtained from the global description replacing data by their types:

$$
\begin{array}{l}
\mathsf{User} \longrightarrow \mathsf{ATM} : \mathsf{String}.\\
\mathsf{ATM} \longrightarrow \mathsf{User} :\\
\quad \begin{array}{lll}
\{ & \mathsf{success} : & \mathsf{User} \longrightarrow \mathsf{ATM} : \\
 & & \begin{array}{lll}
 \{ & \mathsf{deposit} : & \begin{array}{l} \mathsf{User} \longrightarrow \mathsf{ATM} : \mathsf{Real}.\\ \mathsf{ATM} \longrightarrow \mathsf{User} : \mathsf{Real}.\\ \mathsf{end} \end{array}\\
 \Box & \mathsf{withdraw} : & \begin{array}{l} \mathsf{User} \longrightarrow \mathsf{ATM} : \mathsf{Real}.\\ \mathsf{ATM} \longrightarrow \mathsf{User} :\\ \quad \begin{array}{ll} \{ & \mathsf{dispense} : \mathsf{end}\\ \Box & \mathsf{overdraft} : \mathsf{end}\\ \} & \end{array} \end{array}\\
 \} & &
 \end{array}\\
\Box & \mathsf{failure} : & \mathsf{end}\\
\} & &
\end{array}
\end{array} \tag{2}
$$

[2] In [CHY07, HYC08] global descriptions of interaction are more informative than ours, since they also specify the initiation and the channel names on which data and choice labels are communicated.

Now let us look at the User view of this interaction, which can be described by the *session type*:

$$
\begin{array}{llll}
!\,\mathsf{String}. & \&\{\ \mathsf{success}: & \oplus\{\ \mathsf{deposit}: & !\,\mathsf{Real}.\ ?\,\mathsf{Real}.\ \mathsf{end}\\
 & & []\ \mathsf{withdraw}: & !\,\mathsf{Real}.\\
 & & & \&\{\ \mathsf{dispense}:\mathsf{end}\\
 & & & \quad []\ \mathsf{overdraft}:\mathsf{end}\\
 & & & \quad \}\\
 & & \} & \\
 & []\ \ \mathsf{failure}:\mathsf{end} & & \\
 & \} & &
\end{array}
\tag{3}
$$

With respect to the global type (2), the local view of User is obtained by removing the arrows, being the source and the target of the messages determined, and by using ! or ? when the User is the sender or the receiver of a message, respectively; then by writing $\oplus$ when the User *selects* a choice and & when she/he offers a *branching* of choices.

Similarly we get the type for the ATM view; however, being this symmetric w.r.t. the User's view, the same result can be obtained form (3) by interchanging ! with ? and $\oplus$ with &:

$$
\begin{array}{llll}
?\,\mathsf{String}. & \oplus\,\{\ \mathsf{success}: & \&\{\ \mathsf{deposit}: & ?\,\mathsf{Real}.\ !\,\mathsf{Real}.\ \mathsf{end}\\
 & & []\ \mathsf{withdraw}: & ?\,\mathsf{Real}.\\
 & & & \oplus\{\ \mathsf{dispense}:\mathsf{end}\\
 & & & \quad []\ \mathsf{overdraft}:\mathsf{end}\\
 & & & \quad \}\\
 & & \} & \\
 & []\ \ \mathsf{failure}:\mathsf{end} & & \\
 & \} & &
\end{array}
\tag{4}
$$

The types (3) and (4) show a typical *duality* of session types, which is at the basis of *communication safety* and *session fidelity* properties of the processes typable in the session type systems. Communication safety is the extension to a sequence of communications of the standard correctness property of simple types for (polyadic) π-calculus (see [SW01] Chap. 6), namely that only data of the expected type are exchanged. Session fidelity is instead a typical feature of session type systems, where special names, called *session channels*, may carry messages of different types but in a specific sequence. To illustrate the point let us see how a process implementing the User agent might look:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\bar{a}(k).\ k\,!\,\mathsf{identifier}.}\\
k\ \&\{\ \mathsf{success}:\mathsf{if}\ \cdots & \mathsf{then}\ k\oplus & \mathsf{deposit}: & k\,!\,\mathsf{amount}.\ k\,?\,(x).\ \mathbf{0}\\
 & \mathsf{else}\ k\oplus & \mathsf{withdraw}: & k\,!\,\mathsf{amount}.\\
 & & & k\ \&\{\ \mathsf{dispense}:\cdots\\
 & & & \quad []\ \mathsf{overdraft}:\cdots\\
 & & & \quad \}\\
\quad \mathsf{failure}:\ \mathbf{0} & & & \\
\quad \} & & &
\end{array}
\tag{5}
$$

where $\cdots$ represent local computations.

On the other hand a process modelling the ATM could be:

$$
\begin{array}{l}
a(h).\ h\ ?\ (x).\\
\mathsf{if}\ \cdots\ \mathsf{then}\ h\ \oplus\ \mathsf{success}:\ \ h\&\{\ \mathsf{deposit}: h\ ?\ (y).\ \ \cdots\ h\ !\ \mathsf{balance}.\ \ \mathbf{0}\\
\qquad\qquad\qquad\qquad\qquad\quad\ [\!]\ \mathsf{withdraw}: h\ ?\ (z).\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{if}\ \cdots \mathsf{then}\ h\ \oplus\ \mathsf{dispense}:\cdots\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ h\ \oplus\ \mathsf{overdraft}:\cdots\\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \}\\
\qquad\quad \mathsf{else}\ h\ \oplus\ \mathsf{failure}:\mathbf{0}
\end{array}
\tag{6}
$$

The name a is a *session name* (simply a name in [HVK98]) and the prefixes $a(h)$ and $\bar{a}(k)$ are used for *session initiation*, named accept and $\mathsf{receive}$ in the same work; the names k, h are bound names in the respective bodies representing the *session channels*. As soon as the session begins, the session channels are substituted by a new name κ, with a superscript polarity[3] $p \in \{+, -\}$, marking the two end points of the running session, as it is apparent from the rule for *session initiation*:

$$(\bar{a}(k).P) \mid (a(h).Q) \longrightarrow (\nu\kappa)(P\{\kappa^{+}/k\} \mid Q\{\kappa^{-}/h\}).$$

The *polarised channels* $\kappa_1^p, \kappa_2^q, \ldots$ are the medium of all messages between the participants of a session. In fact they are used to *receive and send values*:

$$(\kappa^p\ !\ v.P) \mid (\kappa^{\bar{p}}\ ?\ (x).Q) \longrightarrow P \mid Q\{v/x\}$$

(where $\bar{+} = -$ and $\bar{-} = +$), and to perform *select/branching actions*[4]:

$$(\kappa^p\ \oplus\ \ell_i :\ P) \mid (\kappa^{\bar{p}}\ \&\{\ell_1 : Q_1\ [\!]\ \cdots\ [\!]\ \ell_n : Q_n\}) \longrightarrow P \mid Q_i, \qquad (1 \leq i \leq n).$$

Communication safety and session fidelity (simply called "error freeness" in [HVK98]) are induced by a typing discipline, assigning session types $S, S', \ldots$ to session channels, and the type $[S]$ to a session name carrying a session channel typable by S or by $\overline{S}$. So typing judgements are of the shape[5]:

$$\Gamma \vdash P \triangleright \Delta$$

where Γ associates types to session names and variables, while Δ associates session types to session channels. In the sequel we do not report rules for the polarised channels (that are essentially the same), just for simplicity.

[3] The use of polarities has been proposed in [GH05], and adapted to the original syntax in [YV07].

[4] W.r.t. [HVK98], we write $\kappa^p\ \oplus\ \ell_i :\ P$ for $\kappa^p \triangleleft \ell_i;\ P$ and $\kappa^q\ \&\{\ell_1 : Q_1\,[\!]\cdots[\!]\,\ell_n : Q_n\}$ for $\kappa^q\ \triangleright \{\ell_1 : Q_1\ [\!]\ \cdots\ [\!]\ \ell_n : Q_n\}$, just to keep closer process and the type syntax in the present informal exposition.

[5] The syntax of judgments and rules of the session type system is from [YV07] §3, omitting the bases Θ, of use for process definitions that we disregard in this survey.

The typing rules for session initiation assure that the channels bound by session names have exactly the session types prescribed (writing $\overline{S}$ for the dual of S):

$$\frac{\Gamma, a : [S] \vdash P \triangleright \Delta, k : S}{\Gamma, a : [S] \vdash a(k).P \triangleright \Delta} \qquad \frac{\Gamma, a : [S] \vdash P \triangleright \Delta, k : \overline{S}}{\Gamma, a : [S] \vdash \bar{a}(k).P \triangleright \Delta}$$

The assumption $a : [S]$ declares that the session name a is able to open a session whose session channel k has type S. The session type S is constructed along the use of its subject k in the process P, i.e.:

$$\frac{\Gamma, x : T \vdash P \triangleright \Delta, k : S'}{\Gamma \vdash k\ ?\ (x).P \triangleright \Delta, k :?\, T.S'}$$

whose dual is derived by the rule:

$$\frac{\Gamma \vdash P \triangleright \Delta, k : S'' \quad \Gamma \vdash v : T}{\Gamma \vdash k\ !\ v.P \triangleright \Delta, k :!\, T.S''}$$

Because of these rules, the type $?\,T.S'$ in the conclusion of the first rule tells that over the channel k there will be an input of a value of type T, and then the conversation will continue according to S'; similarly the type $!\,T.S''$ in the conclusion of the second rule tells that the session over k begins with output of a value of type T, and then it continues according to S''. By this we have that $!\,T.S'' = \overline{?\,T.S'}$, provided that $S'' = \overline{S'}$.

Note that to reflect the usage of the session channel in its session type, an almost linear discipline is imposed to the typings Δ. In particular the axiom $\Gamma \vdash \mathbf{0} \triangleright \Delta$ (where $\mathbf{0}$ is the inactive process) requires that Δ associates only the session type end (the type of the completed sessions) to channels. As a consequence weakening of the typing Δ is not admissible but for typings of this form.

We omit the rules for typing selection, branching and parallel composition, which can be found for instance in [YV07].

The actual strength of the π-calculus w.r.t. CCS and similar process algebras consists in the ability to send and receive names. We have seen above that the formalism chosen for the endpoint calculus is essentially a dialect of the π-calculus, extended with session initiation and selection/branching primitives. We will discuss now how a restricted (and more structured) form of mobility allows to express *delegation* in the scenario of sessions and session types.

Consider the more complex version of the User-ATM protocol in Figure 2, which further includes the Bank. The point here is that, to complete its protocol, the ATM asks the Bank to deposit or to withdraw the required amount from the proper bank account. This is accomplished by opening a new session between

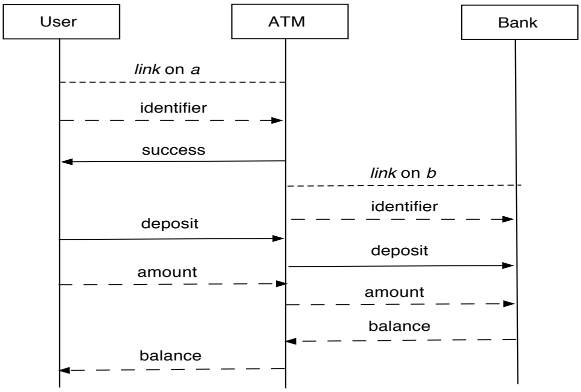


Fig. 2. UML sequence diagram of a User-ATM and Bank interaction

the ATM and the Bank, which is the agent that ultimately is expected to send or to receive the amount determined by the User:

$$
\begin{array}{llll}
\multicolumn{4}{l}{a(h).\ h\ ?\ (x).} \\
\multicolumn{4}{l}{\mathsf{if}\ \ldots\mathsf{then}} \\
\quad h \oplus \mathsf{success} : \bar{b}(k).\ k\ !\ x.\ h\&\{ & \mathsf{deposit} : k \oplus \mathsf{deposit} : & & \\
 & \quad h\ ?\ (y).\ k\ !\ y.\ k\ ?\ (z).\ h\ !\ z.\ \mathbf{0} & & \\
 & [\!]\ \mathsf{withdraw} : k \oplus \mathsf{withdraw} : & & \\
 & \quad h\ ?\ (t).\ k\ !\ t. & & \\
 & \quad k\&\{\ \mathsf{dispense} : h \oplus \mathsf{dispense} : \cdots & & \\
 & \qquad [\!]\ \mathsf{overdraft} : h \oplus \mathsf{overdraft} : \cdots & & \\
 & \qquad \} & & \\
 & \} & & \\
\multicolumn{4}{l}{\mathsf{else}\ h \oplus \mathsf{failure} : \mathbf{0}}
\end{array}
\qquad (7)
$$

The service name $\bar{b}$ is used to require a connection to the Bank, and uses the session channel k. Its first use is to send to the Bank the identifier, received on x from the User. Then the ATM plays just the role of a forwarder between the User and the Bank and vice versa. A quite different approach, however, would be to *delegate* (say just after authentication) all the ATM job to the Bank by:

$$
\begin{array}{ll}
a(h). & h\ ?\ (x). \\
\mathsf{if}\ \cdots & \mathsf{then}\ h \oplus \mathsf{success} : \bar{b}(k).\ k\ !\ x.\ k\ !\ h.\,\mathbf{0} \\
 & \mathsf{else}\ h \oplus \mathsf{failure} : \mathbf{0}
\end{array}
\qquad (8)
$$

In the process (8) the session channel h, which is supposed to carry the conversation with the User, is passed along k to the Bank, that will continue the interaction directly with the User. This is however transparent to the User, who is unaware of the fact that the opposite endpoint is now held by some different partner.

Delegation is achieved by allowing higher-order sessions, i.e. by allowing to send channels over channels[6]:

$$(\kappa^p \; ! \; \kappa_1^q.P) \mid (\kappa^{\bar{p}} \; ? \; (h).Q) \longrightarrow P \mid Q\{\kappa_1^q/h\}$$

How is this reflected in the type system? Is typing able to guarantee to the User that either interaction with the non delegating ATM (7) or with the delegating ATM (8) will always comply with the protocol formalized by the type? As a matter of fact both these issues are addressed by suitably typing the channel exchanges. The rule for the sending process is:

$$\frac{\Gamma \vdash P \triangleright \Delta, k : S_1}{\Gamma \vdash k \; ! \; h.\; P \triangleright \Delta, k : \, !\, S_2.S_1, h : S_2}$$

where h is a fresh name. Because of this the new channel h cannot occur in P, even if it is credited of the (arbitrarily complex) usage described in S_2. This is essential for session fidelity to hold: looking at the example (8), if the ATM could save an occurrence of h that could be used after having been sent to the Bank, then the conversation with the User would be ambiguously directed either to the ATM or to the Bank, and the interaction might end up in some unexpected way. For example the process

$$(\kappa^+ \; ! \; \kappa_1^+.\kappa_1^+ \; ! \; \mathsf{true}.\mathbf{0}) \mid (\kappa^- \; ? \; (h).h \; ! \; \mathsf{false}.\mathbf{0}) \mid (\kappa_1^- \; ? \; (x).P)$$

reduces to

$$(\kappa_1^+ \; ! \; \mathsf{true}.\mathbf{0}) \mid (\kappa_1^+ \; ! \; \mathsf{false}.\mathbf{0}) \mid (\kappa_1^- \; ? \; (x).P)$$

where the linearity of the channel κ_1^+ is lost. The last process can non deterministically give either $(\kappa_1^+ \; ! \; \mathsf{false}.\mathbf{0}) \mid P\{\mathsf{true}/x\}$ or $(\kappa_1^+ \; ! \; \mathsf{true}.\mathbf{0}) \mid P\{\mathsf{false}/x\}$, so no communication protocol is respected.

On the other hand the receiving process will bind a session channel h:

$$\frac{\Gamma \vdash Q \triangleright \Delta, k : S_1, h : S_2}{\Gamma \vdash k \; ? \; (h).Q \triangleright \Delta, k : \, ?\, S_2.S_1}$$

It is indeed essential that the actual usage in Q of the channel h is controlled by the type S_2, which suffices to guarantee that the delegated session will continue as expected by the partner. This implies that, while the type of k obviously changes, the session type of the delegated session in (8) remains the same as in the case of (7) without delegation.

By admitting recursive definitions of processes, also protocols of unbounded sequences of actions can be expressed.

[6] Observe that, since channel names can only be introduced by the initiation of a session, where they occur within the scope of the restriction operator ν, the communicated names are always private, that is only “internal mobility” is permitted (see [SW01], Chap. 5.7). However in [Bor98] it is shown that the internal π-calculus has the same expressive power, up to barbed-bisimulation, as the asynchronous π-calculus, which in turn is known to encode the full π-calculus: see [SW01], Chap. 5.5.

We remark that while global types have straightforward projections into session types, this fails on the process side. Although this is not the case of our examples, the projection map sending global interactions into end-point processes is quite complex. In fact it is a partial map which is defined only if the given interaction satisfies *connectedness*, *well-threadedness* and *coherence* conditions, as they are detected via a further refinement of the global typing system (for more details see [CHY07]).

The interested reader wishing a more technical presentation of the basics of session types might consult [Vas09a], where Vasconcelos presents a reconstruction of session types in a linear π-calculus with a restriction operator binding at the same time two variables and establishing that they are the two end-points of communications.

3 Extensions

In this section we discuss, mainly through schematic examples, some extensions of sessions and session types that allow to increase their expressivity and consequently to widen their applications.

3.1 Extensions of the Calculus

Correspondence Assertions. In the example (7) of the User-ATM-Bank sketched in the previous section, a malicious ATM′ could send to the Bank an amount of money different from that communicated by the User, and consequently altering the balance obtained from the Bank:

$$\begin{array}{l}\mathsf{ATM}' = \\ a(h).\ \cdots \bar{b}(k).\ \cdots \\ \qquad\qquad\qquad \mathsf{deposit} : h\ ?\ (y).\ k\ !\ y-10.\ \ k\ ?\ (z).\ h\ !\ z+10. \\ \qquad\qquad\qquad \cdots \end{array} \tag{9}$$

This change is transparent to the typing, since it does not modify the communication protocol. In order to cope with such kind of misbehaviour, in [BCG05] Bonelli et al. incorporate *correspondence assertions* in the theory of session types. In particular to detect the misbehaviour of ATM′ one is enabled to include two correspondence assertions (which are tagged tuples of expressions) into the codes of the User and of the Bank, intended to state that values of both the amount and the balance are the same:

$$\begin{array}{l}\mathsf{User}' = \ \bar{a}(h).\ \cdots\ h\ !\ \mathsf{amount}.\ \ h\ ?\ (x).\ \mathsf{cBegin}\,\langle \mathsf{amount}, x\rangle.\ \cdots \\ \mathsf{Bank}' = b(k).\ \cdots\ k\ ?\ (y).\ k\ !\ \mathsf{balance}.\ \mathsf{cEnd}\,\langle y, \mathsf{balance}\rangle.\ \cdots\end{array}$$

Then the type system can discover the malicious behaviours of the ATM′ since in the type checking of the process $\mathsf{User}' \mid \mathsf{ATM}' \mid \mathsf{Bank}'$ the tuples $\langle \mathsf{amount}, x\rangle$ and $\langle y, \mathsf{balance}\rangle$, paired by the keywords cBegin and cEnd, do not match.

In general type systems with session types and correspondence assertions can be used to check:

- source of information,
- whether data is propagated as specified across multiple parties,
- if there are unspecified communications between parties, and
- if *the data being exchanged have been modified* by the code in some unexpected way.

Multiparty Sessions. In a multiparty session we can have any number of participants. So a multiparty session forms a unit of structured interactions among many participants which follow a prescribed scenario specified as a global type signature. Multiparty sessions were first designed in [HYC08], but we follow the syntax of [BCD$^+$08], being closer to that one used here for dyadic sessions. For example a global type describing the User-ATM-Bank interaction with three participants is:

$$\begin{array}{lll}
\mathsf{User} \longrightarrow \{\mathsf{ATM}, \mathsf{Bank}\} : & \mathsf{String}. & \\
\mathsf{ATM} \longrightarrow \{\mathsf{User}, \mathsf{Bank}\} : & & \\
\{\ \mathsf{success} : \mathsf{User} \longrightarrow \mathsf{Bank} : & & \\
 & \{\ \mathsf{deposit} : & \mathsf{User} \longrightarrow \mathsf{Bank} : \mathsf{Real}. \\
 & & \mathsf{Bank} \longrightarrow \mathsf{User} : \mathsf{Real}. \\
 & & \mathsf{end} \\
 & [\,]\ \mathsf{withdraw} : & \mathsf{User} \longrightarrow \mathsf{Bank} : \mathsf{Real}. \\
 & & \mathsf{Bank} \longrightarrow \{\mathsf{User}, \mathsf{ATM}\} : \\
 & & \{\ \mathsf{dispose} : \mathsf{end} \\
 & & [\,]\ \mathsf{overdraft} : \mathsf{end} \\
 & & \} \\
 & \} & \\
[\,]\ \mathsf{failure} : \mathsf{end} & & \\
\} & &
\end{array} \tag{10}$$

In this context the arrow does not just indicate the direction of a message: $\mathsf{User} \longrightarrow \{\mathsf{ATM}, \mathsf{Bank}\} : \mathsf{String}$ expresses that the User sends *the same* String to the ATM and to the Bank by means of a unique action. Differently than in the dyadic case, when projecting the global type:

$$\mathsf{User} \longrightarrow \{\mathsf{ATM}, \mathsf{Bank}\} : \mathsf{String}$$

we have to take into account the roles to which the single actions are projected, giving the slightly more verbose session types:

$$!\,\langle\{\mathsf{ATM}, \mathsf{Bank}\}, \mathsf{String}\rangle \qquad ?\,\langle\mathsf{User}, \mathsf{String}\rangle \qquad ?\,\langle\mathsf{User}, \mathsf{String}\rangle$$

for respectively the User, the ATM and the Bank.

On the process side the session initialization primitives declare the role of the single participants (labelled by a natural number), but for one (distinguished by the over-bar on the service name) which being the last one declares the overall number of participants. For example, writing the initial actions of each partner in columns which are separated by the parallel composition operator we get for the previous example:

$$\begin{array}{c|c|c} a[1](k_1). & a[2](k_2). & \bar{a}[3](k_3). \\ k_1 \,!\, \langle\{2,3\},\mathsf{id}\rangle. & k_2 \,?\, \langle 1,x\rangle. & k_3 \,?\, \langle 1,y\rangle. \\ \ldots & \ldots & \ldots \end{array}$$

where each communication specifies either the set of the receivers or the sender.

Concurrent Constraints. Following the approach of [BM07, BM08] the paper [CDC09] proposes a calculus which combines concurrent constraints, name passing and sessions. Public and private constraints specify the requirements of session participants to open new interactions and to conduct them. More precisely the primitives for session initiation allow the programmer to specify a set of constraints whose satisfaction is necessary for starting the session interaction. For example a service could offer different times and prices:

$$a\{\mathsf{deliveryTime} = 3 \mid \mathsf{price} = 10\}(k).\ldots$$
$$a\{\mathsf{deliveryTime} = 5 \mid \mathsf{price} = 7\}(k).\ldots$$

so that a rushed client $\bar{a}\{\mathsf{deliveryTime} \leq 4\}(h).\ldots$ will choose the first option; a thrifty client $\bar{a}\{\mathsf{price} \leq 9\}(h).\ldots$ will take the second one; finally a too demanding client $\bar{a}\{\mathsf{deliveryTime} \leq 4 \mid \mathsf{price} \leq 9\}(h).\ldots$ will refuse the connection at all.

In this calculus we have:

- a *fusion* mechanism that explicitly represents, through the notion of constraint, relations involving private and public names,
- *symmetric data communication* both in input and in output, achieved via the introduction of constraints between channel names.

A simple example showing how communication is realised by fusion - i.e. just by creating a new constraint and putting it in parallel with the process continuations - is:

$$\kappa^{+}(\mathsf{amount}).P \mid \kappa^{-}(x).Q \longrightarrow P \mid Q \mid \mathsf{amount} = x$$

The main technical problem is to preserve the linearity of session channel usage in presence of delegation and constraints.

Lopez et al. [LPO10] encode *a timed extension of multi-party sessions* [HYC08] *into the timed process calculus with concurrent constraints* of [OV08]. The timed extension explicitly includes information on session duration, allows for declarative preconditions within session initiations, and features a construct for session abortion. Since the processes of [OV08] can be interpreted as linear temporal logic formulas, the given encoding allows to verify properties of structured communications.

Code Mobility. Mostrous and Yoshida propose in [MY07, MY09] a calculus of sessions in which processes can be sent and received, i.e. a calculus of sessions with *higher-order processes.* The advantage is to avoid many remote interactions. For example the ATM could send a process to the Bank in order to directly interact with the User. [MY09] discusses also how actions can be permuted in order to increase efficiency.

The main challenge of this approach is the preservation of the linear use of session channels while allowing instantiation of names into executable code.

Exceptions. Carbone et al. in [CHY08] propose a notion of exceptions for sessions which they call *interactional exceptions.* These exceptions demand not only local but also coordinated actions between session participants. The main features of the proposed calculus typed by sessions with exceptions are:

- *flexibility*: exceptions are allowed at any point of a conversation;
- *consistency*: messages in normal and exception conversations are not mixed-up;
- *safety*: communications inside sessions take place linearly and without communication mismatch.

Resource Access Control through Delegation. Capecchi et al. in [CCDR09] enrich the calculus of multiparty sessions with security levels of participants and data. A suitable type system assures that each participant can only receive data of security levels less than or equal to its own security level. For example in a well-typed protocol involving a Customer, a Seller and a Bank, the "secret" credit card number of the Customer is communicated to the Bank, but not to the Seller. This is realised also by making delegation explicit in the typing of the delegated session channel. Typing prevents any leak of information due to selection/branching too.

3.2 Extensions of the Typing

Subtyping. The idea of subtyping, coming from the typed λ-calculus, is that any value of a certain type can be safely placed in a context expecting a value of some more general type: this principle is called *subsumption* (for a handy and clear explanation of the concepts of subtyping and subsumption see [Bru02], Chap. 5). In the setting of the π-calculus, where only names have a type, the subsumption rule takes a dual form (also called *narrowing*): if a type T' describes a more general kind of data than T, written $T \leq T'$, then any name typable by T' in the process P can safely be typed by T in the same process. This is sound with respect to communication safety because for example, an ATM which accepts a Real amount of money can safely communicate with a User who sends an Int amount of money, which is formally expressed by postulating $\mathsf{Int} \leq \mathsf{Real}$ and by deriving $?\mathsf{Int} \leq ?\mathsf{Real}$.

The concept of subtyping, originally conceived for input/output types (see [SW01] Chap. 7, where the covariance/contravariance of input and output actions - respectively - is explained) has been extended to session types by Gay and

Hole in [GH05]. An ATM which offers on a channel both deposit and withdraw can safely communicate through that channel with any User willing just to do a deposit action, which can be expressed by:

$$\&\{\mathsf{deposit} : S_1\} \leq \&\{\mathsf{deposit} : S_1, \mathsf{withdraw} : S_2\}.$$

On the contrary a User who is willing to do a deposit through a certain channel will comply with any environment ready to interact over that channel with someone either asking for a deposit or for a withdraw:

$$\oplus\{\mathsf{deposit} : S_1, \mathsf{withdraw} : S_2\} \leq \oplus\{\mathsf{deposit} : S_1\}.$$

To formalize this in the type system, let us consider the following rule[7]:

$$\frac{\Gamma \vdash P \triangleright \Delta, k : S' \quad S \leq S'}{\Gamma \vdash P \triangleright \Delta, k : S}$$

Then if we type the ATM by $k : \&\{\mathsf{deposit} : S_1, \mathsf{withdraw} : S_2\}$ in the premise, we know that it is offering both actions, so that in particular it will do with just deposit, as stated in the conclusion. On the other hand if we know from the premise that the User will do just a deposit, *a fortiori* she/he will be correctly communicate with an ATM accepting either a deposit or a withdraw selection action, which is spelled out in the conclusion.

Summarizing:

- input is covariant,
- output is contra-variant,
- branching is covariant in the number of branches,
- selection is contra-variant in the number of branches,
- both branching and selection are covariant in the continuation types.

This has the remarkable consequence that, if S, S' are session types and $S \leq S'$, then $\overline{S'} \leq \overline{S}$.

Subtyping enhances expressivity of typing with session types since it allows:

- *refinement of participants* without invalidating type-correctness of the overall system,
- *participants to follow different protocols* which are nevertheless compatible according to the subtype relation.

Bounded Polymorphism. A more precise and flexible specification of protocols is obtained in [Gay07] by introducing bounded polymorphism. In particular *a choice of type in one message may affect the types of future messages.* For example

$$\begin{array}{l}\&\{\ \mathsf{opp}\ (\mathsf{Int} \leq X \leq \mathsf{Complex}) : ?\,X.\ !\,X.\ \mathsf{end} \\ \quad\quad \ldots \\ \ \ \}\end{array}$$

[7] This rule is only admissible in the system studied in [GH05], where a more syntax directed presentation is indeed preferred.

is the type of a calculator which offers an opposite operator working on all numbers whose type is between Int and Complex, returning a number of the same type. A User typed by

$$\oplus\{\ \mathsf{opp} : !\ \mathsf{Real}.\ ?\ \mathsf{Real}.\ \mathsf{end}\}$$

could safely engage a session with such a calculator.

Progress. A very useful property is that once a session is started, the participants will be able to complete all the necessary communications without getting in a deadlock. This property - usually called progress - has been studied for several calculi; in particular Kobayashi has developed very refined techniques for the π-calculus [Kob98, Kob02, Kob05, Kob07].

Session types already assure deadlock-freeness inside single sessions. If distinct sessions do not overlap, then after a session initiation the process is never blocked. This is no longer true if a process contains two or more interleaved sessions: in fact if a session includes another one, then the outermost session might start and wait forever if the innermost session does not find a partner. For example when running the process (7), the session between the User and the ATM opened by a is blocked if there is no Bank hearing on b. In an open scenario we can assume that it is always possible to find the required partners, and therefore we do not consider this kind of cases as deadlocks. There are however situations which cannot be solved by adding suitable partners. A very simple kind of deadlock occurs when two sessions are wrongly interleaved. Consider for example the following typable process:

$$\begin{array}{c|c} a\ (k). & \bar{a}\ (k'). \\ b\ (h). & \bar{b}\ (h'). \\ k\ !\ 2. & h'\ !\ \mathsf{true}. \\ h\ ?\ (x) & k'\ ?\ (y) \\ \dots & \dots \end{array}$$

After the two session initiations we get:

$$\begin{array}{c|c} \kappa_a^+\ !\ 2. & \kappa_b^+\ !\ \mathsf{true}. \\ \kappa_b^-\ ?\ (x) & \kappa_a^-\ ?\ (y) \\ \dots & \dots \end{array}$$

which is blocked as soon as input and output actions are synchronous. Allowing asynchronous output does not avoid this kind of blocks in general, as it is shown for instance by:

$$\begin{array}{c|c} a\ (k). & \bar{a}\ (k'). \\ b\ (h). & \bar{b}\ (h'). \\ h\ ?\ (x). & k'\ ?\ (y) \\ k\ !\ 2 & h'\ !\ \mathsf{true} \\ \dots & \dots \end{array}$$

More interesting examples of deadlocks involve delegation. Type systems assuring progress are discussed in [DCdLY08] for dyadic sessions and in [BCD$^+$08] for multiparty sessions. The key ideas of these works are:

– to take advantage of *nested sessions,*
– to infer the *order of channel usage* for interleaved sessions (following [Kob05]),
– to *forbid "self-delegation"* (opposite polarities of the same session channel cannot be put in sequence).

Action Permutation. As it is well known, in asynchronous π-calculus inputs are blocking while outputs are not (see [SW01], Chap. 5). This asymmetry is the starting point of the work in [HMY09], where Honda et al. propose to *execute outputs before inputs* when possible for increasing efficiency. This change of order is realized by means of an appropriate subtyping theory, which allows automatic action permutation for multiparty sessions while assuring communication safety and session fidelity. Notably action permutation is tricky in presence of recursion and selection/branching.

3.3 Other Extensions

Semantic Subtyping. Semantic subtyping, as proposed in [CF05], is based on the interpretation of types as the sets of their inhabitants, so that subtyping turns out to be set inclusion. Type constructors are indeed interpretable as plain set theoretic operations, so that boolean combinators have their natural meaning.

In [CDCGP09] Castagna et al. propose a theory of session types in which the choices are done on the basis of the type of the messages exchanged. The standard choices through labels are then particular cases in which each label is typed with a singleton type.

An example is the process:

$$\begin{array}{l} \&\{\ k\ ?\ (x : \mathsf{Int}).\ k\ !\ -x.\ \mathbf{0} \\ \qquad k\ ?\ (y : \mathsf{Bool}).\ k\ !\ \neg y.\ \mathbf{0} \\ \quad\ \} \end{array}$$

which, when receiving either an Int or a Bool value, replies differently: in case of an Int number it answers with its opposite; on receiving a Bool value it answers with its negation. The type of the channel k in this process is therefore:

$$\begin{array}{l} \&\{\ ?\mathsf{Int}.\ !\mathsf{Int}.\ \mathsf{end} \\ \qquad ?\mathsf{Bool}.\ !\mathsf{Bool}.\ \mathsf{end} \\ \quad\ \} \end{array}$$

Consider now the slightly different type:

$$\begin{array}{l} \&\{\ ?\mathsf{Real}.\ !\mathsf{Nat}.\ \mathsf{end} \\ \qquad ?\mathsf{Int}.\ !\mathsf{Bool}.\ \mathsf{end} \\ \quad\ \} \end{array} \tag{11}$$

It can be assigned to a channel which, when receiving a Real number replies with a Nat number, and when getting an Int number answers with a Bool value. Being Int a (semantic) subtype of Real, when the channel receives an Int it can react either by sending a value of type Bool, or, by viewing the integer as a Real, a value of type Nat. A session type which is dual of this type can naturally use boolean operators within type syntax:

$$\oplus\{\begin{array}{l} !(\mathsf{Real} \wedge \neg \mathsf{Int}).\ ?\mathsf{Nat}.\ \mathsf{end} \\ !\mathsf{Int}.\ ?(\mathsf{Nat} \vee \mathsf{Bool}).\ \mathsf{end} \end{array}\\ \}$$

This type says that if the channel sends a Real number which is not an Int number, it will receive a Nat number; but if it sends an Int number, then it will receive either a Nat number or a Bool value. In this way one can obtain a finer description of behaviours within the formalism of session types. Note that also the types $!(\mathsf{Real} \wedge \neg\mathsf{Int}).\ ?\mathsf{Nat}.\ \mathsf{end}$ and $!\mathsf{Int}.\ ?(\mathsf{Nat} \vee \mathsf{Bool}).\ \mathsf{end}$ are dual of (11): namely duality is not involutive in this theory.

In [CDCGP09] also duality is defined semantically: two session types are dual if no conversation on a private channel shared by two processes which follow the prescriptions of these two types ever gets stuck.

In this scenario, where types play a computational role, *session types can be interpreted as the sets of their dual types* and the semantics of boolean combinators is set theoretical. This interpretation of session types gives a semantic subtyping relation, since it is safe to replace a channel with another one when every dual of the replacing channel is also a dual of the replaced one.

Hennessy-Milner Logic. Berger et al. in [BYH08] present an extension of Hennessy-Milner Logic suitable to capture the behaviours of session participants. The basic concept of this logic is the *hypothetical parallel composition* formula $A \gg B$, which means: if a process satisfying A is put in parallel with a process that satisfies this formula, then the resulting process will satisfy B. For example the process

$$P \equiv a(k).\ k \ \oplus\ \mathsf{opp}:\ k\ !\ 2.\ k\ ?\ (x).\ h\ !\ x.\mathbf{0}$$

offers a session initiation on the service name a binding the session channel k, then along k it selects the label opp, sends the integer 2, and receives an input which is bound to the variable x. Eventually it sends over the channel h the value of the variable x. Let Q be a process which offers a session initiation on the service name $\bar{a}$ binding the session channel k'; then using the session channel k' it offers a branch labelled opp, receives an input which is bound to the variable y and then sends the opposite of y. It is clear that the process obtained by putting P and Q in parallel may reduce to a process which sends -2 on the channel h. This is expressed in the logic language by saying that P has the property $A \gg h\ !\ -2\ \mathsf{true}$, where

$$A = \forall y^{\mathsf{Int}}.\bar{a}(k').\ k'\ \&\ \mathsf{opp}:\ k'\ ?\ (y).\ k'\ !\ -y.\ \mathsf{true}.$$

4 Session Embedding in Programming Paradigms

In the previous sections sessions have been considered in the context of the π-calculus. In this section instead we will briefly overview how sessions can be incorporated into two mainstream programming paradigms, i.e. the functional and the object-oriented ones. In this way one can achieve powerful type systems which are suited to programming practice, while retaining the benefit of a sound theoretical foundation.

4.1 Functional Paradigm

Vasconcelos et al. [VGR06, Vas09b] transfer the concept of session and session type to a multi-threaded functional language with side-effecting input/output operations. This shows that static checking of session types can be fruitfully added to a language such as Concurrent ML [Rep99] or Concurrent Haskell [JGF96]. For example a functional version of the process (6) would be:

$$\begin{array}{lll}
a\ \ h & = \mathsf{let}\ x = \mathsf{receive}\ h\ \mathsf{in} & \\
 & \quad \mathsf{if}\ \cdots \mathsf{then\ select\ success\ on}\ h\ \mathsf{case}\ h\ \mathsf{of} & \{\ \mathsf{deposit}\ \ \Rightarrow \cdots \\
 & & \quad \mathsf{withdraw} \Rightarrow \cdots \} \\
 & \qquad\quad \mathsf{else\ select\ failure\ on}\ h &
\end{array}$$

Characteristics of this embedding are:

- *the operations on channels are independent terms* rather than prefixes of processes,
- *the communication is asynchronous*,
- *typing is enhanced by subtyping*, which also allows anticipation of outputs with respect to inputs.

In the recent paper [GV10] Gay and Vasconcelos simplify and extend previous work by giving an operational semantics with buffered channels and by proving that *the session type of a channel gives an upper bound on the necessary size of the buffer*. A novel form of subtyping between standard and linear function types reduces the burden of linear typing on the programmer, by allowing standard function types to be inferred by default and converted to linear types if necessary.

4.2 Object-Oriented Paradigm

Moose. Moose (Multi-threaded Object-Oriented calculus with Sessions) is a multi-threaded object-oriented calculus augmented with session primitives, which supports session names as parameters of methods, spawning, iterative sessions and delegation (see [DCDMY09] and the references there). Progress is enhanced by *spawning a new thread when a session channel is received*: in this way self-delegation never happens. *Choice is made on the basis of the class of the object being sent/received* instead of using labels. Through bounded polymorphism the class of a received object may affect the class of the objects which will be sent.

SAM. The design of the SAM (Sessions Amalgamated with Methods) calculus originates from the comparison between sessions and methods in [CCDC$^+$09]. From this comparison a new notion of session is derived, which subsumes the notion of method. In SAM *classes have fields and sessions*, session bodies are selected on the ground of object classes, and channels are created only at run time when sessions are called. Invocation takes place on an object, say a customer asking to withdraw money from a particular ATM machine, and execution of the corresponding session takes place immediately and concurrently with the requesting thread. The body is defined in the class of the receiving object, namely in the class implementing the ATM of our example, and any number of communications interleaved with computations is possible.

For example the ATM class might contain the declaration of a session:

```
void   ?String... atmserver
                  {String x := receive;
                    ...
                  }
```

where void is the return type of the session, ?String... is the session type shown in example (4), atmserver is the session name and the code between brackets is the session body - in this case a translation of the process (6). A User can then call this session on an ATM object by:

```
new  ATM . atmserver {send (identifier);
                       ...
                     }
```

where the code between brackets is in this case a translation of the process (5). Notably there are no channels in the source code, and only polarised channels will be generated at run time.

Delegation is limited since it does not support an initial and a final dialogue before and after the delegation itself. Expressiveness of typing in SAM has been enhanced with union types [BCDC$^+$08] and generics [CCDC$^+$09].

Session Object Calculus. Mostrous and Yoshida propose in [MY08] an extension of Abadi and Cardelli imperative object calculus (see [AC96]) with sessions, naturally integrating session based choices with method invocations. The main features of this typed calculus are:

- *objects can be spawned, updated and cloned*,
- communication is asynchronous,
- subtyping enjoys the minimal subtyping property.

Modular Session Types as Dynamic Interfaces. In the object-oriented calculus of [VGR$^+$09] the availability of methods depends on object states: object interfaces are dynamic. *Each class has a session type which provides a global specification of the availability of methods at each state.* The typing of a method specifies pre- and post-conditions for its object states and static typing guarantees that methods are only called when they are available. A key feature is that the state of an object may depend on the result of a method whose return type is an enumeration.

Inheritance is included; a subtyping relation on session types characterises the relationship between method availability in a subclass and in its superclass.

Building on [VGR$^+$09], Gay et al. show in [GVR$^+$10] that *a session can be modularised by dividing it into distinct methods that can be called separately.* A key idea is to allow a channel to be stored in a field of an object. Several methods can operate on the same channel, thus allowing to effectively encapsulate channels into objects, while retaining the usual object-oriented development practice.

5 Implementations

Naturally implementations of sessions and session types require their embedding in the used languages, so that it is not surprising that implementations have been done using functional and object-oriented languages, even if to the time among the works surveyed in the previous section just [GVR$^+$10] (see the last paragraph of Section 4.2) is implemented by Bica (see the last paragraph of Section 5.2).

On the contrary it is worthwhile to notice the interplay between the theory sketched in Sections 2 and 3 and the actual implementations of sessions and session types mentioned below. For example the Haskell implementation by Sackman and Eisenbach (see Section 5.1) has first realised the action permutation studied then by Mostrous, Yoshida and Honda (see the last paragraph of Section 3.2). Multiparty sessions were first implemented in Scribble (see Section 5.2) and then formalised by Honda, Yoshida, and Carbone (see Section 3.1).

5.1 Functional Languages

The first implementation of sessions and session types was done by Neubauer and Thiemann [NT04] into *Haskell*. The core of this and of the following implementations into Haskell is the definition of a *session monad*. Type classes with functional dependencies model the progression of the current state of the channel. Functions with polymorphic parameters model client and server side of a communication with one specification.

Sackman and Eisenbach give in [SE08] an implementation of sessions as a standard Haskell library. This implementation presents a monadic API to the programmer. In particular the *SMonad* type class is a type indexed monad: it allows to represent a computation from a state to another one which additionally produces a value, where the two states can have different types. Since session types are encoded into Haskell types, no preprocessor, external type checker or modification to the Glasgow Haskell compiler are required.

At the address `http://www.agusa.i.is.nagoya-u.ac.jp/person/sydney/full-sessions.html` one can find a tool providing session type inference in Haskell using Haskell type-level programming.

Bhargavan et al. describe in [BCD$^+$09] a compiler from high-level multiparty session descriptions to custom cryptographic protocols coded as ML modules. In the generated code each participant has strong security guarantees for all her/his messages against any adversary that may control both the network and some participant to the session.

5.2 Object-Oriented Languages

The language Sing# [FAH$^+$06] is a variant of C# which combines session types with ownership types [CNP01], supports message-based communication via a designed heap area (shared memory), and allows interfaces between OS-modules to be described as message passing conversations.

SJ [HYH08] is an extension of Java with syntax for session types and structured communication operations. The main features of SJ are asynchronous message passing, delegation, session subtyping, interleaving, class downloading, and failure handling. The compilation-runtime framework of SJ maps session abstraction onto underlying transports, and guarantees communication safety through static and dynamic session type checking. A User coded into SJ could be:

```
s.request( ); s.send(identifier);
              s.inbranch( ) {case success:  if (···){s.outbranch(deposit);...}
                                                 else {s.outbranch(withdraw);...}
                            }
                            {case failure: }
```

Scribble (http://sourceforge.net/projects/pi4scribble/) is a language for describing global (choreography) and local (service end-point) behaviour. Extensible tools are provided, both as stand alone applications and as Eclipse plugins, to edit the language, to perform validation and to export specifications to other formalisms.

Bica (http://gloss.di.fc.ul.pt/bica/) implements the type system of [GVR$^+$10]. This implementation comprises an extension to the Java 5 compiler that checks conventional Java source code against session type specifications for classes (included in Java annotations). The extension touches the type checker only: if a program satisfies the more stringent type system of [GVR$^+$10], then code is generated as usual. Bica is implemented with Polyglot.

6 Related Concepts and Formalisms

The present section quickly surveys on formalisms which look to be closely related to sessions and session types, and it contains some pointers to the literature.

6.1 Generic Process Types

Igarashi's and Kobayashi's generic type system (GTS: see [IK04]) is a powerful framework from which one can obtain as instances a variety of type systems for the π-calculus guaranteeing strong properties like deadlock and race-freedom. Not surprisingly also systems of session types can be formalised into GTS [Kob07, GGR08] although via non trivial translations. However, as observed by Gay et al.

in [GGR08], this does not invalidate the usefulness of session types mainly because:

1. *session types are valuable for program design*,
2. session types have been developed for calculi/languages different from π-calculus,
3. proofs of type soundness for session types are fairly straightforward,
4. type checking algorithms for session types cannot be easily obtained via translation, since GTS does not yield an algorithm automatically.

6.2 Contracts

Contracts are behavioural descriptions of Web services [MB03]. In [CGP09] Castagna et al. formalise contracts by means of a sublanguage of CCS without τ (see [DH87]), namely with both external and internal choice, but not including the parallel operator. For example a contract for the ATM process (6) would be:

$$\begin{array}{lll} \mathsf{Login}.(& \overline{\mathsf{Success}}. & (\mathsf{Deposit}.\mathsf{Amount}.\overline{\mathsf{Balance}}.\mathbf{0}\ + \\ & & \quad \mathsf{Withdraw}.\mathsf{Amount}.(\overline{\mathsf{Dispense}}.\cdots \oplus \overline{\mathsf{Overdraft}}.\cdots)) \\ \quad \oplus & \overline{\mathsf{Failure}}.\mathbf{0}) & \end{array}$$

Names and co-names model the input and output actions, respectively; the external choice $+$ is a selection by the ATM counterpart, while the internal choice $\oplus$ represents decisions by the ATM itself.

The main difference between contracts and session types is that contracts record the overall behaviour of a process, while session types project this behaviour onto the private channels that a process uses.

A prominent feature of the theory of contracts is the subcontract relation: if σ is a subcontract of τ, written $\sigma \preceq \tau$, then any client which is satisfied with a service described by σ, will comply with a service described by τ, since the latter possibly includes more capabilities than those described in σ.

In [LP08] Laneve and Padovani give two encodings, from contracts to session types and from session types to contracts[8]. It is also shown that, if $\sigma \preceq \tau$, then the translation of σ is a subtype of the translation of τ in the sense of [GH05].

As remarked in [BCdL10], however, when allowing session delegation, the direct formalisation of the idea that a subcontract can be the description of some "shorter interaction" (as it is in [CGP09]), leads to the collapse of the subtyping relation; this can be avoided at the price of considering subtyping and subcontract as different notions.

The distance between contracts and session types has been narrowed in [CP09] by defining a theory of contracts with explicit channels, so that delegation becomes expressible.

Padovani in [Pad09] presents session types roughly speaking as projections of contracts. The main contributions of this work are:

[8] These encodings are however far more complex than what the last example seems to suggest.

- session types are generalised to processes similar to value-passing CCS,
- session types can be composed by a parallel composition operator (as conversation types, see Section 6.3),
- participants can use channels for communicating after delegating them.

A last remark is that there is a clear similarity between global types and session types on one side and choreography and contracts (as defined in [BZ07]) on the other side. We think that such a relation should be further investigated in order to gain a deeper view of both formalisms.

6.3 Conversation Calculus

The conversation calculus (see [CV09] and the references there) organizes behaviour around places of conversation, which slightly resemble Boxed Ambients [BCC04]. The conversation types record the overall behaviour of processes and assure progress, while accounting for dynamical join/leave of a possibly unanticipated number of participants.

An example of [CV09] showing how the conversation calculus takes advantage of localities is the following composition of two conversation contexts, named *Buyer* and *Seller*:

$$\begin{array}{l} \mathit{Buyer} \blacktriangleleft [\ \mathbf{new}\ \mathit{Seller} \cdot \texttt{startBuy} \Leftarrow \mathsf{buy}!\mathsf{prod}.\ \texttt{price}?(v)\ ]\ | \\ \mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB}\ |\ \mathbf{def}\ \texttt{startBuy} \Rightarrow \texttt{buy}?(\mathit{prod}).\ \texttt{askPrice}^{\uparrow}!\mathit{prod}. \\ \hspace{20em} \texttt{readVal}^{\uparrow}?(v).\ \texttt{price}!v\ ] \end{array}$$

The code $\mathbf{new}\ \mathit{Seller} \cdot \texttt{startBuy} \Leftarrow$ calls the service $\texttt{startBuy}$ located at *Seller*. This system reduces to

$$\begin{array}{l} (\nu\ c)(\ \mathit{Buyer} \blacktriangleleft [c \blacktriangleleft [\ \texttt{buy}!\mathsf{prod}.\ \texttt{price}?(v)\ ]\ ]\ | \\ \qquad \mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB}\ |\ c \blacktriangleleft [\texttt{buy}?(\mathit{prod}).\ \texttt{askPrice}^{\uparrow}!\mathit{prod}. \\ \hspace{16em} \texttt{readVal}^{\uparrow}?(v).\ \texttt{price}!v\ ]\ ]\ ) \end{array}$$

where c is the fresh name of the new created conversation context. The code in the *Buyer* side of c sends a product and receives a price, both in the current conversation c. The code in the *Seller* side of c first receives a product in the conversation c, then it consults the database *PriceDB* by means of the messages superscripted by ↑ which are targeted to the parent conversation (*Seller*), and finally it sends a price in the conversation c.

6.4 Calculi for Web Services

The work on this subject is documented by a large and rapidly growing body of literature, which cannot be accounted for shortly in the present survey. Therefore we just mention three calculi that look more closely related to sessions and session types: the Service Centred Calculus (SCC) [BBC$^+$06], the Calculus of

Sessions and Pipelines (CaSPiS) [BBDNL08], and the Calculus for Orchestration of Web Services (COWS) [LPT07]. Common features of these calculi are:

- a clear *distinction between users and services*,
- that *services are permanent*,
- that sessions can only be nested,
- the presence of operators for *explicit closure of sessions*,
- that values can be communicated from an inner session to an outer one (*pipeline*).

For instance, the SCC process $\mathsf{succ} \ \Rightarrow \ (x)x + 1$ models a service that, received an integer, gives its successor. A client for this service will be written $\mathsf{succ}\ \{(y)(z)\ \mathsf{return}\ z\} \Leftarrow 5$: after the invocation both x and y are bound to the argument 5, the client waits for a value from the server and the received value (in this case 6) is substituted for z and hence returned as the result of the service invocation.

7 Conclusions

Session types allow the framing of newly emerged issues, in the world of communication centred programming and web services, into the mainstream of type theories and systems, familiar from the functional and object-oriented languages theory and practice. As it is inherent to such approach, they are based on abstractions which just approximate the desired goal of detecting and certifying that certain desirable properties are satisfied by given pieces of code or system specifications. This has however the advantage of being a well-understood technique, that can be implemented efficiently and, by the way, embodied into compilers or software development tools assisting programmers and system designers. On the other hand this should be contrasted with the modelling of processes into process algebras, where powerful but unfeasible concepts of equivalence are used to abstract from implementation details and to distil a precise notion of behaviour.

We think that, as it happens in other fields, it is a matter of balance between expressivity and feasibility, which can be reached only via a deeper understanding of the involved concepts and of their intrinsic complexity. This seems to be the reason why session types look like processes, or why processes - possibly involving few computational combinators - are often thought of as specifications, rather than as concrete implementations. Because of these reasons we think that the work of comparing session types and related systems with other process-based formalisms is worthy and might be fruitful to step to a new generation of calculi and reasoning tools apt to the emerging challenges of the world-wide computing.

Acknowledgments. We gratefully thank Giuseppe Castagna, Ilaria Castellani, Vasco Vasconcelos, Simon Gay, and Luca Padovani for their comments and suggestions on an early draft of the present paper.

References

[AC96] Abadi, M., Cardelli, L.: A Theory of Objects. Springer, Heidelberg (1996)

[BBC$^+$06] Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V., Zavattaro, G.: SCC: a Service Centered Calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)

[BBDNL08] Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and Pipelines for Structured Service Programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 19–38. Springer, Heidelberg (2008)

[BCC04] Bugliesi, M., Castagna, G., Crafa, S.: Access Control for Mobile Agents: The Calculus of Boxed Ambients. ACM Transactions on Programming Languages and Systems 26(1), 57–124 (2004)

[BCD$^+$08] Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global Progress in Dynamically Interleaved Multiparty Sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)

[BCD$^+$09] Bhargavan, K., Corin, R., Deniélou, P.-M., Fournet, C., Leifer, J.J.: Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. In: CSF 2009, pp. 124–140. IEEE Computer Society, Los Alamitos (2009)

[BCDC$^+$08] Bettini, L., Capecchi, S., Dezani-Ciancaglini, M., Giachino, E., Venneri, B.: Session and Union Types for Object Oriented Programming. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 659–680. Springer, Heidelberg (2008)

[BCdL10] Barbanera, F., Capecchi, S., de'Liguoro, U.: Typing Asymmetric Client-Server Interaction. In: Sirjani, M. (ed.) FSEN 2009. LNCS, vol. 5961, pp. 97–112. Springer, Heidelberg (2010)

[BCG05] Bonelli, E., Compagnoni, A., Gunter, E.: Correspondence Assertions for Process Synchronization in Concurrent Communications. Journal of Functional Programming 15(2), 219–248 (2005)

[BM07] Buscemi, M., Montanari, U.: CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 18–32. Springer, Heidelberg (2007)

[BM08] Buscemi, M., Montanari, U.: Open Bisimulation for the Concurrent Constraint Pi-Calculus. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 254–268. Springer, Heidelberg (2008)

[Bor98] Boreale, M.: On the Expressiveness of Internal Mobility in Name-Passing Calculi. Theoretical Computer Science 195(2), 205–226 (1998)

[Bru02] Bruce, K.: Foundations of Object-Oriented Languages: Types and Semantics. MIT Press, Cambridge (2002)

[BYH08] Berger, M., Yoshida, N., Honda, K.: Completeness and Logical Full Abstraction in Modal Logics for Typed Mobile Processes. In: Aceto, L., Damg*r*ard, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 99–111. Springer, Heidelberg (2008)

[BZ07] Bravetti, M., Zavattaro, G.: Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007)

[CCDC$^+$09] Capecchi, S., Coppo, M., Dezani-Ciancaglini, M., Drossopoulou, S., Giachino, E.: Amalgamating Sessions and Methods in Object Oriented Languages with Generics. Theoretical Computer Science 410, 142–167 (2009)

[CCDR09] Capecchi, S., Castellani, I., Dezani, M., Rezk, T.: Session Types for Access and Information Flow Control (2009), `http://www.di.unito.it/~dezani/ccdr.pdf`

[CDC09] Coppo, M., Dezani-Ciancaglini, M.: Structured Communications with Concurrent Constraints. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 104–125. Springer, Heidelberg (2009)

[CDCGP09] Castagna, G., Dezani-Ciancaglini, M., Giachino, E., Padovani, L.: Foundations of Session Types. In: PPDP 2009, pp. 219–230. ACM Press, New York (2009)

[CF05] Castagna, G., Frisch, A.: A Gentle Introduction to Semantic Subtyping. In: PPDP 2005, pp. 198–208. ACM Press, New York (2005) (full version); Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 30–34. Springer, Heidelberg (2005) Joint ICALP-PPDP keynote talk.

[CGP09] Castagna, G., Gesbert, N., Padovani, L.: A Theory of Contracts for Web Services. ACM Transactions on Programming Languages and Systems article n.19, 31(5), p. 51 (2009)

[CHY07] 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)

[CHY08] Carbone, M., Honda, K., Yoshida, N.: Structured Interactional Exceptions for Session Types. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer, Heidelberg (2008)

[CNP01] Clarke, D., Noble, J., Potter, J.: Simple Ownership Types for Object Containment. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 53–76. Springer, Heidelberg (2001)

[CP09] Castagna, G., Padovani, L.: Contracts for Mobile Processes. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 211–228. Springer, Heidelberg (2009)

[CV09] Caires, L., Vieira, H.T.: Conversation Types. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 285–300. Springer, Heidelberg (2009)

[DCdLY08] Dezani-Ciancaglini, M., de' Liguoro, U., Yoshida, N.: On Progress for Structured Communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007 and FODO 2008. LNCS, vol. 4912, pp. 257–275. Springer, Heidelberg (2008)

[DCDMY09] Dezani-Ciancaglini, M., Drossopoulou, S., Mostrous, D., Yoshida, N.: Session Types for Object-Oriented Languages. Information and Computation 207(5), 595–641 (2009)

[DH87] De Nicola, R., Hennessy, M.: CCS Without τ's. In: Ehrig, H., Levi, G., Montanari, U. (eds.) CAAP 1987 and TAPSOFT 1987. LNCS, vol. 249, pp. 138–152. Springer, Heidelberg (1987)

[FAH$^+$06] 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, ACM SIGOPS, pp. 177–190. ACM Press, New York (2006)

[Gay07] Gay, S.: Bounded Polymorphism in Session Types. Mathematical Structures in Computer Science 18(5), 895–930 (2007)

[GGR08] Gay, S., Gesbert, N., Ravara, A.: Session Types as Generic Process Types. In: PLACES 2008, pp. 16–21 (2008), `http://gloss.di.fc.ul.pt/places08/Places08Proceedings.pdf`

[GH05] Gay, S., Hole, M.: Subtyping for Session Types in the Pi-Calculus. Acta Informatica 42(2/3), 191–225 (2005)

[GV10] Gay, S., Vasconcelos, V.: Linear Type Theory for Asynchronous Session Types. Journal of Functional Programming 20(1), 19–50 (2010)

[GVR$^+$10] Gay, S., Vasconcelos, V., Ravara, A., Gesbert, N., Caldeira, A.: Modular Session Types for Distributed Object-Oriented Programming. In: POPL 2010, pp. 299–312. ACM Press, New York (2010)

[HMY09] Honda, K., Mostrous, D., Yoshida, N.: Global Principal Typing in Partially Commutative Asynchronous Sessions. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 316–332. Springer, Heidelberg (2009)

[HVK98] Honda, K., Vasconcelos, V., Kubo, M.: Language Primitives and Type Disciplines for Structured Communication-based Programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)

[HYC08] Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM, New York (2008)

[HYH08] Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 516–541. Springer, Heidelberg (2008)

[IK04] Igarashi, A., Kobayashi, N.: A Generic Type System for the Pi-Calculus. Theoretical Computer Science 311(1-3), 121–163 (2004)

[JGF96] Jones, S.P., Gordon, A., Finne, S.: Concurrent Haskell. In: POPL 1996, pp. 295–308. ACM Press, New York (1996)

[Kob98] Kobayashi, N.: A Partially Deadlock-Free Typed Process Calculus. ACM Transactions on Programming Languages and Systems 20(2), 436–482 (1998)

[Kob02] Kobayashi, N.: A Type System for Lock-Free Processes. Information and Computation 177, 122–159 (2002)

[Kob03] Kobayashi, N.: Type Systems for Concurrent Programs. In: Aichernig, B.K., Maibaum, T. (eds.) Formal Methods at the Crossroads. LNCS, vol. 2757, pp. 439–453. Springer, Heidelberg (2003)

[Kob05] Kobayashi, N.: Type-Based Information Flow Analysis for the Pi-Calculus. Acta Informatica 42(4-5), 291–347 (2005)

[Kob06] Kobayashi, N.: A New Type System for Deadlock-Free Processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)

[Kob07] Kobayashi, N.: Type Systems for Concurrent Programs. In: Extended version of [Kob03], Tohoku University (2007)

[LP08] Laneve, C., Padovani, L.: The Pairing of Contracts and Session Types. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 681–700. Springer, Heidelberg (2008)

[LPO10] López, H., Pérez, J., Olarte, C.: Towards a Unified Framework for Declarative Structured Communications. In: PLACES 2009. EPTCS, vol. 17, pp. 1–16 (2010)

[LPT07] 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)

[MB03] Meredith, G., Bjorg, S.: Contracts and Types. Communications of the ACM 46(10), 41–47 (2003)

[MY07] Mostrous, D., Yoshida, N.: Two Sessions Typing Systems for Higher-Order Mobile Processes. In: Della Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 321–335. Springer, Heidelberg (2007)

[MY08] Mostrous, D., Yoshida, N.: A Session Object Calculus for Structured Communication-Based Programming (2008), `http://www.doc.ic.ac.uk/~mostrous/sesobj.pdf`

[MY09] Mostrous, D., Yoshida, N.: Session-Based Communication Optimisation for Higher-Order Mobile Processes. In: Curien, P.-L. (ed.) TLCA 2009. LNCS, vol. 5608, pp. 203–218. Springer, Heidelberg (2009)

[NT04] Neubauer, M., Thiemann, P.: An Implementation of Session Types. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 56–70. Springer, Heidelberg (2004)

[OV08] Olarte, C., Valencia, F.: Universal Concurrent Constraint Programming: Symbolic Semantics and Applications to Security. In: SAC 2008, pp. 145–150. ACM, New York (2008)

[Pad09] Padovani, L.: Session Types at the Mirror. In: ICE 2009. EPTCS, vol. 12, pp. 71–86 (2009)

[Rep99] Reppy, J.H.: Concurrent Programming in ML. Cambridge University Press, Cambridge (1999)

[SE08] Sackman, M., Eisenbach, S.: Session Types in Haskell (Updating Message Passing for the 21st Century) (2008), `http://pubs.doc.ic.ac.uk/session-types-in-haskell/session-types-in-haskell.pdf`

[SW01] Sangiorgi, D., Walker, D.: The π-Calculus: a Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)

[THK94] 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)

[Vas09a] Vasconcelos, V.: Fundamentals of Session Types. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 158–186. Springer, Heidelberg (2009)

[Vas09b] Vasconcelos, V.: Session Types for Linear Multithreaded Functional Programming. In: PPDP 2009, pp. 1–6. ACM Press, New York (2009)

[VGR06] Vasconcelos, V., Gay, S., Ravara, A.: Typechecking a Multithreaded Functional Language with Session Types. Theoretical Computer Science 368, 64–87 (2006)

[VGR$^+$09] Vasconcelos, V., Gay, S., Ravara, A., Gesbert, N., Caldeira, A.: Dynamic Interfaces. In: FOOL 2009 (2009), `http://www.cs.cmu.edu/~aldrich/FOOL09/vasconcelos.pdf`

[YV07] Yoshida, N., Vasconcelos, V.: Language Primitives and Type Disciplines for Structured Communication-based Programming Revisited. In: SecReT 2006. ENTCS, vol. 171, pp. 73–93. Elsevier, Amsterdam (2007)

Choreography Rehearsal*

Chiara Bodei and Gian Luigi Ferrari

Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo, 3, I-56127,
Pisa, Italy
{chiara,giangi}@di.unipi.it

Abstract. We propose a methodology for statically predicting the possible interaction patterns of services within a given choreography. We focus on choreographies exploiting the event notification paradigm to manage service interactions. Control Flow Analysis techniques statically approximate which events can be delivered to match the choreography constraints and how the multicast groups can be optimised to handle event notification within the service choreography.

1 Introduction

The ability of supporting programmable coordination policies of heterogeneous services is a key element in the success of the *Service Oriented Computing* (SOC) paradigm. Two different approaches are usually adopted to assemble services: *orchestration* and *choreography.* In the service orchestration, an intermediate entity, the orchestrator, arranges service activities according to the given business process. The service choreography, instead, involves all parties and their associated interactions providing a global view of the system. Relevant standard technologies are the Business Process Execution Language (BPEL) [23], for the orchestration, and Web Service Choreography Description Language (WS-CDL) [24], for the choreography. Notably, the orchestration-choreography issues have led to the development of a variety of foundational models (see e.g. [19,12,2,7,18,10] to cite only a few). We refer to the surveys in [9,22] for an analysis of the approaches.

In [15,21] a middleware, called Java Signal Core Layer (JSCL), supporting the design and implementation of service coordination policies has been introduced. The middleware consists of a set of API for assembling services by exploiting the *event notification* paradigm. A distinguished feature of JSCL consists of the strict interplay among formal semantic foundations, implementation pragmatics and experimental evaluation of the resulting programming mechanisms. More precisely, the programming facilities available in JSCL have been semantically motivated. At the abstract level, the middleware takes the form of the Signal Calculus (SC) [17]. The SC calculus is an asynchronous process calculus with explicit primitives to deal with (multicast) event notification and service

[*] Research supported by the EU within the FET-GC II Integrated Project IST-2005-016004 SENSORIA and by the Italian PRIN Project “SOFT”.

C. Laneve and J. Su (Eds.): WS-FM 2009, LNCS 6194, pp. 29–45, 2010.

distribution. The SC-JSCL framework allows one to specify and program service coordination policies (orchestration and choreography) relying on multicast notification only. Moreover, it features sessions as a mechanism to synchronise behaviours of distributed and independent services. Remarkably, the middleware does not assume any centralized mechanism for publishing, subscribing and notifying events. Hence, SC and JSCL have to be properly regarded as a foundational framework and its programming counterpart for specifying, verifying and programming coordination policies of distributed services.

The JSCL framework has also been equipped with a *model driven* development methodology [13,14,16]. The methodology exploits a suitable choreography model that takes the form of a process calculus, called Network Coordination Policies (NCP). The two calculi (SC and NCP) lay at two different levels of abstraction. The former is tailored to support the (formal) design of services, the latter is the specification language to declare the coordination policies. Policies are processes that specify service behaviour as seen by an observer standing from a global point of view, hence capable of observing the interactions that are expected to happen, and how these are interleaved. Indeed, certain features can be described at both levels: the NCP specification declares what is expected from the service network infrastructure, while the SC design specifies how to implement it. The gap between the local and global abstraction levels has been formally filled in [13,17]. It has been proved that for each SC design, there exists an NCP choreography that reflects all the properties of the design. The conformance of an SC design with respect to an NCP specification is formally proved by checking weak asynchronous bisimilarity [1] between them. This notion of conformance has the main benefit of supporting the development of systems in a model driven development fashion. The designer can define a suitable chain of SC models that implement the choreography: each model is obtained by refinement steps that add more details. The conformance of each model with respect to the NCP specification provides the formal machinery to choose the required level of abstraction, so that one can focus on coordination of services, without considering the implementation details, or focus on service design, just trying to match the abstract policies requirements.

The present paper aims at contributing to this line of research. Our long-term goal is to equip the JCSL middleware with semantic-based toolkits supporting its design, development, and deployment. In particular, this paper develops *static* reasoning techniques for the JSCL middleware. We use a specific static technique, *Control Flow Analysis*, based on *Flow Logic* [20]. This kind of static analysis provides a variety of automatic and decidable methods and tools for analysing properties of computing system.

Our first contribution is the definition of a Control Flow Analysis for the SC process calculus, that it is shown to be sound. For simplicity, the analysis is introduced in two stages: first it is developed for a basic fragment of the calculus considering flows and multicast. In the second stage, session management is taken into account. This analysis *safely* approximates the behaviour of an SC design, statically predicting the possible structure of event notifications. This information offers a basis for studying dynamic properties, by suitably handling the approximation of the static analysis constructs. We have indeed an over-approximation of the *exact* behaviour of a system. This means that all those interactions that the analysis *does not* include will *never* take place, while all the interactions that the analysis *does* include *can* happen, i.e. they are only possible. Therefore, the result of the analysis can be used to predict at compile time all the *possible* event flows emanating from a certain service. Implicitly, this amounts to providing the maximal flow of an event notification and, consequently, an upper bound on the structure of the multicast group implementing the notification. Hence, the analysis provides formal basis to optimise the management of multicast groups of the JSCL run-time.

Our second contribution consists in the development of a Control Flow Analysis for the NCP calculus. The analysis, that computes a safe over-approximation of event interactions, can be used to verify whether certain choreography constraints are satisfied. We can assert that events of a certain type have not to be captured by a service and then we can statically verify, by inspecting the analysis results, that this assertion is violated or not. In other words, the analysis acts in a *descriptive* fashion: if no property violation is statically found then no violation of the property can occur at run-time. However, within the NCP choreography model, the analysis can also be exploited in a *prescriptive* fashion. Intuitively, the analysis can suggest how to instrument the SC design to avoid occurrences of a property violation. For instance, the constraints on event handling mentioned above can be satisfied, by instrumenting the multicast group with a filter discarding the events referring to the unauthorized event.

Our static machinery has been applied to several process calculi, amongst which π-calculus (e.g. [5]) and LySa [4] to establish security properties. In particular, the mixed descriptive/prescriptive approach offered by Control Flow Analysis has been introduced in [3] to deal with type flaws in crypto-protocols.

Plan of the Paper. In Section 2, we present the simplest version of the SC calculus focussing on multicast notification. In Section 3, we completely introduce the Control Flow Analysis for this version of the calculus. This analysis is extended in Section 4 to manage the SC notion of session. The NCP calculus and its Control Flow Analysis are described in Section 5. In Section 6, we show how consistency between a network of SC components and the global coordination policy expressed by NCP specifications is reflected by the correspondence between the analysis results. For lack of space, all the proofs are omitted, but are reported in the extended version of the paper [6].

2 The Calculus

The Signal Calculus (SC) [17], is a process calculus specifically designed to describe coordination of services distributed over a network. The calculus is based on the event notification paradigm. SC building blocks are called *components*, which interact by issuing/reacting to *events*. A component contains a behaviour, for instance, a “simple” service, interacting through an asynchronous signal passing mechanism. Each component stores information about the collection of components that must be notified whenever events are issued (*event flow*). When an event is raised by a component, several envelopes are generated to notify all components in the flow (*multicast notification*). Each envelope, also called *signal*, contains the event itself and the address of the target component. Each component owns a set of signal handlers associated to type event. Usually, in the event notification literature, the type of an event is called *topic*. Signal handlers, called *reactions*, are responsible for the management of the reception of an event notification. Indeed, the reception of a signal acts like a trigger that activates the execution of a new behaviour, described by the compatible reaction within the component.

The component *interface* is defined by its *reactions* and *flows*. The language primitives allow one to *dynamically* modify the component interfaces topology of the coordination, by adding new flows and reactions. Finally, components are structured to build a *network* of services. A network provides the facility to transport signals containing the events exchanged among components.

Let $\mathcal{A}$, ranged over by a, b, c..., be a finite set of components names, and $\mathcal{T}$, ranged over by $\tau_1, ..., \tau_k$, be a finite set of topics. We use $\tilde{a}$ to denote a set of names $a_1, ..., a_n$. A *component* is written as $a[B]^R_F$ and represents the service uniquely identified by the name a, i.e. its public address. Each component has internal behaviour B, reaction R and flow F.

The syntax of SC is presented in Fig. 1. A reaction R is a multiset, possibly empty, of unit reactions. A unit reaction $\tau \gg B$ triggers the execution of the behaviour B upon reception of a signal tagged by the topic τ. A flow F is a set, possibly empty, of unit flows. A unit flow $\tau \rightsquigarrow \tilde{a}$ describes the set of component names $\tilde{a}$ where raised events having τ as topic have to be delivered. We define $F \downarrow_\tau$ as the set of $\tilde{b}$ such that $\tau \rightsquigarrow \tilde{b}$ occurs in F.

A behaviour B is a multiset of simple behaviours. The reaction part of the component interface can be extended by the reaction update $\mathsf{rupd}(R); B$. Similarly, the flow update $\mathsf{fupd}(F); B$ extends the component flows. The asynchronous event

$N ::=$		*networks*
	$\mid \mathbf{0}$	empty network
	$\mid N \| N$	parallel composition
	$\mid a[B]^R_F$	component
	$\mid \langle\tau\rangle@a$	signal envelope

$B ::=$		*behaviour*
	$\mid 0$	empty behaviour
	$\mid \mathsf{rupd}(R); B$	reaction update
	$\mid \mathsf{fupd}(F); B$	flow update
	$\mid \mathsf{out}\langle\tau\rangle; B$	event emission
	$\mid \epsilon; B$	internal behaviuor
	$\mid B \mid B$	parallel composition

$R ::=$		*reactions*
	$\mid 0$	empty reaction
	$\mid \tau \gg B$	unit reaction
	$\mid R \mid R$	parallel composition

$F ::=$		*flows*
	$\mid 0$	empty flow
	$\mid \tau \rightsquigarrow \tilde{a}$	unit flow
	$\mid F \mid F$	parallel composition

Fig. 1. Syntax of SC, version 1

emission $\mathsf{out}\langle\tau\rangle; B$ first spawns into the network a set of envelopes containing the event, one for each component name declared in the flow having topic τ, and then activates B. The behaviour $\epsilon; B$ abstracts from the internal activities performed by the component (at the end of its execution, the component activates the continuation B). Finally, the inactive behaviour $\mathbf{0}$ and the parallel composition $B|B$ have the standard meanings. Reactions, flows and behaviours are defined up-to a structural congruence ($\equiv$). Indeed we assume that $(F, |, 0)$, $(R, |, 0)$ and $(B, |, 0)$ are commutative monoids, i.e. parallel composition is commutative, associative and 0 is the identity. Moreover, we have that $\tau \rightsquigarrow \tilde{a}|\tau \rightsquigarrow \tilde{b} \equiv \tau \rightsquigarrow \tilde{a} \cup \tilde{b}$. We omit the trailing occurrences of 0.

Networks (N) describe the distribution of components and carry signals exchanged among them. The signal envelope $\langle\tau\rangle@a$ describes a message containing the topic τ, whose target component is named a. The empty network $\mathbf{0}$ and the parallel composition have the standard meanings. In the following, we will use $\prod_{b_i \in \tilde{b}} \langle\tau\rangle@b_i$, with $\tilde{b}$ a finite set of component names, to represent the parallel composition of messages having topic τ.

The operational semantics is defined in the reduction style and states how components, at each step, communicate and update their interfaces. Reduction rules of SC are given in Fig. 2. Rule (SKIP) describes the execution of an internal action, i.e. an action that has no side effects on the system. Rule (RUPD) extends the component reactions with a further unit reaction (the parameter of the primitive). Rule (FUPD) extends the component flows with a unit flow. Rule (OUT) first takes the set of component names $\tilde{a}$ that are linked to the component for the topic τ and then spawns into the network an envelope for each component name in the set. Rule (IN) allows a signal envelope to react with the component whose name is specified inside the envelope. Note that signal emission rule (OUT) and signal receiving rule (IN) do not consume, respectively, the flow and the reaction of the component, i.e. flows and reactions are *persistent*. Finally, rules (STRUCT) and (PAR) are standard.

$$\text{(SKIP)} \quad \frac{}{a[\epsilon; B_1 \mid B_2]_F^R \to a[B_1 \mid B_2]_F^R}$$

$$\text{(RUPD)} \quad \frac{}{a[\mathsf{rupd}(R_1); B_1 \mid B_2]_F^R \to a[B_1 \mid B_2]_F^{R|R_1}}$$

$$\text{(FUPD)} \quad \frac{}{a[\mathsf{fupd}(F_1); B_1 \mid B_2]_F^R \to a[B_1 \mid B_2]_{F|F_1}^R}$$

$$\text{(OUT)} \quad \frac{F \downarrow_\tau = \tilde{b}}{a[\mathsf{out}\langle\tau\rangle; B_1 \mid B_2]_F^R \to a[B_1 \mid B_2]_F^R || \Pi_{b_i \in \tilde{b}} \langle\tau\rangle@b_i}$$

$$\text{(IN)} \quad \frac{}{\langle\tau\rangle@a || a[B_1]_F^{R|\tau > B_2} \to a[B_1|B_2]_F^{R|\tau > B_2}}$$

$$\text{(PAR)} \quad \frac{N \to N_1}{N || N_2 \to N_1 || N_2}$$

$$\text{(STRUCT)} \quad \frac{N \equiv N_1 \to N_2 \equiv N_3}{N \to N_3}$$

Fig. 2. Reduction Semantics of SC

Example 1. Multicast Notification. Let us consider a component s that requires a set of resources to provide a certain functionality. This component is exploited by several clients c_i, with $i = 1, .., n$, to achieve a common goal. All clients collaborate to the activation of the service supplied by s, providing the required resources. The process is summarized as follows:

$$\begin{array}{lcl} N & \overset{def}{=} & s[out\langle\tau_r\rangle]^{\tau_0 \gtrdot \mathsf{out}\langle\tau_r\rangle|\tau_0 \gtrdot B}_{t_r \rightsquigarrow \{c_1,c_2,c_3\}} ||C_1||C_2||C_3 \\ C_i & \overset{def}{=} & c_i[0]^{t_r \gtrdot \mathsf{out}\langle\tau_0\rangle|\tau_r \gtrdot 0}_{\tau_0 \rightsquigarrow \{s\}} \end{array}$$

Initially there is a bid phase, in which the service S issues an event to notify its demand of resources.

$$s\,[\mathtt{out}\langle\tau_r\rangle]^{\tau_o \gtrdot \mathtt{out}\langle\tau_r\rangle \;|\; \tau_o \gtrdot B}_{\tau_r \rightsquigarrow \{c_1,c_2,c_3\}} \rightarrow s\,[0]^{\tau_o \gtrdot \mathtt{out}\langle\tau_r\rangle \;|\; \tau_o \gtrdot B}_{\tau_r \rightsquigarrow \{c_1,c_2,c_3\}} \parallel \langle\tau_r\rangle@c_1 \parallel \langle\tau_r\rangle@c_2 \parallel \langle\tau_r\rangle@c_3$$

Upon the reception of a resource request, a client non-deterministically activates one of its two reactions: it can ignore the service demand, or as shown below,

$$c_i\,[0]^{\tau_r \gtrdot \mathtt{out}\langle\tau_o\rangle \;|\; \tau_r \gtrdot 0}_{\tau_o \rightsquigarrow \{s\}} \parallel \langle\tau_r\rangle@c_i \rightarrow c_i\,[\mathtt{out}\langle\tau_o\rangle]^{\tau_r \gtrdot \mathtt{out}\langle\tau_o\rangle \;|\; \tau_r \gtrdot 0}_{\tau_o \rightsquigarrow \{s\}}$$

the client raises events τ_o to notify their agreement to provide a resource.

$$c_i\,[\mathtt{out}\langle\tau_o\rangle]^{\tau_r \gtrdot \mathtt{out}\langle\tau_o\rangle \;|\; \tau_r \gtrdot 0}_{\tau_o \rightsquigarrow \{s\}} \rightarrow c_i\,[0]^{\tau_r \gtrdot \mathtt{out}\langle\tau_o\rangle \;|\; \tau_r \gtrdot 0}_{\tau_o \rightsquigarrow \{s\}} \parallel \langle\tau_o\rangle@s$$

Upon the reception of a resource bid, the service non-deterministically activates one of its two reactions. If no client responds to the service demand, the bid fails and the functionality is not provided; otherwise, if it receives a sufficient amount of resources the bid phase terminates, the functionality can be provided, as shown below.

$$s\,[0]^{\tau_o \gtrdot \mathtt{out}\langle\tau_r\rangle \;|\; \tau_o \gtrdot B}_{\tau_r \rightsquigarrow \{c_1,c_2,c_3\}} \parallel \langle\tau_o\rangle@s \rightarrow s\,[B]^{\tau_o \gtrdot \mathtt{out}\langle\tau_r\rangle \;|\; \tau_o \gtrdot B}_{\tau_r \rightsquigarrow \{c_1,c_2,c_3\}}$$

3 The Control Flow Analysis for SC

We now introduce the Control Flow Analysis for SC. The aim of the analysis is to over-approximate all the possible behaviour of SC processes. In particular, we focus on how components communicate and update their interface. The result of analysing a network N is a tuple $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$, called *estimate* for N, that satisfies the judgements defined by the axioms and rules in the upper (lower, resp.) part of Table 1. Given a certain component a, $\mathcal{B}(a)$ gives an approximation of the possible behaviours of a; $\mathcal{R}(a)$ gives an approximation of the possible reactions of a; $\mathcal{F}(a)$ gives an approximation of the possible flows of a: and $\mathcal{E}(a)$ gives an approximation of the possible envelopes to be received by a.

To *validate* the correctness of a proposed estimate $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$ we state a set of clauses operating upon judgements for analysing processes $\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models N$, defined in the flavour of Flow Logic [20].

Validation. The analysis is specified in two phases. First, we check that the estimate $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$ describes the initial process. This is done in the upper part of Table 1, where the clauses amount to a structural traversal of process syntax. The clauses rely on the auxiliary functions $\mathbf{A}_B$, $\mathbf{A}_R$, $\mathbf{A}_F$, that given a behaviuor B, reaction R or flow F, keep track of the single unit behaviour occurring in B, reaction actions in R and flows in F, respectively. Their definitions are reported at the beginning of Table 1. In the second phase, we check that $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$ also takes into account the possible dynamics of the process under analysis. This is expressed by the closure conditions in the lower part of Table 1 that mimic the semantics, by modelling, without exceeding the precision boundaries of the analysis, the semantic preconditions and the consequences of the possible actions. More precisely, preconditions check, in terms of $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$, for the possible presence of the redexes necessary for actions to be performed. The conclusion imposes the additional requirements on $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$, necessary to give a valid prediction of the analysed action. For instance, in the penultimate clause in Table 1, if (i) there exists an occurrence of $\mathsf{out}\langle\tau\rangle$ in $\mathcal{B}(a)$, and (ii) there exists an occurrence of (τ, b) in $\mathcal{F}(a)$, then there is a signal envelope with topic τ to be received by b, i.e. a possible `out` action is predicted.

Table 1. Analysis for SC Processes

$\mathbf{A}_B(0) = \emptyset$	
$\mathbf{A}_B(\epsilon; B) = \mathbf{A}_B(B)$	
$\mathbf{A}_B(b; B) = \{b\} \cup \mathbf{A}_B(B)$ where $b ::= \mathsf{fupd}(F) \vert \mathsf{rupd}(R) \vert \mathsf{out}\langle\tau\rangle$	
$\mathbf{A}_B(B_0 \vert B_1) = \mathbf{A}_B(B_0) \cup \mathbf{A}_B(B_1)$	
$\mathbf{A}_R(0) = \emptyset$	
$\mathbf{A}_R(\tau \gtrdot B) = \{(\tau, B)\}$	
$\mathbf{A}_R(R_0 \vert R_1) = \mathbf{A}_R(R_0) \cup \mathbf{A}_R(R_1)$	
$\mathbf{A}_F(0) = \emptyset$	
$\mathbf{A}_F(\tau \rightsquigarrow \tilde{a}) = \{(\tau, a_i) \vert a_i \in \tilde{a}\}$	
$\mathbf{A}_F(F_0 \vert F_1) = \mathbf{A}_F(F_0) \cup \mathbf{A}_F(F_1)$	
$\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models \mathbf{0}$	iff *true*
$\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models N_0 \vert N_1$	iff $\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models N_0 \ \wedge \ \mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models N_1$
$\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models \langle\tau\rangle@a$	iff $\tau \in \mathcal{E}(a)$
$\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models a[B]_F^R$	iff $\mathbf{A}_B(B) \subseteq \mathcal{B}(a) \wedge \mathbf{A}_R(R) \subseteq \mathcal{R}(a) \wedge \mathbf{A}_F(F) \subseteq \mathcal{F}(a)$
$\mathsf{fupd}(F) \in \mathcal{B}(a) \Rightarrow \mathbf{A}_F(F) \subseteq \mathcal{F}(a)$	
$\mathsf{rupd}(R) \in \mathcal{B}(a) \Rightarrow \mathbf{A}_R(R) \subseteq \mathcal{R}(a)$	
$\mathsf{out}\langle\tau\rangle \in \mathcal{B}(a) \wedge (\tau, b) \in \mathcal{F}(a) \Rightarrow \tau \in \mathcal{E}(b)$	
$\tau \in \mathcal{E}(a) \wedge (\tau, B) \in \mathcal{R}(a) \Rightarrow \mathbf{A}_B(B) \subseteq \mathcal{B}(a)$	

Example 2 (Multicast notification). Back to our example, we report the main entries of the analysis in Table 2. It is possible to check that $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$ is a valid estimate, by following the two stages explained above. The analysis correctly approximates the behaviour of N; for instance it predicts that three envelopes $\langle\tau_r\rangle@c_i$ can be spawn (as proved by the fact that $\mathcal{E}(c_i) \ni \tau_r$ for $i = 1, 2, 3$).

Table 2. Some Analysis Entries of the Multicast Notification Example

$\mathcal{E}(c_i) \ni \tau_r$	$\mathcal{E}(s) \ni \tau_0$
$\mathcal{B}(s) \ni \mathsf{out}\langle\tau_r\rangle, \mathcal{A}_B(B)$	$\mathcal{B}(c_i) \ni \mathsf{out}\langle\tau_0\rangle$
$\mathcal{R}(s) \ni (\tau_0, \mathsf{out}\langle\tau_r\rangle), (\tau_0, B)$	$\mathcal{R}(c_i) \ni (\tau_r, \mathsf{out}\langle\tau_0\rangle), (\tau_r, 0)$
$\mathcal{F}(s) \supseteq \{(\tau_r, c_i) \mid c_i \in \{c_1, c_2, c_3\}\}$	$\mathcal{F}(c_i) \ni (\tau_0, \{s\})$

We prove that our analysis is safe with respect to the given semantics, i.e. a valid estimate enjoys the following subject reduction property.

Theorem 1. ***(Subject Reduction)***
If $N \to N'$ and $\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models N$ then also $\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models N'$.

Proof Sketch. The proof is by induction on $N \to N'$.

The above result can be made more precise, by looking at the single analysis components. As an example, we just show that the analysis component $\mathcal{F}$ captures all the flows that involve the components of a network N. Clearly, similar results hold for the other components of the analysis.

Theorem 2. ***(Flows $\mathcal{F}$)*** *If $\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models N$ and $N \to^* N' \to N''$, such that the last transition $N' \to N''$ is derived using the rule (FUPD) on the set F in a component* a*, then $\mathbf{A}_F(F) \subseteq \mathcal{F}(a)$.*

Proof Sketch. By Theorem 1, we have that $\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models N'$. Therefore, the proof proceeds by induction on the transition rules used to derive $N' \to N''$.

Our Control Flow Analysis approximates the behaviour of the network under consideration. It provides a *safe over-approximation* of the *exact* behaviour of services: at least all the valid behaviours are captured. More precisely, all those interactions that the analysis does not consider as possible will *never* occur. On the other hand, the interactions deemed as possible may, or may not, occur in the actual dynamic evolution of the network. Therefore, by exploiting the analysis's soundness, we can prove several properties. As an example, we discuss a property related to the flow of a certain service. First, we introduce some auxiliary notions. Given a network N, the set of networks reachable from N is defined as $\mathit{Reach}(N) = \{N' | N \to^* N'\}$. Let the flows emanating from a in N be defined as $F(N)(a) = \{F | a[B]^R_F \text{ occurs in } N\}$. The analysis component $\mathcal{F}$ can be used to predict, at compile time, all the *possible* flows emanating from a certain component in a network at run time, as stated by the following result.

Theorem 3. *Given a network N, including a component* a*, and an estimate $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$ such that $\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models N$, we have that $\{F(N')(a) | N' \in \mathit{Reach}(N)\} \subseteq \mathcal{F}(a)$.*

Proof Sketch. Immediate by Theorems 1 and 2.

From this static result, we can infer the maximal possible dimension that a flow emanating from a certain component in a network can reach at run time, just by computing the cardinality of the set $\mathcal{F}(a)$.

Note that similar static machineries can be exploited in the back-end of JCSL compiler, to optimise the code and the structure of the network interface.

4 Managing Session: A New Version of SC and a New Version of the Analysis

In the first version of SC, information associated to signals is not structured and topics cannot be created dynamically. Furthermore, the notion of session is missing: components cannot keep track of concurrent event notifications. A refined version of SC, whose syntax is presented in Fig. 3, tackles sessions management.

$N ::=$	*networks*
$\mid \mathbf{0}$	empty network
$\mid N \vert\vert N$	parallel composition
$\mid a[B]^R_F$	component
$\mid \langle \tau_{\copyright}\tau' \rangle @a$	signal envelope

$R ::=$	*reactions*
$\mid 0$	empty reaction
$\mid R \vert R$	parallel composition
$\mid \tau_{\copyright}\tau' \gtrdot B$	check reaction
$\mid \tau\lambda\tau' \gtrdot B$	lambda reaction

$B ::=$	*behaviour*
$\mid 0$	empty behaviour
$\mid \mathsf{rupd}(R); B$	reaction update
$\mid \mathsf{fupd}(F); B$	flow update
$\mid \mathsf{out}\langle \tau_{\copyright}\tau' \rangle; B$	event emission
$\mid \epsilon; B$	internal behaviuor
$\mid B \vert B$	parallel composition

$F ::=$	*flows*
$\mid 0$	empty flow
$\mid \tau \rightsquigarrow \tilde{a}$	unit flow
$\mid F \vert F$	parallel composition

Fig. 3. Syntax of SC, version 2

Events are pairs including a topic and a session identifier. The syntax of behaviors is modified by the signal emission primitive ($\mathtt{out}\langle \tau_{\copyright}\tau' \rangle$). Note that both topics and sessions are names and are freely interchangeable. As far as the reactive part is concerned, a *lambda reaction* $\tau\lambda\tau' \gtrdot B$ handles all signals with topic τ, regardless of their session. In the behaviour B, τ' is bound by the lambda reaction. A *check reaction* $\tau_{\copyright}\tau' \gtrdot B$ can instead handle only signals having the topic τ issued for the session τ' and does not declare bound names. The syntax of flows has been not changed. The *envelope* $\langle \tau_{\copyright}\tau' \rangle @a$ now carries both the topic τ and the session identifier τ'. For the sake of simplicity, we skip the restriction construct.

In Fig. 4, we only give the rules that are different from the ones in Fig. 2 and the new ones. Similarly, in Table 3, we just give the CFA rules that are different from the ones in Table 1. The subject reduction result stated on the previous version of SC can be easily extended to the present version.

$$\text{(OUT)} \quad \frac{F \downarrow_{\tau} = \tilde{b}}{a[\mathsf{out}\langle \tau_{\copyright}\tau' \rangle; B_1 \mid B_2]^R_F \rightarrow a[B_1 \mid B_2]^R_F \vert\vert \mathit{\Pi}_{b_i \in \tilde{b}} \langle \tau_{\copyright}\tau' \rangle @b_i}$$

$$\text{(CHECK)} \quad \frac{}{\langle \tau_{\copyright}\tau' \rangle @a \mid\mid a[B_1]^{R \mid \tau_{\copyright}\tau' \gtrdot B_2}_F \rightarrow a[B_1 \vert B_2]^R_F}$$

$$\text{(LAMBDA)} \quad \frac{}{\langle \tau_{\copyright}\tau' \rangle @a \mid\mid a[B_1]^{R \mid \tau\lambda\tau'' \gtrdot B_2}_F \rightarrow a[B_1 \vert \{\tau'/\tau''\} B_2]^{R \mid \tau\lambda\tau'' \gtrdot B_2}_F}$$

Fig. 4. Reduction Semantics of SC

Table 3. Analysis for SC Processes, version 2

$\mathbf{A}_B(b; B) = \{b\} \cup \mathbf{A}_B(B)$ where $b ::= \mathsf{fupd}(F) \mid \mathsf{rupd}(R) \mid \mathsf{out}\langle \tau_{\copyright}\tau' \rangle$ $\mathbf{A}_B((\nu\tau)B) = \mathbf{A}_B(B)$ $\mathbf{A}_R(\tau_{\copyright}\tau' \gtrdot B) = \{(\tau_{\copyright}\tau', B)\}$ $\mathbf{A}_R(\tau\lambda\tau' \gtrdot B) = \{(\tau\lambda\tau', B)\}$
$\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models \langle \tau_{\copyright}\tau' \rangle @a$ iff $\tau_{\copyright}\tau' \in \mathcal{E}(a)$
$\mathsf{out}\langle \tau_{\copyright}\tau' \rangle \in \mathcal{B}(a) \wedge (\tau, b) \in \mathcal{F}(a) \Rightarrow \tau_{\copyright}\tau' \in \mathcal{E}(b)$ $\tau_{\copyright}\tau' \in \mathcal{E}(a) \wedge (\tau_{\copyright}\tau', B) \in \mathcal{R}(a) \Rightarrow \mathbf{A}_B(B) \subseteq \mathcal{B}(a)$ $\tau_{\copyright}\tau' \in \mathcal{E}(a) \wedge (\tau\lambda\tau'', B) \in \mathcal{R}(a) \Rightarrow \mathbf{A}_B(\{\tau'/\tau''\}B) \subseteq \mathcal{B}(a)$

5 The Network Coordination Policies Calculus and Its Analysis

The Calculus. We now conclude the presentation of JCSL framework, by introducing the choreography model. This takes the form of an asynchronous calculus, called Network Coordination Policies (NCP) [17]. Intuitively, SC is used to support the design of services, while NCP is the specification language used to declare the coordination policies. Policies are processes that represent the behavior as observed from a *global* point of view, i.e. by observing all the public interactions on the network infrastructure. Hence, an NCP process describes the interactions that are expected to happen and how these are interleaved. The NCP specification declares *what* is expected from the service network infrastructure, whereas the SC design specifies *how* to implement it. NCP adheres to the multicast notification mechanism of SC, however, while SC exploits the notion of flows, NCP manages this information by a global point of view, introducing the notion of *network topologies*. In other words, a network topology represents the flows of all components involved by the coordination.

A NCP specification consists of two entities: a policy and a network topology. The former describes the actions that should be performed by components, while the latter describes the component inter-connection. A *network topology* is a structure $G = (V, E)$, where $V \subseteq A$ consists of the restricted component names of the network and $E \subseteq A \times \mathcal{T} \times A$ are the flow connections among components: $(a, \tau, b) \in E$ represents a flow form a towards b for signal of topic τ. Note that G induces a directed labelled graph, called *topic-graph.* We will use the following auxiliary notations: (i) the *flows emanating from a in G*, $G(a) = \{(\tau, b) | (a, \tau, b) \in E\}$; (ii) the *topic-graph of τ in G*, $G(\tau) = \{(a, \tau', b) \in E | \tau' = \tau\}$; (iii) the *flow projection of τ for a in G*, $G(\tau, a) = \{b | (\tau, b) \in G(a)\}$.

The syntax of NCP is presented in Fig. 5. For the sake of simplicity, we consider the restriction-free fragment of NCP. As a consequence in the semantics, we will skip the rules (OPEN), (CLOSE) and (NEW). Let G be an NCP topology and P an NCP policy, then the pair $\langle G; P \rangle$ is called *NPC state.* NPC states represent the specifications of a system.

$$\begin{array}{lll}
P ::= & & \textit{coordination policies} \\
 & \mid \sum_{i\in I} p_i@a_i.P_i & \text{non-det. guarded choice} \\
 & \mid \overline{\tau}\tau'@a.P & \text{policy} \\
 & \mid \langle \tau \copyright \tau' \rangle @a & \text{signal envelope} \\
 & \mid \mathsf{fupd}(F)@a.P & \text{flow update} \\
 & \mid \iota.P & \text{internal activity} \\
 & \mid P \| P & \text{parallel composition}
\end{array}
\qquad
\begin{array}{lll}
p ::= & & \\
 & \mid \tau(\tau') & \text{lambda input} \\
 & \mid \tau\tau' & \text{check input}
\end{array}$$

Fig. 5. Syntax of NCP

$$\begin{array}{lll}
\alpha ::= & & \textit{actions} \\
 & \mid \epsilon & \text{silent action} \\
 & \mid \tau\tau'@a & \text{free reaction activation} \\
 & \mid (\tau\tau'@a) & \text{message reception} \\
 & \mid \langle \tau \copyright \tau' \rangle @a & \text{bound event notification}
\end{array}$$

Fig. 6. NCP actions

An NCP process is called a *coordination policy.* Non-deterministic (guarded) choice is denoted as $\sum_{i\in I} p_i@a_i.P_i$; a policy $p@a.P$ represents an action p executed by the component a with continuation P; prefix $\tau(\tau')$ allows to receive on τ and is called *lambda input* since it corresponds to the SC lambda reaction; $\tau\tau'$ allows to receive signals having topic τ and session τ' and is therefore called *check input.* Since a lambda input can handle events regardless their sessions, the name τ' represents a binder for the received session identifier. The policy $\overline{\tau}\tau'$ raises an event on session τ' with topic τ. The component delivers the corresponding notifications to all services that are subscribed on the topic τ. The *envelope* $\langle \tau \copyright \tau' \rangle @a$ represents a pending message/notification on the network towards a. Notice that only the target of the envelope is declared. Also in NCP, the emission of an event and its reception are performed in two phases. Initially, the emitter spawns into the network the proper envelopes, according with the actual network topology. Subsequently, a subscriber can react to the received envelope. The policy $\mathsf{fupd}(F)$ adds F to the flows departing from a. Prefix $\iota.P$ represents the execution of an internal activity before the execution of P. Finally, coordination policies can be composed in parallel.

The operational semantics of NCP is specified by the labelled transition system (LTS), reported in Fig. 7. Labels α are defined in Fig. 6.

Rule (SKIP) trivially fires the silent action. Rule (FUPD) changes the network topology, by appending the sub-network $a \boxtimes F$ to the environment G, i.e. all the flows departing from a in F. Rule (EMIT) allows for multicast communications: it spawns in the network an envelope for each subscriber in $G(\tau)(a)$. Note that the continuation policy P is executed regardless the reception of envelopes as typical in asynchronous communications. Notification of envelopes is ruled by (NOTIFY) as much like as the output in the asynchronous π-calculus. Rules (LAMBDA) and (CHECK) model input actions. In the former, the selected input p_j reads any signal with topic τ and binds τ_1 to τ_1' in an early-style

$$\text{(SKIP)} \quad \langle G; \iota.P\rangle \xrightarrow{\epsilon} \langle G; P\rangle$$

$$\text{(FUPD)} \quad \langle G; \mathsf{fupd}(F)@a.P\rangle \xrightarrow{\epsilon} \langle G \uplus (a \boxtimes F); P\rangle \text{ where } a \boxtimes F = \{(a,\tau,b) | (\tau,b) \in F\}$$

$$\text{(EMIT)} \quad \langle G; \overline{\tau}\tau'@a.P\rangle \xrightarrow{\epsilon} \langle G; P || \Pi_{b \in G(\tau,a)} \langle \tau \copyright \tau' \rangle @b\rangle$$

$$\text{(NOTIFY)} \quad \langle G; \langle \tau \copyright \tau' \rangle @a\rangle \xrightarrow{\tau \copyright \tau' @a} \langle G; \mathbf{0}\rangle$$

$$\text{(LAMBDA)} \quad \frac{j \in I \quad p_j = \tau(\tau_1)}{\langle G; \sum_{i \in I} p_i@a_i.P_i\rangle \xrightarrow{\tau\tau_1'@a} \langle G \uplus \tau_1' \boxdot T; \{\tau_1'/\tau_1\}P_j || p_j@a_j.P_j\rangle}$$
$$\text{where } \tau_1' \boxdot T = \{(a,\tau,b) | (a,b) \in T\}$$

$$\text{(CHECK)} \quad \frac{j \in I \quad p_j = \tau\tau'}{\langle G; \sum_{i \in I} p_i@a_i.P_i\rangle \xrightarrow{p_j@a_j} \langle G; P_j\rangle}$$

$$\text{(ASYNCH)} \quad \frac{}{\langle G; P\rangle \xrightarrow{(\tau\tau'@a)} \langle G; P || \langle \tau \copyright \tau' \rangle @a\rangle}$$

$$\text{(COM)} \quad \frac{\langle G; P_0\rangle \xrightarrow{\tau\tau'@a} \langle G; P_0'\rangle \quad \langle G; P_1\rangle \xrightarrow{\langle \tau \copyright \tau' \rangle @a} \langle G; P_1'\rangle}{\langle G; P_0 || P_1\rangle \xrightarrow{\epsilon} \langle G; P_0' || P_1'\rangle}$$

$$\text{(PAR)} \quad \frac{\langle G; P_0\rangle \xrightarrow{\alpha} \langle G'; P_0'\rangle}{\langle G; P_0 || P_1\rangle \xrightarrow{\alpha} \langle G'; P_0' || P_1\rangle}$$

Fig. 7. NPC LTS

semantics. When a check input is selected, only envelopes of topic τ in session τ_1 can be consumed. Notice that the reception by a check reaction of a topic does not change the network topology, because the two topics involved by the communication are already known. The reception of a fresh name (τ_1') by a lambda reaction, instead, can extend the environment knowledge of the component: the receiver can discover all the existing linkages involving the received name τ_1'. In the spirit of early-style semantics, we allow the rule to extend the topology with any possible graph (T). Differently from SC, these two rules can express external non-deterministic choice and can involve several components. Rule (ASYNCH) permits to any NCP state to perform an input, simply storing the received message for subsequent usages, allowing to arbitrarily delay the communication. Rule (COM) allows the communication of a free session name τ'. Finally, rule (PAR) has the standard meaning.

The Control Flow Logic for NCP. We develop a Control Flow Analysis for NCP, with the aim of over-approximating all the possible behaviour of NCP processes. The analysis, still specified in two phases, is reported in Table 4, where $\mathsf{sbj}(P)$ collects all the component names included in P. To emphasise the relation between the two calculi, we overload the analysis component names $\mathcal{B}$ and $\mathcal{E}$ and we use the judgement $\mathcal{B}, \mathcal{E}, G_S \models \langle G; P\rangle$ (and, in turn, $\mathcal{B}, \mathcal{E}, G_S \models P$), that we make more precise, i.e. $\mathcal{B}_{NCP}, \mathcal{E}_{NCP}, G_S \models \langle G; P\rangle$, when needed. There, G_S stands for the static abstraction of the graph of topics. It includes the initial graph and all the possible arcs and vertices that can be added during the computation. The clauses rely on the auxiliary function $\mathbf{A}_P$, that given a process

Table 4. Analysis for NCP

$$
\begin{array}{l}
\mathbf{A}_P(\sum_{i\in I} p_i@a_i.P_i) = \bigcup_{i\in I}\mathbf{A}_P(p_i@a_i.P_i)\\
\mathbf{A}_P(p@a.P) = \{((p,P),a)\}\\
\mathbf{A}_P(\overline{\tau}\tau'@a.P) = \{(\overline{\tau}\tau',a)\} \cup \mathbf{A}_P(P)\\
\mathbf{A}_P(\langle\tau\copyright\tau'\rangle@a) = \emptyset\\
\mathbf{A}_P(\mathsf{fupd}(F)@a.P) = \{(\mathsf{fupd}(F),a)\} \cup \mathbf{A}_P(P)\\
\mathbf{A}_P(P_0|P_1) = \mathbf{A}_P(P_0) \cup \mathbf{A}_P(P_1)\\
\mathbf{A}_P(\iota.P) = \mathbf{A}_P(P)\\
\mathbf{A}_P(P)(a) = \{el|(el,a) \in \mathbf{A}_P(P)\}\\
\mathbf{E}_P(P) = \begin{cases}\{(\tau\copyright\tau',a)\} & \text{if } P = \langle\tau\copyright\tau'\rangle@a\\ \emptyset & \text{otherwise}\end{cases}\\
\mathbf{E}_P(P)(a) = \{el|(el,a) \in \mathbf{E}_P(P)\}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{B},\mathcal{E},G_S \models \langle G;P\rangle & \mathsf{iff}\ G \subseteq G_S \wedge \mathcal{B},\mathcal{E},G_S \models P\\
\mathcal{B},\mathcal{E},G_S \models P & \mathsf{iff}\ \forall a \in \mathsf{sbj}(P).\mathbf{A}_P(P)(a) \subseteq \mathcal{B}(a) \wedge \mathbf{E}_P(P)(a) \subseteq \mathcal{E}(a)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fupd}(F) \in \mathcal{B}(a) \Rightarrow \mathbf{A}_F(F) \subseteq G_S(a)\\
\overline{\tau}\tau' \in \mathcal{B}(a) \wedge (\tau,b) \in G_S(a) \Rightarrow \tau\copyright\tau' \in \mathcal{E}(b)\\
\tau\copyright\tau' \in \mathcal{E}(a) \wedge (\tau\tau',P) \in \mathcal{B}(a) \Rightarrow \mathcal{B},\mathcal{E},G_S \models P\\
\tau\copyright\tau' \in \mathcal{E}(a) \wedge (\tau(\tau''),P) \in \mathcal{B}(a) \Rightarrow G(\tau') \subseteq G_S \wedge \mathcal{B},\mathcal{E},G_S \models \{\tau'/\tau''\}P
\end{array}
$$

P, keeps track of the single actions in P, and whose definition is in the upper part of Table 4. Hereafter, we denote with el the generic element of a set. This analysis is correct with respect to the given semantics. Furthermore, we prove that G_S captures all the flows arising in the topology.

Theorem 4. ***(Subject Reduction)***
Let S a NPC state $\langle G;P\rangle$. If $S \xrightarrow{\alpha} S'$ and $\mathcal{B},\mathcal{E},G_S \models S$ then also $\mathcal{B},\mathcal{E},G_S \models S'$.

Proof Sketch. The proof is by induction on $S \xrightarrow{\alpha} S'$.

Theorem 5. ***(Flows $\mathcal{F}$)*** *If $\mathcal{B},\mathcal{E},G_S \models S$ and $S \rightarrow^* S' \xrightarrow{\alpha} S''$, such that the last transition $S' \xrightarrow{\alpha} S''$ is derived using the rule (FUPD) on the set F in a component* a, *then $\mathbf{A}_F(F) \subseteq G_S(a)$.*

Proof Sketch. By Theorem 4, we have that $\mathcal{B},\mathcal{R},\mathcal{F},\mathcal{E} \models S'$. Therefore, the proof proceeds by induction on the transition rules used to derive $S' \xrightarrow{\alpha} S''$.

Note that the above theorem formally represents the projection of the choreography over a component. Namely, it provides the local view of the choreography policy.

The NCP control flow analysis can be used to verify whether certain choreography constraints are satisfied, for instance, on the security side. We can assert when a service does not capture a certain topic, and then statically verify, by inspecting the analysis results, whether this assertion is not violated.

Given a policy P and a graph G, let the set of systems reachable from $\langle G; P\rangle$ be defined as $Reach(\langle G; P\rangle) = \{\langle G'; P'\rangle | \langle G; P\rangle \rightarrow^* \langle G'; P'\rangle\}$. Let the flows emanating from a in $\langle G; P\rangle$ be defined as $F_{topic}(\langle G; P\rangle)(a) = \{(\tau, b) | (a, \tau, b) \in G\}$ and $F_{topic}(\langle G; P\rangle)(a, \tau) = \{b | (a, \tau, b) \in G\}$.

A service a does not capture a certain topic τ, when the flow projection of τ for a is empty in the initial graph G and in every graph reachable from it.

Definition 1. *Given a process P, a graph G, a topic τ, and component* a *occurring in P, we say that a* does not capture τ *if $F_{topic}(\langle G'; P'\rangle)(a, \tau) = \emptyset$ for all $\langle G'; P'\rangle \in Reach(\langle G; P\rangle)$.*

Again, an analysis component, G_S, can be used to predict at compile time whether the constraint is respected. Actually, because of safety, we can assess that if the property is statically guaranteed, then it will also be at run time, as stated by the following result, whose proof is based on Theorem 5.

Theorem 6. *Given a process P, a graph G, a topic τ, and component* a *occurring in P, if $G_S(a) = \emptyset$ then* a does not capture τ.

Proof Sketch. The proof proceeds by contradiction, by assuming that *a does capture* τ.

Here, our analysis acts in a *descriptive* way, i.e. it describes if a property violation is possible and because of soundness, we can prove that if no violation is found, no violation can arise at run-time. In the same setting, our approach can have a *prescriptive* value. In this case, we aim at *preventing* violation to arise, by suggesting how to instrument the code with the necessary checks, e.g. by enriching the multicast group with a filter discarding the events referring to the unauthorized topic.

6 Checking Choreography

Consistency between network of SC components and the global coordination policies expressed by NCP specifications is formally verified in [17]. Verification is based on the encoding from SC networks to NCP policies, presented in Table 5, and on bisimilarity. This result can also suggest a model driven development approach. The designer can define successive SC models for implementing a choreography model, obtained by incremental refinement.

The basic idea of the encoding is to transform SC reductions into NCP transitions labeled with ϵ. The encoding uses the following functions: (i) $[\![B]\!]_a$ which takes an SC behaviour B, localised within a, and maps it into an NCP policy; (ii) $[\![R]\!]_a$ which takes a reaction R, installed in the interface of a, and maps it into a policy; and (iii) $[\![N]\!]$ which takes a network N and maps it into a state.

Control Flow Analysis provides us with an approximation of behaviours, both for the choreography model (NCP) and the actual design (SC). The consistency result is reflected by the correspondence between the analysis estimate $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$ of a network N and that $(\mathcal{B}_{NCP}, \mathcal{E}_{NCP}, G_S)$ of its encoding $[\![N]\!]$. We need the following auxiliary function that maps each element possibly occurring in $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$, in the corresponding element occurring in $(\mathcal{B}_{NCP}, \mathcal{E}_{NCP}, G_S)$.

Table 5. Encoding of behaviours, reactions and networks

$\llbracket 0\rrbracket_a = \mathbf{0}$	$\llbracket B\vert B'\rrbracket_a = \llbracket B\rrbracket_a \vert\vert \llbracket B'\rrbracket_a$
$\llbracket \epsilon; B\rrbracket_a = \iota.\llbracket B\rrbracket_a$	$\llbracket \mathsf{out}\langle \tau{\copyright}\tau'\rangle B\rrbracket_a = \overline{\tau}\tau' @a\llbracket B\rrbracket_a$
$\llbracket \mathsf{rupd}(R); B\rrbracket_a = \iota.\llbracket R\rrbracket_a \vert\vert \llbracket B\rrbracket_a$	$\llbracket \mathsf{fupd}(F); B\rrbracket_a = \mathsf{fupd}(F)@a.\llbracket B\rrbracket_a$
$\llbracket 0\rrbracket_a = \mathbf{0}$	$\llbracket R\vert R'\rrbracket_a = \llbracket R\rrbracket_a \vert\vert \llbracket R'\rrbracket_a$
$\llbracket \tau{\copyright}\tau' \gtrdot B\rrbracket_a = \tau\tau'@a.\llbracket B\rrbracket_a$	$\llbracket \tau\lambda\tau' \gtrdot B\rrbracket_a = \tau(\tau')@a\llbracket B\rrbracket_a$
$\llbracket \emptyset\rrbracket = \langle \mathbf{0}; \mathbf{0}\rangle$ $\llbracket \langle \tau{\copyright}\tau'\rangle @a\rrbracket = \langle \mathbf{0}; \langle \tau{\copyright}\tau'\rangle @a\rangle$	
$\dfrac{\llbracket N\rrbracket = \langle G; P\rangle \quad \llbracket N'\rrbracket = \langle G'; P'\rangle}{\llbracket N\vert\vert N'\rrbracket = \langle G \uplus G'; P\vert\vert P'\rangle}$	
$\left\llbracket a[B]^R_F\right\rrbracket = \langle G; \llbracket B\rrbracket_a \vert\vert \llbracket R\rrbracket_a\rangle$ where $G = a \boxtimes F$	

$$\begin{array}{ll}
Enc(\mathsf{out}\langle \tau{\copyright}\tau'\rangle) = \overline{\tau}\tau' & Enc(\mathsf{fupd}(F)) = \mathsf{fupd}(F) \\
Enc((\tau{\copyright}\tau', B)) = (\tau\tau', Enc(B)) & Enc((\tau\lambda\tau', B)) = (\tau(\tau'), Enc(B)) \\
Enc(\tau{\copyright}\tau') = \tau{\copyright}\tau' & Enc((\tau, b)) = (\tau, b)
\end{array}$$

Example 3. We illustrate this correspondence on the following example, given by a network N having two components: a and b.

$$N = a[0]^{\tau\lambda\tau' \gtrdot \mathsf{out}\langle \tau{\copyright}\tau'\rangle}_{\tau \rightsquigarrow \{b\}} \vert\vert\ b[0]^{\tau\lambda\tau' \gtrdot \mathsf{out}\langle \tau_1{\copyright}\tau'\rangle}_{\tau_1 \rightsquigarrow \tilde{c}} \vert\vert \langle \tau{\copyright}\tau''\rangle @a \vert\vert \langle \tau{\copyright}\tau'''\rangle @b$$

The corresponding encoding is given by the following state S:

$$S = \langle (\emptyset, \{(a, \tau, \{b\}), (b, \tau, \tilde{c})\}); \tau(\tau')@a.\overline{\tau}\tau'@a \vert\vert \tau(\tau')@b.\overline{\tau_1}\tau'@b\rangle \vert\vert \langle \tau{\copyright}\tau''\rangle @a \vert\vert \langle \tau{\copyright}\tau'''\rangle @b$$

The analyses of N and of S, reported in Table 6, show the correspondence between the estimates components.

Table 6. Some Entries of the Analysis of N (upper part) and of S (lower part)

$\mathcal{E}(a) \ni \tau{\copyright}\tau''$	$\mathcal{E}(b) \ni \tau{\copyright}\tau''', \tau{\copyright}\tau''$
$\mathcal{B}(a) \ni \mathsf{out}\langle \tau_1{\copyright}\tau''\rangle$	$\mathcal{B}(b) \ni \mathsf{out}\langle \tau_1{\copyright}\tau'''\rangle$
$\mathcal{R}(a) \ni (\tau\lambda\tau', \mathsf{out}\langle \tau{\copyright}\tau'\rangle)$	$\mathcal{R}(b) \ni (\tau\lambda\tau', \mathsf{out}\langle \tau_1{\copyright}\tau'\rangle)$
$\mathcal{F}(a) \supseteq \{(\tau, b)\}$	$\mathcal{F}(b) \supseteq \{(\tau_1, c_i) \vert c_i \in \tilde{c}\}$
$\{(a, \tau, \{b\}), (b, \tau, \tilde{c})\} \in G_S$	$G(\tau'') \cup G(\tau''') \in G_S$
$\mathcal{E}_{NCP}(a) \ni \tau{\copyright}\tau''$	$\mathcal{E}_{NCP}(b) \ni \tau{\copyright}\tau''', \tau{\copyright}\tau''$
$\mathcal{B}_{NCP}(a) \ni (\tau(\tau'), \overline{\tau}\tau'@a), \overline{\tau}\tau''$	$\mathcal{B}_{NCP}(b) \ni (\tau(\tau'), \overline{\tau_1}\tau'@b), \overline{\tau_1}\tau'''$

Now, we formally state the correspondence between the two analyses.

Theorem 7. *Given a network N and $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$, such that $\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E} \models N$, let $\langle G; P\rangle = \llbracket N\rrbracket$ and $(\mathcal{B}, \mathcal{E}, G_S)$ such that $\mathcal{B}, \mathcal{E}, G_S \models \langle G; P\rangle$. We have that for all* a *in the domain of $(\mathcal{B}, \mathcal{R}, \mathcal{F}, \mathcal{E})$, we have that:*

- $\forall el \in \mathcal{B}(a) : Enc(el) \in \mathcal{B}_{NCP}(a)$ • $\forall el \in \mathcal{R}(a) : Enc(el) \in \mathcal{B}_{NCP}(a)$
- $\forall el \in \mathcal{E}(a) : Enc(el) \in \mathcal{E}_{NCP}(a)$ • $\forall (\tau, b) \in \mathcal{F}(a) : Enc(\tau, b) \in G_S(a)$

Proof Sketch. The proof proceeds by structural induction.

The correspondence of the two analyses is made easier by our assumption on the absence of restriction and scope extrusion in NCP. As a consequence, the treatment of internal actions is strongly simplified. The more involved reasonings, needed to cope with the full calculus, require further investigation.

7 Concluding Remarks

We have introduced Control Flow Analysis for the SC-NCP framework for service coordination. Our approach is based on a two layer calculus (in the spirit of [11,12,8]). The abstract level (NCP) provides a declarative framework to specify the service coordination, while the concrete level (SC) allows us to design the behavior of services. The distinguished feature of our approach is given by the mixed descriptive-prescriptive mechanism, offered by the Control Flow Analysis and experimented to prove security properties of cryptographic protocols [3]. This provides us flexible facilities to manage a wide range of properties.

The SC-NCP programming model has provided the foundational basis to design and implement the JSCL middleware for services. The correspondence result, stated in Section 6, provides a further formal hook to freely move inside the two-level structure of JSCL. Depending on the level of the structure, one can focus on either the design or the choreography, with the guarantee that the key features are preserved. Differently from the dynamic mechanism of bisimulation, used in [17], static analysis predicts the possible interaction patterns of services in a given choreography, allowing for a sort of choreography rehearsal.

We plan to equip the JSCL framework to include the reasoning machineries available by implementing the analyses developed in the present paper. We intend to exploit the analysis to statically verify that a design is compliant with the specification of the choreography demands, and to instrument the code to avoid the occurrences of certain events at run-time.

References

1. Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-calculus. Theor. Comput. Sci. 195(2), 291–324 (1998)
2. Bartoletti, M., Degano, P., Ferrari, G., Zunino, R.: Secure service orchestration. In: Aldini, A., Gorrieri, R. (eds.) FOSAD 2007. LNCS, vol. 4677, pp. 24–74. Springer, Heidelberg (2007)
3. Bodei, C., Brodo, L., Degano, P., Gao, H.: Detecting and preventing type flaws at static time. Journal of Computer Security (to appear, 2009)
4. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Nielson, H.R.: Static validation of security protocols. Journal of Computer Security 13(3), 347–390 (2005)
5. Bodei, C., Degano, P., Nielson, F., Nielson, H.R.: Static analysis for the π-calculus with their application to security. Info. & Computat. 165, 68–92 (2001)
6. Bodei, C., Ferrari, G.L.: Choreography rehearsal. Technical Report TR-09-11, Dipartimento di Informatica, Univ. Pisa (2009)

7. Boreale, M., Bruni, R., Caires, L., Nicola, R.D., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V.T., Zavattaro, G.: SCC: A service centered calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
8. Bravetti, M., Zavattaro, G.: A foundational theory of contracts for multi-party service composition. Fundam. Inform. 89(4), 451–478 (2008)
9. Bruni, R.: Calculi for service oriented computing. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 1–41. Springer, Heidelberg (2009)
10. Bruni, R., Lanese, I., Melgratti, H.C., Tuosto, E.: Multiparty sessions in soc. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 67–82. Springer, Heidelberg (2008)
11. 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)
12. 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)
13. Ciancia, V., Ferrari, G.L., Guanciale, R., Strollo, D.: Checking correctness of transactional behaviors. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 134–148. Springer, Heidelberg (2008)
14. Ciancia, V., Ferrari, G.L., Guanciale, R., Strollo, D.: Global coordination policies for services. In: FACS 2008. Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam (2009) (to appear)
15. Ferrari, G.L., Guanciale, R., Strollo, D.: Jscl: A middleware for service coordination. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 46–60. Springer, Heidelberg (2006)
16. Ferrari, G.L., Guanciale, R., Strollo, D., Tuosto, E.: Refactoring long running transactions. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 127–142. Springer, Heidelberg (2009)
17. Guanciale, R.: The Signal Calculus: Beyond Message-based Coordination for Services. PhD thesis, Institute for Advanced Studies, IMT, Lucca (2009)
18. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: A calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 327–338. Springer, Heidelberg (2006)
19. 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)
20. Nielson, H.R., Nielson, F.: Flow logic: A multi-paradigmatic approach to static analysis. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 223–244. Springer, Heidelberg (2002)
21. Strollo, D.: Designing and Experimenting Coordination Primitives for Service Oriented Computing. PhD thesis, Institute for Advanced Studies, IMT, Lucca (2009)
22. Su, J., Bultan, T., Fu, X., Zhao, X.: Towards a theory of web service choreographies. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 1–16. Springer, Heidelberg (2008)
23. TC, O.: Business process execution language for web services version 2.0, `http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.html`
24. Web services choreography description language version 1, `http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/`

A Graph Syntax for Processes and Services*

Roberto Bruni, Fabio Gadducci, and Alberto Lluch Lafuente

Department of Computer Science, University of Pisa
{bruni,gadducci,lafuente}@di.unipi.it

Abstract. We propose a class of hierarchical graphs equipped with a simple algebraic syntax as a convenient way to describe configurations in languages with inherently hierarchical features such as sessions, fault-handling scopes or transactions. The graph syntax can be seen as an intermediate representation language, that facilitates the encoding of structured specifications and, in particular, of process calculi, since it provides primitives for nesting, name restriction and parallel composition. The syntax is based on an algebraic presentation that faithfully characterises families of hierarchical graphs, meaning that each term of the language uniquely identifies an equivalence class of graphs (modulo graph isomorphism). Proving soundness and completeness of an encoding (i.e. proving that structurally equivalent processes are mapped to isomorphic graphs) is then facilitated and can be done by structural induction. Summing up, the graph syntax facilitates the definition of faithful encodings, yet allowing a precise visual representation. We illustrate our work with an application to a workflow language and a service-oriented calculus.

1 Introduction

As witnessed by a large literature, graphs offer a convenient ground for the specification and analysis of modern software systems with features such as distribution, concurrency and mobility. Among the graph-based formalisms used for such purposes, we recall those based on traditional Graph Transformation [14], Bigraphical Reactive Systems [17] and Synchronized Hyperedge Replacement [13]. Building a graphical representation of an existing language involves two major challenges: encoding states and encoding the operational semantics. A correct state encoding should map structurally equivalent states into equivalent (typically isomorphic) graphs. In addition, the state encoding should also facilitate the encoding of the operational semantics, which typically means mimicking term rewrites via suitable graph rewrites.

The use of graph isomorphism as state equivalence has several advantages. Visually, it offers an intuitive normal form representation for system states, abstracting from the concrete identity of the single components. Operationally, it enables graph transformations, which have (sub)graph isomorphism at the base of the matching mechanism used for the application of rules, for simulating state evolution. Sometimes, though, capturing equivalence of configurations via graph

* Research supported by the EU FET integrated project Sensoria, IST-2005-016004.

C. Laneve and J. Su (Eds.): WS-FM 2009, LNCS 6194, pp. 46–60, 2010.

isomorphism it is not always possible or convenient, and additional axioms must be taken into account. However, also in such situations the reuse of standard graph transformation techniques may turn to be useful. For instance, one typical solution is to consider (equivalence classes of) normal forms for graphs and a set of confluent and terminating rewrite rules that reduce graphs into the normal form of their equivalence class.

The encoding of states of a process calculus is facilitated by their algebraic structure (since processes are terms) and it is typically defined inductively on such structure. However, the syntax of graph formalisms is often not provided with suitable features for names, name restrictions or hierarchical aspects. Hence, typical solutions consist in developing ad-hoc algebraic syntaxes that are often significantly different from those of standard process calculi. They require advanced skills and are based on sophisticated techniques involving the set-theoretic definition of graphs with interfaces (e.g. [14,15]), the use of enriched type systems (e.g. [8,16]) or the representation of hierarchies as trees (e.g. [17,15]), and they may result in layered specifications and complex correctness proofs.

Our goal is to develop a technique for simplifying the definition of state encodings into graphical structures and the proof of their correctness and such that the associated graph rewriting rules are automatically determined from the state encoding and the original operational semantics.

In a companion paper [3] we propose to fill the gap between the different levels of abstraction at which process calculi and graphical structures reside by introducing a specification formalism made of an algebra of hierarchical graphs, plus a sound and complete set of axioms equating two terms whenever they represent essentially the same hierarchical graph. The graph algebra is equipped with primitives and mechanisms for dealing with names, name restriction, parallel composition and, most importantly, nesting in the same way as they are used in process calculi. In particular, the nesting mechanism allows for easily defining graphical presentations for process calculi with inherently hierarchical aspects such as sessions, transactions or locations: features of fundamental relevance, e.g. in the area of service-oriented computing. Besides facilitating the visual specification of processes, the graph algebra simplifies the proofs of correctness: the algebraic structure of states and graphs enables proofs by structural induction.

In this paper we validate the proposal sketched above by using our graph algebra (§ 2) to encode the configurations of process calculi with service-inherent features that have a certain *hierarchical* nature. In particular, we provide novel, correct graphical encodings for two languages. The first one (§ 3) is a simple workflow language, vaguely reminiscent of BPEL: it is used for showing the basic features of the graph syntax and getting the reader acquainted with the approach. The second example (§ 4) regards a sophisticated calculus for the description of service-oriented applications, namely, CaSPiS [2], whose features pose further challenges to visualisation, due to the interplay of name handling, nested sessions and a pipeline operator.

2 An Algebra of Hierarchical Graphs

We offer an overview of our algebra of typed, hierarchical (hyper)graphs that we call *designs*, referring to [3] for a detailed presentation.

Definition 1 (design). *A* design *is a term of sort* $\mathbb{D}$ *generated by the grammar*

$$\begin{array}{lll} \mathbb{D} & ::= & L_{\overline{x}}[\mathbb{G}] \\ \mathbb{G} & ::= & \mathbf{0} \;\mid\; x \;\mid\; l(\overline{x}) \;\mid\; \mathbb{G} \,|\, \mathbb{G} \;\mid\; (\nu x)\mathbb{G} \;\mid\; \mathbb{D}\langle \overline{x} \rangle \end{array}$$

where l and L are respectively drawn from vocabularies $\mathcal{T}$ and $\mathcal{NT}$ of edge and design labels, $\mathcal{N}$ is the set of nodes, $x \in \mathcal{N}$ and $\overline{x} \in \mathcal{N}^*$.

The algebraic reading is as usual, where each syntactical category and vocabulary is considered as a sort and productions are considered as functions. This allows us, for instance, to consider open terms (i.e. terms with typed variables), useful for defining encodings by means of derived operators.

Terms generated by $\mathbb{G}$ and $\mathbb{D}$ are meant to represent, respectively, hierarchical graphs and hierarchical graphs with (edge-like) interfaces. The syntax has the following informal meaning: $\mathbf{0}$ is the empty graph, x is a discrete graph containing node x only, $l(\overline{x})$ is a graph formed by an l-labelled (hyper-)edge attached to nodes $\overline{x}$ (the i-th tentacle to the i-th node in $\overline{x}$), $\mathbb{G} \mid \mathbb{H}$ is the graph resulting from the parallel composition of graphs $\mathbb{G}$ and $\mathbb{H}$ (their disjoint union up to the coalescing of common nodes), $(\nu x)\mathbb{G}$ is the graph $\mathbb{G}$ after making node x not visible from the environment (borrowing nominal calculus jargon we say that the node x is *restricted*), and $\mathbb{D}\langle \overline{x} \rangle$ is a graph formed by attaching design $\mathbb{D}$ to nodes $\overline{x}$ (the i-th node in the interface of $\mathbb{D}$ to the i-th node in $\overline{x}$). A term $L_{\overline{x}}[\mathbb{G}]$ is a design of type L, with body graph $\mathbb{G}$ and exposing nodes $\overline{x}$ in its interface.

Restriction $(\nu x)\mathbb{G}$ acts as a binder for x in $\mathbb{G}$ and similarly $L_{\overline{x}}[\mathbb{G}]$ binds $\overline{x}$ in $\mathbb{G}$. As usual, restricted and interface nodes lead us to the notion of *free* nodes. To this end, we let $\lfloor \overline{x} \rfloor$ denote the set of elements of a vector $\overline{x}$.

Definition 2 (free nodes). *The* free *nodes of a design or a graph are denoted by the function $fn(\cdot)$, defined as follows*

$$\begin{array}{llll} fn(\mathbf{0}) = \emptyset & fn(x) = x & fn(l(\overline{x})) = \lfloor \overline{x} \rfloor & fn(\mathbb{G} \mid \mathbb{H}) = fn(\mathbb{G}) \cup fn(\mathbb{H}) \\ fn((\nu x)\mathbb{G}) = fn(\mathbb{G}) \setminus \{x\} & & fn(\mathbb{D}\langle \overline{x} \rangle) = fn(\mathbb{D}) \cup \lfloor \overline{x} \rfloor & fn(L_{\overline{x}}[\mathbb{G}]) = fn(\mathbb{G}) \setminus \lfloor \overline{x} \rfloor \end{array}$$

We also use $L_{\langle \overline{y} \rangle}[\mathbb{G}\{^{\overline{y}}/_{\overline{x}}\}]$ as a shorthand for a (well-formed: see Definition 3) term $L_{\overline{x}}[\mathbb{G}]\langle \overline{y} \rangle$ whenever it holds $\lfloor \overline{y} \rfloor \cap fn(\mathbb{G}) = \emptyset$, where $\{^{\overline{y}}/_{\overline{x}}\}$ denotes the pairwise, capture-avoiding substitution mapping of names in $\overline{x}$ into names in $\overline{y}$ (given by the set $\{\overline{x}[n] \mapsto \overline{y}[n] \mid n \in |\overline{x}|\}$).

The example below offers a first glance at the algebra of hierarchical graphs.

Example 1. Let $a \in \mathcal{T}$, $A \in \mathcal{NT}$, $u, v, w, x, y \in \mathcal{N}$. We depict in Fig. 1 some terms of our algebra: u (top-left), $a(u, v)$ (top-second), $a(u, w) \mid a(w, v)$ (top-third), $(\nu w)(a(u, w) \mid a(w, v))$ (top-right), and $A_{(u,v)}[(\nu w)(a(u, w) \mid a(w, v))]\langle x, y \rangle$

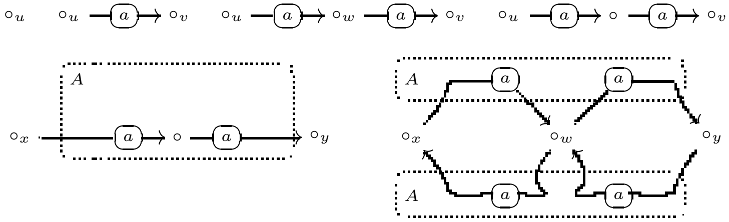


Fig. 1. Some terms of the graph algebra

(bottom-left), also abbreviated as $A_{\langle x,y\rangle}[(\nu w)(a(x,w) \mid a(w,y))]$ Nodes are represented by circles, edges by boxes, and designs by dotted boxes. The first tentacle of an edge is represented by a plain arrow with no head, while the second one is denoted by a normal arrow. Nodes subscripted with their identities are free.

Note that this representation is informal and aims at offering an intuitive visualisation. Figure 1 also includes term $A_{\langle x,y\rangle}[a(x,w) \mid a(w,y)] \mid A_{\langle y,x\rangle}[a(y,w) \mid a(w,x)]$ (bottom-right), where two designs are composed in circle by attaching them symmetrically to x and y, sharing (as a common name) node w.

Sort $\mathbb{D}$ is partitioned over the set $\mathcal{NT} = \{L_1, \ldots, L_n\}$, i.e. we consider sorts $L_1, \ldots, L_n$ and a membership predicate $\mathbb{D} : L$ that holds whenever $\mathbb{D} = L_{\overline{x}}[\mathbb{G}]$ for some $\overline{x}$ and $\mathbb{G}$. Thus, design labels play the role of design (sub-)types. Likewise, we consider the set of nodes $\mathcal{N}$ to be partitioned over different sorts.

Each label of $\mathcal{T}$ and $\mathcal{NT}$ has a fixed arity and for each rank a fixed node type. Intuitively, the typed arity of a label denotes the ordered and typed tentacles of edges with that label. We say that a design (or a graph) is well-typed if for each occurrence of a typed operator $L_{\overline{x}}[\mathbb{G}]$ we have that the (vectors of) types of $\overline{x}$ and L coincide, and similarly for typed operators $\mathbb{D}\langle\overline{x}\rangle$ and $l(\overline{x})$.

Definition 3 (well-formedness). *A design or graph is* well-formed *if (1) it is well-typed; (2) for each occurrence of design $L_{\overline{x}}[\mathbb{G}]$ we have $\lfloor\overline{x}\rfloor \subseteq fn(\mathbb{G})$; and (3) for each occurrence of graph $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle$, the substitution mapping $^{\overline{x}}/_{\overline{y}}$ is a function.*

Intuitively, the restriction on the mapping $^{\overline{x}}/_{\overline{y}}$ forbids two distinct nodes at the higher level to be mapped to the same node in $\mathbb{G}$. This is needed for avoiding implicit name fusions (equivalently, node coalescing, which would require explicit fusion operators and further axioms not needed here) as the result of applying a *flattening* axiom, as shown below. From now on, we restrict our attention to well-formed designs: all the axioms are going to preserve well-formedness and all the derived operators used for the encodings will be well-formed.

In order to have a notion of syntactically equivalent designs (i.e. to consider designs up to isomorphism), the algebra includes the structural graph axioms of [9] such as associativity and commutativity for | (with identity **0**) and name extrusion (respectively, axioms DA1–DA3 and DA4–DA6). In addition, it includes axioms to α-rename bound nodes (DA7–DA8), an axiom for making immaterial the addition of a node to a graph where that same node is already free (DA9) and another one ensuring that global names are not local (DA10). Finally, note that, with respect to the laws presented in [3], we included an explicit axiom for node filtering in designs (DA11): it just states that node restriction may occur at any level of the graph hierarchy, hence, possibly only at the top level.

Definition 4 (design axioms). *The structural congruence* $\equiv_{\mathrm{D}}$ *over well-formed designs and graphs is the least congruence satisfying all the axioms*

$$\begin{array}{rl rll}
\mathbb{G} \mid \mathbb{H} \equiv \mathbb{H} \mid \mathbb{G} & (\mathsf{DA1}) & \mathbb{G} \mid (\nu x)\mathbb{H} \equiv (\nu x)(\mathbb{G} \mid \mathbb{H}) & \textit{if } x \notin \textit{fn}(\mathbb{G}) & (\mathsf{DA6}) \\
\mathbb{G} \mid (\mathbb{H} \mid \mathbb{I}) \equiv (\mathbb{G} \mid \mathbb{H}) \mid \mathbb{I} & (\mathsf{DA2}) & L_{\overline{x}}[\mathbb{G}] \equiv L_{\overline{y}}[\mathbb{G}\{^{\overline{y}}/_{\overline{x}}\}] & \textit{if } \lfloor \overline{y} \rfloor \cap \textit{fn}(\mathbb{G}) = \emptyset & (\mathsf{DA7}) \\
\mathbb{G} \mid \mathbf{0} \equiv \mathbb{G} & (\mathsf{DA3}) & (\nu x)\mathbb{G} \equiv (\nu y)\mathbb{G}\{^{y}/_{x}\} & \textit{if } y \notin \textit{fn}(\mathbb{G}) & (\mathsf{DA8}) \\
(\nu x)(\nu y)\mathbb{G} \equiv (\nu y)(\nu x)\mathbb{G} & (\mathsf{DA4}) & x \mid \mathbb{G} \equiv \mathbb{G} & \textit{if } x \in \textit{fn}(\mathbb{G}) & (\mathsf{DA9}) \\
(\nu x)\mathbf{0} \equiv \mathbf{0} & (\mathsf{DA5}) & L_{\overline{x}}[z \mid \mathbb{G}]\langle \overline{y} \rangle \equiv z \mid L_{\overline{x}}[\mathbb{G}]\langle \overline{y} \rangle & \textit{if } z \notin \lfloor \overline{x} \rfloor & (\mathsf{DA10}) \\
 & & L_{\langle \overline{x} \rangle}[(\nu y)\mathbb{G}] \equiv (\nu y)L_{\langle \overline{x} \rangle}[\mathbb{G}] & \textit{if } y \notin |\overline{x}| & (\mathsf{DA11})
\end{array}$$

where the substitutions are required to be functions (to avoid node coalescing) in axiom (DA7) *and to respects the typing (to preserve well-formedness) in all axioms.*

Structural congruence respect free nodes, i.e. $\mathbb{G} \equiv_{\mathrm{D}} \mathbb{H}$ implies $\textit{fn}(\mathbb{G}) = \textit{fn}(\mathbb{H})$.

We call a graph *flat* whenever there is no design in its body. Flattening a design is done by a kind of hyper-edge replacement [12] in the form of axioms that are sometimes useful to be included in the structural congruence.

Definition 5 (flattening axiom). *A flattening axiom* $\mathsf{flat}_{\mathsf{L}}$ *for some design label* L *is of the form* $L_{\langle \overline{y} \rangle}[\mathbb{G}] \equiv \mathbb{G}$

In the following example we see how flattening is fundamental to characterise classes of graphs by means of derived operators.

Example 2. Suppose that we want to characterise the set of a-labelled, acyclic, and connected sequences. We can define an algebra with an element α in the sequence, and a binary sequential composition $\_;\_$. Both are derived operators defined by $\alpha \overset{\mathbf{def}}{=} A_{(u,v)}[a(u,v)]$ and $X;Y \overset{\mathbf{def}}{=} A_{(u,v)}[(\nu w)(X\langle u,w \rangle \mid Y\langle w,v \rangle)]$, where X and Y have type A. Clearly, the algebra as such constructs hierarchical sequences, where e.g. $(\alpha;(\alpha;\alpha))\langle x,y \rangle$ and $((\alpha;\alpha);\alpha)\langle x,y \rangle$ are not equivalent graphs due to different nestings of A-labeled edges. Introducing $\mathsf{flat}_{\mathsf{A}}$ in the algebra, instead, we have that the two former terms are identified, and intuitively correspond to the normal form $(\nu w_1, w_2)(a(x,w_1) \mid a(w_1,w_2) \mid a(w_2,y))$.

The above example illustrates the two roles of the nesting operator: as a means to wrap a graph and as a sort of typed interface to enable disciplined graph compositions. The presence of flattening axioms makes the first role immaterial.

3 Graphical Interpretation of Workflows

This section presents the use of our graph-based language for algebraic workflow specifications. We consider a minimal language that includes, nonetheless, typical workflow ingredients and offers an attractive presentation of our technique.

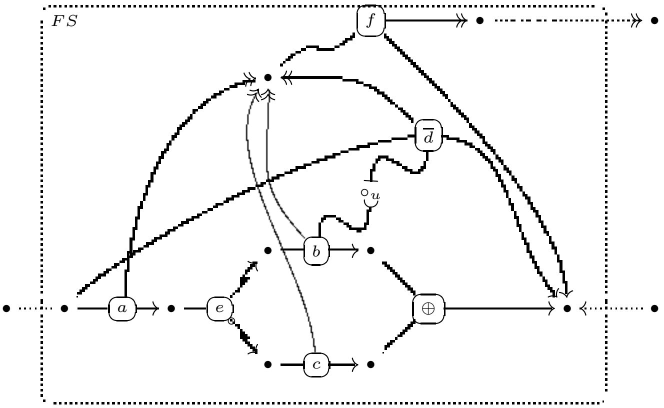


Fig. 2. A simple workflow

A simple language for workflows. Our simple workflow language considers workflows of activities that can be composed in sequence, parallel or by branching. The control flow is restricted to one entry point and two exit points: one for the successful completion and one for error raising. Error-handling activities and error scopes are also considered. In addition, we consider synchronisation links as present in some workflow languages like BPEL.

Figure 2 depicts a very simple example including all the ingredients: we see a main composed flow with a failure handler f. The main flow starts with activity d (whose overlining will be explained later) in parallel with a workflow consisting of activity a, followed by a conditional choice between activities b and c (depending on some expression e). A data dependency is imposed between b and d.

More precisely, the syntax of the workflow language is formally defined below.

Definition 6 (workflow). *Let $\mathcal{A}$ be a set of* activity names*, $\mathcal{E}$ a set of* expressions*, and $\mathcal{U}$ a set of* synchronisation points*. The set $\mathcal{W}$ of all* workflows *is the set of terms generated by F in following syntax (where $a \in \mathcal{A}$, $e \in \mathcal{E}$ and $u \in \mathcal{U}$)*

$$\begin{array}{lcl} F & ::= & A \quad | \quad F;F \quad | \quad \mathsf{if}\ e\ \mathsf{then}\ F\ \mathsf{else}\ F \quad | \quad F|F \quad | \quad \mathsf{try}\ F\ \mathsf{catch}\ F \\ A & ::= & a \quad | \quad a(u) \quad | \quad a[u] \end{array}$$

Informally, a is an asynchronous activity, $a(u)$ is an activity of type a with source link u, $a[u]$ is an activity of type a with target link u, $G; H$ is the sequential composition of structured flows G and H, $\mathsf{if}\ e\ \mathsf{then}\ G\ \mathsf{else}\ H$ introduces a binary branch, i.e. a choice between flows G and H depending on the evaluation of e, $G \mid H$ is the parallel composition of structured flows G and H, and $\mathsf{try}\ G\ \mathsf{catch}\ H$ inserts a new error scope for flow G with fault handler H. As an example, the workflow of Fig. 2 corresponds to the following workflow term $\mathsf{try}\ d[u] \mid a; (\mathsf{if}\ e\ \mathsf{then}\ b(u)\ \mathsf{else}\ c)\ \mathsf{catch}\ f$.

We consider a structural congruence that basically models the fact that sequential composition is associative and parallel composition is associative and commutative. This is formally defined as follows.

Definition 7 (structural congruence). *The* structural congruence *for workflows is the relation* $\equiv_{\mathrm{W}} \subseteq \mathcal{W} \times \mathcal{W}$*, closed under workflow construction and inductively generated by the following set of axioms*

$$G;(H;I) \equiv (G;H);I \quad (w\mathsf{A1}) \quad G \mid (H \mid I) \equiv (G \mid H) \mid I \quad (w\mathsf{A2}) \; G \mid H \equiv H \mid G \quad (w\mathsf{A3})$$

Workflow encoding. The encoding of our workflow language is depicted in Fig. 3. We explain our graphical notation of design operations in detail here. An edge is represented by a rounded box with its label inside. We see that the node types that we use are • and ∘, which respectively represent control flow and synchronisation links. We use an encircled circle ◎ to denote an argument of type ∘ in the encoding of activities. In case of encodings with more than one argument for a type we denote the argument order explicitly by subscripting (see, e.g., the two arguments of type F in the encoding of failure handling, sequence, branching and parallel composition). Terminal edge labels include $a, \overline{a}$ for each $a \in \mathcal{A}$, $e \in \mathcal{E}$ and $\oplus$ which are respectively used to represent activities (with overline for those with target links), expressions and to denote branch closing (xor join). To improve visualisation impact, we use different kinds of arrows to denote tentacles. A plain tentacle represents an entry point, while a simple arrow indicates an ordinary exit point. A double arrow indicates the fault exit point. In a conditional choice, the *then* branch is denoted with an ordinary arrow, while the *else* branch is denoted with an arrow with a small circle on its tail. A bar-ended tentacle denotes the synchronisation link of an activity. Link sources and targets are respectively represented by concave and bar-ended tentacles. Finally, we consider the non-terminal type F to stand for workflows. Dotted arrows denote node exposure and an enclosing dotted box represents a design with its type on the upper-left corner. All nodes are bound except for the argument names (as in the case of synchronisation points in the encoding of activities).

The formal definition by means of our graph algebra is as simple as follows.

Definition 8 (workflow interpretation). *The interpretation of the operators of the workflow language over the design algebra is given by:*

$$\begin{aligned}
a &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[a(in,out,fail)]\\
a(u) &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[a(in,out,fail,u)]\\
a[u] &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[\overline{a}(in,out,fail,u)]\\
G;H &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[(\nu\, mid)(G\langle in,mid,fail\rangle \mid H\langle mid,out,fail\rangle)]\\
\mathsf{if}\ e\ \mathsf{then}\ G\ \mathsf{else}\ H &\stackrel{\mathbf{def}}{=} (\nu\, th1,th2,el1,el2)F_{(in,out,fail)}[e(in,th1,el1)\\
&\qquad \mid G\langle th1,th2,fail\rangle \mid H\langle el1,el2,fail\rangle \mid \oplus(th2,el2,out)]\\
G \mid H &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[G\langle in,out,fail\rangle \mid H\langle in,out,fail\rangle]\\
\mathsf{try}\ G\ \mathsf{catch}\ H &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[(\nu\, mid)FS_{\langle in,out,fail\rangle}[G\langle in,out,mid\rangle\\
&\qquad \mid H\langle mid,out,fail\rangle]]
\end{aligned}$$

together with axiom $\mathsf{flat}_{\mathsf{F}}$.

It is easy to prove that structural congruence amounts to design equivalence, i.e. equivalent workflows amount to equivalent graphs. Note that thanks to axiom

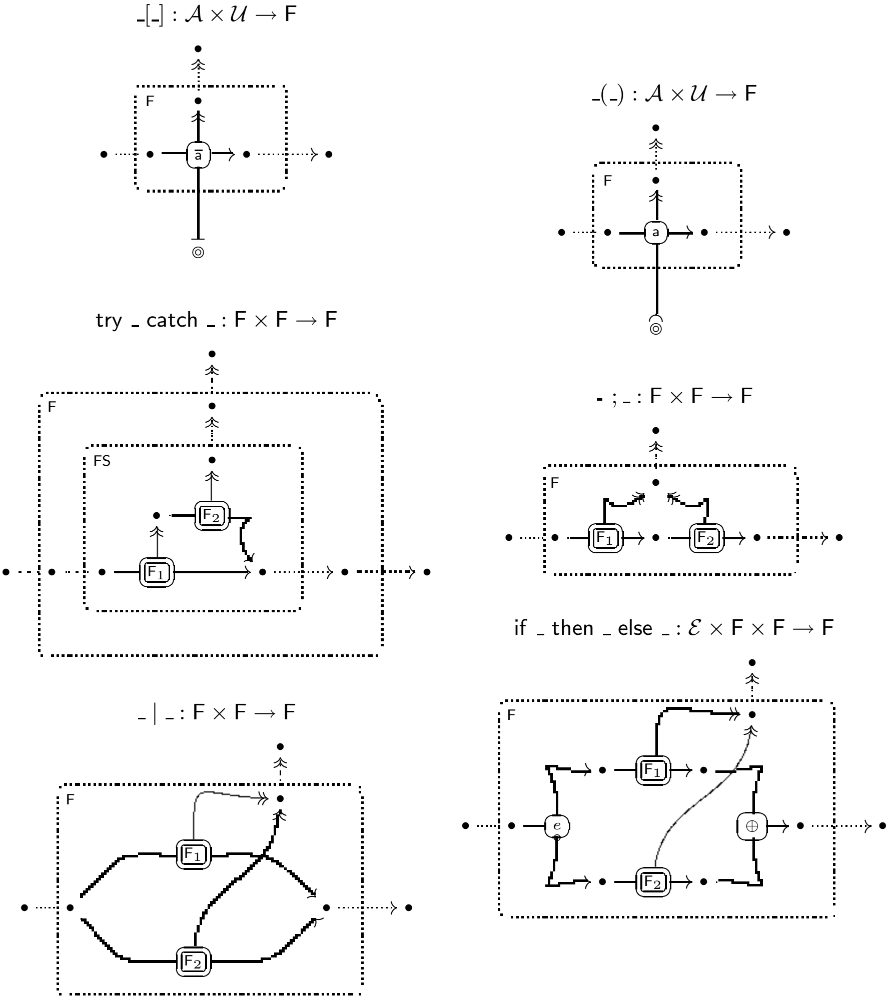

_[_] : $\mathcal{A} \times \mathcal{U} \to$ F
F
ā
⊚
_(_) : $\mathcal{A} \times \mathcal{U} \to$ F
F
a
⊚
try _ catch _ : F × F → F
F
FS
F2
F1
_ ; _ : F × F → F
F
F1
F2
if _ then _ else _ : $\mathcal{E}$ × F × F → F
F
F1
e
⊕
F2
_ | _ : F × F → F
F
F1
F2

Fig. 3. Graphical encoding of a simple workflow language

DA11 node restriction can be equivalently placed in the innermost design (see $G; H$), at the topmost level (see if e then G else H) or at any intermediate level of nesting (see try G catch H).

Proposition 1. *For any two workflows G and H we have $G \equiv_{\mathrm{W}} H$ iff $G \equiv_{\mathrm{D}} H$.*

4 Graphical Interpretation of CaSPiS

This section presents the graphical representation of CaSPiS by defining each CaSPiS syntactic constructor as a derived operator of our graph algebra. We offer a minimal presentation of CaSPiS and refer to [2] for a more detailed description.

Definition 9 (CaSPiS syntax). *Let $\mathcal{R}$ be a set of session names, $\mathcal{S}$ a set of service names and $\mathcal{V}$ a set of value names. A CaSPiS* process *P is a term generated by the syntax*

$$\begin{array}{lcl} P & ::= & \mathbf{0} \mid r \triangleright P \mid P > Q \mid (\nu w)P \mid P \mid P \mid A.P \\ A & ::= & s \mid \overline{s} \mid (?x) \mid \langle u \rangle \mid \langle u \rangle^{\uparrow} \end{array}$$

where $s \in \mathcal{S}$, $r \in \mathcal{R}$, $u \in \mathcal{V}$, $w \in \mathcal{V} \cup \mathcal{R}$ and x is a value variable.

Service definitions and invocations are written like input and output prefixes in CCS. Thus $s.P$ defines a service s that can be invoked by $\overline{s}.Q$. Synchronisation of $s.P$ and $\overline{s}.Q$ leads to the creation of a new session, identified by a fresh name r that can be viewed as a private, synchronous channel binding caller and callee. Since client and service may be far apart, a session naturally comes with two sides, written $r \triangleright P$ and $r \triangleright Q$, with r bound somewhere above them by (νr). Rules governing creation and scoping of sessions are based on those of the restriction operator in the π-calculus. Note that nested invocations to services will yield separate sessions and thus hierarchies of nested sessions.

When two partner sides $r \triangleright P$ and $r \triangleright Q$ are deployed, intra-session communication is done via output and input actions $\langle u \rangle$ and $(?x)$: values produced by P can be consumed by Q, and vice-versa.

Values can be returned outside a session to the enclosing environment using the return operator $\langle \cdot \rangle^{\uparrow}$. Values can be consumed by other sessions, or used locally to invoke other services, or to start new activities. This is achieved using the pipeline operator $P > Q$. Here, a new instance of process Q is activated each time P emits a value that Q can consume. Notably, the new instance of Q will run within the same session as $P > Q$, not in a fresh one.

CaSPiS processes can be considered up to the structural congruence $\equiv_{\mathrm{C}}$.

Definition 10 (CaSPiS congruence). *The structural congruence $\equiv_{\mathrm{C}}$ is the least congruence induced by the following laws*

$$\begin{array}{clrll} P \mid (Q \mid R) \equiv (P \mid Q) \mid R & (\mathsf{CA1}) & P \mid (\nu n)Q \equiv (\nu n)(P \mid Q) & \textit{if } n \notin \mathit{fn}(P) & (\mathsf{CA6}) \\ P \mid Q \equiv Q \mid P & (\mathsf{CA2}) & (\nu n)P \equiv (\nu m)P\{^{m}/_{n}\} & \textit{if } m \notin \mathit{fn}(P) & (\mathsf{CA7}) \\ P \mid \mathbf{0} \equiv P & (\mathsf{CA3}) & A.(\nu n)P \equiv (\nu n)A.P & \textit{if } n \notin A & (\mathsf{CA8}) \\ (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P & (\mathsf{CA4}) & r \triangleright (\nu n)P \equiv (\nu n)r \triangleright P & \textit{if } n \neq r & (\mathsf{CA9}) \\ (\nu n)\mathbf{0} \equiv \mathbf{0} & (\mathsf{CA5}) & ((\nu n)Q) > P \equiv (\nu n)(Q > P) & \textit{if } n \notin \mathit{fn}(P) & (\mathsf{CA10}) \\ & & (?x).P \equiv (?y).P\{^{y}/_{x}\} & \textit{if } y \notin \mathit{fn}(P) & (\mathsf{CA11}) \end{array}$$

CaSPiS encoding. We first define the alphabets of edge labels and nodes. The set $\mathcal{NT}$ of design labels is composed by P, S, D, I, F and T which respectively stand for Parallel processes, Sessions, service Definitions, service Invocations and pipes (From and To). Sort T is further partitioned over $2^{\mathcal{V} \cup \mathcal{R}}$ (denoting each subsort as T^{N}) to deal with a common problem when encoding replicated processes (the target process of a pipe is implicitly replicated for each value generated by the source). The set $\mathcal{T}$ of edge labels contains def (service definition), inv (service invocation), in (input), out (output) and ret (return). The node sorts considered are $\circ$ (channels), $\bullet$ (control points), $*$ (service names, i.e. $\mathcal{S}$) and $\Box$ (values, i.e. $\mathcal{V}$). We assume that for each session name r there is a channel node.

The graphical representation of each design and edge label and their respective types can be found in Fig. 4. For instance, designs of type P are all of the form $P_{(p,t,o,i)}[G]$ where p is the control point representing the process start of execution, t is the returning channel, i is the input channel and o is the output channel. Designs of type D and I only expose the starting point of execution.

Definition 11 (CaSPiS interpretation). *The interpretation of CaSPiS constructors as derived operators of the design algebra is given by*

$$
\begin{aligned}
s.Q &\stackrel{\mathbf{def}}{=} P_{(p,t,o,i)}[\, t|o|i|\, D_{\langle p\rangle}[\, (\nu q, t', o', i')(\mathsf{def}(p,s,q)|Q\langle q,t',o',i'\rangle)\,]\,] \\
\overline{s}.Q &\stackrel{\mathbf{def}}{=} P_{(p,t,o,i)}[\, t|o|i|\, I_{\langle p\rangle}[\, (\nu q, t', o', i')(\mathsf{inv}(p,s,q)|Q\langle q,t',o',i'\rangle)\,]\,] \\
r \rhd Q &\stackrel{\mathbf{def}}{=} P_{(p,t,o,i)}[\, t|i|\, S_{\langle p,o\rangle}[\, Q\langle p,o,r,r\rangle\,]\,] \\
Q > R &\stackrel{\mathbf{def}}{=} P_{(p,t,o,i)}[\, (\nu q, m)(\, F_{\langle p,t,m,i\rangle}[\, Q\langle p,t,m,i\rangle\,] \\
&\qquad\qquad\qquad\qquad\quad |\, T^{fn(R)}_{\langle m\rangle}[\, (\nu q, t', o')R\langle q,t',o',m\rangle\,]\,)\,] \\
Q|R &\stackrel{\mathbf{def}}{=} P_{(p,t,o,i)}[\, Q\langle p,t,o,i\rangle|R\langle p,t,o,i\rangle\,] \\
(\nu w)Q &\stackrel{\mathbf{def}}{=} P_{(p,t,o,i)}[(\nu w)Q\langle p,t,o,i\rangle] \\
\mathbf{0} &\stackrel{\mathbf{def}}{=} P_{(p,t,o,i)}[\, p|t|o|i\,] \\
\langle u\rangle.Q &\stackrel{\mathbf{def}}{=} P_{(p,t,o,i)}[\, (\nu q)(\mathsf{out}(p,q,u,o)|Q\langle q,t,o,i\rangle)\,] \\
\langle u\rangle^{\uparrow}.P &\stackrel{\mathbf{def}}{=} P_{(p,t,o,i)}[\, (\nu q)(\mathsf{ret}(p,q,u,t)|Q\langle q,t,o,i\rangle)\,] \\
(?x).P &\stackrel{\mathbf{def}}{=} P_{(p,t,o,i)}[\, (\nu q, x)(\mathsf{in}(p,q,x,i)|Q\langle q,t,o,i\rangle)\,]
\end{aligned}
$$

Part of the above definition is graphically represented in Fig. 4. As in Section 3 we use different arrow types to denote the different (ordered, typed) tentacles of each edge. For example, for a design representing a process, a double arrow represents its returning channel, an outgoing arrow its output channel, an incoming arrow its input channel and a plain arrow its control point. Again, arguments of an operation are denoted by encircling the corresponding symbol. For instance, double boxes correspond to design variables, while node arguments of type $\circ$, $\star$ and $\Box$ are represented by $\circledcirc$, $\circledast$ and $\boxdot$, respectively.

We introduce flattening axioms $\mathsf{flat_P}$ into $\equiv_{\mathrm{D}}$, but not $\mathsf{flat_S}$, $\mathsf{flat_D}$, $\mathsf{flat_I}$, $\mathsf{flat_F}$ and $\mathsf{flat^N_T}$. Hence, edges of type P are immaterial (they can be considered as type annotations) and the only explicit hierarchies are given by session nesting (S), service definition (D), service invocation (I) and pipelining (F and T). Flattening processes allows for getting rid of the axioms for parallel composition (see [14]). The explicit embedding of sessions provides an intuitive visual representation.

We explain just a few representative operations in detail. The session operations are interpreted as graph operations that wrap a process into a hierarchical S-typed graph which exposes the control point and a return channel. The first is associated to the control point of the resulting P-typed design, while the second is connected to its output channel. Note how session embedding hides the input and output channels of the embedded process: they are connected directly to the dedicated inter-communication node of the session. Another interesting operation is the pipeline. Here, the source and target process of the pipeline are embedded in F- and T-typed designs. It is worth noting how the input and output channels of each process are connected in a complementary way. The target process hides

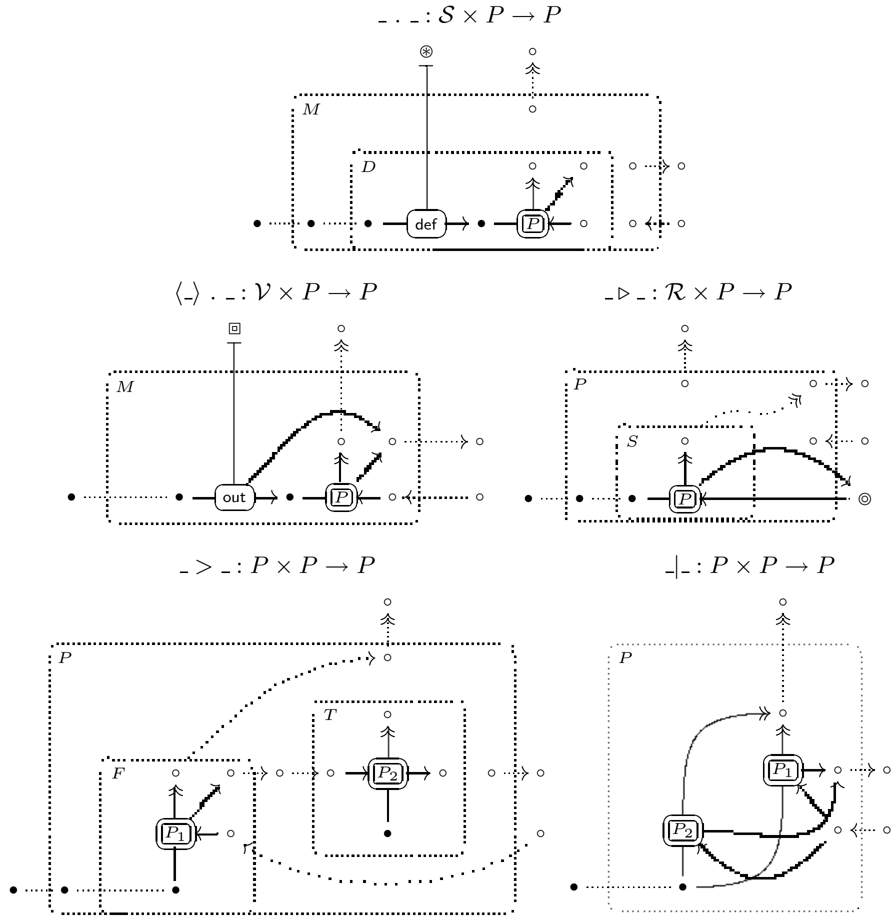


Fig. 4. Graphical representation of some CaSPiS interpreted operators

its control point and communication channels to denote that it is a non-active process. When the source of the pipe is ready to send a value, a copy of the target process will be created and the control and channel nodes will be connected as expected. Moreover, we note that the actual type for the target of the pipe is $T^{fn(R)}$: in words, the type is indexed with the free names of R. This is necessary to avoid node extrusion in a case in which we have no corresponding name extrusion (CaSPiS congruence does not allow to extrude restricted names of the target process of a type). In particular when $w \in fn(R)$ the CaSPiS processes (νw) $(Q > R)$ and $Q > (\nu w)R$ are not congruent, but neither are their corresponding graphs $P_{(p,t,i,o)}[(\nu q, m, w)(F_{\langle p,t,m,i\rangle}[Q\langle p,t,m,i\rangle] | T^{fn(R)}_{\langle m\rangle}[(\nu q, t', i')R\langle q, t', o', m\rangle])]$ and $P_{(p,t,i,o)}[(\nu q, m)(F_{\langle p,t,m,i\rangle}[Q\langle p,t,m,i\rangle] | T^{fn(R)\setminus\{w\}}_{\langle m\rangle}[(\nu q, t', i', w)R\langle q, t', o', m\rangle])]$, because they carry different T subtypes.

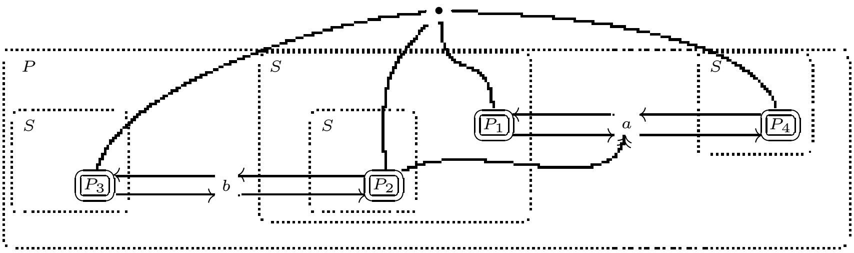


Fig. 5. Example of session nesting

Example 3. Let us illustrate our encoding with a simple example of session nesting. Consider process $(\nu a)(\nu b)(a \rhd (P_1|b \rhd P_2)|a \rhd P_3|b \rhd P_4)$. Two sessions a and b have been created (as the result of two service invocations). Agent $a \rhd (P_1|b \rhd P_2)$ participates to sessions a and b (assume P_1 is the protocol for a and P_2 the one for b), with the b side nested in a. The counter-party protocols for a and b are P_3 and P_4, respectively. Figure 5 depicts a simplified graphical representation of our example, where the graph has been elaborated (e.g. merging nodes for intra-session communication, omitting isolated nodes and irrelevant tentacles) to focus on the main issues and make immediate the correspondence with the process term. It is worth to note that the graph highlights the fact that the return channel of a nested session is pipelined into the output channel of the enclosing session. More precisely, the return channel of the immediate session where P_2 lives (i.e. b) is connected to the output channel of the session containing it, i.e. the session channel a.

Example 4. As another illustrative example consider processes $P_1 > (P_2 > P_3)$ whose (simplified) graphical representation is in Fig. 6. The graphical representation highlights various aspects of interest: the flow of the information via the input and output channels, the fact that P_2 and P_3 are *inactive* protocols, and the pipe nesting. Since $>$ is not associative $P_1 > (P_2 > P_3)$ and $(P_1 > P_2) > P_3$ are not structurally equivalent and this is faithfully reflected in the graphs.

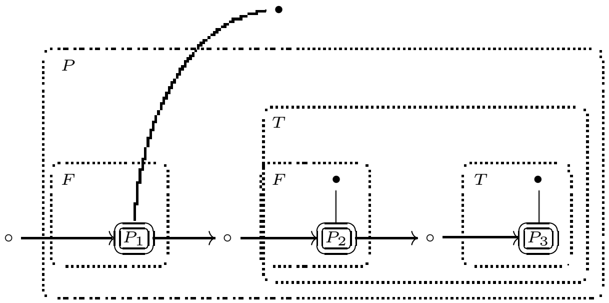


Fig. 6. Example of pipelining

A main result of our work is that structural congruence amounts to design equivalence, i.e. equivalent processes are mapped to isomorphic graphs.

Proposition 2. *For any two processes* P *and* Q *we have* $P \equiv_{\mathrm{C}} Q$ *iff* $P \equiv_{\mathrm{D}} Q$.

5 Conclusion

We presented a preliminary step towards a general technique for the graphical presentation of (possibly service-oriented) process calculi.

More precisely, we used our novel specification formalism based on a convenient algebra of hierarchical graphs [3] to define encodings of process calculi with inherently hierarchical aspects such as sessions, transactions or locations: features which are of fundamental relevance, e.g. in the area of service-oriented computing. In particular, together with the encoding of a simple language for structured workflow with nested scope, we presented a novel graphical encoding of CaSPiS, a recently proposed session-centered calculus.

The chosen encodings highlight the virtues of our graph algebra. First, its syntax resembles the standard syntax of a process calculus, thus offering the possibility of providing intuitive and simple encoding definitions. Second, we can exploit the algebraic structure of both processes and graphs to show encoding properties by structural induction. Indeed, the main result of [3] already guarantees that equivalent designs correspond to isomorphic graphs.

As explained in [3], the particular model of hierarchical graphs puts on a common ground other approaches that have been issued for modelling purposes like the algebra of graphs of Corradini et al. [9], the interface graphs of the second author [14] (a flat model for encoding process calculi with names), the hierarchical graphs of Plump et al. [11] (a suitable extension of traditional graph transformation) and the Bigraphs of Milner et al. [17]. We refer to [3] for a more accurate comparison and we only remark here that one of the advantages of our model of graphs regards the use of hierarchical edges over trees to model processes with an explicit hierarchical structure (thus also recursive processes in the form of replication operators, pipes, etc.). In unstructured cases the approaches are basically equivalent, with hierarchies possibly offering a more attractive visualisation.

Our final goal is to completely mimick the operational semantics of encoded processes and in this line we believe that our model enjoys some good properties, the main being that (though not shown here) the category of hierarchical graphs admits pushouts along monos, which puts the basis for a pushout-based graph rewriting mechanism. While the mimicking of reduction semantics seems rather straightforward we also point to the more ambitious goal of mimicking labelled transition system semantics, possibly in the form of SOS rules. For that purpose we expect that recent approaches based on borrowed contexts [1] or structured graph transformation [10] can be a good start point. The development of a suitable dynamics for our algebras, and its characterisation in terms of graph rewriting mechanisms, is the subject of ongoing work.

We believe that our approach can serve as an inspiration to equip well-known graphical models of communication with syntactical notations that facilitate the definition of intuitive and correct encodings of process calculi. We remark that due to the lack of space we did not discuss the encoding of non-finite processes in full detail. For instance, dealing with replication operators is by no means difficult, by exploiting the hierarchical structure. Of course, the axiom $!P \equiv !P \mid P$ would not hold, since the two terms would have different graphical representations. However, it would suffice to introduce an unfolding operation, possibly parametric in the free names of P, as it happens for the encoding of pipe operators in CaSPiS.

We already applied our technique to other calculi. For instance, we developed an encoding of the best-known nominal calculus, the π-calculus. The encoding is roughly equivalent to the one in [14]. We also focused on service oriented calculi testing our technique on a calculus of transactions called *sagas* [7], and a calculus with locations and multi-party sessions called μse [4].

We plan to propose our algebra as primitive syntax for ADR [6], our graph-based approach to architectural design and reconfiguration. As a matter of fact, we are working to integrate the presented approach in our prototypical implementation of ADR [5]. A preliminary version is available in the form of a visualiser (see `www.albertolluch.com/adr2graphs/`). We remark that our approach proposes a *graphical* representation of configurations but it is not intended to be a *visual* presentation: the graph structures we propose might be used as the formal, facilitating support of some particular kind of appealing diagrams. This issue remains to be investigated.

References

1. Bonchi, F., Gadducci, F., Monreale, G.V.: Reactive systems, barbed semantics, and the mobile ambients. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 272–287. Springer, Heidelberg (2009)
2. Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and pipelines for structured service programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 19–38. Springer, Heidelberg (2008)
3. Bruni, R., Gadducci, F., Lluch Lafuente, A.: An algebra of hierarchical graphs. In: Hofmann, M., Wirsing, M. (eds.) TGC 2010. LNCS. Springer, Heidelberg (to appear 2010), `http://www.albertolluch.com/papers/adr.algebra.pdf`
4. Bruni, R., Lanese, I., Melgratti, H.C., Tuosto, E.: Multiparty sessions in SOC. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 67–82. Springer, Heidelberg (2008)
5. Bruni, R., Lluch Lafuente, A., Montanari, U.: Hierarchical design rewriting with maude. In: Rosu, G. (ed.) WRLA 2008. Electronic Notes in Theoretical Computer Science, vol. 238(3), pp. 45–62. Elsevier, Amsterdam (2009)
6. Bruni, R., Lluch Lafuente, A., Montanari, U., Tuosto, E.: Style Based Architectural Reconfigurations. Bulletin of the European Association for Theoretical Computer Science (EATCS) 94, 161–180 (2008)
7. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Palsberg, J., Abadi, M. (eds.) POPL 2005, pp. 209–220. ACM, New York (2005)

8. Bundgaard, M., Sassone, V.: Typed polyadic pi-calculus in bigraphs. In: Bossi, A., Maher, M.J. (eds.) PPDP 2006, pp. 1–12. ACM, New York (2006)
9. Corradini, A., Montanari, U., Rossi, F.: An abstract machine for concurrent modular systems: CHARM. Theoretical Computer Science 122(1-2), 165–200 (1994)
10. Drewes, F., Hoffmann, B., Janssens, D., Minas, M., Eetvelde, N.V.: Shaped generic graph transformation. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 201–216. Springer, Heidelberg (2008)
11. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. Journal on Computer and System Sciences 64(2), 249–283 (2002)
12. Drewes, F., Kreowski, H.-J., Habel, A.: Hyperedge replacement, graph grammars. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. 1, pp. 95–162. World Scientific, Singapore (1997)
13. Ferrari, G.L., Hirsch, D., Lanese, I., Montanari, U., Tuosto, E.: Synchronised hyperedge replacement as a model for service oriented computing. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 22–43. Springer, Heidelberg (2006)
14. Gadducci, F.: Term graph rewriting for the pi-calculus. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 37–54. Springer, Heidelberg (2003)
15. Gadducci, F., Monreale, G.V.: A decentralized implementation of mobile ambients. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 115–130. Springer, Heidelberg (2008)
16. Grohmann, D., Miculan, M.: An algebra for directed bigraphs. In: Mackie, I., Plump, D. (eds.) TERMGRAPH 2007. Electronic Notes in Theoretical Computuer Science, vol. 203(1), pp. 49–63. Elsevier, Amsterdam (2008)
17. Jensen, O.H., Milner, R.: Bigraphs and mobile processes. Technical Report 570, Computer Laboratory, University of Cambridge (2003)

A Formalisation of Adaptable Pervasive Flows*

Antonio Bucchiarone[1], Alberto Lluch Lafuente[2],
Annapaola Marconi[1], and Marco Pistore[1]

[1] FBK-IRST, via Sommarive 18, 38050, Trento, Italy
{bucchiarone,marconi,pistore}@fbk.eu
[2] Department of Computer Science, University of Pisa
lafuente@di.unipi.it

Abstract. *Adaptable Pervasive Flows* is a novel workflow-based paradigm for the design and execution of pervasive applications, where dynamic workflows situated in the real world are able to modify their execution in order to adapt to changes in their environment. In this paper, we study a formalisation of such flows by means of a formal flow language. More precisely, we define APFoL (Adaptable Pervasive Flow Language) and formalise its textual notation by encoding it in Blite, a formalisation of WS-BPEL. The encoding in Blite equips the language with a formal semantics and enables the use of automated verification techniques. We illustrate the approach with an example of a Warehouse Case Study.

1 Introduction

Flows are models defining a set of activities to be done, and their relations with each other. Flows are deeply seated in many fields, including business processes and service oriented computing. The flow modeling paradigm is often used either implicitly or explicitly in many real life situations. In this paper, we concentrate on a novel usage of flows, which is being investigated by the ALLOW project [1]: the usage of flows as a new programming paradigm for human-oriented pervasive applications. More precisely, *Adaptable Pervasive Flows* (APFs) [10] are proposed as an extension of traditional workflow concepts [19] in order to make them more flexible with respect to their pervasive execution environment. APFs are dynamic workflows situated in the real world that modify their execution in order to adapt to changes in their environment. This requires on the one hand that a flow must be context-aware: during execution it must be possible to obtain information on the underlying environment (e.g. relevant information on world entities, status of other flows, human activities). On the other hand flow models must be flexible enough to allow an easy and continuous adaptation. APFs are based on WS-BPEL [5], a well-known language for specifying flows in a Web Service setting, and extend it in order to implement all the aspects related to pervasive applications.

In [16] the authors define one of these extensions to WS-BPEL. More precisely, they define a set of constructs that allow a convenient way of embedding the adaptation logic within the specification of an APF and show how WS-BPEL can be extended

* Research supported by the EU, STREP project Allow IST-324449 and Sensoria, IST-2005-016004.

C. Laneve and J. Su (Eds.): WS-FM 2009, LNCS 6194, pp. 61–75, 2010.

to support the proposed constructs. These constructs allow for capturing interesting cases of adaptation in pervasive applications that are difficult to address with classical workflows and with the standard WS-BPEL language. In this paper, we extend the work in [16], by providing a formal model for APFs and for the extensions of WS-BPEL related to the adaptation logics. This is a significant extension of the previous work since, due to the high dynamicity of pervasive applications, formal methods become crucial to drive design disciplines, equip existing languages with well-defined semantics and increase the reliability by means of automated verification.

Web services are a good example where many efforts are being invested on the development and application of formal methods. For instance, there have been various proposals to define formal semantics for WS-BPEL, typically by means of process calculi (e.g. [12], [15]), Petri nets (e.g. [14], [18]), or graphs (e.g. [7]).

In this paper we propose APFoL, a formal language for adaptable pervasive flows. Amongst the various formal approaches to flow languages we have chosen Blite [13] as a starting point. Blite is a process calculus that captures a significant part of the WS-BPEL language. We build our language as an almost straightforward extension of Blite. More precisely, we equip the language with abbreviations to deal with adaptation mechanisms, flow constructs and activity types typical of adaptable pervasive flows. The formal language proposed permit us to formally specify the built-in adaptation constructs informally proposed in [16].

This document is structured as follows. Section 2 introduces a scenario from the Warehouse Case Study of the Allow project [1] while Section 3 presents the background that is used in section 4 where our flow language is introduced. Section 5 illustrates our language with an example drawn from the Warehouse Case Study. Finally, Section 6 draws conclusions and outlines current and future work.

2 The Running Example: Warehouse Management

We present a scenario from the Warehouse Case Study of the Allow project [1], namely the management of a warehouse that we will use as a reference for all the examples within this document. The main aim of warehouse management is to organize and control the transport and storage of goods within a warehouse. This is achieved through the definition and processing of complex transactions, including shipping, receiving, put-away, picking and issuing of goods. The objective of the warehouse management system is to provide a set of computerised procedures supporting all the aforementioned activities: from the handling of goods reception, storage and shipping, to the management of all the physical storage facilities.

Warehouse management often utilizes auto AIDC (Automatic Identification and Data) technology, such as bar-code scanners, mobile computers, wireless LANs and potentially RFID (Radio Frequency Identification) to efficiently monitor the flow of products. Current systems require that, once data has been collected, synchronization with a centralized system is performed. The centralized system is in charge of controlling all aspects of warehouse management. Pervasive flows offer the possibility to distribute the control logics and hence to improve flexibility and context awareness of the executed processes.

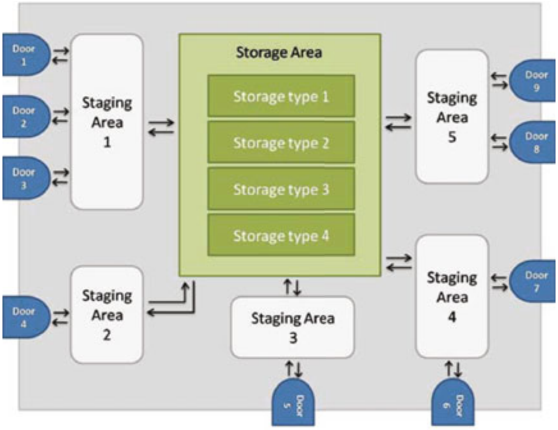


Fig. 1. Warehouse structure

The (logical and physical) structure of a warehouse is described by Figure 1. Doors are the locations where the goods arrive at or leave the warehouse. Trucks drive up to the doors of a warehouse in order to unload or load goods there. Staging areas are used for interim storage of goods in the warehouse. These are located in close proximity to the doors associated to them. The storage area is organized in several zones corresponding to different storage types. Storage types are physical or logical subdivisions of a warehouse complex, characterized by its warehouse technique, the space used, its organizational form, or its function.

Warehouse management requires the execution of different procedures which refer to the different objects and human actors, including *good receipt*, *issuing* and *transfer*. Here we focus on the first procedure, which describes the three steps (see Figure 2) that have to be performed when the goods arrive at the warehouse: *delivery* (a truck has reached the warehouse and is docked at a door); *unload* (goods are unloaded and temporally stored in the staging area); and *put-away* (goods are moved to the storage area).

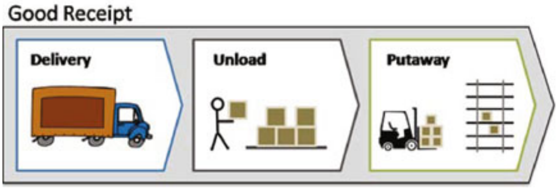


Fig. 2. Good receipt procedure

3 Background

Our work is strongly based on a proposal for the extension of WS-BPEL to deal with adaptation [16] and the process calculus Blite [13].

3.1 WS-BPEL and APFs

Similar to the well-known workflows, APFs consist of a set of activities and a corresponding execution order, which is specified using control elements such as sequence, choice or parallel operators. After a deep analysis and comparison of todays workflow standards using criteria such as industry impact, robustness aspect and extensibility [2], the ALLOW project has chosen WS-BPEL as a nucleus for an APF language.

A particular feature of APFs is that they are situated in the real world. This realizes the *pervasiveness* of the flows and is achieved in two ways. First, the flows are logically attached to physical entities (which can be either objects or humans) and move with them through different contexts. Secondly, they run on physical devices (e.g. PDAs, desktops). Todays workflow languages (e.g. WS-BPEL) provide no possibility to explicitly model the context model and constraints on the workflow environment. The pervasiveness of a WS-BPEL workflow can be specified only through ad-hoc interactions with external context-aware services. Clearly, this solution affects the readability and transparency of the pervasive aspect within the specified workflow. A first extension [9] that has been done aims at providing a modeling approach to annotate WS-BPEL processes with contextual constraints and an execution model to monitor those constraints during process execution.

Another important aspect of APFs is their *adaptiveness*. A flow is a dynamic entity that modifies its execution in order to adapt to changes in the execution environment. A key enabling factor for automated adaptation mechanisms is a convenient way of embedding the adaptation logic within the specification of a flow. Adaptation mechanisms cannot be limited to standard recovery constructs (e.g. fault/event/compensation handlers in WS-BPEL), but should also support the specification of flexible context-aware reactions to adaptation needs that can be used to handle run-time flow deviations without requiring a flow recovery/failure. The aim of the work in [16] is to present a set of primitives and principles that can support the encoding of context-aware run-time deviations and changes within a flow model in a secure (from an execution perspective) and convenient (from a modelling perspective) way. The authors propose a set of *built-in adaptation* modeling constructs that can be useful to add dynamicity and flexibility to flow models and for each construct they define the corresponding WS-BPEL extension. In particular, the proposed constructs are (i) conditional branches within flows with context conditions as guard conditions, (ii) context handlers that allow to automatically react to context conditions violation during the execution of the flow, and (iii) constructs that allow to specify a set of alternative scopes, each handling a specific execution context, and that allow to jump at run-time from one scope to another, whenever the context changes or the assumptions on the context turn out to be wrong.

3.2 WS-BPEL and Blite

Blite [13] is a formal language for describing web service orchestrations. It has been designed as a formal model to capture the essentials of WS-BPEL, a de-facto standard for describing web services. Blite is a process calculus and as such it has a well-defined notion of syntax and operational semantics. The language includes features such as

Table 1. Syntax of Blite

Basic activities	$\mathsf{b} ::=$	$\mathsf{inv}\,\ell^i\,\mathsf{o}\,\bar{\mathsf{x}} \mid \mathsf{rcv}\,\ell^r\,\mathsf{o}\,\bar{\mathsf{x}} \mid \mathsf{x} := \mathsf{e}$	invoke, receive, assign
	$\mid$	$\mathsf{empty} \mid \mathsf{throw} \mid \mathsf{exit}$	empty, throw, exit
Structured activities	$\mathsf{a} ::=$	$\mathsf{b} \mid \mathsf{if(e)\{a_1\}\{a_2\}} \mid \mathsf{while(e)\,\{a\}}$	basic, conditional, iteration
	$\mid$	$\mathsf{a_1\,;a_2} \mid \sum_{j\in J} \mathsf{rcv}\,\ell^r_j\,\mathsf{o}_j\,\bar{\mathsf{x}}_j\,;\mathsf{a}_j$	sequence, choice (with $\lvert J \rvert > 1$)
	$\mid$	$\mathsf{a_1} \mid \mathsf{a_2} \mid [\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c]$	parallel, scope
Start activities	$\mathsf{r} ::=$	$\mathsf{rcv}\,\ell^r\,\mathsf{o}\,\bar{\mathsf{x}} \mid \sum_{j\in J} \mathsf{rcv}\,\ell^r_j\,\mathsf{o}_j\,\bar{\mathsf{x}}_j\,;\mathsf{a}_j$	receive, choice
	$\mid$	$\mathsf{r\,;a} \mid \mathsf{r_1} \mid \mathsf{r_2} \mid [\mathsf{r} \bullet \mathsf{a}_f \star \mathsf{a}_c]$	sequence, parallel, scope
Services	$\mathsf{s} ::=$	$[\mathsf{r} \bullet \mathsf{a}_f] \mid \mu \vdash \mathsf{a} \mid \mu \vdash \mathsf{a}\,,\mathsf{s}$	definition, instance, multiset
Deployments	$\mathsf{d} ::=$	$\{\mathsf{s}\}_\mathsf{c} \mid \mathsf{d_1} \parallel \mathsf{d_2}$	deployment, composition

service definition and instantiation, typical flow constructs, communication primitives and failure handling and compensation mechanisms. We offer here a brief, intuitive overview of Blite and refer to [13] for a detailed presentation.

The syntax of the Blite language is summarised in Table 1 (borrowed from [13]). *Basic activities* include variable assignments, flow success related operations (throw and exit) and communication primitives to send (inv) or receive (rcv) values from partner links. *Structured activities* organise basic activities in flows by using typical flow constructs such as branches, sequences, loops, fork&join and (input-prefixed) choices[1]. In addition, scopes can be defined with appropriate failure and compensation activities. *Start activities* are structured activities starting with a choice of receive operations. The reason for this is that service definitions are inactive until they receive a request. Services already instanced, instead are represented by their memory μ (an assignment of values to variables) and their flow (a structured activity). Deployments (i.e. the system) are sets of correlated services.

4 A Formal Language for Adaptable Pervasive Flows

This section presents our formalisation of adaptable pervasive flows in terms of a flow language that we call APFoL.

We present here our language for adaptable pervasive flows, describing each ingredient and giving its encoding in Blite. It is worth mentioning that the encoding automatically equips our language with a formal semantics.

We start offering an informal presentation of our visual notation, which we plan to formalise in the future, possibly by means of a graphical encoding of our language. Here we just present the informal visual notation as Figures 3, 4 and 5, that act as an illustration of the textual notation that we shall describe in detail.

We now present the textual notation of APFoL (summarised in Table 2), which basically extends the syntax of Blite (c.f. Table 1) with ad-hoc constructs for built-in adaptation mechanisms.

[1] Called *pick* in [13] but *choice* here to avoid confusion with the pick flow construct of APFoL.

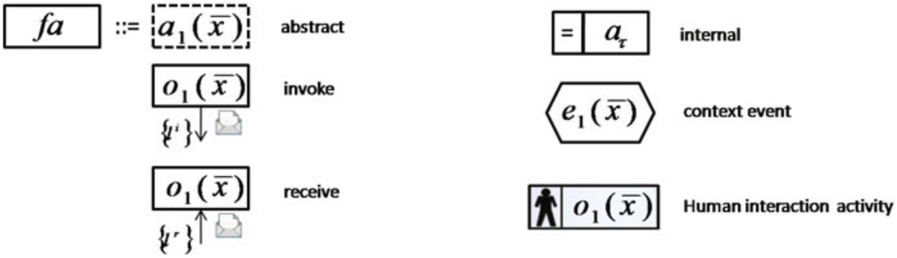


Fig. 3. Visual syntax for Flow Activities of Table 2

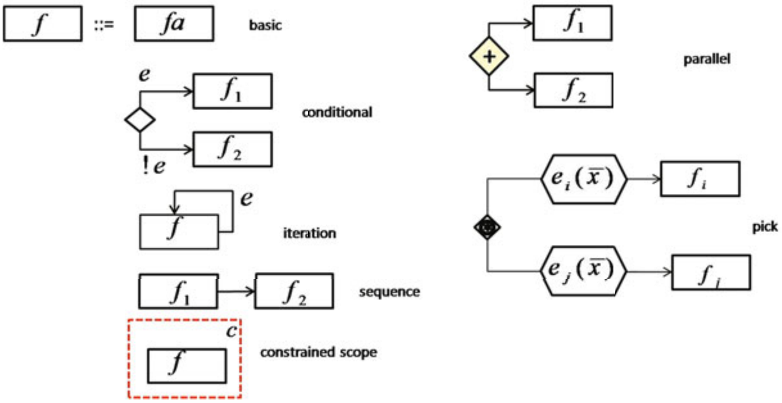


Fig. 4. Visual syntax for Flow Instances (Part-I) of Table 2

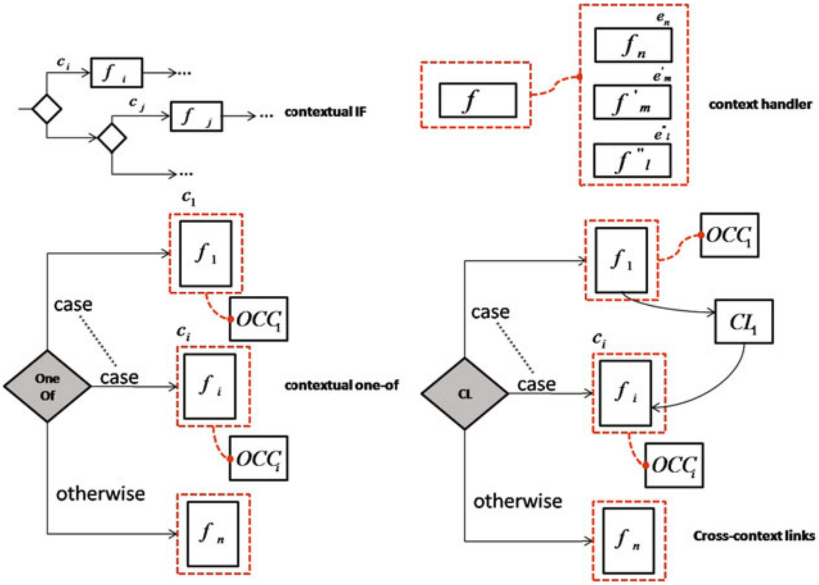


Fig. 5. Visual syntax for Flow Instances (Part-II) of Table 2

The main differences with respect to Blite's syntax regard basic and structured activities. We call them called *basic and structured flows* in APFoL to avoid confusion and stick to the APF slang. They include some new constructs to model the relevant primitives of adaptable pervasive flows. Services and deployments remain identical but, again, we call them differently (*flows and flow* systems) to avoid confusion. Finally, we avoid presenting the productions for *start flows* for simplicity: they are a straightforward adaptation of those for *start activities* in the same way as flow instances are an adaptation of activity instances.

First, APFoL includes the same control flow constructs of Blite. In addition, even if not part of the primitive syntax summarised in Table 1, APFoL includes typical control flow constructs such as different forms of branching (e.g. switch) and looping (e.g. loop-exit) which are straightforwardly encoded in Blite.

An *abstract activity* $A(\overline{x})$ represents either a partial design-time specification of a flow model or an abbreviation of a complex activity. Abstract activities are modelled just as function symbols A of type fa (flow activity) or fr (start activity), thus $A(\overline{x})$: fa∪fr in Table 2. For each such symbol we assume a definition to exist (for abbreviations) or to be given at run-time (for partial designs). Refining an abstract activity then means

Table 2. Syntax of APFoL

Flow activities	fa ::=	sinv l o x̄ $\mid$ srcv l o x̄ $\mid$ a_τ	invoke, receive, internal
	$\mid$	$\langle e\overline{x}\rangle \mid A(\overline{x})$	context event, abstract activity
Flows instances	f ::=	fa $\mid$ if(e){f$_1$}{f$_2$} $\mid$ while(e) {f}	basic, conditional, iteration
	$\mid$	f$_1$; f$_2$ $\mid$ f$_1$ $\mid$ f$_2$	sequence, parallel
	$\mid$	$[\![f]\!]^c_a \mid \mathsf{pick}_{i\in J}(e_i \rightarrow f_i)$	constrained scope, pick
	$\mid$	$\mathsf{cIF}_{i\in J}(c_i \rightarrow f_i)$	contextual IF
	$\mid$	$[\![f]\!]\lfloor \overset{e_1}{f_1}\rfloor\lfloor \overset{e_n}{f_n}\rfloor\lfloor \overset{e'_1}{f'_1}\rfloor\lfloor \overset{e'_m}{f'_m}\rfloor\lfloor \overset{e''_1}{f''_1}\rfloor\lfloor \overset{e''_l}{f''_l}\rfloor$	context handler
	$\mid$	$\mathsf{one-of}[\![f_1]\!]^{c_1}_{r_1} \ldots [\![f_n]\!]^{c_n}_{r_n}$	contextual one-of
Flows	ff ::=	[fr • f] $\mid$ $\mu \vdash$ f $\mid$ $\mu \vdash$ f , ff	definition, instance, multiset
Flow system	fs ::=	{ff}$_\mathsf{c}$ $\mid$ fs$_1$ ‖ fs$_2$	deployment, composition

Table 3. Blite encoding of the main ingredients of APFoL

Sending activity	$\mathsf{sinv}\ q\ o\ \bar{x} \overset{\text{def}}{=} \mathsf{inv}\ \langle \mathsf{id}, q\rangle\ o\ \bar{x}\ ;\ \mathsf{rcv}\ \langle \mathsf{id}\rangle\ o\ ack$
Receiving activity	$\mathsf{srcv}\ q\ o\ \bar{x} \overset{\text{def}}{=} \mathsf{rcv}\ \langle \mathsf{id}, q\rangle\ o\ \bar{x}\ ;\ \mathsf{inv}\ \langle q\rangle\ o\ ack$
Context event	$\langle e\ \bar{x}\rangle \overset{\text{def}}{=} \mathsf{rcv}\ ContextManager\ get_e\ \bar{x}$
Internal event	$\langle e\rangle \overset{\text{def}}{=} while(e)\{empty\}$
Constrained scope	$[\![f]\!]^c_a \overset{\text{def}}{=} \langle c\rangle\ [\{f\}^{\neg c} \mid ((\langle\neg c\rangle;\ \mathsf{throw})\ \bullet\ a * \mathsf{empty}]$
Pick	$\mathsf{pick}_{i\in J}(e_i\bar{x}_i \rightarrow a_i) \overset{\text{def}}{=} \sum_{i\in J}(\langle e_i\bar{x}_i\rangle\ ;\ a_i)$
Contextual IF	$\mathsf{cIF}_{i\in J}(c_i \rightarrow f_i) \overset{\text{def}}{=} \mathsf{switch}_{i\in J}\mathsf{c_i} \rightarrow [\![f_i]\!]^{c_i}_{\mathsf{throw}}$

replacing the left-hand side of a definition by its right-hand side. Clearly, this is not a real extension of the language and is standard machinery of all algebraic specifications.

Communication activities allow for sending (resp. receiving) a message to (resp. from) another flow. Invoke and receive activities are synchronous. Thus we encode them as suggested in [13] by the authors of Blite, namely by a pair of receive and invoke actions. In the definition, id stands for the flow instance identity. A sending activity is thus encoded as the invocation of operation o at partner l, where the identity of the flow is passed to receive the response. Similarly, the receive activity expects to receive an invocation of operation o at himself (id) together with the invoker's identity q which is used to send the response.

Data manipulation activities are internal activities that change the value of local variables and do not interact with their environment. Data manipulation activities are modelled as structured activities whose component basic activities a_τ are all assignments. Note that the grammar for a_τ can be given but, for the sake of a clear presentation, prefer to avoid adding an ad-hoc syntactical category and its (rather redundant) productions.

Human interaction activities are activities that require an interaction with a human, e.g. displaying or getting information through a device. Human interaction activities are modelled as communication operations, since devices are represented by flows.

Context events are a special type of activities for receiving events broadcasted by a particular entity called *Context Manager*. More precisely, during this activity the flow execution waits until the event is received. We model context events as particular receive operations. More precisely, we shall model context managers as services that broadcast their events e via replicated sending operations named get_e. Thus, the reception of the event is modelled as a reception operation with the context manager as partner, operation get_e and the corresponding tuple of values.

We shall also use a sort of *internal events*, denoted by $\langle e \rangle$ whose meaning is to wait until the expression (i.e. a condition or trigger) e is true.

A *constrained scope* $[\![f]\!]^c_a$ is a flow f enclosed into a scope with unique entry and exit points, a constraint c and adaptation a (triggered if the constraint is not valid). They are represented in our syntax by terms of the form $[\![f]\!]^c_a$, where f is the normal flow, c is the constraint and a is the adaptation to be performed in case the condition fails inside the scope. This is modelled in Blite by exploiting the failure mechanism. First, we wait until the condition is true. Then we open a Blite scope where we put a condition observer flow in parallel with the normal flow conditioned to $\neg c$[2]. Conditioning is necessary to avoid the normal flow to progess in case the context condition violated. Note that this semantics does not really interrupt the flow. The exception is raised and the failure code performs the adaptation activity, only when the observer is executed.

A *pick* is a branching point in the process where the alternatives are based on events, rather than the evaluation of expressions. More precisely, it is the receipt of a message or of a context event that determines which of the paths will be taken. Event-based decision is modelled by a non deterministic choice of activities preceded by the corresponding triggering event.

[2] This is done by inserting a condition (in form of an *internal event*) between each activity and can be easily defined in a recursive manner.

$$
\begin{array}{l}
[\![main]\!] \lfloor \overset{fault_1}{ff_1} \rfloor \dots \lfloor \overset{fault_n}{ff_n} \rfloor \lfloor \overset{event_1}{ef_1} \rfloor \dots \lfloor \overset{event_m}{ef_m} \rfloor \lfloor \overset{block_1}{bf_1} \rfloor \dots \lfloor \overset{block_l}{bf_l} \rfloor \overset{\mathbf{def}}{=} \\
\qquad \mathsf{new}\ done; \\
\qquad done := false; \\
\qquad [\![\{main\}^{suspend} \\
\qquad\qquad | \, fault_1 \rightarrow throw \\
\qquad\qquad | \dots \\
\qquad\qquad | \, fault_n \rightarrow throw \\
\qquad\qquad | \, event_1 \rightarrow ef_1 \\
\qquad\qquad | \dots \\
\qquad\qquad | \, event_m \rightarrow ef_m \\
\qquad\qquad | \, block_1 \rightarrow bf_1; done := true \\
\qquad\qquad | \dots \\
\qquad\qquad | \, block_l \rightarrow bf_l; done := true \\
\qquad]\!]^{fault_1 \vee \dots \vee fault_1 n}_{switch\{fault_1 \rightarrow ff1 \dots fault_n \rightarrow ff_n\}}
\end{array}
$$

$$suspend \overset{\mathbf{def}}{=} (\neg done \wedge block_1) \vee \dots \vee (\neg done \wedge block_l)$$

Fig. 6. Context handler in Blite

A *contextual IF* allows to define several flow fragments as possible branches in the execution of the flow. Each flow fragment has an associated *context condition*. We can define also one flow fragment without a context condition, which will encode the default behaviour. The operational semantics of this construct is similar to a traditional if: the first fragment for which the *context condition* holds will be selected and executed.

A *context handler* is a particular flow associated to a *main* flow or flow scope. It specifies an alternative flow (in the form of a contextual IF) to be applied if a corresponding scope condition is violated. There are various forms of conditions within the context handler flow.

- *Fault-triggering* suspend all active tasks in the main flow and execute the corresponding error-handler. If the main flow is a flow scope, the fault is propagated to the enclosing scope.
- *Event-triggering* conditions can be *non-blocking* (execution of the main flow proceeds normally and the corresponding flow is executed concurrently) or *blocking* (execution of the main flow suspends, then the corresponding flow is executed and finally the main flow is resumed).

The encoding of context handlers in Blite is defined in Figure 6. The construct consists of a main flow *main* and three sets of observers: fault-handlers, non-blocking event handlers, and blocking event handlers, whose triggers are respectively denoted by *fault*, *event* and *block*, while the corresponding flows are respectively denoted by *ff*, *ef* and *bf*. The idea is as follows: the main flow (conditioned to *suspend*) is put in parallel with various observers one for each fault and event trigger. When a fault condition triggers, an error is raised and handled by the error handler which selects a fault and fires the corresponding flow. Non-blocking events trigger the corresponding flow. Blocking events perform similarly, but note that the *suspend* condition depends on the blocking event

```
$\text{one} - \text{of}[\![f_1]\!]_{r_1}^{c_1} \dots [\![f_n]\!]_{r_n}^{c_n} \overset{\text{def}}{=}$
    new $success$;
    loop
        switch
            $c_1 \rightarrow [\![f_1; success := true]\!]_{r_1}^{c_1}$
            $\dots$
            $c_n \rightarrow [\![f_n; success := true]\!]_{r_n}^{c_n}$
        if($success$) exit;
```

Fig. 7. Contextual One-of in Blite

conditions: if one of them is true and the corresponding flow has not been performed, the main flow remains suspended until *done* becomes true[3].

A *contextual one-of* consists of a set of alternative flow fragments, each of them associated to a contextual condition modeling the contextual assumption for that fragment, and a rollback flow that can be executed to undo the partial and unsuccessful work of the fragment. At run-time, the first flow fragment for which the contextual condition holds is chosen and executed. During the fragment execution, its context condition is monitored and, as soon as it is violated, the following actions are performed:

1. *stop execution:* all running activities within the fragment are stopped;
2. *undo partial work:* the roll-back flow associated to the current fragment is executed;
3. *context jump:* the first fragment for which the associated context holds is executed and its context condition is monitored.

Roll-back flows can throw fault/exceptions (e.g. to handle the fact that the work done within the fragment cannot be undone), and in this case the flow is terminated following normal flow fault handling. If this is not the case, and the roll-back flow completes successfully, the main flow is considered successfully running.

The encoding in Blite is rather easy, a loop is used to guarantee that performing a rollback returns to the selection of one of the choices. The only way to exit a loop is to successfully finish one of main flows.

When using the *contextual one-of*, it may be the case that, when jumping from one execution context to another, we do not want to undo the work done or the complete flow rollback is not possible. The *cross-context link(CL)* is designed especially for this case. CLs connect two activities of different scopes within a *contextual one-of*. CLs allow adapting to a context change by jumping from a certain execution state of the current activity (*source* activity) to an execution activity (*target* activity) of another fragment suitable for the actual context. After the jump the flow instance must be in a consistent state. Therefore, a CL has an associated flow needed to prepare the flow to the jump. At runtime, if the contextual condition associated to the running scope turns out to be false, two possibilities are considered:

[3] Note that in order to guarantee a unique *done* variable we declare it as *new* at the beginning of the encoding. This is not a feature of Blite but can be added straightforwardly by considering the local store μ as a stack of assignment sets instead of a plain set.

```
CL$[\![f_1]\!]_{r_1}^{c_1} \ldots [\![f_n]\!]_{r_n}^{c_n} \overset{\text{def}}{=}$
    new $success$; new $next\_flow$; $next\_flow := any$;
    loop
        switch
            $c_1 \wedge proceed(1) \rightarrow [\![f_1; success := true]\!]_{r_1}^{|c_1|}$
            $\ldots$
            $c_n \wedge proceed(n) \rightarrow [\![f_n; success := true]\!]_{r_n}^{|c_n|}$
        if($success$) exit;
```

Fig. 8. Contextual One-of with CLs in Blite

1. if there exists some context link leaving the active activity for which the context condition holds:
   (a) the roll-back flow associated to the cross-context link is executed
   (b) the monitoring for the new context condition is activated
   (c) the flow execution is re-started from the target activity of the CL
2. otherwise the condition violation is handled as described for the standard *contextual one-of*.

The encoding in Blite is similar to the encoding of ordinary contextual one-of. The first difference is that the guard of each flow f_i is enriched with *proceed*(i) which is an abbreviation for $next\_flow = i \vee next\_flow = any$. This serves to control which flow should be executed next. The second difference is that each compensation c must take the ad-hoc roll-back flows for cross-context jumps into consideration. With $|c|$ we denote the introduction of a choice that decides whether to apply the ordinary compensation c or the roll-back flow associated to the jump to the next flow.

5 The Box Unloading Example

In this section we present a complete example that summarizes most elements introduced before. For exemplification, we consider the box flow. A first problem that can occur here is that the box can be damaged. The damage may have occurred either before, during transportation, but it may also occur at any point while the box is being unloaded to the staging area, or moved to the storage area.

In Figure 9 we use the Contextual OneOf construct to model the handling of damaged boxes. In case the box is not damaged, the first flow scope is chosen and executed. If at any point the box gets damaged, the context condition *not(b.damaged)* is violated and the *onContextChange* flow is executed. That is, pending activities for unloading and/or storing are canceled (e.g. the reserved staging/storage location is made available for other boxes, the request for unloading/storing sent to workers are revoked). The specific activities to be performed clearly depend on the state of execution, due to this abstract activities are specified, namely *Cancel Unloading* and *Cancel Storing* and at run-time they will be refined with context-specific concrete activities. Once the *onContextChange* flow is executed, the control goes back to the *OneOf* and the scope handling

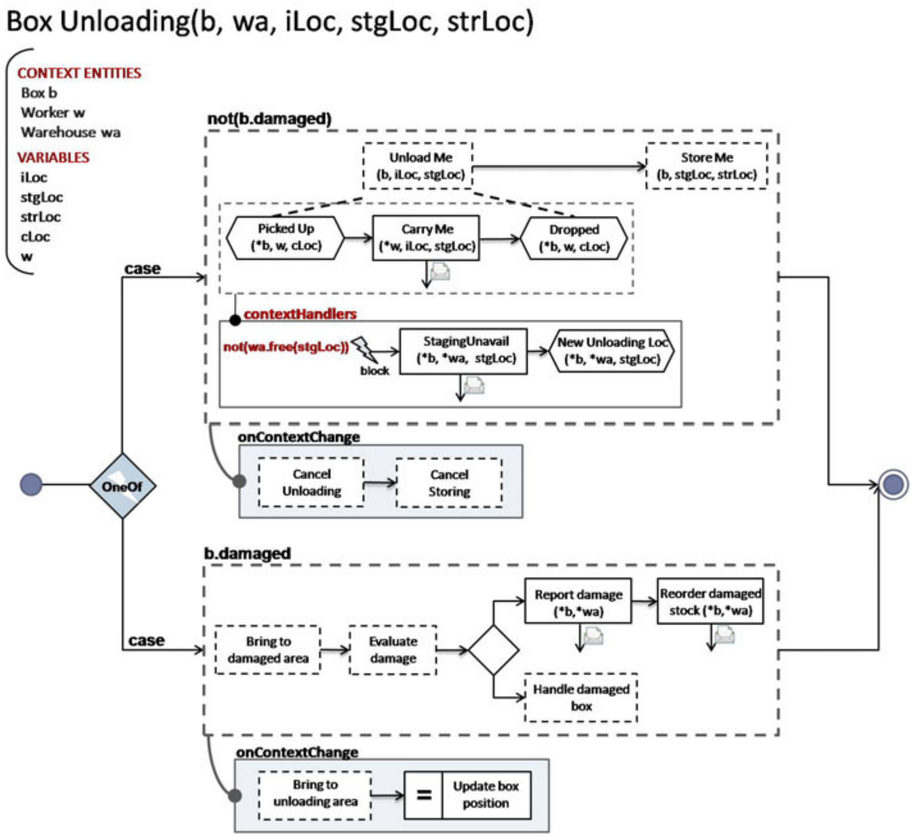


Fig. 9. The Box Unloading example

damaged boxes is executed. If at some point in the execution of the flow scope for handling damaged boxes the box is repaired, the context condition associated to the scope (*b.damaged*) is violated, the execution stops and the *onContextChange* flow is executed. This way, the box is brought to a waiting area and its position is updated, and then the scope for handling undamaged boxes can start.

Another built-in construct exploited within the example in Figure 9 is the *contextHandler*. In particular, during the execution of the *Unload Me* refinement, it may be the case that the assigned staging location is no more available. If this is the case, the contextual constraint *wa.free(stgLoc)*, monitored during the whole execution of the refinement flow, is violated and the contextHandler is executed. Since the handler is defined as a blocking event, the execution of the main scope is suspended, then the handler flow is executed and then the main scope is resumed.

The APFoL code of this example is flow Box Unloading shown in Figure 5, while its WS-BPEL code is listed in [3].

$\mathsf{Box\ Unloading} \stackrel{\mathrm{def}}{=} \mathsf{one-of}[\![notDamaged]\!]^{c_1}_{occ_1}\,[\![Damaged]\!]^{c_2}_{occ_2}$
$\quad c_1 \stackrel{\mathrm{def}}{=} not(b.damaged)$
$\quad occ_1 \stackrel{\mathrm{def}}{=} CancelUnloading;$
$\qquad\quad CancelStoring$
$\quad c_2 \stackrel{\mathrm{def}}{=} b.damaged$
$\quad occ_2 \stackrel{\mathrm{def}}{=} BringtoUnloadingArea;$
$\qquad\quad UpdateBoxPosition$

$notDamaged \stackrel{\mathrm{def}}{=} UnloadMe(b, iLoc, stgLoc);$
$\quad StoreMe(b, stgLoc, strLoc)$
$\quad UnloadMe(b, iLoc, stgLoc) \stackrel{\mathrm{def}}{=} [\![main]\!]\lfloor \overset{block_l}{bf_l} \rfloor$
$\quad StoreMe(b, stgLoc, strLoc) \stackrel{\mathrm{def}}{=} \ldots$

$\quad main \stackrel{\mathrm{def}}{=} \langle PickedUp\ (*b, w, cLoc)\rangle;$
$\qquad\quad \mathsf{sinv}\ w\ \mathsf{CarryMe}\ (*\mathsf{w}, \mathsf{iLoc}, \mathsf{stgLoc});$
$\qquad\quad \langle Dropped\ (*b, w, cLoc)\rangle$
$\quad block_1 \stackrel{\mathrm{def}}{=} not(wa.free(stgLoc))$
$\quad bf_1 \stackrel{\mathrm{def}}{=} \mathsf{sinv}\ wa\ \mathsf{StagingUnavail}\ (*\mathsf{b}, *\mathsf{wa}, \mathsf{stgLoc})\ ;$
$\qquad \langle NewUnloadingLoc\ (*b, *wa, stgLoc)\rangle$

$Damaged \stackrel{\mathrm{def}}{=} BringtoDamagedArea;$
$\qquad\quad EvaluateDamage;$
$\qquad\quad \mathsf{if(repairable)\{f_1\}\{f_2\}}$

$\quad f_1 \stackrel{\mathrm{def}}{=} HandleDamagedBox$
$\quad f_2 \stackrel{\mathrm{def}}{=} \mathsf{sinv}\ wa\ \mathsf{ReportDamage}\ (*\mathsf{b}, *\mathsf{wa});$
$\qquad \mathsf{sinv}\ wa\ \mathsf{ReorderDamagedStock}\ (*\mathsf{b}, *\mathsf{wa})$

Fig. 10. APFoL encoding of flow Box Unloading

6 Conclusion and Future Work

We have described a preliminary version of APFoL, a language for adaptable pervasive flows with formal support. More precisely, we have presented a language whose textual notation is based on Blite [13], a process calculus for WS-BPEL.

The formalisation of the language equips the language with a well-defined (and hence non-ambiguous) semantics. More precisely, the formalisation as a process calculus (Blite) facilitates the use of automated verification techniques. Some support for the analysis and verification of Blite specifications exists (an encoding from Blite into another calculus with tool support [4]), but we are working in a direct implementation of Blite semantics in the rewrite engine Maude [8], in order to exploit its generic built-in capabilites in form of analysis tools such as a model checker and a theorem prover. With suchan implementation at hand, APFoL can be implemented as a derived rewrite theory in Maude.

Our approach has been illustrated with examples from the Warehouse case study of the Allow project [1].

We plan to develop a graph-based formalisation of our visual notation, possibly basing existing techniques for the graphical encoding of process calculi (e.g. [6]). The main goals of having a formal graph-based representation is formalise the relation betweeen textual and visual notation and to enable the use of graph transformation techniques, and their corresponding tools.

In the future we would like to investigate the connection with apparently similar approaches in the data base community around the notion of *business artifacts* [17,11], which are flows attached to physical objects moving through different contexts.

Acknowledgments

The authors would like to thank Roberto Bruni for sharing with us his knowledge on WS-BPEL formalisations and the authors of Blite (Rosario Pugliese, Francesco Tiezzi and Alessandro Lapadula) for providing us with useful material.

References

1. EU-FET project 213339 ALLOW, `http://www.allow-project.eu/`
2. D3.1 Basic flow-model and language for Adaptable Pervasive Flows. ALLOW Project Deliverable (November 2008)
3. APFoL homepage, `http://www.antoniobucchiarone.it/APFoL.html`
4. Blite: A formal account of WS-BPEL, `http://rap.dsi.unifi.it/blite/`
5. OASIS WSBPEL Tecnical Committee. Web Services Business Process Execution Language, version 2.0 (2007), `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0`
6. Bruni, R., Gadducci, F., Lluch Lafuente, A.: A graph syntax for processes and services. In: 9th International Workshop on Web Services and Formal Methods, WS-FM 2009 (2009)
7. Bundgaard, M., Glenstrup, A.J., Hildebrandt, T.T., Højsgaard, E., Niss, H.: Formalizing higher-order mobile embedded business processes with binding bigraphs. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 83–99. Springer, Heidelberg (2008)
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
9. Eberle, H., Fll, S., Herrmann, K., Leymann, F., Marconi, A., Unger, T., Wolf, H.: Enforcement from the inside: Improving quality of bussiness in process management. In: IEEE 7th International Conference on Web Services, ICWS 2009 (2009) (to appear)
10. Herrmann, K., Rothermel, K., Kortuem, G., Dulay, N.: Adaptable Pervasive Flows - An Emerging Technology for Pervasive Adaptation. In: Workshop on Pervasive Adaptation (PerAda), September 2008. IEEE Computer Society, Los Alamitos (2008)
11. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part II. LNCS, vol. 5332, pp. 1152–1163. Springer, Heidelberg (2008)
12. Laneve, C., Zavattaro, G.: Web-pi at work. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 182–194. Springer, Heidelberg (2005)
13. Lapadula, A., Pugliese, R., Tiezzi, F.: A formal account of WS-BPEL. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 199–215. Springer, Heidelberg (2008)

14. 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)
15. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for WS-BPEL. Journal of Logic and Algebraic Programming 70(1), 96–118 (2007)
16. Marconi, A., Pistore, M., Sirbu, A., Eberle, H., Leymann, F.: Enabling adaptation of pervasive flows: Built-in contextual adaptation. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC 2009. LNCS, vol. 5900, pp. 389–403. Springer, Heidelberg (2009)
17. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal 42(3), 428–445 (2003)
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 Computer Programming 67(2-3), 162–198 (2007)
19. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press, Cambridge (2002)

Compliance Preorders for Web Services

Michele Bugliesi, Damiano Macedonio, Luca Pino, and Sabina Rossi

Dipartimento di Informatica, Università Ca' Foscari Venezia
{michele,mace,lpino,srossi}@dsi.unive.it

Abstract. Compliance is a basic property of web-service architectures that ensures the absence of deadlocks and livelocks during execution. Following recent attempts in the literature, we interpret compliance as an experiment, much like the experiments made by a test process in testing theories, and use it as the basis for a notion of *compliance preserving substitution* of components within a composition of web services.

We review the different notions of compliance in the literature, analyze their relative strengths and weaknesses, and formalize their interrelationships by providing a uniform formal framework where we reconcile the different perspectives that characterize them.

1 Introduction

Compliance is a basic property that characterizes the correct behavior of concurrent distributed systems. It is used widely in the context of Service Oriented Architectures (SOA) as a formal device to identify well-formed service compositions, those whose interactions are free of synchronization errors.

Formal theories of compliance have been developed within different settings, most notably with *session types* and *behavioral contracts*. Session types [11] have originally been conceived as a generalization of channel types [16] for the static control of interaction patterns in which the same channel is used to send and/or receive payloads of different types at different times. Recently, systems of session types have applied widely in the analysis of various kinds of interaction and conversation structures [6,12,5].

Behavioral contracts, our focus in the present paper, arise in process algebraic settings, and provide abstract descriptions of system behavior by means of terms of some process algebra. Formal theories of contracts have first been introduced in [7], and then further developed along independent lines of research in [13,8,9], and in [3,4].

All these papers share the main motivations and the overall technical setup, inspired by the theory of testing in process algebra [15]. In particular, they all interpret compliance as a basic test for investigating services, extract the preorder relationships induced by the test, and justify a compliance-preserving substitution principle for services based on that. On the other hand, the approaches differ significantly in the notion of compliance adopted as well as in the settings where they apply it. In [13,8,9], compliance is targeted at preventing deadlocks, and the theory is developed for a client-server setting to provide safety guarantees for

C. Laneve and J. Su (Eds.): WS-FM 2009, LNCS 6194, pp. 76–91, 2010.

the client. Compliant servers are those which will never get their clients stuck: the compliance preorder is established similarly in terms of the ability of servers to satisfy their clients. In [3,4], instead, compliance is a stronger condition that ensures the absence of deadlocks *and* livelocks, and the application setting is that of choreographies: compliant choreographies are those whose computations never get stuck or trapped into infinite loops without chances to exit. The compliance pre-order, in turn, is induced on the component contracts, in terms of their ability to preserve the compliance of the choreographies they are part of.

In this paper, we review the existing definitions in the literature to formalize their inter-relationships. As a result of our analysis we provide a uniform framework where we (*i*) reconcile the different perspectives that characterize the existing definitions of compliance, (*ii*) fill some of the existing gaps among them, and thus (*iii*) maximize the potential of cross-fertilization among the different approaches. We start with an analysis of the deadlock-safe notion of compliance developed in [8,13] for client-server settings, and propose an equivalent formulation that scales naturally to multi-party service compositions. Then, we give a fully-abstract co-inductive characterization of the induced compliance-preorder and discuss its use for the safe replacement of services inside multi-party compositions. We also analyze the stronger definition of compliance proposed by [4] for choreographies, showing that it effectively constitutes a conservative generalization of the deadlock-safe definition of [8,13] (when the latter is lifted to multiparty compositions). We generalize the coinductive construction of the deadlock-safe preorder to obtain a sound (but not complete) characterization of the stronger preorder. For both pre-orders, we also show how the *filters* from [8] may be employed to achieve a flexible compliance-preserving substitution principle inside choreographies.

Plan. Section 2 introduces the contract language we use for our analysis. Section 3 analyzes the notions of compliance in the literature, and introduces our own variations of such notions. Section 4 develops the coinductive characterizations of the associated compliance preorders. Section 5 discusses generalized versions of the preorders, and their counductive characterizations. Section 6 concludes the presentation.

2 A Core Contract Language

We start introducing a small language for contracts and contract compositions that we use for our analysis. Contracts are represented as (single-threaded) terms of a CCS-like [14] process calculus that includes recursion and operators for external and internal choice. Parallel composition arises in *contract compositions.* We presuppose a denumerable set of action names $\mathcal{A}$, ranged over by a, b, c. Actions represent the basic units of observable behavior of the underlying services, namely input, noted a, and output, note $\overline{a}$. We let α range over actions and co-actions, and note $\overline{\alpha}$ the co-action corresponding to α.

$$\begin{array}{ll} \textit{Contracts} & \sigma \ ::= \mathbf{1} \mid \mathbf{x} \mid \alpha.\sigma \mid \sigma + \sigma \mid \sigma \oplus \sigma \mid \mathtt{rec}(\mathbf{x})\,\sigma \\ \textit{Compositions} & C ::= \sigma \mid C \parallel C \end{array}$$

$\mathbf{1}$ signals that the service has reached a successful state, $\alpha;\sigma$ describes a service that performs action α and then behaves as σ; $\sigma + \sigma'$ denotes an external choice, guided by the environment, while $\sigma \oplus \sigma'$ indicates a local choice between σ and σ' made irrespective of the structure of the interacting environment; $\mathtt{rec}(\mathbf{x})\,\sigma$ is a recursively defined contract. We assume a standard contractivity condition for the recursion operator, requiring that recursion variables be guarded by a prefix. We let $in(\sigma)$ and $out(\sigma)$ note the set of input and output actions in σ, respectively.

Table 1. Dynamics of contracts and compositions

Contract Satisfaction: $\sigma \checkmark$

$$\mathbf{1}\checkmark \qquad \frac{\sigma_i \checkmark}{\sigma_1 + \sigma_2 \checkmark} \qquad \frac{\sigma\{\mathbf{x} := \mathtt{rec}(\mathbf{x})\,\sigma\}\checkmark}{\mathtt{rec}(\mathbf{x})\,\sigma \checkmark}$$

Contract transitions: $\sigma \xrightarrow{\dot\alpha} \sigma'$

$$\alpha.\sigma \xrightarrow{\alpha} \sigma \qquad\qquad \sigma_1 \oplus \sigma_2 \longrightarrow \sigma_i \quad (i = 1, 2)$$

$$\frac{\sigma_i \xrightarrow{\dot\alpha} \sigma}{\sigma_1 + \sigma_2 \xrightarrow{\dot\alpha} \sigma}\ (i = 1, 2) \qquad \frac{\sigma\{\mathbf{x} := \mathtt{rec}(\mathbf{x})\,\sigma\} \xrightarrow{\dot\alpha} \sigma'}{\mathtt{rec}(\mathbf{x})\,\sigma \xrightarrow{\dot\alpha} \sigma'}$$

Composition satisfaction and transitions:

$$\frac{C_1\checkmark \quad C_2\checkmark}{C_1 \parallel C_2 \checkmark} \qquad \frac{C_1 \xrightarrow{\alpha} C_1' \quad C_2 \xrightarrow{\overline{\alpha}} C_2'}{C_1 \parallel C_2 \longrightarrow C_1' \parallel C_2'} \qquad \frac{C_1 \xrightarrow{\dot\alpha} C_1'}{C_1 \parallel C_2 \xrightarrow{\dot\alpha} C_1' \parallel C_2}$$

Dynamics of contracts and compositions. We define the dynamics of the calculus with a labelled transition system (and a success predicate), with rules reported in Table 1: $\dot\alpha$ indicates the label α or no label. The first block of rules defines the successful states of a contract, i.e., those that expose the success term $\mathbf{1}$ at top level, or immediately under an external choice (up-to recursive unfoldings). The rules in the second and third blocks define the transitions for contracts and compositions, and are mostly self-explanatory. Notice that a composition reaches a successful state only when all component contracts are themselves successful.

We write $\Longrightarrow$ to note the reflexive and transitive closure of $\longrightarrow$, and $\sigma \overset{\alpha}{\Longrightarrow} \sigma'$ for $\sigma \Longrightarrow\xrightarrow{\alpha}\Longrightarrow \sigma'$. Similarly, for $w = \alpha_1 \ldots \alpha_n$ the transition $\sigma \overset{w}{\Longrightarrow} \sigma'$ stands for $\sigma \overset{\alpha_1}{\Longrightarrow} \cdots \overset{\alpha_n}{\Longrightarrow} \sigma'$. We omit the target of a (weak) transition when immaterial, writing $\sigma \xrightarrow{\alpha}$ and $\sigma \overset{\alpha}{\Longrightarrow}$ to signal that σ has a (weak) transition on α to some σ', and $init(\sigma)$ for the set $\{\alpha \mid \sigma \overset{\alpha}{\Longrightarrow}\}$. Finally, with $\sigma \downarrow \mathrm{R}$ we note that $\mathrm{R} \subseteq (\mathcal{A} \cup \{\checkmark\})$ is the smallest non-empty set such that $\sigma \xrightarrow{\alpha}$ implies $\alpha \in \mathrm{R}$, and $\sigma \checkmark$ implies $\checkmark \in \mathrm{R}$. Similarly, $\sigma \Downarrow \mathrm{R}$ whenever $\sigma \Longrightarrow \sigma'$ with $\sigma' \downarrow \mathrm{R}$.

A computation for a composition C is a sequence $C \equiv C_0 \longrightarrow C_1 \longrightarrow \ldots$; the computation is *maximal* if either it is infinite or there exists C_n such that $C \Longrightarrow C_n \not\longrightarrow$. A composition C is finite if all its maximal computations are finite. Throughout, we presuppose the following conditions on contracts and compositions.

Definition 1 (Determinacy). *A contract σ is* determinate *if for every action α there exists at most one contract σ' such that $\sigma \Longrightarrow\xrightarrow{\alpha} \sigma'$, and σ' is itself determinate. A composition $\sigma_1 \parallel \ldots \parallel \sigma_n$ is determinate, if so are the σ_i, and in addition, $in(\sigma_i) \cap in(\sigma_j) = \emptyset$ whenever $i \neq j$.*

When $\sigma \overset{\alpha}{\Longrightarrow}$, we note $\sigma(\alpha)$ the unique σ' such that $\sigma \Longrightarrow\xrightarrow{\alpha} \sigma'$. ($\sigma(\alpha)$ is always defined for determinate contracts). Determinacy is technically convenient for our analysis, and represents a fairly mild assumption (cf. [8] for a similar assumption, which in that case is enforced directly by the transition relation). Indeed, contracts can be made determinate, by factoring, without affecting their external behavior. To illustrate, the non-determinate contract $(a.\sigma_1 \oplus b.\sigma_2) + (a.\sigma_3 \oplus b.\sigma_4)$ may equivalently be expressed (up to, internal moves) as the determinate contract $a.(\sigma_1 \oplus \sigma_3) \oplus b.(\sigma_2 \oplus \sigma_4)$. As to compositions, determinacy, simply amounts to interpreting the actions of each contract in a composition as being associated with services located and accessible at univocally identified ports/sites (as done, for instance, in [4]).

3 Compliance Tests

We start with a review of the asymmetric, deadlock-safe notion of compliance (ds-compliance, for short), as proposed by [8,13] for client-server settings, and introduce its asymmetric and symmetric variants for general, multi-party compositions. Then, we discuss the definition of *safe* compliance, that is sensitive to both deadlocks and livelocks.

3.1 Deadlock-Safe Compliance

For two contracts ρ and σ, let $\rho \bowtie \sigma$ signal that ρ and σ may synchronize, possibly after a sequence of internal moves. Formally $\rho \bowtie \sigma$ iff $init(\sigma) \cap \overline{init(\rho)} \neq \emptyset$, where $\overline{init(\rho)}$ is the set of co-actions corresponding to the actions in $init(\rho)$. The asymmetric presentation distinguishes two roles (client and server, respectively) for the contracts involved in the compliance test.

Definition 2 (Client-server ds-compliance [8,13]). *A client ρ and a server σ are* $\mathtt{ds}$-compliant*, written $\rho \dashv^{\mathsf{ds}} \sigma$ iff whenever $\rho \parallel \sigma \Longrightarrow \rho' \parallel \sigma'$, either $\rho' \bowtie \sigma'$, or $\checkmark \in \mathrm{R}$ for all R such that $\rho' \Downarrow \mathrm{R}$.*

Accordingly, $\rho \dashv^{\mathsf{ds}} \sigma$ if and only if whenever the interaction between ρ and σ gets stuck (as there is no chance of synchronization) ρ may independently terminate with success. The asymmetric nature of the definition, and its clear bias in favor of ρ, is a consequence of the different intended roles of the two contracts in the composition. In [13], the definition has the following, additional proviso: in case $\sigma'\!\uparrow$ then for all R such that $\rho' \Downarrow \mathrm{R}$, $\mathrm{R} = \{\checkmark\}$. The intuition here is that if the interaction between client and server gets the server trapped into an internal loop (noted $\sigma'\!\uparrow$), then the client must terminate successfully without expecting any further synchronization with the server. The proviso holds vacuously in our contract language, given that recursive contracts are formally contractive, hence they may not loop without interaction.

Definition 2 may be restated equivalently by stipulating that ρ and σ are compliant if whenever $\rho \parallel \sigma \Longrightarrow \rho' \parallel \sigma' \not\longrightarrow$, one has $\rho'\checkmark$. That the two definitions are equivalent follows again from our formal-contractiveness assumption, which implies that $\rho \not\bowtie \sigma$ iff $\rho \parallel \sigma \Longrightarrow \rho' \parallel \sigma' \not\longrightarrow$. The new formulation is interesting as it suggests the following, natural lifting of the client-server notion of compliance to multi-party compositions.

Definition 3 (Asymmetric ds-compliance). *A contract ρ is* $\mathtt{ds}$-compliant *with a composition C, written $\rho \dashv^{\mathsf{ds}} C$, iff whenever $\rho \parallel C \Longrightarrow \rho' \parallel C' \not\longrightarrow$, one has $\rho'\checkmark$.*

Asymmetric compliance, in turn, is readily made symmetric by simply removing the bias in favour of any of the component services.

Definition 4 (Symmetric ds-compliance). *A contract composition C is* $\mathtt{ds}$-compliant*, noted $C\searrow^{\mathsf{ds}}$, if whenever $C \Longrightarrow C' \not\longrightarrow$, one has $C'\checkmark$.*

3.2 Safe Compliance

One weakness of $\mathtt{ds}$-compliance is that it is insensitive to livelocks. The following example helps illustrate the problem in the asymmetric setting. Consider the two contracts $\sigma = \mathtt{rec}(\mathbf{x})\,(a.\mathbf{x} \oplus b.\mathbf{x})$ and $\sigma' = \mathtt{rec}(\mathbf{x})\,a.\mathbf{x}$, and take the contract $\rho = \mathtt{rec}(\mathbf{x})\,(\overline{a}.\mathbf{x} + \overline{b}.\mathbf{1})$. Applying Definition 3, it is a routine check to verify that ρ is $\mathtt{ds}$-compliant with both σ and σ', namely $\rho \dashv^{\mathsf{ds}} \sigma$, and $\rho \dashv^{\mathsf{ds}} \sigma'$. This is not exactly desirable, as the two contracts determine quite different behaviors for ρ: indeed, while σ is acceptable to ρ, as there is always a chance for ρ to reach a successful state, σ' is not as it leaves ρ trapped into a livelock.

If livelocks are to be avoided, we need to strengthen the compliance test. The following definition, that we take verbatim from [4], does the job with a test inspired by the theory of *should*-testing [17].

Definition 5 (Symmetric safe compliance [4]). *A contract composition C is* s-compliant*, noted $C{\searrow}^{\mathsf{s}}$, if for every C' such that $C \Longrightarrow C'$ there exists C'' such that $C' \Longrightarrow C''\checkmark$.*

In other words, s-compliance ensures that at each intermediate step of the computation in a choreography, each component service has a way to reach a successful state (either autonomously, or via synchronizations within the choreography). This is enough to avoid livelocks. To illustrate, consider the composition $C \stackrel{\mathit{def}}{=} \mathtt{rec}(\mathbf{x})\,(a.\mathbf{x}) \parallel \mathtt{rec}(\mathbf{x})\,(\bar{a}.\mathbf{x})$. C is ds-compliant even though it never reaches any successful state: indeed, the condition imposed by Definition 4 holds vacuously as C has only infinite computation. This is rectified in Definition 5 by demanding that all intermediate computation states offer a path to success for all the services of the composition.

Safe compliance is readily re-cast into an asymmetric presentation, by simply re-introducing the bias in favour of one component and stipulating that $\rho \dashv^{\mathsf{s}} C$ if for every $\rho' \parallel C'$ such that $\rho \parallel C \Longrightarrow \rho' \parallel C'$ there exists $\rho'' \parallel C''$ such that $\rho' \parallel C' \Longrightarrow \rho'' \parallel C''$ and $\rho''\checkmark$. The new predicate conveys the desired guarantees for the client-server setting (when C is a single server): if we go back to our problematic servers σ and σ' above, we now see that $\rho \dashv^{\mathsf{s}} \sigma$ but $\rho \not\dashv^{\mathsf{s}} \sigma'$, as desired.

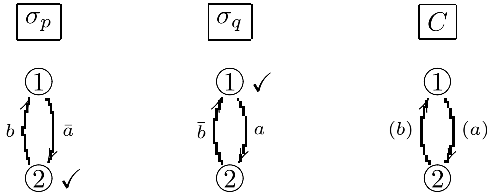
3.3 Symmetric vs Asymmetric Compliance

While there is clearly a strong connection between the two presentations of compliance (symmetric vs asymmetric), we are not aware of results establishing formal relationships. We give two such results below, first showing that for deadlock-safe compliance the asymmetric presentation can be defined in terms of the symmetric one, in the following sense. Given $C = \sigma_1 \parallel \ldots \parallel \sigma_n$, we note $C/_i$ the composition "C drop σ_i", defined as follows: $C/_i \stackrel{\mathit{def}}{=} \sigma_1 \parallel \ldots \parallel \sigma_{i-1} \parallel \sigma_{i+1} \parallel \ldots \parallel \sigma_n$.

Theorem 6 (Symmetric vs. Asymmetric ds-compliance). *Let C be the composition $\sigma_1 \parallel \ldots \parallel \sigma_n$. Then $C{\searrow}^{\mathsf{ds}}$ if and only if $\sigma_i \dashv^{\mathsf{ds}} C/_i$ for all $1 \leq i \leq n$.*

Proof. ($\Longrightarrow$) From the hypothesis, for every C' such that $C \Longrightarrow C' \not\longrightarrow$ we have $C'\checkmark$. This means that each component of C' is in a success state. Since this is true of all $C' \not\longrightarrow$ reachable from C, it must be the case that $\sigma_i \dashv^{\mathsf{ds}} C/_i$ for all i. ($\Longleftarrow$) The hypothesis is that for every i, $\sigma_i \dashv^{\mathsf{ds}} C/_i$. Now, for all the C' such that $C \Longrightarrow C' \not\longrightarrow$ we have that $\sigma'_i\checkmark$ (where σ'_i is the state corresponding to σ_i in C'). Since this is true of every i, it follows that $C{\searrow}^{\mathsf{ds}}$. □

In the forward direction Theorem 6 carries over to the case of safe compliance. Instead, somewhat surprisingly, this is not true of the backward direction. To see that, consider the following counter-example. Take $\sigma_p = \mathtt{rec}(\mathbf{x})\,(\bar{a}.(b.\mathbf{x} + \mathbf{1}))$ and $\sigma_q = \mathtt{rec}(\mathbf{x})\,(a.\bar{b}.\mathbf{x} + \mathbf{1})$, and form the composition $C = \sigma_p \parallel \sigma_q$. The following diagrams show the transitions for each contract and for the composition: the success states are marked by the satisfaction predicate.



Now, $\sigma_p \dashv^{\mathsf{s}} \sigma_q$ because, for every state reached by the computation of C, we can extend the computation to reach the state 2, where the σ_p reaches its success. $\sigma_q \dashv^{\mathsf{s}} \sigma_p$ holds for the same reason, because it's always possible to reach the state 1 where σ_q reaches its success. However $C \not\searrow^{\mathsf{s}}$ because, as outlined by the graph above, there is not a reachable state where both the contracts are successful.

4 Compliance Preorders

Associated with a compliance test, which we mark as $\bullet$ to generalize[1], one defines a corresponding semantics for service contracts. In client-server setting, this is stated in terms of the sets of the compliant clients $[\sigma]^{\bullet} \overset{def}{=} \{\rho \mid \rho \dashv^{\bullet} \sigma\}$; in multi-party compositions, it is defined similarly in terms of sets of compliant compositions: $[\sigma]^{\bullet} \overset{def}{=} \{C \mid \sigma \parallel C\searrow^{\bullet}\}$. Then, based on the contract semantics, one defines the contract preorder, uniformly as follows: $\sigma \sqsubseteq^{\bullet} \sigma' \overset{def}{=} [\sigma]^{\bullet} \subseteq [\sigma']^{\bullet}$. Defined this way, compliance preorders may be employed to justify a substitution principle for servers, and more generally for services inside choreographies, namely: given two contracts σ and σ', it is safe to substitute (a service described by) σ with (a service described by) σ' as long as $\sigma \sqsubseteq^{\bullet} \sigma'$.

In this section, we give coinductive versions of the two preorders $\sqsubseteq^{\mathsf{ds}}$ and $\sqsubseteq^{\mathsf{s}}$ induced by the compliance tests introduced in the previous section. As we will show, for the deadlock-safe case, the coinductive version provides a fully abstract characterization of $\sqsubseteq^{\mathsf{ds}}$. For the safe preorder, the characterization is only sound, not complete. In both cases, being our compositions finite, and our contracts finite-state, the characterizations provide an effective construction for deciding the preorders.

4.1 Characterizing the Deadlock-Safe Preorder

We start with the deadlock safe preorder. The definition we give here, and the full-abstraction proof refine those in [13,9] to account for the symmetric nature of the compliance test.

Definition 7 (Coinductive ds-preorder). *$\mathcal{R}$ is a coinductive ds-preorder if $\sigma \mathrel{\mathcal{R}} \rho$ implies that: (i) if $\rho \longrightarrow \rho'$, then $\sigma \mathrel{\mathcal{R}} \rho'$; (ii) if $\rho \downarrow \mathrm{R}$, then there exists $\mathrm{S} \subseteq \mathrm{R}$ such that $\sigma \Downarrow \mathrm{S}$; (iii) for every action α if $\rho \xrightarrow{\alpha} \rho'$, then $\sigma \overset{\alpha}{\Longrightarrow}$ and $\sigma(\alpha) \mathrel{\mathcal{R}} \rho'$. We note $\preceq^{\mathsf{ds}}$ the greatest ds-preorder.*

[1] In the paper $\bullet$ will stand either for 'ds', defined in §3.1, or 's', defined in §3.2.

The next lemma shows that the coinductive $\mathtt{ds}$-preorder is preserved not only on single actions, but also on sequences of actions.

Lemma 8. *If $\sigma \preceq^{\mathtt{ds}} \rho$ and $\rho \stackrel{w}{\Longrightarrow} \rho'$, then $\exists\ \sigma'$ such that $\sigma \stackrel{w}{\Longrightarrow} \sigma'$ and $\sigma' \preceq^{\mathtt{ds}} \rho'$.*

Proof. By induction on the length of w. □

Theorem 9 (Soundness). *If $\sigma \preceq^{\mathtt{ds}} \rho$, then $\sigma \sqsubseteq^{\mathtt{ds}} \rho$.*

Proof. Take a composition C such that $(\sigma \parallel C)\searrow^{\mathtt{ds}}$ and let

$$\rho \parallel C = \rho_1 \parallel C_1 \longrightarrow \rho_2 \parallel C_2 \longrightarrow \cdots \longrightarrow \rho_n \parallel C_n$$

be a maximal computation from C. We must show that $(\rho_n \parallel C_n)\checkmark$. By Lemma 8, we know that $\sigma \parallel C \Longrightarrow \sigma_n \parallel C_n$ and $\sigma_n \preceq^{\mathtt{ds}} \rho_n$. Also, $\rho_n \downarrow \mathrm{R}$ for some (non-empty) R, because otherwise $\rho_n \longrightarrow$ and the computation from C we are considering would not be maximal. Let then S be such that $\sigma_n \Downarrow \mathrm{S}$ and $\mathrm{S} \subseteq \mathrm{R}$, and take σ'_n such that $\sigma_n \Longrightarrow \sigma'_n \not\longrightarrow$ and $\sigma'_n \downarrow \mathrm{S}$. Obviously $\sigma \parallel C \Longrightarrow \sigma'_n \parallel C_n$; furthermore, $C_n \not\longrightarrow$ since C_n is part of the final state of a maximal computation, and $\sigma'_n \not\longrightarrow$ by construction. Then, for all α such that $C_n \stackrel{\alpha}{\longrightarrow}$ it must be the case that $\sigma'_n \not\xrightarrow{\overline{\alpha}}$ because otherwise from $\mathrm{S} \subseteq \mathrm{R}$ we would derive $\rho_n \xrightarrow{\overline{\alpha}}$, which is impossible since $\rho_n \parallel C_n \not\longrightarrow$. It follows, then, that $\sigma \parallel C \Longrightarrow \sigma'_n \parallel C_n \not\longrightarrow$, and given that $(\sigma \parallel C)\searrow^{\mathtt{ds}}$ we have $(\sigma'_n \parallel C_n)\checkmark$. This, in turn, implies $C_n\checkmark$ and $\sigma'_n\checkmark$, and hence $\checkmark \in \mathrm{S}$. Now, from $\mathrm{S} \subseteq \mathrm{R}$ we know that $\rho_n\checkmark$, hence $(\rho_n \parallel C_n)\checkmark$ as desired. □

A further lemma shows that all contracts may be composed into at least one $\mathtt{ds}$-compliant choreography. This is a direct consequence of the syntactic structure of our contracts (all choices in a contract must either end up with $\mathbf{1}$ or with a recursion variable) and the definition of $\mathtt{ds}$-compliance.

Lemma 10. *For all σ there exists C such that $(\sigma \parallel C)\searrow^{\mathtt{ds}}$.*

Theorem 11 (Completeness). *If $\sigma \sqsubseteq^{\mathtt{ds}} \rho$, then $\sigma \preceq^{\mathtt{ds}} \rho$.*

Proof. We need to manipulate contract compositions in order to force their behavior. In particular we note $\alpha_1.C_1 + \alpha_2.C_2$ the composition such that if $\alpha_1.C_1 + \alpha_2.C_2 \stackrel{\gamma}{\Longrightarrow} C$, then $\gamma \in \{\alpha_1, \alpha_2\}$ and in addition $\gamma = \alpha_i$ implies $C \equiv C_i$ (for $i = 1, 2$, respectively). We omit the (rather lengthy) details of how these compositions can be formed so that they are determinate (in the sense of Definition 1), and move with the proof of our claim.

We show that $\mathcal{R} \stackrel{\mathit{def}}{=} \{(\sigma, \rho) \mid \sigma \sqsubseteq^{\mathtt{ds}} \rho\}$ is a coinductive $\mathtt{ds}$-preorder. Assume $(\sigma, \rho) \in\mathcal{R}$: we examine the three clauses of the definition in turn.

Assume $\rho \longrightarrow \rho'$. By our hypothesis we know that for all C $(\sigma \parallel C)\searrow^{\mathtt{ds}}$ implies $(\rho \parallel C)\searrow^{\mathtt{ds}}$, so obviously $(\rho' \parallel C)\searrow^{\mathtt{ds}}$ and thus $(\sigma, \rho') \in\mathcal{R}$.

Take R such that $\rho \downarrow \mathrm{R}$: we reason by contradiction. Assume that there is no S such that $\sigma \Downarrow \mathrm{S}$ and $\mathrm{S} \subseteq \mathrm{R}$: we show that $\sigma \not\sqsubseteq^{\mathtt{ds}} \rho$. Let then $A = \{\alpha \in \mathcal{A} \mid \sigma \Downarrow \mathrm{S} \text{ and } \alpha \in \mathrm{S} \setminus \mathrm{R}\}$. By Lemma 10, for each $\alpha \in A$ there exists C_α such that

$(\sigma(\alpha) \parallel C_\alpha)\searrow^{\mathsf{ds}}$. Now, let $C = \sum_{\alpha\in A} \overline{\alpha}.C_\alpha$. By construction, $(\rho \parallel C)\not\searrow^{\mathsf{ds}}$, and similarly $(\rho \parallel C + \mathbf{1})\not\searrow^{\mathsf{ds}}$ if $\rho\not\checkmark$. On the other hand, again by construction, one easily sees that $(\sigma \parallel C{+}\mathbf{1})\searrow^{\mathsf{ds}}$. Furthermore, when $\rho\checkmark$, by our initial assumption we know that for all S such that $\sigma \Downarrow \mathrm{S}$, it must be the case that $\mathrm{S} \supsetneq \{\checkmark\}$ (for otherwise $\mathrm{S} \subseteq \mathrm{R}$): hence, in this case, we also have $(\sigma \parallel C)\searrow^{\mathsf{ds}}$. Summarizing, when $\rho\not\checkmark$, we have $(\sigma \parallel C)\searrow^{\mathsf{ds}}$ and $(\rho \parallel C)\not\searrow^{\mathsf{ds}}$. When $\rho\checkmark$, the same is true of $C + \mathbf{1}$. In both cases we have the desired contradiction.

Let $\rho \xrightarrow{\alpha} \rho'$. Again we reason by contradiction, on the two possible cases.

- $\sigma\stackrel{\alpha}{\not\Longrightarrow}$. By Lemma 10, there exists a composition C such that $C\stackrel{\overline{\alpha}}{\not\Longrightarrow}$ and $(\sigma \parallel C)\searrow^{\mathsf{ds}}$. Now, choose a name c fresh for ρ and C, and form the composition $C' = C{+}\overline{\alpha}.\overline{c}.\mathbf{1}$. We have that $(\sigma \parallel C')\searrow^{\mathsf{ds}}$ and $(\rho \parallel C')\not\searrow^{\mathsf{ds}}$, which contradicts the hypothesis that $\sigma \sqsubseteq^{\mathsf{ds}} \rho$ as desired.
- $\sigma \stackrel{\alpha}{\Longrightarrow}$ and there exists C such that $(\sigma(\alpha) \parallel C)\searrow^{\mathsf{ds}}$ but $(\rho' \parallel C)\not\searrow^{\mathsf{ds}}$. Then we define

$$C' = \left(\sum_{\sigma\stackrel{\beta}{\Longrightarrow}\text{ and }\beta\neq\alpha} \overline{\beta}.C_\beta\right) + \overline{\alpha}.C$$

with C_β such that $(\sigma(\beta) \parallel C_\beta)\searrow^{\mathsf{ds}}$. Again, $(\sigma \parallel C')\searrow^{\mathsf{ds}}$ and $(\rho \parallel C')\not\searrow^{\mathsf{ds}}$ as desired. □

4.2 Characterizing the Safe-Preorder

The coinductive construction of the s-preorder arises as an extension of the one we just discussed. To motivate the construction, consider the following two contracts:

$$\sigma = \mathtt{rec}(\mathbf{x})\,.(a.(b.\mathbf{x} + \mathbf{1}) \oplus c.\mathbf{1}) \quad \text{and} \quad \sigma' = \mathtt{rec}(\mathbf{x})\,.(a.(b.\mathbf{x} + \mathbf{1})). \tag{1}$$

Applying Definition 7, one verifies that $\sigma \preceq^{\mathsf{ds}} \sigma'$. On the other hand, given the composition $C = \mathtt{rec}(\mathbf{x})\,.(\overline{a}.(\overline{b}.\mathbf{x}){+}\overline{c}.\mathbf{1})$ one has $(\sigma \parallel C)\searrow^{\mathsf{ds}}$ whereas $(\sigma' \parallel C)\not\searrow^{\mathsf{ds}}$.

In other words, the example in (1) shows that $\preceq^{\mathsf{ds}}$ is not sound for $\sqsubseteq^{\mathsf{s}}$. A closer look at the example shows that the problem is in the second of the clauses that define $\preceq$: in particular, to show that $\sigma \preceq^{\mathsf{ds}} \sigma'$ it is enough for σ' to match (with R) any one of the action sets S such that $\sigma \Downarrow \mathrm{S}$, disregarding the remaining ones. This is fine as long as we only look at *finite* maximal computations, as the choice of any of action sets is arbitrary and effectively excludes the others; it is unsound, instead, when the computation in σ may go back to the same point of choice, as a result of a loop, and select another set. This observation suggests how to rectify the construction, by keeping track of the states reached in the simulation game, and make a sound choice at the looping states.

A *contract-indexed* relation over contracts is a binary relation indexed by sets of contracts. We let H range over sets of contracts, and write $\sigma\ \mathcal{R}_H\ \rho$ to mean that σ and ρ are related by $\mathcal{R}$ at H.

Definition 12 (Coinductive safe-preorder). *A coinductive* $\mathtt{s}$-preorder $\mathcal{R}$ *is contract-indexed relation such that* $\sigma\ \mathcal{R}_H\ \rho$ *implies the following conditions:*

- *if* $\rho \longrightarrow \rho'$, *then* $\sigma\ \mathcal{R}_{H\cup\{\rho\}}\ \rho'$
- *if* $\rho \downarrow \mathrm{R}$, *then*
  - *if* $\rho \notin H$, *then there exists* $\mathrm{S} \subseteq \mathrm{R}$ *such that* $\sigma \Downarrow \mathrm{S}$,
  - *if* $\rho \in H$, *then for every* S *such that* $\sigma \Downarrow \mathrm{S}$ *it holds* $\mathrm{S} \subseteq \mathrm{R}$,
- *if* $\rho \xrightarrow{\alpha} \rho'$, *then* $\sigma \overset{\alpha}{\Longrightarrow}$ *and* $\sigma(\alpha)\ \mathcal{R}_{H\cup\{\rho\}}\ \rho'$

We write $\sigma\mathcal{R}\rho$ *when* $\sigma\mathcal{R}_H\rho$ *for some* H, *and note* $\preceq^{\mathtt{s}}$ *the greatest* $\mathtt{s}$*-preorder.*

Notice that the coinductive $\mathtt{s}$-preorder is a conservative extension of the corresponding $\mathtt{ds}$-preorder. The next lemma proves the same result as Lemma 8, now for the $\preceq^{\mathtt{s}}$ preorder.

Lemma 13. *Let* $\mathcal{R}$ *be a coinductive* $\mathtt{s}$*-preorder. If* $\sigma\ \mathcal{R}_H\ \rho$ *for some* H *and* $\rho \overset{w}{\Longrightarrow} \rho' \not\longrightarrow$, *then there exist* σ', H' *such that* $\sigma \overset{w}{\Longrightarrow} \sigma'$ *with* $\sigma'\ \mathcal{R}_{H'}\ \rho'$ *and* $H \subseteq H'$. *Furthermore, if* $\rho' = \rho$ *(with* w *non empty), then* $\rho' \in H'$.

Proof. We show that given $\sigma\ \mathcal{R}_H\ \rho$, if $\rho \overset{\alpha_1}{\Longrightarrow} \rho_1 \overset{\alpha_2}{\Longrightarrow} \rho_2 \cdots \rho_{n-1} \overset{\alpha_n}{\Longrightarrow} \rho_n = \rho'$, then $\sigma \overset{\alpha_1}{\Longrightarrow} \sigma_1 \overset{\alpha_2}{\Longrightarrow} \sigma_2 \cdots \sigma_{n-1} \overset{\alpha_n}{\Longrightarrow} \sigma_n$ and $\sigma_i\ \mathcal{R}_{H_i}\ \rho_i$ for $i = 1 \dots n$ with $H \subseteq H_1 \subseteq \dots \subseteq H_{n-1} \subseteq H_n$. We proceed by induction on the length n of the sequence. For the basic step, we have that if $\rho \Longrightarrow \rho'$, then $\sigma\ \mathcal{R}_{H'}\ \rho'$ with $H \subseteq H'$ by the first item of Definition 12. For the induction step, assume that $n > 0$. Then, by the induction hypothesis, $\sigma \overset{\alpha_1}{\Longrightarrow} \sigma_1 \dots \overset{\alpha_{n-1}}{\Longrightarrow} \sigma_{n-1}$ with $\sigma_i\ \mathcal{R}_{H_i}\ \rho_i$ for $i = 1 \dots n-1$ and $H \subseteq H_1 \subseteq \dots \subseteq H_{n-1}$. Now, since $\rho_{n-1} \overset{\alpha_n}{\Longrightarrow} \rho_n = \rho'$, by repeated applications of the first item, and one application of the last item in Definition 12 we obtain $\sigma_{n-1} \overset{\alpha_n}{\Longrightarrow}$ and $\sigma_{n-1}(\alpha_n)\ \mathcal{R}_{H_n}\ \rho'$ with $H_{n-1} \subseteq H_n$. We are done, as we can choose $\sigma_n = \sigma_{n-1}(\alpha_n)$.

In case $\rho' = \rho$, by Definition 12 we can immediately prove that when w is non empty $\rho \in H_1$, and thus also $\rho \in H_n$. □

Lemma 14. *Let* $\mathcal{R}$ *be a coinductive* $\mathtt{s}$*-preorder. If* $\sigma\ \mathcal{R}\ \rho$ *and* $\rho \overset{w_1}{\Longrightarrow} \rho' \overset{w_2}{\Longrightarrow} \rho'$ *with* $\rho' \not\longrightarrow$, *then (i)* $\sigma \overset{w_1,w_2}{\Longrightarrow} \sigma'$ *with* $\sigma'\ \mathcal{R}\ \rho'$ *and (ii) for every* α *such that* $\sigma' \Longrightarrow \widehat{\sigma} \xrightarrow{\alpha} \sigma''$, *then* $\rho' \xrightarrow{\alpha} \rho''$ *and* $\sigma''\ \mathcal{R}\ \rho''$. *Furthermore,* $\widehat{\sigma}\checkmark$ *implies* $\rho'\checkmark$.

Proof. Let H be an index such that $\sigma\ \mathcal{R}_H\ \rho$ for some H. Then item *(i)* is a direct consequence of Lemma 13. For item *(ii)*, Lemma 13 applied to $\rho' \overset{w_2}{\Longrightarrow} \rho'$ says that $\sigma'\ \mathcal{R}_{H'}\ \rho'$ with $\rho' \in H'$. Since $\rho' \not\longrightarrow$ we apply the second item of Definition 12. In particular $\rho' \downarrow \mathrm{R}$ with $\mathrm{S} \subseteq \mathrm{R}$ for all S such that $\sigma' \Downarrow \mathrm{S}$, and this proves the thesis. □

Lemma 15. *Let* C *be a contract composition without finite maximal computations. Then there exists* C' *such that (i)* $C \Longrightarrow C'$ *and (ii) for every* C'' *such that* $C' \Longrightarrow C''$ *then also* $C'' \Longrightarrow C'$.

Proof. Since a composition expressed as a term of our language is a finite-state system, we prove the lemma by induction on the number of the different states reachable from C. Basic step, suppose that C can reach a single state. Then the only maximal computation is $C \longrightarrow C \longrightarrow C \longrightarrow \cdots$ and the thesis is verified with $C' = C$. Induction step, assume that C can reach n different states, if $C' \Longrightarrow C$ for every C' such that $C \Longrightarrow C'$ then the thesis is verified. Otherwise, there exists C' such that $C \Longrightarrow C'$ and $C' \not\Longrightarrow C$. Then C' can reach at most $n-1$ states. Since C has no finite maximal computation, so does C'. Thus we apply the inductive hypothesis and we have: there exists C'', with $C \Longrightarrow C' \Longrightarrow C''$ such that if $C'' \Longrightarrow C'''$, then $C''' \Longrightarrow C''$. Hence we conclude the thesis. □

Corollary 16. *Let C be a contract composition without finite maximal computations. Then there exists a computation $C \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_n$ such that for every C' such that $C_n \Longrightarrow C'$ there exists $i \leq n$ such that $C' = C_i$.*

Proof. Lemma 15 says that there exists C' such that $C \Longrightarrow C'$ and, if $C_1, \ldots, C_n$ are all the states reachable from C' then $C_i \Longrightarrow C'$ for $i = 1, \ldots n$. Then consider the computation $C \Longrightarrow C' \Longrightarrow C_1 \Longrightarrow C' \Longrightarrow C_2 \Longrightarrow C' \cdots \Longrightarrow C_n \Longrightarrow C$ and conclude the thesis. □

Theorem 17 (Soundness). *If $\sigma \preceq^{\mathsf{s}} \rho$, then $\sigma \sqsubseteq^{\mathsf{s}} \rho$.*

Proof. We reason by contradiction. We assume that there exists C such that $(\sigma \parallel C)\searrow^{\mathsf{s}}$ but $(\rho \parallel C)\not\searrow^{\mathsf{s}}$, this means that *(i)* there exists a computation $\rho \parallel C \Longrightarrow \rho' \parallel C'$ such that $(\rho'' \parallel C'') \not\checkmark$ for every $\rho' \parallel C' \Longrightarrow \rho'' \parallel C''$.

If there exists a finite computation $\rho' \parallel C' \Longrightarrow \rho'' \parallel C''$ with *(ii)* $\rho'' \parallel C'' \not\Longrightarrow$ and $(\rho'' \parallel C'') \not\checkmark$, hence $\rho'' \not\longrightarrow$ and $C'' \not\longrightarrow$, moreover either $\rho'' \not\checkmark$ or $C'' \not\checkmark$. Now consider the list w of actions such that $\rho \overset{w}{\Longrightarrow} \rho''$ and $C \overset{\bar{w}}{\Longrightarrow} C''$, where $\bar{w}$ represents the list of actions performed by C to synchronize with w in order to reduce the whole composition $(\rho \parallel C)$ to $(\rho'' \parallel C'')$. From $\sigma \preceq^{\mathsf{s}} \rho$ and Lemma 13, there exist H, H' such that $\sigma\ \mathcal{R}_H\ \rho$, $\sigma'\ \mathcal{R}_{H'}\ \rho'$ with $H \subseteq H'$ and $\sigma \overset{w}{\Longrightarrow} \sigma'$. Let R such that $\rho' \downarrow \mathrm{R}$. Since $\sigma'\ \mathcal{R}_{H'}\ \rho'$, then $\sigma' \Longrightarrow \sigma'' \not\longrightarrow$ with *(iv)* $\sigma'' \downarrow \mathrm{S}$ and $\mathrm{S} \subseteq \mathrm{R}$. Hence we found the computation $\sigma \parallel C \Longrightarrow \sigma' \parallel C' \Longrightarrow \sigma'' \parallel C''$, where the first part is the synchronization on w between σ and C. Due to *(ii)*, *(iii)* and *(iv)*, then $\sigma'' \parallel C'' \not\longrightarrow$ and $\sigma'' \parallel C'' \not\checkmark$. We have $\sigma \parallel C\not\searrow^{\mathsf{s}}$, against the hypothesis.

Thus we conclude that $\rho' \parallel C'$ has no finite maximal computations. Then Corollary 16 says that $\rho' \parallel C' \longrightarrow \rho_1 \parallel C_1 \longrightarrow \cdots \longrightarrow \rho_n \parallel C_n$ and for every C^* such that $\rho_n \parallel C_n \Longrightarrow C^*$ there exists $i \leq n$ with $C^* = \rho_i \parallel C_i$. As done above, consider the list $w = w_1, w_2$ of actions such that $\rho \overset{w_1}{\Longrightarrow} \rho' \overset{w_2}{\Longrightarrow} \rho_n$ and $C \overset{\bar{w}}{\Longrightarrow} C_n$, were again $\bar{w}$ represents the list of actions performed by C to synchronize with w in order to reduce the whole composition $(\rho \parallel C)$ to $(\rho_n \parallel C_n)$. Again, by Lemma 13 *(v)* there exist H, H' such that $\sigma\ \mathcal{R}_H\ \rho$, $\sigma_n\ \mathcal{R}_{H'}\ \rho_n$ with $H \subseteq H'$ and $\sigma \overset{w}{\Longrightarrow} \sigma_n$.

We distinguish two cases. (1) If for every C^* such that $\rho_n \parallel C_n \Longrightarrow C^*$ we have $C^* = \rho_n \parallel \widehat{C}$, thus ρ_n does not perform any more action along the computation. Note that $\rho_n \not\longrightarrow$, then let R such that $\rho_n \downarrow \mathrm{R}$. Moreover, due to *(i)*, it holds $\rho_i \parallel C_i \not\checkmark$ for $i = 1 \ldots n$, hence *(vi)* either $\rho_i \not\checkmark$ or $C_i \not\checkmark$ for $i = 1 \ldots n$. Moreover, consider *(v)* and let σ_n' such that $\sigma_n \Longrightarrow \sigma_n'$ and $\sigma_n' \downarrow \mathrm{S}$

with $\textsc{s} \subseteq \textsc{r}$. Due to the assumption on $\rho_n \parallel C_n$ and *(vi)*: for every C^* such that $\sigma_n \parallel C_n \Longrightarrow C^*$ we have $C^* = \sigma_n \parallel \widehat{C}$, and also either $\sigma_n \not\checkmark$ or $C_i \not\checkmark$ for $i = 1 \ldots n$. Thus $C^* \not\checkmark$ for every C^* such that $\sigma_n \parallel C_n \Longrightarrow C^*$. We conclude that $\sigma \parallel C \not\searrow^{\mathsf{s}}$, against the hypothesis. Thus the only possible case is (2): there exists at least one $\widehat{\rho} \neq \rho_n$ such that $\rho_n \parallel C_n \Longrightarrow \widehat{\rho} \parallel \widehat{C}$ and moreover for every $\widehat{\rho}$ such that $\rho_n \parallel C_n \Longrightarrow \widehat{\rho} \parallel \widehat{C}$ it holds $\widehat{\rho} = \rho_i$ for some $i < n$. Thus for every $\widehat{\rho}$ such that $\rho_n \parallel C_n \Longrightarrow \widehat{\rho} \parallel \widehat{C}$ it holds: $\rho \overset{v_1}{\Longrightarrow} \widehat{\rho} \overset{v_2}{\Longrightarrow} \rho_n \overset{v_3}{\Longrightarrow} \widehat{\rho}$. Thanks to Lemma 14 and *(v)* we conclude the following

Fact 18. *For every $\widehat{\rho}$ such that $\rho_n \parallel C_n \Longrightarrow \widehat{\rho} \parallel \widehat{C}$ with $\rho_n \overset{\widehat{w}}{\Longrightarrow} \widehat{\rho}$ and $\widehat{\rho} \not\longrightarrow$ there exist $\widehat{\sigma}$ and H' such that $\sigma_n \overset{\widehat{w}}{\Longrightarrow} \widehat{\sigma}$ and $\sigma_n \parallel C_n \Longrightarrow \widehat{\sigma} \parallel \widehat{C}$ with $\widehat{\sigma}\, \mathcal{R}_{H'}\, \widehat{\rho}$ and for every $\widehat{\sigma} \Longrightarrow \widehat{\sigma}' \xrightarrow{\alpha} \sigma''$ then also $\widehat{\rho} \xrightarrow{\alpha} \widehat{\rho}'$ and $\widehat{\sigma}''\, \mathcal{R}_{H''}\, \widehat{\rho}'$ for some H''. Furthermore $\widehat{\sigma}'' \checkmark$ implies $\widehat{\rho}' \checkmark$.*

Since $(\sigma \parallel C \searrow^{\mathsf{s}})$ there exists a computation $\sigma_n \parallel C_n \Longrightarrow \sigma^* \parallel C^*$ such that $(\sigma^* \parallel C^*) \checkmark$. In particular $\sigma_n \overset{w^*}{\Longrightarrow} \sigma^*$. If the sequence w^* is empty, then Fact 18 says that also $\widehat{\rho} \checkmark$. Hence $\rho_n \parallel C_n \Longrightarrow \rho_n \parallel C^*$ and $(\rho_n \parallel C^*) \checkmark$. In case w^* is not empty, we prove that also

$$\rho_n \overset{w^*}{\Longrightarrow} \rho^* \text{ with } \sigma^*\, \mathcal{R}_{H^*}\, \rho^*. \tag{2}$$

We proceed by induction on the length of w. Fact 18 gives the basis of the induction. For the induction step, let $w = \alpha, w'$. Then $\sigma_n \Longrightarrow \sigma_n' \xrightarrow{\alpha} \sigma''$ and Fact 18 says that $\rho_n \xrightarrow{\alpha} \rho_n'$ with $\sigma_n''\, \mathcal{R}_{H''}\, \rho_n'$. Now the induction hypothesis holds for σ_n'' and ρ_n', and we are done. Now, from (2), it follows that $\rho_n \parallel C_n \Longrightarrow \rho^* \parallel C^*$ and the fact that $\sigma^*\, \mathcal{R}_{H^*}\, \rho^*$ says that $(\rho^* \parallel C^*) \checkmark$. Hence we found a contradiction and we conclude $(\rho \parallel C) \searrow^{\mathsf{s}}$. □

The converse of Theorem 17 does not hold. Here is a counter-example. Let:

$$\sigma = \mathtt{rec}(\mathbf{x})\,(a.\mathbf{x} \oplus b.\mathbf{1}) \qquad \rho = \mathtt{rec}(\mathbf{x})\,(\,a.(a.\mathbf{x} \oplus b.\mathbf{1})\,) \tag{3}$$

We can easily see that $\sigma \sqsubseteq^{\mathsf{s}} \rho$, because $\sigma \overset{w}{\Longrightarrow}$ for all w such that $\rho \overset{w}{\Longrightarrow}$. In fact, given an arbitrary C, C must be able to respond to all of these traces, with a corresponding dual trace in $\sigma \parallel C$. Given our observation, the same is true in the composition $\rho \parallel C$. On the other hand, $\sigma \not\preceq^{\mathsf{s}} \rho$ because, after one round of the loop, $\rho \downarrow \{a\}$, whereas $\sigma \Downarrow \{b\}$ and $\{b\} \not\subseteq \{a\}$, which breaks the condition required by the coinductive game. The problem would seemingly be solved by replacing the third condition with the following, slightly weaker requirement: if $\rho \xrightarrow{\alpha} \rho'$, then there exists σ' such that $\sigma \overset{\alpha}{\Longrightarrow} \sigma'$ and $\sigma'\, \mathcal{R}_{H \cup \{\rho\}}\, \rho'$. On the other hand, this coarser definition is unsound: taking the two contracts we discussed earlier in (1), one verifies that $\sigma \preceq^{\mathsf{s}} \sigma'$ even though $\sigma \not\sqsubseteq^{\mathsf{ds}} \sigma'$, hence $\sigma \not\sqsubseteq^{\mathsf{s}} \sigma'$.

As it turns out, characterizing the safe preorder exactly, is hard. Indeed, to our knowledge, no such characterization exists in the literature. The only related result we are aware of is the trace-based full-abstract characterization of *should* testing in [17]. However, the construction is non-effective, as it requires infinite sums to characterize the infinite traces that capture the possible test processes.

5 Filtered Preorders

As the last step of our analysis, we show how the *filters* introduced in [8,9] can be recast into the setting of general service compositions to achieve an expressive compliance-preserving substitution principle inside choreographies. Following [8,9], we define filters as behavioral coercions that specify the legal flow of actions for individual contracts. Their syntax is defined by the following productions:

$$f \;:=\; \mathbf{0} \mid \alpha.f \mid f \times f \mid f \otimes f \mid \mathtt{x} \mid \mathtt{rec}(\mathtt{x})\, f.$$

Their semantics, in Table 2, is best understood by viewing a filter as a finite-state automaton accepting possibly infinite strings of actions, with × and ⊗ noting the intersection and union automata. Then, applying a filter f to a contract σ, as in $f(\sigma)$, corresponds to verify that the sequence of visible transitions made by the contract σ forms a string of the filter's language.

Table 2. Dynamics of Filtered Contracts

Transitions for filters

$$\alpha.f \xmapsto{\alpha} f \qquad \frac{f\{\mathtt{x} := \mathtt{rec}(\mathtt{x})\, f\} \xmapsto{\alpha} f'}{\mathtt{rec}(\mathtt{x})\, f \xmapsto{\alpha} f'} \qquad \frac{f \xmapsto{\alpha} f_\alpha \qquad g \xmapsto{\alpha} g_\alpha}{f \otimes g \xmapsto{\alpha} f_\alpha \otimes g_\alpha}$$

$$\frac{f \xmapsto{\alpha} f_\alpha \qquad g \xmapsto{\alpha} g_\alpha}{f \times g \xmapsto{\alpha} f_\alpha \times g_\alpha} \qquad \frac{f \xmapsto{\alpha} f_\alpha \qquad g \not\xmapsto{\alpha}}{f \times g \xmapsto{\alpha} f_\alpha} \qquad \frac{f \not\xmapsto{\alpha} \qquad g \xmapsto{\alpha} g_\alpha}{f \times g \xmapsto{\alpha} g_\alpha}$$

Transitions for filtered contracts

$$\frac{\sigma \xrightarrow{\alpha} \sigma' \qquad f \xmapsto{\alpha} f'}{f(\sigma) \xrightarrow{\alpha} f'(\sigma')} \qquad \frac{\sigma \longrightarrow \sigma'}{f(\sigma) \longrightarrow f(\sigma')} \qquad \frac{\sigma \checkmark}{f(\sigma) \checkmark}$$

In their original, client-server formulation by [8,9], filters generalize the notion of *contract interface* introduced in [13]. Like filters, interfaces are intended to constrain the behavior of contracts, by defining the set of actions that a contract may legally engage in. However, while with filters this set may vary dynamically as the contract unfolds, with interfaces the set is determined statically and does not change over time. In addition, filters also play a role in strengthening the substitution principle based on compliance preorders. Specifically, given any compliance preorder $\leq^{\bullet}$ (whether coinductive or not), one defines a corresponding filtered preorder as the following relation: $\sigma \leq^{\mathtt{F}\bullet} \rho$ if there exists a filter f such that $\sigma \leq^{\bullet} f(\rho)$. Thus, even when $\sigma \not\leq^{\bullet} \rho$, one may still rely on a filter f to justify the replacement of σ with $f(\rho)$, provided that $\sigma \leq^{\bullet} f(\rho)$. In a client-server setting,. the filtered preorder of [8,9] generalizes the interface-indexed preorder of [13], which relates contracts based on a their ability to comply with clients that follow the discipline imposed by the indexing interface, disregarding all clients that do not follow that discipline.

Our present use of filters provides corresponding generalizations of the concepts of *input-output sets* and *input-output indexed preorder* by [3,4] that parallel the notions of interface and interface indexed preorder in the analysis of multiparty compositions. In [4], the authors provide an effective decision procedure for their preorder based on the theory of should-testing [17]. In the rest of this section, we provide an effective construction for the filtered version of the coinductive safe-preorder. The same construction can be given, *mutatis mutandis*, for the deadlock-safe preorder.

Definition 19 (filtered $\mathtt{s}$-preorder). *A filtered* $\mathtt{s}$*-preorder is a contract indexed relation* $\mathcal{F}$ *such that if* $\sigma \mathcal{F}_H \rho$*, then*

1. *if* $\rho \longrightarrow \rho'$*, then* $\sigma \mathcal{F}_{H'} \rho'$ *with* $H' = H \cup \{\rho\}$*,*
2. *else if* $\rho \downarrow \mathrm{R}$*, then*
   (a) *if* $\rho \notin H$*, then there exists* $\mathrm{S}_{\mathrm{R}} \subseteq \mathrm{R}$ *such that* $\sigma \Downarrow \mathrm{S}_{\mathrm{R}}$*, and for every* $\alpha \in \mathrm{S}_{\mathrm{R}}$ *it is the case that* $\rho \xrightarrow{\alpha} \rho'$ *and* $\sigma \overset{\alpha}{\Longrightarrow}$ *with* $\sigma(\alpha) \mathcal{F}_{H'} \rho'$ *and* $H' = H \cup \{\rho\}$*.*
   (b) *if* $\rho \in H$*, then for every* S *such that* $\sigma \Downarrow \mathrm{S}$ *it holds* $\mathrm{S} \subseteq \mathrm{R}$*, and for every action* $\alpha \in \bigcup_{\sigma \Downarrow \mathrm{S}} \mathrm{S}$ *if* $\rho \xrightarrow{\alpha} \rho'$*, then* $\sigma \overset{\alpha}{\Longrightarrow}$ *with* $\sigma(\alpha) \mathcal{F}_{H'} \rho'$ *and* $H' = H \cup \{\rho\}$*.*

We write $\sigma \mathcal{F} \rho$ *whenever* $\sigma \mathcal{F}_H \rho$ *for some* H*, and note* $\sigma \preceq^{\mathtt{Fs}} \rho$ *the greatest filtered* $\mathtt{s}$*-preorder.*

Theorem 20. $\sigma \preceq^{\mathtt{Fs}} \rho$ *iff there exists a filter* f *such that* $\sigma \preceq^{\mathtt{s}} f(\rho)$*.*

Proof. Define *contract bisimilarity*, noted $\sim$, is the greatest symmetric relation such that $\sigma \sim \rho$ implies (i) $\sigma \checkmark$ iff $\rho \checkmark$, and (ii) if $\sigma \xrightarrow{\dot{\alpha}} \sigma'$, then also $\rho \xrightarrow{\dot{\alpha}} \rho'$ and $\sigma' \sim \rho'$. Clearly, $\sim \subseteq \preceq^{\mathtt{F}\bullet}$.

We proceed with the proof of the theorem, in the two directions in turn.
$(\Longrightarrow)$ Take a filtered $\mathtt{s}$-preorder $\mathcal{F}$. For every H and $(\sigma, \rho) \in \mathcal{F}_H$ we define

$$Set_H(\sigma, \rho) \overset{def}{=} \begin{cases} \bigcup_{\rho \Downarrow \mathrm{R}} \mathrm{S}_R & \text{if } \rho \notin H \text{ and } \mathrm{S}_R \text{ is given by item } 2.a \text{ of Definition 19} \\ \bigcup_{\sigma \Downarrow \mathrm{S}} \mathrm{S} & \text{if } \rho \in H \end{cases}$$

Then, given a set D of pairs of contracts, we define

$$f^D_{\sigma,\rho,H} \overset{def}{=} \begin{cases} \mathtt{rec}(\mathbf{x}_{(\sigma,\rho)}) \ \times_{\alpha \in Set_H(\sigma,\rho)} \alpha . f^{D \cup \{(\sigma,\rho)\}}_{\sigma(\alpha),\rho(\alpha),H \cup \{\rho\}} & \text{if } (\sigma,\rho) \notin D \\ \mathbf{x}_{(\sigma,\rho)} & \text{otherwise} \end{cases}$$

and let $f_{\sigma,\rho,H} = f^{\emptyset}_{\sigma,\rho,H}$. Since the reachable states are finite, f is well defined. Furthermore note that, if $f_{\sigma,\rho,H}(\rho) \overset{\alpha}{\Longrightarrow} f'(\rho')$ and $\sigma \overset{\alpha}{\Longrightarrow}$, then $f'(\rho') \sim f_{\sigma(\alpha),\rho',H \cup \{\rho\}}(\rho')$. Finally we define the following contract indexed relation:

$$\mathcal{R}_H \overset{def}{=} \{(\sigma, f(\rho)) \, : \, (\sigma, \rho) \in \mathcal{F}_H \text{ and } f(\rho) \sim f_{\sigma,\rho,H}(\rho)\}$$

We prove that $\mathcal{R}$ is a coinductive $\mathtt{s}$-preorder, by a case analysis of the items in the Definition 12. Given a filter f, and a set $\mathrm{R} \subseteq \mathcal{A} \cup \{\checkmark\}$, let $\mathrm{R}_{|f}$ note the set $\{\alpha \in \mathrm{R} \mid f \xmapsto{\alpha}\}$. Let then $(\sigma, f(\rho)) \in \mathcal{R}_H$.

If $\rho \longrightarrow \rho'$, then also $f(\rho) \longrightarrow f(\rho')$ and we have $(\sigma, \rho') \in \mathcal{F}_{H'}$ with $H' = H \cup \{\rho\}$, so $(\sigma, f(\rho')) \in\mathcal{R}_{H'}$. If instead $\rho \downarrow \mathrm{R}$, we distinguishes two cases. *(i)* If $\rho \notin H$, then, since $(\sigma, \rho) \in \mathcal{F}_H$, there exists $\mathrm{S}_R \subseteq \mathrm{R}$ such that $\sigma \Downarrow \mathrm{S}_\mathrm{R}$ and by definition we have $\mathrm{S}_\mathrm{R} \subseteq \mathit{Set}_H(\sigma, \rho)$; so $\mathrm{S}_\mathrm{R} \subseteq \mathrm{R}_{|(f_{\sigma,\rho,H})}$. *(ii)* If $\rho \in H$, then, since $(\sigma, \rho) \in \mathcal{F}_H$, for every S such that $\sigma \Downarrow \mathrm{S}$ it holds $\mathrm{S} \subseteq \mathrm{R}$ and also $\mathrm{S} \subseteq \mathit{Set}_H(\sigma, \rho)$, hence $\mathrm{S} \subseteq \mathrm{R}_{|(f_{\sigma,\rho,H})}$.

Assume now $f_{\sigma,\rho,H}(\rho) \xrightarrow{\alpha}$. Then there exists ρ' such that $\rho \xrightarrow{\alpha} \rho'$, so $f_{\sigma,\rho}(\rho) \xrightarrow{\alpha} f'(\rho')$, where $f'(\rho') \sim f_{\sigma',\rho',H'}(\rho')$ with $\sigma' = \sigma(\alpha)$ and $H' = H \cup \rho$. Since $f_{\sigma,\rho,H} \xmapsto{\alpha}$ then $\sigma \xRightarrow{\alpha}$ with $(\sigma(\alpha), \rho') \in \mathcal{F}_{H'}$ and $H' = H \cup \{\rho\}$. We conclude that $(\sigma(\alpha), f'(\rho')) \in \mathcal{R}_{H'}$.

($\Longleftarrow$) Let $\mathcal{R}$ be a coinductive $\mathtt{s}$-preorder. Given H, we define

$$\mathcal{F}_H \stackrel{\mathit{def}}{=} \{(\sigma, \rho) \,|\, \text{there exists } f \text{ such that } (\sigma, f(\rho)) \in\mathcal{R}_{f(H)}\}$$

where $f(H) = \{f(\rho) \,:\, \rho \in H\}$. We show that $\mathcal{F}$ is a filtered $\mathtt{s}$-preorder. Take $(\sigma, \rho) \in \mathcal{F}_H$, i.e. $(\sigma, f(\rho)) \in\mathcal{R}_{f(H)}$.

If $\rho \longrightarrow \rho'$, then $f(\rho) \longrightarrow f'(\rho')$ and $(\sigma, f'(\rho')) \in\mathcal{R}_{f(H')}$ with $H' = H \cup \{\rho\}$, hence $(\sigma, \rho') \in \mathcal{F}_{H'}$. If instead $\rho \downarrow \mathrm{R}$, we distinguishes two cases. *(i)* If $\rho \notin H$, then also $f(\rho) \notin f(H)$ hence there exists S such that $\mathrm{S} \subseteq \mathrm{R}_{|f}$. Since $\mathrm{R}_{|f} \subseteq \mathrm{R}$ we have $\mathrm{S} \subseteq \mathrm{R}$. Take now $\alpha \in \mathrm{S}$, then $\alpha \in \mathrm{R}_{|f}$ so $f(\rho) \xrightarrow{\alpha} f'(\rho')$. Now, from $(\sigma, f(\rho)) \in\mathcal{R}_{f(H)}$ we know that $(\sigma(\alpha), f'(\rho')) \in\mathcal{R}_{f(H')}$ with $H' = H \cup \{f(\rho)\}$, so $(\sigma(\alpha), \rho') \in \mathcal{F}_{H'}$. (*(ii)* If $\rho \in H$, then also $f(\rho) \in f(H)$. Now, for every S such that $\sigma \Downarrow \mathrm{S}$ it holds $\mathrm{S} \subseteq \mathrm{R}_{|f}$, hence $\mathrm{S} \subseteq \mathrm{R}_{|f}$ and for every action $\alpha \in \bigcup_{\sigma \Downarrow \mathrm{S}} \mathrm{S}$ if $f(\rho) \xrightarrow{\alpha} f(\rho')$, then $\sigma \xRightarrow{\alpha}$ with $(\sigma(\alpha), f(\rho')) \in \mathcal{R}_{f(H')}$ and $H' = H \cup \{\rho\}$, hence also $(\sigma(\alpha), \rho') \in \mathcal{F}_{H'}$. □

6 Conclusion

We have developed a formal framework for the analysis of different theories of compliance in the literature. Besides investigating the relationships between the existing definitions of compliance, we have also shown how to obtain compliance preorders for multiparty service compositions, by recasting and generalizing the theory of behavioral coercions from [8,9] to this setting. Our present endeavor continues on the line of work we initiated in [2]. There, we used filters to provide a new solution to the problem of web service adaptation within service compositions [18,1,10]. Specifically, we showed how filters may be employed as adapters to enforce the compliance of a choreography, by blocking the transition paths in all the components that may get the choreography stuck or trapped into a livelock. Here, our focus has been on providing effective techniques for the construction of expressive compliance preorders supporting contract replacement. Collectively, the resulting theory constitutes an elegant support for a formal analysis of component/service compliance, adaptation and replacement inside choreographies.

Acknowledgements. We gratefully acknowledge comments from the anonymous referees.

References

1. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic service composition based on behavioral descriptions. Int. J. Cooperative Inf. Syst. 14(4), 333–376 (2005)
2. Bernardi, G., Bugliesi, M., Macedonio, D., Rossi, S.: A theory of adaptable contract-based service composition. In: Global Comp. IEEE Computer Society, Los Alamitos (2008)
3. Bravetti, M., Zavattaro, G.: Contract based multi-party service composition. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 207–222. Springer, Heidelberg (2007)
4. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007)
5. Caires, L., Vieira, H.T.: Conversation types. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 285–300. Springer, Heidelberg (2009)
6. 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)
7. Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A formal account of contracts for web services. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 148–162. Springer, Heidelberg (2006)
8. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. In: POPL 2008, pp. 261–272. ACM Press, New York (2008)
9. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Transactions on Programming Languages and Systems (to appear, 2009)
10. De Giacomo, G., Sardiña, S.: Automatic synthesis of new behaviors from a library of available behaviors. In: Veloso, M.M. (ed.) IJCAI, pp. 1866–1871 (2007)
11. Honda, K., Vasconcelos, V., 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)
12. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL 2008, pp. 273–284. ACM Press, New York (2008)
13. Laneve, C., Padovani, L.: The *must* preorder revisited. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 212–225. Springer, Heidelberg (2007)
14. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1989)
15. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science 34, 83–133 (1984)
16. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Mathematical Structures in Computer Science 6(5), 409–453 (1996)
17. Rensink, A., Vogler, W.: Fair testing. Information and Computation 205(2), 125–198 (2007)
18. Traverso, P., Pistore, M.: Automated composition of semantic web services into executable processes. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 380–394. Springer, Heidelberg (2004)

A Formal Semantics for the WS-BPEL Recovery Framework

The π-Calculus Way

Nicola Dragoni[1] and Manuel Mazzara[2]

[1] DTU Informatics, Technical University of Denmark, Denmark
ndra@imm.dtu.dk
[2] School of Computing Science, Newcastle University, UK
manuel.mazzara@newcastle.ac.uk

Abstract. While current studies on Web services composition are mostly focused — from the technical viewpoint — on standards and protocols, this work investigates the adoption of formal methods for dependable composition. The Web Services Business Process Execution Language (WS-BPEL) — an OASIS standard widely adopted both in academic and industrial environments — is considered as a touchstone for concrete composition languages and an analysis of its ambiguous Recovery Framework specification is offered. In order to show the use of formal methods, a precise and unambiguous description of its (simplified) mechanisms is provided by means of a conservative extension of the π-calculus. This has to be intended as a well known case study providing methodological arguments for the adoption of formal methods in software specification. The aspect of verification is not the main topic of the paper but some hints are given.

1 Introduction

Service Oriented Architectures and the related paradigm are modern attempts to cope with old problems connected to Business-to-Business (B2B) and information interchange. Many implementations of this paradigm are possible and the so called Web services look to be the most prominent, mainly because the underlying architecture is already there; it is simply the web which has been extensively used in the last 15 years and where we can easily exploit HTTP [21], XML [5], SOAP [8] and WSDL [3]. The World Wide Web provides a basic platform for the interconnection on a point-to-point basis of different companies and customers but one of the B2B complications is the management of causal interactions between different services and the way in which the messages between them need to be handled (e.g., not always in a sequential way). This area of investigation is called composition, *i.e.,* the way to build complex services out of simpler ones [4]. These days, the need for workflow technology is becoming quite evident and the positive aspect is that we had investigated this technology for decades and we also have excellent modeling tools providing verification features that are grounded in the very active field of concurrency theory research.

C. Laneve and J. Su (Eds.): WS-FM 2009, LNCS 6194, pp. 92–109, 2010.

1.1 BPEL and Its Ambiguous Specification

Several organizations worked on composition proposals. The most important in the past have been IBM's WSFL [1] and Microsoft's XLANG [2]. These two have then converged into Web Services Business Process Execution Language [18] (BPEL for short) which is presently an OASIS standard and, given its wide adoption, it will be used as a touchstone for composition languages in this paper. BPEL allows workflow-based composition of services. In the committee members' words the aim is *"enabling users to describe business process activities as Web services and define how they can be connected to accomplish specific tasks"*. The problem with BPEL was that the earlier versions of the language were not very clear, the specification was huge and many points confusing, especially in relation to the Recovery Framework (RF) and the interactions between different mechanisms (fault handlers and compensation handlers). BPEL indeed represents a business tradeoff where not necessarily all the single technical choices have been made considering all the available options. Although in the final version of the specification (which is lighter and cleaner) fault handling during compensation has been simplified, we strongly believe that the sophisticated mechanism of recovery still needs a clarification.

1.2 Contribution of the Paper

In this paper we aim to reduce this ambiguity providing an *easily readable formal semantics* of the BPEL Recovery Framework (BPEL RF for short). This goal requires at least two different contributions:

1. a *formal semantics* of the framework, focusing on its essential mechanisms
2. an *easily readable* specification of these mechanisms

We provide both contributions following a "π-calculus way", that is using the π-calculus as formal specification language. It is worth noting that here the actual challenge is to provide not only a formal semantics for the BPEL RF but also an *easily readable* specification. Indeed, other attempts might be found in literature providing the first contribution only. For instance, in [12] such encoding has been proposed by one of the authors. However, one of the unsatisfactory aspects about that encoding is that it is hardly readable and complex. The actual challenge here is to reduce such complexity while keeping a formal and rigorous approach. As a result, in this paper we contribute with a better understanding of how the BPEL RF works. Moreover, the case study allows us to show the real power of the $\texttt{web}\pi_\infty$ calculus (*i.e.,* the π-calculus based formal language exploited for the mentioned encoding) not only in terms of simplicity of the resulting BPEL specification, but also sketching how $\texttt{web}\pi_\infty$ can contribute to the implementation of real orchestration engines.

Finally, we would like to stress that different formal models might be chosen for this goal. As discussed in the next section, our choice is primarily motivated by the "foundational feature" of the π-calculus, namely *mobility*, i.e. the possibility of transmitting channel names that will be, in turn, used by any receiving process. It is worth noting that in the specific contribution of this paper this feature is not really exploited or totally necessary since the modeled mechanisms requested us to pay more attention to process synchronization and concurrency than to full mobility. Anyway, in the general case, we have the strong opinion that mobility is an essential feature that composition languages should exhibit [13]. This aspect will be better discussed in section 2.

Outline. The paper is organized as follows. Section 2 will discuss the rationale behind our "π-calculus way" choice, briefly motivating why the π-calculus could be considered a formal foundation for dependable Web services composition. Section 3 will present $\texttt{web}\pi_\infty$ discussing its syntax and semantics. Sections 4 and 5 will contribute with a clarification of the BPEL RF semantics. In particular, Section 4 will show how it appears in the original (ambiguous) specification, and Section 5 will propose the actual simplification and formal specification. Section 6 will add some conclusive remarks.

2 The π-Calculus Way to Dependable Composition

The need for formal foundation has been discussed widely in the last years, although many attempts to use formal methods in this setting have been speculative. Some communities, for example, criticized the process algebra options [19] promoting the Petri nets choice. The question here is whether we need a formal foundation and, if that is the case, which kind of formalism we need. While sequential computation has well established foundations in the λ-calculus and Turing machines, when it comes to concurrency things are far from being settled. The π-calculus ([17] and [16]) emerged during the eighties as a theory of mobile systems providing a conceptual framework for expressing them and reasoning about their behavior. It introduces mobility generalizing the channel-based communication of CCS by allowing channels to be passed as data through rendezvous over other channels. In other words, it is a model for prescribing (specification) and describing (analysis) concurrent systems consisting of agents which mutually interact and in which the communication structure can dynamically evolve during the execution of processes. Here, a communication topology is intended as the linkage between processes which indicates who can communicate with whom. Thus, changing the communication links means, for a process, moving inside this abstract space of linked entities.

A symmetry between λ-calculus and π-calculus could be suggested and the option to build concurrent languages (and so workflow languages as well) on a formal basis could actually make sense. It has indeed been investigated in many works, even in the BPEL context. But, while formal methods are expected to bring mathematical precision to the development of computer systems (providing precise notations for specification and verification), so far BPEL — despite having been subject of a number of formalizations (for example [10], [7] and [22]) — has not yet been proved to be built on an exact and specific mathematical model, including process algebras (this argument has been carefully developed in [13]). Thus, we do not have any conceptual and software tools for analysis, reasoning and software verification. If we are not able to provide this kind of tools, any hype about mathematical rigor becomes pointless.

It is also worth noting that, although many papers use the term π-calculus and process algebra interchangeably, there is a difference between them. Algebra is a mathematical structure with a set of values and a set of operations on the values. These operations enjoy algebraic properties such as commutativity, associativity, idempotency, and distributivity. In a typical process algebra, processes are values and parallel composition is defined to be a commutative and associative operation on processes. The π-calculus is an algebra but it differs from previous models for concurrency precisely for the fact that it includes a notion of mobility, i.e. the possibility of transmitting channel names that will be, in turn, used by receiving processes. This allows a sort of dynamic reconfiguration with the possibility of creating (and deleting) processes through the alteration of the process topology (although it can be argued that, even if the link to a process disappears, the process itself disappears only from "an external point of view").

The π-calculus looks interesting because of its treatment of component bindings as first class objects, which enables this dynamic reconfiguration to be expressed simply. So, the question now is: do we need this additional feature of the π-calculus or should we restrict our choice to models, like CCS, without this notion of mobility? Why all this hype over the π-calculus and such a rare focus on its crucial characteristic? We have the strong opinion that mobility is an essential feature that composition languages should exhibit. Indeed, while in some scenarios services can be selected already at design-time, in others some services might only be selected at runtime and this selection has then to be propagated to different parties. This phenomenon is called link passing mobility and it is properly approached in [6].

It is worth noting that in the specific contribution of this paper this feature is not really exploited or totally necessary since the modeled mechanisms requested we pay more attention to process synchronization and concurrency than to full mobility. This aspect has been instead essential in the full formalization of BPEL. In [13] it has been shown how it plays an important role in the encoding of interactions of the kind request-response. Indeed, in that case the invoker must send a channel name to be used then to return the response. This is a typical case of the so called output capability of the π-calculus, i.e. a received name is used as the subject of outputs only. The full input capability of the π-calculus — i.e. when a received name is used also as the subjects of inputs — has been not exploited in the BPEL encoding (and neither it is in this work). Indeed in [13] a specific well-formedness constraint imposes that *"received names cannot be used as subjects of inputs or of replicated inputs"*. Thus, at the present moment we remain agnostic regarding the need of the π-calculus input capability in the description of BPEL mechanism. We realize that this admission could be an argument for discussing again the choice of the original model.

2.1 Our Approach

WS-standards for dependability only concerns SOAP when employed as an XML messaging protocol (e.g. OASIS WS-Reliability and WS-Security), *i.e.,* at the message level. However, things are more complicated than this since loosely coupled components like Web services, being autonomous in their decisions, may refuse requests or suspend their functionality without notice, thus making their behavior unreliable to other activities. Henceforth, most of the web languages also include the notion of loosely coupled transaction – called *web transaction* [11] in the following – as a unit of work involving loosely coupled activities that may last long periods of time. These transactions, being orthogonal to administrative domains, have the typical atomicity and isolation properties relaxed, and instead of assuming a perfect roll-back in case of failure, support the explicit programming of compensation activities. Web transactions usually contain the description of three processes: *body*, *failure handler*, and *compensation.* The failure handler is responsible for reacting to events that occur during the execution of the body; when these events occur, the body is blocked and the failure handler is activated. The compensation, on the contrary, is installed when the body commits; it remains available for outer transactions to require some undo of previously performed actions. BPEL also uses this approach.

Our approach to recovery is instead described in [13], where it has been shown that different mechanisms for error handling are not necessary and the BPEL semantics has been presented in terms of $\mathtt{web}\pi_\infty$, which is based on the idea of event notification as the unique error handling mechanism. This result allows us to extend any semantic considerations about $\mathtt{web}\pi_\infty$ to BPEL. $\mathtt{web}\pi_\infty$ (originally in [14]) has been introduced to investigate how process algebras can be used as a foundation in this context. It is a simple and conservative extension of the π-calculus where the original algebra is augmented with an operator for asynchronous events raising and catching in order to enable the programming of widely accepted error handling techniques (such as long running transactions and compensations) with reasonable simplicity. We addressed the problem of composing services starting directly from the π-calculus and considering this proposal as a foundational model for composition simply to verify statements regarding any mathematical foundations of composition languages and not to say that the π-calculus is more suitable than other models (such as Petri nets) for these purposes. The calculus is presented in detail in section 3 while in section 4 and 5 it is showed how it can be useful to clarify the BPEL RF semantics.

3 The Composition Calculus

In this section we present a proposal to cope with the issues presented in section 2. Although $\mathtt{web}\pi_\infty$ is ambitious, for sure we do not pretend to solve all the problems and to give the ultimate answer to all the questions. Giving all the details about the language and its theory is beyond the scope of this paper which is giving a brief account about how $\mathtt{web}\pi_\infty$ can be considered in the overall scenario of formal methods for dependable Web services. You can find all the relevant details in some previous work, especially in [12], [13] and [15].

3.1 Syntax

The syntax of $\texttt{web}\pi_\infty$ *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.

$$
\begin{array}{lll}
P \ ::= & & \\
 & \mathbf{0} & (\textbf{nil}) \\
 & \mid \overline{x}\,\widetilde{u} & (\textbf{output}) \\
 & \mid \sum_{i \in I} x_i(\widetilde{u_i}).P_i & (\textbf{alternative composition}) \\
 & \mid (x)P & (\textbf{restriction}) \\
 & \mid P \mid P & (\textbf{parallel composition}) \\
 & \mid\ !x(\widetilde{u}).P & (\textbf{guarded replication}) \\
 & \mid \langle\!| P \ ; \ P |\!\rangle_x & (\textbf{workunit})
\end{array}
$$

A process can be the inert process $\mathbf{0}$, an output $\overline{x}\,\widetilde{u}$ sent on a name x that carries a tuple of names $\widetilde{u}$, an alternative composition consisting of input guarded processes that consumes a message $\overline{x_i}\,\widetilde{w_i}$ and behaves like $P_i\{\widetilde{w_i}/\widetilde{u_i}\}$, a restriction $(x)P$ that behaves as P except that inputs and messages on x are prohibited, a parallel composition of processes, a replicated input $!x(\widetilde{u}).P$ that consumes a message $\overline{x}\,\widetilde{w}$ and behaves like $P\{\widetilde{w}/\widetilde{u}\} \mid\ !x(\widetilde{u}).P$, or a workunit $\langle\!| P \ ; \ Q |\!\rangle_x$ that behaves as the *body* P until an abort $\overline{x}$ is received and then behaves as the *event handler* Q.

Names x in outputs, inputs, and replicated inputs are called *subjects* of outputs, inputs, and replicated inputs, respectively. It is worth to notice that the syntax of $\texttt{web}\pi_\infty$ processes simply augments the asynchronous π-calculus with workunit process. The input $x(\widetilde{u}).P$, restriction $(x)P$ and replicated input $!x(\widetilde{u}).P$ are binders of names $\widetilde{u}$, x and $\widetilde{u}$ respectively. The scope of these binders is the process P. We use the standard notions of α-equivalence, *free* and *bound names* of processes, noted $\mathrm{fn}(P)$, $\mathrm{bn}(P)$ respectively.

3.2 Semantics

We give the semantics for the language in two steps, following the approach of Milner [16], separating the laws that govern the static relations between processes from the laws that rule their interactions. The first step is defining a static structural congruence relation over syntactic processes. A structural congruence relation for processes equates all agents we do not want to distinguish. It is introduced as a small collection of axioms that allow minor manipulation on the processes' structure. This relation is intended to express some intrinsic meanings of the operators, for example the fact that parallel is commutative. The second step is defining the way in which processes evolve dynamically by means of an operational semantics. This way we simplify the statement of the semantics just closing with respect to $\equiv$, *i.e.,* closing under process order manipulation induced by structural congruence.

Definition 1. *The* structural congruence $\equiv$ *is the least congruence satisfying the Abelian Monoid laws for parallel and summation (associativity, commutativity and* $\mathbf{0}$ *as identity) closed with respect to* α*-renaming and the following axioms:*

1. *Scope laws:*
$$(u)\mathbf{0} \equiv \mathbf{0}, \qquad (u)(v)P \equiv (v)(u)P,$$
$$P \mid (u)Q \equiv (u)(P \mid Q)\ , \quad \textit{if } u \notin \mathrm{fn}(P)$$
$$\llparenthesis (z)P \ ;\ Q \rrparenthesis_x \equiv (z)\llparenthesis P \ ;\ Q \rrparenthesis_x\ , \quad \textit{if } z \notin \{x\} \cup \mathrm{fn}(Q)$$
2. *Workunit laws:*
$$\llparenthesis \mathbf{0} \ ;\ Q \rrparenthesis_x \equiv \mathbf{0}$$
$$\llparenthesis \llparenthesis P \ ;\ Q \rrparenthesis_y \mid R \ ;\ R' \rrparenthesis_x \equiv \llparenthesis P \ ;\ Q \rrparenthesis_y \mid \llparenthesis R \ ;\ R' \rrparenthesis_x$$
3. *Floating law:*
$$\llparenthesis \overline{z}\,\widetilde{u} \mid P \ ;\ Q \rrparenthesis_x \equiv \overline{z}\,\widetilde{u} \mid \llparenthesis P \ ;\ Q \rrparenthesis_x$$

The scope laws are standard while novelties regard workunit and floating laws. The law $\llparenthesis \mathbf{0} \ ;\ Q \rrparenthesis_x \equiv \mathbf{0}$ 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 $\llparenthesis \llparenthesis P \ ;\ Q \rrparenthesis_y \mid R \ ;\ R' \rrparenthesis_x \equiv \llparenthesis P \ ;\ Q \rrparenthesis_y \mid \llparenthesis R \ ;\ R' \rrparenthesis_x$ moves workunit outside parents, thus flattening the nesting. Notwithstanding this flattening, parent workunits may still affect the children ones by means of names. The law $\llparenthesis \overline{z}\,\widetilde{u} \mid P \ ;\ Q \rrparenthesis_x \equiv \overline{z}\,\widetilde{u} \mid \llparenthesis P \ ;\ Q \rrparenthesis_x$ 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 defined by the reduction relation where we use the shortcut:

$$\llparenthesis P \ ;\ Q \rrparenthesis \stackrel{\text{def}}{=} (z)\llparenthesis P \ ;\ Q \rrparenthesis_z \text{ where } z \notin \mathrm{fn}(P)\ \cup\ \mathrm{fn}(Q)$$

Definition 2. *The* reduction relation $\rightarrow$ *is the least relation satisfying the following axioms and rules, and closed with respect to* $\equiv$, $(x)\_$, $\_ \mid \_$, *and* $\llparenthesis\ \_ \ ;\ Q \rrparenthesis_z$*:*

(COM)
$$\overline{x_i}\,\widetilde{v} \mid \textstyle\sum_{i\in I} x_i(\widetilde{u_i}).P_i \quad\rightarrow\quad P_i\{\widetilde{v}/\widetilde{u_i}\}$$
(REP)
$$\overline{x}\,\widetilde{v} \mid !x(\widetilde{u}).P \quad\rightarrow\quad P\{\widetilde{v}/\widetilde{u}\} \mid !x(\widetilde{u}).P$$
(FAIL)
$$\overline{x} \mid \llparenthesis \textstyle\prod_{i\in I} \sum_{s\in S} x_{is}(\widetilde{u_{is}}).P_{is} \mid \prod_{j\in J} !x_j(\widetilde{u_j}).P_j \ ;\ Q \rrparenthesis_x \quad\rightarrow\quad \llparenthesis Q \ ;\ \mathbf{0} \rrparenthesis$$

where$J \neq \emptyset \vee (I \neq \emptyset \wedge S \neq \emptyset)$

Rules (COM) and (REP) are standard in process calculi and models 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), for this reason we close the workunit restricting its name.

Interested readers may find all the definitions and proofs with an extensive explanation for the extensional semantics, the notions of barb, process contexts and barbed bisimulation in [13]. Definitions for Labelled Semantics, asynchronous bisimulation, labelled bisimilarity and the proof that it is a congruence are also present. Finally, results relating barbed bisimulation and asynchronous labeled bisimulation as well as many examples are discussed. A core BPEL is encoded in $\texttt{web}\pi_\infty$ and a few properties connected to this encoding are proved for it.

4 A Case Study: The BPEL RF

One of the unsatisfactory things about the encoding of the BPEL RF we presented in [12] is that it was hardly readable for humans. The goal was to capture in that encoding all the hidden details of the BPEL semantics and working out the full theory also for verification purpose. But surely we lost something in readability since the target for that encoding were not humans but machines. Many people who approached our work justified their problems in understanding the encoding claiming that was exactly the proof of the BPEL recovery framework complexity. This is definitely true but, in order to be really useful, that work needs to be understandable also to non-specialists (and humans in general). With the goal of better understanding how the BPEL RF works, in this section we analyze a case study where $\texttt{web}\pi_\infty$ shows its power. We will firstly report the description of the mechanisms following the original BPEL specification, then we will consider a simplification of the actual mechanisms giving a simplified semantics and a simplified explanation. In this way some details will be lost but we will improve readability. The first simplification is considering only the case in which a single handler exists for each of the three different type (fault, compensation and event). Furthermore, we do not consider interdependencies between the mechanisms: default handlers with automatic compensation of inner scope. This study is an integration of what done before in [12] and [15]. The semantics provided is not the one implemented by the engines supporting BPEL, we have already given a formalization for the Oracle BPEL Manager in [13]. While in [12] you can find a complete description, here we want to focus only on the essence of the single mechanisms to understand at which stage of the execution they play their role and in which way.

4.1 Details from the BPEL Specification

Instead of assuming a perfect roll-back in case of failure, BPEL supports in its RF the notion of the so-called *loosely coupled transactions* and the explicit programming of compensation activities. This kind of transactions lasts long periods (atomicity needs to be relaxed wrt ACIDity), crosses administrative domains (isolation needs to be relaxed) and possibly fails because of services unavailability etc... They usually contain the description of three processes:

- body
- fault handler
- compensation handler

BPEL also adds the possibility to have a third kind of handler called the event handler. The whole set of activities is included in a construct called `scope` introduced as follows in the specification:

> “A `scope` provides the context which influences the *execution behavior* of its enclosed activities. This behavioral context includes variables, partner links, message exchanges, correlation sets, *event handlers*, *fault handlers*, a *compensation handler*, and a termination handler [...]
>
> Each `scope` has a required primary activity that defines its normal behavior. The primary activity can be a complex structured activity, with many nested activities to arbitrary depth. All other syntactic constructs of a `scope` activity are optional, and some of them have default semantics. The context provided by a `scope` is shared by all its nested activities.”

In the following, we report the way in which the concepts of the Recovery Framework and the need for it are motivated in [18].

Compensation Handler

> “Business processes are often of long duration. They can manipulate business data in back-end databases and line-of-business applications. Error handling in this environment is both difficult and business critical. The use of ACID transactions is usually limited to local updates because of trust issues and because locks and isolation cannot be maintained for the long periods during which fault conditions and technical and business errors can occur in a business process instance. As a result, the overall business transaction can fail or be cancelled after many ACID transactions have been committed. The partial work done must be undone as best as possible. Error handling in BPEL processes therefore leverages the concept of compensation, that is, *application-specific activities that attempt to reverse the effects of a previous activity that was carried out as part of a larger unit of work that is being abandoned.* There is a history of work in this area regarding the use of Sagas and open nested transactions. BPEL provides a variant of such a compensation mechanism by providing the ability for flexible control of the reversal. BPEL achieves this by providing the ability to define fault handling and compensation in an application-specific manner, in support of Long-Running Transactions (LRT’s) [...] *BPEL allows scopes to delineate that part of the behavior that is meant to be reversible* in an application-defined way by specifying a compensation handler. Scopes with compensation and fault handlers can be nested without constraint to arbitrary depth.[...]
> A compensation handler can be invoked by using the `compensateScope` or `compensate` (together referred to as the “compensation activities”). A compensation handler for a scope MUST be made *available for invocation only when the scope completes successfully. Any attempt to compensate a scope, for which the compensation handler either has not been installed or has been installed and executed, MUST be treated as executing an* `empty` *activity.* [...]”

Fault Handler

> “Fault handling in a business process can be thought of as *a mode switch from the normal processing in a scope*. Fault handling in BPEL is designed to be treated as “reverse work” in that its aim is *to undo the partial and unsuccessful work of a scope in which a fault has occurred.* The completion of the activity of a fault handler, even when it does not rethrow the handled fault, is not considered successful completion of the attached scope. *Compensation is not enabled for a scope that has had an associated fault handler invoked.*
>
> Explicit fault handlers, if used, attached to a scope provide a way to define a set of custom fault-handling activities, defined by catch and catchAll constructs. Each catch construct is defined to intercept a specific kind of fault, defined by a fault QName. An optional variable can be provided to hold the data associated with the fault. If the fault name is missing, then the catch will intercept all faults with the same type of fault data. The fault variable is specified using the faultVariable attribute in a catch fault handler. The variable is deemed to be implicitly declared by virtue of being used as the value of this attribute and is local to the fault handler. It is not visible or usable outside the fault handler in which it is declared. A catchAll clause can be added to catch any fault not caught by a more specific fault handler.”

Event Handler

> “Each scope, including the process scope, can have a set of event handlers. *These event handlers can run concurrently and are invoked when the corresponding event occurs* [...] There are two types of events. First, events can be inbound messages that correspond to a WSDL operation. Second, events can be alarms, that go off after user-set times.”

5 Formal Semantics of a (Simplified) BPEL RF

The plain text description of these mechanisms taken from the specification should give an idea of the complexity of this framework. The main difficulty we have found at the beginning of this investigation was to clarify the basic difference between failure and compensation handlers, since many words have been spent on this but the true essence of these mechanisms has never been given in a concise and simple way. In the past we also promoted a complete explanation of the mechanisms focusing on inessential minor details. Here we want to give the basic idea explaining that failure and compensation handlers differ mainly because they play their role at different stages of computation: failure handler is responsible for reacting to signals that occur during the normal execution of the body; when these occur, the body is interrupted and the failure handler is activated. On the contrary, compensation handler is installed only when the body successfully terminates. It remains available if another activity requires some undo of the committed activity. In some sense, failures regard "living" (not terminated) processes, while compensation is only for "successfully terminated process". The key point regarding event handlers is instead bound to the sentence reported above: they are *invoked concurrently* to the body of a scope that meanwhile continues running. This is very different from what happens for failures that interrupt the main execution and compensations which run only after the completion of the relative body.

The difficulty of the encoding we gave in [12] lies in the nontrivial interactions between the different mechanisms and it is due to the sophisticated implicit mechanism of recovery activated when designer-defined fault or compensation handlers are absent. Indeed, in this case, BPEL provides backward compensation of nested activities on a causal dependency basis relying on two rules:

- control dependency: links and sequence define causality
- peer-scope dependency: the basic control dependency causality is reflected over peer scopes

These two rules resemble some kind of structural inductive definition, as is usually done in process algebra. It is exactly our goal to skip these details here and to clarify the semantics.

5.1 Syntax

Let $(\mathtt{A};\ \mathtt{H})_s$ be a scope named s where A is the main activity (body) and H a handler. Both A and H have to be intended as BPEL activities coming from a subset of the ones defined in [12]. Practically, that work was limited to basic activities, structured activities and error handling. The idea now is to represent a simplified BPEL scope called s having a single handler H, so we are providing a semantics for the error handling mechanisms alternative to the previous one. For the sake of simplicity, we start considering a single handler at a time. Afterward we will consider the full scope construct. In the following subsection the formal semantics derived from $\mathtt{web}\pi_\infty$ will be presented, here we just define the syntax giving an informal explanation.

Definition 3 (Compensation Handler). *We define the compensation handler as follows:*

$$(\mathtt{A};\mathtt{COMP}\,s \rightarrow \mathtt{C})_s$$

If s is invoked after the successful termination of A, then run the allocated compensation C.

Definition 4 (Fault Handler). *We define the fault handler as follows:*

$$(\mathtt{A};\mathtt{FAULT}\,f \rightarrow \mathtt{F})_s$$

If f is invoked in A, then abort immediately the body A and run F.

Definition 5 (Event Handler). *We define the event handler as follows:*

$$(\mathtt{A};\mathtt{EVENT}\,e \rightarrow \mathtt{E})_s$$

If e is invoked in A then run E in parallel while the body A continues running still listening for another event e.

5.2 Semantics

The formal semantics of the three mechanisms is defined here in terms of $\texttt{web}\pi_\infty$. These constructs are encoded in $\texttt{web}\pi_\infty$ which has a formal semantics, as a consequence the semantic of the constructs themselves is given. The continuation passing style technique is used like in [12]. Briefly, $[\![A]\!]_y$ means that the encoding of the BPEL activity A completes with a message sent over the channel y. More details can be found also in [13]. In that work the function $[\![A]\!]_y : A_{BPEL} \rightarrow Process$ has been used to map BPEL activities into $\texttt{web}\pi_\infty$ processes flagging out y to signal termination.

5.3 Compensation Handler

Definition 6 (Compensation Handler). *The semantics of the single Compensation Handler scope is defined in terms of* $\texttt{web}\pi_\infty$ *as follows:*

$$(\mathit{A}; \mathit{COMP}\ s \rightarrow \mathit{C})_s \overset{\text{def}}{=} (y)(y')(\langle\!|\, [\![A]\!]_y \ ; \ s().[\![C]\!]_{y'} \,|\!\rangle_y)$$

The reader will realize that there are two new names y and y' defined at the outer level. This means that all the interactions related to this name are local to this process, *i.e.,* interferences from the outside are not allowed (they are restricted names). Then you have a workunit containing the main process and the compensation handler. Both these processes are, in turn, contained by the double brackets, which means that their encodings need to be put here. As you can see the compensation is blocked until a message on s (the name of the scope) is received and C will be available only after the successful termination of A signaled on the local channel y. This expresses exactly the fact that the compensation is available only after the successful termination of the body as required in the BPEL specification. The reason for which C is activated after the termination of A stands in the $\texttt{web}\pi_\infty$ rule (FAIL) which activates the workunit handler $s().[\![C]\!]_{y'}$ when the signal y (the workunit name) is received. This name is precisely sent by A when it terminates (because of the continuation passing style encoding).

5.4 Fault Handler

Definition 7 (Fault Handler). *The semantics of the single Fault Handler scope is defined in terms of* $\texttt{web}\pi_\infty$ *as follows:*

$$(\mathit{A}; \mathit{FAULT}\ f \rightarrow \mathit{F})_s \overset{\text{def}}{=} (f)(y)(y')(\langle\!|\, [\![A]\!]_y \ ; \ [\![F]\!]_{y'} \,|\!\rangle_f)$$

The fault handler has a semantic very close to the $\texttt{web}\pi_\infty$ workunit. For this reason the encoding here is basically an isomorphism. The handler is triggered when receiving the signal f which interrupts the normal execution of the body. Since the activation of the fault handler is internal to the scope itself, the scope name is not relevant in the right hand side.

5.5 Event Handler

Definition 8 (Event Handler). *The semantics of the single Event Handler scope is defined in terms of* webπ_∞ *as follows:*

$$(A; \mathtt{EVENT}\ e \rightarrow E)_s \overset{\text{def}}{=} (e)(y)(y')(\langle\!|[\![A]\!]_y\ ;\ \mathbf{0}|\!\rangle_y \mid !e().[\![E]\!]_{y'})$$

The event handler is interesting. The main point here is that the body execution is not interrupted when e is received. Consider indeed that E is outside the workunit and it is triggered only by e. The handler, receiving e and activating E, will run in parallel with A without interrupting it. It is worth noting also that the presence of the replication allows e to be received many times during the execution of A, each time running a new handler. The event handler will stay active without any risk of being stopped by other scopes since all the names inside the handler are local to E (bound names) due to the way in which BPEL activities are encoded by the function $[\![A]\!]_y$. This is a simplification to clarify the mechanism, it actually represents a deviation from the BPEL standard where the events are not restricted in this way.

5.6 BPEL Scope

Now that we have understood each mechanism let us put all together. We define a scope construct including all the three handlers. Again, we consider single handlers of each type with no interactions, no default handler and no automatic compensation of inner scopes.

Definition 9 (Full Scope Construct). *The semantics of the full scope construct is defined in terms of* webπ_∞ *as follows:*

$$\begin{aligned}&(A; \mathtt{FAULT}\ f \rightarrow F; \mathtt{EVENT}\ e \rightarrow E; \mathtt{COMP}\ s \rightarrow C)_s \overset{\text{def}}{=}\\&(e)(f)(y)(y')(y'')(y''')(\langle\!|[\![A]\!]_y\ ;\ [\![F]\!]_{y'}|\!\rangle_f\\&\mid !e().[\![E]\!]_{y''} \mid \langle\!|(x)x()\ ;\ s().[\![C]\!]_{y'''}|\!\rangle_y)\end{aligned}$$

It is worth noting that here the name s is a free global name (undefined) available to all the scopes which possibly run in parallel. The technical problem is that, in this way, the encoding is not compositional. Actually, this problem is easily fixed when the encoding is extended to the complete set of BPEL constructs, including the top level process where all the scopes are defined since there you can restrict all the names of the inner scopes. This has been done previously in [12]. The purpose of this work is just to explain in a clearer way the differences between the mechanisms of the recovery framework without presenting again the whole encoding. A synergy between this result and what we have done in [12] is left as future work.

5.7 Example

Let us now show an example of how this mechanism works in practice. To do this we will run a process description on the "reduction semantics machine" of $\texttt{web}\pi_\infty$. This example serves as a clarification for all the concepts presented in this paper, especially for those readers who are not very familiar with the mathematical tools exploited in our investigation. Let us consider the following process where, for simplicity, the body and the handlers are already presented in terms of $\texttt{web}\pi_\infty$:

$$((z)(\overline{f} \mid z().\mathbf{0}); \texttt{FAULT}\ \ f \rightarrow \overline{warning}\,; \texttt{EVENT}\ \ e \rightarrow \mathbf{0}; \texttt{COMP}\ \ s \rightarrow \mathbf{0})_s$$

Looking at the previous encoding it results in the following full $\texttt{web}\pi_\infty$ process:

$$\begin{array}{l}(e)(f)(y)(y')(y'')(y''')(\langle\!|(z)(\overline{f} \mid z().\mathbf{0}) \mid \overline{y}\ ;\ \overline{warning} \mid \overline{y'}\,|\!\rangle_f \\ \mid !e().\overline{y''} \mid \langle\!|(x)x()\ ;\ s().\overline{y'''}\,|\!\rangle_y)\end{array}$$

where *warning* is some global channel handling the actual warning (for example displaying a message on the screen). This is a specific instance of the Full Scope Construct as defined above where event and compensation handlers are empty while the fault handler sends an empty message on the warning channel. The process $z().\mathbf{0}$ expresses the fact that we want the process to fail without allocating the compensation handler and it has to be the standard encoding when raising a failure signal to indicate that there is no successful termination. Now, applying the (FAIL) rule and the floating law, we have:

$$\begin{array}{l}(e)(f)(y)(y')(y'')(y''')(\langle\!|\overline{warning} \mid \overline{y'}\ ;\ \mathbf{0}|\!\rangle \\ \mid !e().\overline{y''} \mid \langle\!|(x)x()\ ;\ s().\overline{y'''}\,|\!\rangle_y)\end{array}$$

which will lead to a warning on the appropriate channel *without* activating the compensation (which would need a message on y) since the scope did not successfully complete. It is worth noting that the event handler remains ready to accept events but it never activates in this scenario. This happens because the channel on which the event handler listens is restricted, and this is consistent with the expected behaviour.

5.8 Is It Really Simpler?

The intention of this work is to demonstrate, in real life scenarios, the added value of formal methods. We believe that what has been introduced so far can been really useful in the clarification of the BPEL RF semantic. Just to stress better this point, let us recall only the complete Event Handler compilation presented in [12]:

$$\left(\begin{array}{ll}EH(S_e, y_{eh}) = & (y')(\{e_x \mid x \in h_e(S_e)\}) \\ & en_{eh}().(\langle\!| \prod_{(x,\widetilde{u},A)\in S_e}\ !\ x(\widetilde{u}).\overline{e_x}\,\widetilde{u}\ ;\ \overline{y_{eh}}\,|\!\rangle_{dis_{eh}} \\ & \mid \prod_{(x,\widetilde{u},A_x)\in S_e}\ !\ e_x(\widetilde{u}).[\![A_x]\!]_{\overline{y'}}\,)\end{array}\right)$$

while the new one is:

$$(\texttt{A}; \texttt{EVENT}\ \ e \rightarrow \texttt{E})_s \overset{\text{def}}{=} (e)(y)(y')(\langle\!|[\![A]\!]_y\ ;\ \mathbf{0}|\!\rangle_y \mid !e().[\![E]\!]_{y'})$$

For the proper background please refer to [13] where you can find a detailed explanation of the encodings and all the theory. Here the idea is just to give a flavor of how this work contributes (in terms of simplification) to the improvement of the BPEL specification.

5.9 Design of BPEL Orchestration Engines

Although this paper has to be intended as investigating a well known case study and providing methodological arguments for the adoption of formal methods in software specification, the aspect of verification is not alien to our work and here we intend to give some hints in this regard. The most common formalization of behavioral equivalence is through barbed congruence, which guarantees that equated processes are indistinguishable by external observers, even when put in arbitrary contexts. For instance, equivalent Web services remain indistinguishable also when composed to form complex business transactions. The barbed congruence in this scenario has been presented in [15]. The proposed encoding, based on the function $[\![A]\!]_y : A_{BPEL} \rightarrow Process$, can be used to test the equivalence of BPEL processes on the basis of the barbed congruence developed in the theory. The idea is to inherit the equivalence notion from $\mathtt{web}\pi_\infty$ to decide BPEL processes equivalence. Here, as a further contribution, we want to show that, despite its simplicity, there are many ways in which BPEL can benefit from this work exploiting this idea of behavioral equivalence. For example, our proposal can contribute to the implementation of real orchestration engines. The application example comes from one of the theorems proved in [13]:

$$\langle\!|\, !z(u).P \,|\, Q \;;\; \overline{v} \,|\!\rangle_x \approx_a (y)(\langle\!|\, !z(u).P \;;\; \overline{y} \,|\!\rangle_x \,|\, \langle\!|\, Q \,|\, (w)w(u) \;;\; \overline{v} \,|\!\rangle_y)$$

where the symbol $\approx_a$ has to be intended as barbed congruence, i.e. the process on its left and the one on its right exhibit the same behavior. It is worth noting that the process $(w)w(u)$ is necessary to prevent v from disappearing in the case the workunit on the right would terminate succesfully. This theorem suggests a transformation where it is always possible to separate the body and the recovery logics of the workunit expressing, for example, the event handler behavior. This is possible not only when the recovery logic is a simple output (as in this case) but in all the other cases, on the basis of another theorem showed in the same work:

$$\langle\!|\, P \;;\; Q \,|\!\rangle_x \approx_a (x')(\langle\!|\, P \;;\; \overline{x'} \,|\!\rangle_x \,|\, \langle\!|\, x'().Q \;;\; \mathbf{0} \,|\!\rangle)$$

Now we have the design option to compile the mechanism in an alternative way allowing a logical separation of code which can lead to an actual physical separation. For example, different workunits could be loaded on different machines. Although BPEL typically allows a centralized control and a local compilation, this result gives us further insights in the direction of distribution. Consider, for example, the case in which different scopes can share instances of the same handler loaded on a specific dedicated machine. This result can also be interpreted in a choreographic perspective.

6 Summary, Related Works and Criticisms

The goal of this paper was to show how a variant of the π-calculus can be of some use in the context of dependable Web services composition. The specific case study presented aimed at reducing the ambiguity of the BPEL RF providing a (simplified) formal semantics opposed to the complete one already given in [12]. This is what we have called the "π-calculus way", *i.e.,* using the π-calculus as formal specification language. As we have already underlined, several different formal notations might have been chosen for this purpose. Our choice depended on the "foundational feature" of *mobility.* It has been noted that in the specific contribution the mobility feature has not been fully exploited since the modeled mechanisms required us to pay more attention to process synchronization and concurrency than to full mobility. Anyway, we have realized that, in the general case, mobility is an essential feature of composition languages and this point is discussed more in detail in [13].

Although before this work [15] and [12] have been earlier attempts at defining a formal semantics for WS-BPEL and unifying and simplifying its recovery mechanisms, those papers are far from being complete and from providing the ultimate BPEL formal semantics. Many other works have been presented recently that significantly improved what has been done there. For example, Blite [10] is a "lightweight BPEL" with formal semantics taking into account also dynamic aspects (e.g. dynamic compensations) that have not been directly part of our investigation. Another relevant work adding dynamic compensation features is [20]. In this paper the interested reader can find a comparison between different compensation mechanisms presented in the recent literature. The criticism in this work is that in $\texttt{web}\pi_\infty$ completed transactions cannot be compensated. This is of course true but, as shown in this paper, this aspect can be easily modeled (look for example at the encoding of the BPEL compensation handler). The basic idea behind $\texttt{web}\pi_\infty$ is indeed to provide a unifying theory for Web services composition as discussed in [15] where different mechanisms can be easily mapped without being directly supported. A good analysis of fault, compensation and termination (FCT) in WS-BPEL is also discussed in [7]. Here the BPEL approach to FCT with related formal semantics is given, thus covering termination handler that has not been part of our work. Furthermore, the authors in [22] recognize that in [12] the lack of support for control links has to be seen as a major drawback. And this is a criticism that we do not hide and we find relevant. The same paper proposes an alternative formalization of WS-BPEL 2.0 based on the π-calculus and then compares different approaches (including the one in [12]) from the complexity point of view for verification purposes. The authors found out that their approach presents a smaller number of states deriving from the neglect of internal activity states. Indeed, while the encoding in [12] requires every activity to signal (at least) its termination (due to the continuation passing style technique used), in [22] the activity lifecycle is not modeled. Apart from the criticisms presented in the recent literature (the list included here is not exhaustive anyway), other interesting questions have been asked regarding this approach to the BPEL RF, for example if we intend to capture fault tolerance behavior depending on external factors, for example timeout. This topic indeed has not been central to our investigation. Other authors worked on these aspects, in particular [9] discusses timed transactions.

Although we know that much needs to be done yet, we are confident that the issues we have identified are worth investigating. We have to admit that sometimes we have doubts regarding what we are doing and the solution we are adopting, so we usually look for some reassurance in the famous words of Descartes: “Dubium Sapientiae initium”, i.e. “Doubt is the origin of wisdom”.

Acknowledgments. The paper has been improved during the useful conversations with Cliff Jones, Alexander Romanovsky and Ani Bhattacharyya. For some of the ideas discussed here we have to thank Cosimo Laneve, Roberto Lucchi, Claudio Guidi and Gianluigi Zavattaro. Very useful comments came also by Joey Coleman, Felix Loesch and Michael Jastram that kindly provided other written reviews for this work (Ani Bhattacharyya also provided a written review). Finally, we have also to thank the anonymous WSFM reviewers for their contribution. This work has been partially funded by the EU FP7 DEPLOY Project (Industrial deployment of system engineering methods providing high dependability and productivity). More details at http://www.deploy-project.eu/.

References

1. Web services flow language (wsfl 1.0), `http://www.ebpml.org/wsfl.htm`
2. Xlang: Web services for business process design, `http://www.ebpml.org/xlang.htm`
3. Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: Web services description language (wsdl 1.1), W3C Recommendation (June 26, 2007), `http://www.w3.org/TR/wsdl20/`
4. Chris, P.: Web services orchestration and choreography. Computer 36(10), 46–52 (2003)
5. World Wide Web Consortium. Extensible markup language (xml) 1.0. W3C Recommendation: `http://www.w3.org/XML/`
6. Decker, G., Leymann, F., Weske, M.: Bpel4chor: Extending bpel for modeling choreographies. In: Proceedings International Conference on Web Services, ICWS (2007)
7. Eisentraut, C., Spieler, D.: Fault, compensation and termination in ws-bpel 2.0 – a comparative analysis. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 107–126. Springer, Heidelberg (2009)
8. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., Nielsen, H.F., Karmarkar, A., Lafon, Y.: Simple object access protocol (soap) 1.1, W3C Recommendation (April 27, 2007), `http://www.w3.org/TR/soap12-part1/`
9. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
10. Lapadula, A., Pugliese, R., Tiezzi, F.: A formal account of WS-BPEL. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 199–215. Springer, Heidelberg (2008)
11. Little, M.: Web services transactions: Past, present and future, `http://www.jboss.org/dms/jbosstm/resources/presentations/XML2003.pdf`
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. Mazzara, M.: Towards Abstractions for Web Services Composition. PhD thesis, Department of Computer Science, University of Bologna (2006)

14. 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)
15. 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)
16. Milner, R.: Functions as processes. Mathematical Structures in Computer Science 2(2), 119–141 (1992)
17. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press, Cambridge (1999)
18. OASIS Web Services Business Process Execution Language (WSBPEL) TC. Web services business process execution language version 2.0., `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
19. van der Aalst, W.M.P.: Pi calculus versus Petri nets: Let us eat humble pie rather than further inflate the Pi hype (2004), `http://is.tm.tue.nl/research/patterns/download/pi-hype.pdf`
20. Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 201–215. Springer, Heidelberg (2009)
21. W3C. Http - hypertext transfer protocol, `http://www.w3.org/protocols`
22. Weidlich, M., Decker, G., Weske, M.: Efficient analysis of bpel 2.0 processes using pi-calculus. In: APSCC 2007: Proceedings of the 2nd IEEE Asia-Pacific Service Computing Conference, Washington, DC, USA, 2007, pp. 266–274. IEEE Computer Society, Los Alamitos (2007)

Realizability Is Controllability

Niels Lohmann and Karsten Wolf

Universität Rostock, Institut für Informatik, 18051 Rostock, Germany
{niels.lohmann,karsten.wolf}@uni-rostock.de

Abstract. A *choreography* describes the interaction between services. It may be used for specification purposes, for instance serving as a contract in the design of an inter-organizational business process. Typically, not all describable interactions make sense which motivates the study of the *realizability* problem for a given choreography.

In this paper, we show that realizability can be traced back to the problem of *controllability* which asks whether a service has compatible partner processes. This way of thinking makes algorithms for controllability available for reasoning about realizability. In addition, it suggests alternative definitions for realizability. We discuss several proposals for defining realizability which differ in the degree of coverage of the specified interaction.

1 Introduction

When designing an inter-organizational business process, the involved parties (e.g., enterprises or business units) need to agree on many aspects of their interaction. One of these aspects is the order of exchanged messages between the parties. To this end, *choreographies* have been proposed. A choreography specification aims at specifying the interaction without revealing unnecessary details about the internal control flow of the involved parties. Keeping internals secret may have several reasons. On the one hand, trade secrets may be involved as the parties may be competitors. On the other hand, an internal control flow may not exist when the choreography is specified in a design-by-contract scenario.

Several languages have been proposed for specifying choreographies (see [1] for a survey). They all have in common that they permit to specify unreasonable interactions. An example for a potentially unreasonable interaction is to require that a message from party A to party B must be exchanged before another message from C to D. As long as no other messages are passed between A and C or B and C this requirement cannot be satisfied. For distinguishing between reasonable and unreasonable interaction, the concept of *choreography realizability* was introduced for example in [2,3].

In this paper, we address the following issues in existing approaches to choreographies and realizability notions. *First*, several approaches seem to focus on synchronous interaction. Asynchronous interaction is either not considered at all, or is brought into the approach as a derivative of the synchronous approach. For instance, some approaches specify only the order in which messages are sent but

C. Laneve and J. Su (Eds.): WS-FM 2009, LNCS 6194, pp. 110–127, 2010.

say nothing about the order in which they should be received. Consequently, we propose a formalism for modeling choreographies where synchronous and asynchronous communications are both first-class citizens. In our setting, causality between the receipt of a message and sending another one can be specified. *Second*, there appear to be several proposals for defining realizability. Consequently, we propose a hierarchy of realizability notions that includes and extends existing concepts. The hierarchy is systematically obtained by relating the problem of realizability to the problem of *controllability* in the sense of [4].

Controllability asks whether a given service has compatible partners. Existing techniques for answering the controllability problem are capable of synthesizing a compatible partner if it exists. Hence, *third*, we suggest techniques to synthesize internals of realizing partners. By relating the realizability problem to controllability, we, *fourth*, get the opportunity to study specifications that involve both a choreography and the specification of the internal behavior of some of the parties. This way, we marry the choreography approach with the orchestration approach. Both approaches have so far been conceived as complementary paradigms for building up complex processes from services.

The rest of this paper is organized as follows. In Sect. 2, we introduce a formal framework which allows us to reason about choreographies in a formal and language-independent manner. In Sect. 3, we recall different realizability notions and introduce the novel concept of *distributed realizability*, which seamlessly complements existing notions. The main contribution of the paper is presented in Sect. 4: the realizability problem can be formulated in terms of controllability and algorithms for controllability can be used to prove realizability by synthesizing realizing services. Section 5 is dedicated to issues arising when asynchronous communication is considered. In Sect. 6, we show how the relationship between controllability and realizability can be used to combine aspects from interaction modeling and interconnected models. Section 7 discusses related work and Sect. 8 concludes the paper and gives directions for future research.

2 A Formal Framework for Choreographies

To formally reason about choreographies, we first introduce a formal framework that employs automata to model single services as well as whole service choreographies. Throughout this paper, fix a finite set of message channels M that is partitioned into asynchronous message channels M_A and synchronous message channels M_S. From M, derive a set of message events $E := {!E} \cup {?E} \cup {!?E}$, consisting of asynchronous send events ${!E} := \{!x \mid x \in M_A\}$, asynchronous receive events ${?E} := \{?x \mid x \in M_A\}$, and synchronization events ${!?E} := \{!?x \mid x \in M_S\}$. Furthermore, we distinguish a non-communicating event $\tau \notin E$. In this paper, we assume that asynchronous messages may overtake each other. We claim that this is — compared to FIFO queues for communicating finite state machines [5] — a more natural approach to model asynchronicity, because it makes less assumptions about the underlying infrastructure.

Definition 1 (Peer, collaboration). *A* peer $P = [I, O]$ *consists of a set of input message channels* $I \subseteq M$ *and a set of output message channels* $O \subseteq M$, $I \cap O = \emptyset$. *A* collaboration *is a set* $\{[I_1, O_1], \ldots, [I_n, O_n]\}$ *of peers such that* $I_i \cap I_j = \emptyset$ *and* $O_i \cap O_j = \emptyset$ *for all* $i \neq j$, *and* $\bigcup_{i=1}^{n} I_i = \bigcup_{i=1}^{n} O_i$.

A collaboration is a set of bilaterally communicating peers and can be seen as the structure or syntactic signature of a choreography, because the internal behavior of the peers is left unspecified and only their message channels are given. Figure 1(a) shows the graphical notation we use in this paper to depict collaborations. The desired observable behavior of a collaboration can be specified with a finite state automaton whose transitions are labeled with message events.

Definition 2 (Peer automaton). *A* peer automaton $A = [Q, \delta, q_0, F, \mathcal{P}]$ *is a tuple such that* Q *is a finite set of states,* $\delta \subseteq Q \times (E_I \cup E_O \cup \{\tau\}) \times Q$ *is a transition relation,* $q_0 \in Q$ *is an initial state,* $F \subseteq Q$ *is a set of final states, and* $\mathcal{P} = \{[I_1, O_1], \ldots, [I_n, O_n]\}$ *is a nonempty set of peers. Thereby,* $E_I := \{?x \mid x \in M_A \cap \bigcup_{i=1}^{n} I_i\} \cup \{!?x \mid x \in M_S \cap \bigcup_{i=1}^{n} I_i\}$ *are the input events of* A *and* $E_O := \{!x \mid x \in M_A \cap \bigcup_{i=1}^{n} O_i\} \cup \{!?x \mid x \in M_S \cap \bigcup_{i=1}^{n} O_i\}$ *are output events of* A.

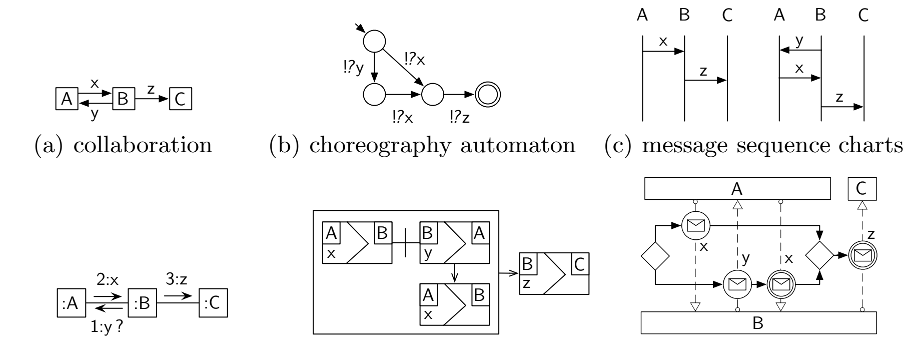
A implements *the peers* $\mathcal{P}$, *and for* $(q, x, q') \in \delta$, *we also write* $q \xrightarrow{x} q'$. A *is called a* single-peer automaton, *if* $|\mathcal{P}| = 1$. A *is called a* multi-peer automaton, *if* $|\mathcal{P}| > 1$ *and* $\mathcal{P}$ *is a collaboration.* A *is called* τ-free *if* $q \xrightarrow{x} q'$ *implies* $x \neq \tau$ *for all* $q, q' \in Q$. *A* run *of* A *is a sequence of events* $x_1 \cdots x_m$ *such that* $q_0 \xrightarrow{x_1} \cdots \xrightarrow{x_m} q_f$ *with* $q_f \in F$.

We use the standard graphical representation for automata (cf. Fig. 1(b)). A τ-free peer automaton can be used to define a choreography.

Definition 3 (Choreography). *Let* $A = [Q, \delta, q_0, F, \mathcal{P}]$ *be a multi-peer automaton. A run* ρ *of* A *is a* conversation *if no* τ *transition occurs in* ρ *and, for all* $x \in M_A$, $\#_{!x}(\rho) = \#_{?x}(\rho)$ *and for every prefix* ρ' *of* ρ *holds:* $\#_{!x}(\rho') \geq \#_{?x}(\rho')$. *Thereby,* $\#_x(\rho)$ *denotes the number of occurrences of the message event* x *in the run* ρ. *A* choreography *is a set of conversations. For a run* ρ, *define the event sequence of* ρ *as* $\rho_{|E}$ *(i.e.,* ρ *without* τ*-steps). The* language *of* A, *denoted* $\mathcal{L}(A)$, *is the union of the event sequences of all runs of* A. A *is a* choreography automaton, *if* A *is* τ*-free and* $\mathcal{L}(A)$ *is a choreography.*

The requirements for a conversation state that asynchronous events are always paired (messages do not get lost), and a send event always occurs before the respective receive event.

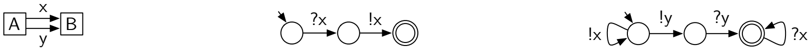
A mapping from existing interaction modeling languages such as interaction Petri nets [6], Let's Dance [7], message sequence charts [3], collaboration diagrams [8], or iBPMN [9] to choreography automata is straightforward. While these languages differ in syntax and semantics, concepts such as an underlying collaboration (i.e., the peers and their message channels) and the choreography (i.e., the intended global behavior) can be easily derived from these languages. Figure 1 shows different models specifying the same globally observable behavior.



(a) collaboration (b) choreography automaton (c) message sequence charts

(d) collaboration diagram (e) Let's Dance model (f) iBPMN model

Fig. 1. Different models specifying the choreography {!?x !?z, !?y !?x !?z}



(a) collaboration (b) dependency violation (c) unbounded message buffer

Fig. 2. The multi-peer automata (b) and (c) do not specify a choreography

We decided to use an automaton model, because it lacks structural restrictions and is naturally linked to regular languages.

However, not every τ-free multi-peer automaton specifies a choreography. Figure 2(b) depicts a peer automaton that violates the causal dependency between an asynchronous send and the respective receive event. Another problem arises in settings such as shown in Fig. 2(c) in which an arbitrary number of x messages needs to be buffered. In Sect. 5, we will show how bounded message buffers can be enforced and all runs that are not conversations can be removed from a peer automaton.

Finally, not every choreography can be expressed by a multi-peer automaton, for example the context-free choreography $(\mathsf{!?d})^i(\mathsf{!?e})^i = \{\epsilon, \mathsf{!?d!?e}, \mathsf{!?d!?d!?e!?e}, \dots\}$. In this paper, we only consider regular choreographies, because language equivalence and language containment is undecidable for context-free languages, and hence realizability is undecidable for context-free choreographies.

3 Realizability Notions

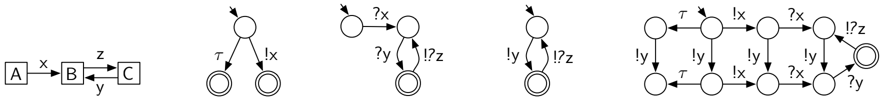
As discussed in the introduction, not every choreography can be implemented by peers. To relate the specified interactions of a choreography and the interactions between peers, we first define the behavior of composed single-peer automata.

In the composition, pending asynchronous messages are represented by a multiset. Denote the set of all multisets over M_A with $Bags(M_A)$, the empty multiset with $[\,]$, and the multiset containing only one instance of $x \in M_A$ with $[x]$. Addition of multisets is defined pointwise.

Definition 4 (Composition of single-peer automata). *Let $A_1, \dots, A_n$ be single-peer automata ($A_i = [Q_i, \delta_i, q_{0_i}, F_i, \{P_i\}]$ for $i = 1, \dots, n$) such that their peers form a collaboration. Define the* composition $A_1 \oplus \cdots \oplus A_n$ *as the multi-peer automaton $[Q, \delta, q_0, F, \{P_1, \dots, P_n\}]$ with $Q := Q_1 \times \cdots \times Q_n \times Bags(M_A)$, $q_0 := [q_{0_1}, \dots, q_{0_n}, []]$, $F := F_1 \times \cdots \times F_n \times \{[]\}$, and, for all $i \neq j$ and $B \in Bags(M_A)$ the transition relation δ contains exactly the following elements:*

- $[q_1, \dots, q_i, \dots, q_n, B] \xrightarrow{\tau} [q_1, \dots, q_i', \dots, q_n, B]$, *iff $[q_i, \tau, q_i'] \in \delta_i$ (internal move by A_i),*
- $[q_1, \dots, q_i, \dots, q_n, B] \xrightarrow{!x} [q_1, \dots, q_i', \dots, q_n, B + [x]]$, *iff $[q_i, !x, q_i'] \in \delta_i$ (asynchronous send by A_i),*
- $[q_1, \dots, q_i, \dots, q_n, B + [x]] \xrightarrow{?x} [q_1, \dots, q_i', \dots, q_n, B]$, *iff $[q_i, ?x, q_i'] \in \delta_i$ (asynchronous receive by A_i), and*
- $[q_1, \dots, q_i, \dots, q_j, \dots, q_n, B] \xrightarrow{!?x} [q_1, \dots, q_i', \dots, q_j', \dots, q_n, B]$, *iff $[q_i, !?x, q_i'] \in \delta_i$ and $[q_j, !?x, q_j'] \in \delta_j$ (synchronization between A_i and A_j).*

The composition of single-peer automata yields a multi-peer automaton (see Fig. 3 for an example). Its behavior can be related to a specified choreography which leads to the concept of complete realizability [1,2,3,6,8].



(a) collaboration (b) peer A (c) peer B (d) peer C (e) composition A ⊕ B ⊕ C

Fig. 3. Composition of single peer automata (unreachable states are omitted)

Definition 5 (Complete realizability). *Let C be a choreography automaton implementing the peers $\{P_1, \dots, P_n\}$. The single-peer automata $A_1, \dots, A_n$* completely realize *C if, for all i, A_i implements $\{P_i\}$ and $\mathcal{L}(A_1 \oplus \cdots \oplus A_n) = \mathcal{L}(C)$.*

Complete realizability is a strong requirement, because it demands that the observable behavior of the endpoints exactly matches the choreography. In reality, it is often the case that not all aspects of a choreography can be implemented. To this end, Zaha et al. [10] introduce the notion *local enforceability* (also called *partial realizability* or *weak realizability*) which only demands that a subset of the choreography is realized by the peer implementations:

Definition 6 (Partial realizability). *Let C be a choreography automaton implementing the peers $\{P_1, \dots, P_n\}$. The single-peer automata $A_1, \dots, A_n$* partially realize *C if, for all i, A_i implements $\{P_i\}$ and $\emptyset \neq \mathcal{L}(A_1 \oplus \cdots \oplus A_n) \subseteq \mathcal{L}(C)$.*

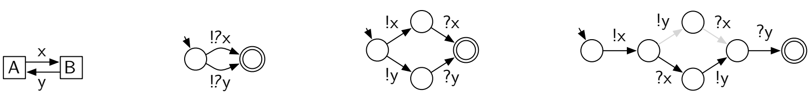
Obviously, complete realizability implies partial realizability. Though this weaker notion ensures that all constraints of the choreography are fulfilled, it still only considers a single tuple of peer automata. If there does not exist such tuple of automata that realizes the *complete* choreography, there might still exist a *set* of tuples — each partially realizing the choreography — which distributedly realizes the complete choreography:

Definition 7 (Distributed realizability). *Let C be a choreography automaton implementing the peers $\{P_1, \dots, P_n\}$. The tuples of single-peer automata* $[A_{1_1}, \dots, A_{n_1}], \dots, [A_{1_m}, \dots, A_{n_m}]$ distributedly realize C *if, for* $i = 1, \dots, n$ *and* $j = 1, \dots, m$, *(i)* A_{i_j} *implements* $\{P_i\}$, *(ii)* $\emptyset \neq \mathcal{L}(A_{1_j} \oplus \cdots \oplus A_{n_j}) \subseteq \mathcal{L}(C)$, *and (iii)* $\bigcup_{j=1}^{m} \mathcal{L}(A_{1_j} \oplus \cdots \oplus A_{n_j}) = \mathcal{L}(C)$.

Distributed realizability allows for design time coordination between peers: From a set of different possible implementations, we can choose a specific tuple of peer implementations that are coordinated in the sense that each peer can rely on the other peer's behavior. In addition, every conversation that is specified by the choreography can be realized by at least one tuple of implementing peers; that is, the choreography does not contain "dead code". While being a stronger notion than partial realizability (i.e., more of the choreography's behavior is implemented), it is still a weaker notion than complete realizability.

As an example, consider the collaboration depicted in Fig. 4(a). The choreography in which the peers communicate synchronously (b) is completely realizable by a set of peers which synchronize at runtime via message x or y. In case the messages are sent asynchronously (c), this is no longer possible. This choreography is not completely realizable, because there does not exist a single pair of peer automata that implement the specified behavior. However, the implementations can be coordinated at design time: either peer A sends a message and peer B is quiet or the other way around. These two pairs distributedly realize the whole choreography (cf. Fig. 5). Finally, choreography (d) can only be partially realized, because the conversation !x!y?x?y cannot be implemented by the peers without also producing the unspecified conversations !y!x?x?y or !y!x?y?x.

In this paper, the different realizability notions are defined in terms of trace containment. This is motivated by existing literature [1,2,3,8,10]. Other



(a) (b) (c) (d)

Fig. 4. For the collaboration (a), the choreography (b) is completely realizable, (c) is distributedly realizable, and (d) is partially realizable

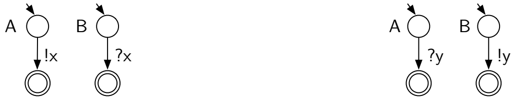


Fig. 5. Two pairs of peers which distributedly realize the choreography of Fig. 4(c)

approaches such as [6] also consider branching (i. e., a conversation is a tree). This conflicts our understanding of a choreography to specify observable behavior, because branching cannot be easily observed. Finally, we only consider finite conversations rather than infinite words. This again is motivated by existing literature and decidability issues.

4 From Choreographies to Orchestrations

In this section, we link realizability to controllability [4], a correctness criterion that was originally defined for service orchestrations [11]. We first recall how choreography conformance can be checked using a monitor. Then, we derive an orchestration service from this monitor and show how its partner services are related to peer implementations. Finally, we show how the algorithm to check controllability can be used to synthesize peer implementations.

4.1 Choreography Monitor Service

Choreography realization plays an important rule in inter-organizational business processes where a choreography is used to specify a business protocol the parties agreed to follow. In this setting, it is not sufficient to prove realizability of the choreography alone, but also to constantly monitor whether the agreed protocol is followed by the peers. In that setting, an *enterprise service bus* [12] is used to provide the connection between the individual peers. It monitors the message exchanges between the peers. Protocol violations can then be later prosecuted by examining log files or even during runtime as proposed in [13].

In this setting, it is important that the interactions are monitored in an unobtrusive way; that is, the interactions must not be altered by the monitor and the peers must not be aware of the monitor. By definition, the choreography automaton exactly determines the desired interactions. However, it blocks unspecified interactions and hence lacks the monitor property. To this end, the states of the choreography automaton needs to be made *deterministic*. This is a standard operation for regular automata and does not restrict generality. It ensures that in every state q and for each event x there is exactly one x-labeled edge leaving q. In case such a transition was not specified, the new introduced edge leads to a non-final deadlock state. A composition of peers is monitored as follows.

Definition 8 (Monitored composition). *Let* $C = [Q_C, \delta_C, q_{0_C}, F_C, \mathcal{P}]$ *be a deterministic choreography automaton with* $\mathcal{P} = \{P_1, \dots, P_n\}$ *and let* $A_1, \dots, A_n$ *be single-peer automata (*$A_i = [Q_i, \delta_i, q_{0_i}, F_i, \{P_i\}]$ *for* $i = 1, \dots, n$*) such that their peers form a collaboration. Define the* monitored composition $C \otimes (A_1 \oplus \cdots \oplus A_n)$ *as the multi-port peer automaton* $[Q, \delta, q_0, F, \mathcal{P}]$ *with* $Q := Q_C \times Q_1 \times \cdots \times Q_n \times \mathit{Bags}(M_A)$*,* $q_0 := [q_{0_C}, q_{0_1}, \dots, q_{0_n}, [\,]]$*,* $F := F_M \times F_1 \times \cdots \times F_n \times [\,]]$*, and, for all* $i \neq j$ *and* $\mathit{Bags}(M_A)$*, the transition relation* δ *contains exactly the following elements:*

- $[q, q_1, \ldots, q_i, \ldots, q_n, B] \xrightarrow{\tau} [q, q_1, \ldots, q'_i, \ldots, q_n, B]$, *iff* $[q_i, \tau, q'_i] \in \delta_i$ *(internal move by* A_i*, invisible to* C*),*
- $[q, q_1, \ldots, q_i, \ldots, q_n, B] \xrightarrow{!x} [q', q_1, \ldots, q'_i, \ldots, q_n, B + [x]]$, *iff* $[q_i, !x, q'_i] \in \delta_i$ *and* $[q, !x, q'] \in \delta_C$ *(asynchronous send by* A_i*, monitored by* C*),*
- $[q, q_1, \ldots, q_i, \ldots, q_n, B + [x]] \xrightarrow{?x} [q', q_1, \ldots, q'_i, \ldots, q_n, B]$, *iff* $[q_i, ?x, q'_i] \in \delta_i$ *and* $[q, !?x, q'] \in \delta_C$ *(asynchronous receive by* A_i*, monitored by* C*),*
- $[q, q_1, \ldots, q_i, \ldots, q_j, \ldots, q_n, B] \xrightarrow{!?x} [q', q_1, \ldots, q'_i, \ldots, q'_j, \ldots, q_n, B]$, *iff* $[q_i, !?x, q'_i] \in \delta_i$, $[q_j, !?x, q'_j] \in \delta_j$, *and* $[q, !?x, q'] \in \delta_C$ *(synchronization between* A_i *and* A_j*, monitored by* C*).*

The monitor synchronizes with the message events of the single-peer automata, but does not constrain their behavior. The monitor only has an effect on the final states of the composition. Only if all single-peer automata *and* the monitor reach a final state, this state is final in the monitored composition.

We can now change the point of view and regard the monitor as a service that is communicating with several other services by synchronous message events. Again, this service will reach a final state iff the message events from the environment are observed in the correct order. Note that a choreography only considers message *events* (i.e., sending or receipt of a message) rather than the messages itself (e.g., asynchronously sent messages that are pending on a channel). Hence, all message events of the monitor service automaton are synchronous.

Definition 9 (Monitor service). *Let* $C = [Q, \delta, q_0, F, \mathcal{P}]$ *be a deterministic choreography automaton. Define the* monitor service $M_C := [Q, \delta_M, q_0, F, \mathcal{P}]$ *with the transition relation* $\delta_M \subseteq Q \times \{!?\langle x\rangle \mid x \in E\} \times Q$ *with* $[q, !?\langle x\rangle, q'] \in \delta_M$ *iff* $(q, x, q') \in \delta$.

The monitor service of a choreography can now be interpreted as an orchestrator that communicates synchronously with its peers. The introduction of a synchronous event $!?\langle x\rangle$ for event x is a technical necessity to "encode" the original nature of the event x. In the final peer implementations, we will later replace the message event $!?\langle x\rangle$ by x again. For example, an asynchronous event $?a$ is monitored as $!?\langle ?a\rangle$.

4.2 Link to Controllability

In this section, we show how the different realizability notions are related to controllability [4]. Controllability is a correctness criterion for services: a service A is controllable iff there exists a compatible service B (i.e., $A \oplus B$ is deadlock free). Controllability can be extended to multi-port services.

Definition 10 (Decentralized controllability [4]). *Let* A *be a multi-peer automaton implementing the peer* $\{[I_1, O_1], \ldots, [I_n, O_n]\}$*.* A *is* decentralized controllable *iff there exists a tuple of single-peer automata* $[B_1, \ldots, B_n]$ *(called* strategy *of* A*) such that* B_i *implements the peer* $\{[O_i, I_i]\}$ *and* $A \oplus B_1 \oplus \cdots \oplus B_n$ *is deadlock free; that is, every reachable state* $q \notin F$ *has a successor.*

Note that the single-peer automata $B_1, \dots, B_n$ only communicate with A and do not share message channels. Hence, they cannot communicate directly with each other during runtime. Only during design time of $B_1, \dots, B_n$ it is possible to coordinate their behavior.

While realizability notions require the existence of single-peer automata whose composition realizes a certain parts of a given choreography, decentralized controllability requires the existence of single-peer automata which, when composed to a given service, result in deadlock-free communication. When considering the monitor automaton of a choreography, controllability and realizability coincide which is the main result of this paper:

Theorem 1 (Realizability is controllability). *Let C be a choreography automaton implementing the peers $\{P_1, \dots, P_n\}$ and M_C a monitor service automaton for C.*

(1) C is partially realizable iff M_C is decentralized controllable.
(2) C is distributedly realizable iff M_C is decentralized controllable and for the set of strategies $\mathcal{S}$ holds: $\bigcup_{[A_1,\dots,A_n]\in\mathcal{S}} \mathcal{L}(A_1 \oplus \cdots \oplus A_n) = \mathcal{L}(C)$.
(3) C is completely realizable iff M_C is decentralized controllable and there exists a strategy $[A_1, \dots, A_n]$ such that $\mathcal{L}(A_1 \oplus \cdots \oplus A_n) = \mathcal{L}(C)$.

The proof of Thm. 1 follows immediately from the definition of decentralized controllability, the definition of the monitor service (any unspecified behavior will lead to a deadlock) and the definitions of partial, distributed, and complete realizability.

Theorem 1 links several notions of realizability, the central correctness criterion for choreographies, to controllability. The latter was originally proposed as a "soundness notion for services" and was used to analyze service orchestrations [11].

4.3 Synthesizing Realizing Peers

In the remainder of this section, we sketch the algorithm from [4] to check for decentralized controllability. It consists of four steps: (1) peer overapproximation, (2) removal of reachable deadlocks, (3) resolution of dependencies between peers, and (4) peer projection. We will explain the steps in more detail below.

Firstly, an over-approximation of the possible interactions with the given multi-peer automaton is calculated. This step is necessary in the setting of asynchronous communication, because the decoupling of sending and receiving actions limits the observability of actions. When considering a monitor automaton, we can skip this step, because the monitor automaton communicates entirely synchronously (cf. Def. 9).

In a second step, all reachable deadlocks and states from which no final state is reachable are removed. Thereby, a deadlock is considered unreachable if the event leading to it is impossible in the respective state. An example would be a receipt of an asynchronous message in the initial state, for instance an edge labeled with "!?⟨?x⟩". Keeping these states is a technical necessity and is further discussed in [4].

From the remaining automaton, we have to make sure that message events that cannot be coordinated by the peers are independent, meaning they can occur in any order. We call two message events *distant* if there exists no peer that can observe both:

Definition 11 (Distant message events). *Let $A = [Q, \delta, q_0, F, \mathcal{P}]$ be a choreography automaton. Two message events $a, b \in E$ are* distant *iff there exist no peer $[I, O] \in \mathcal{P}$ such that $\{a, b\} \subseteq (E_I \cup E_O)$.*

No we can define independence between distant events as follows.

Definition 12 (Independence [4]). *Let $A = [Q, \delta, q_0, F, \mathcal{P}]$ be a τ-free multi-peer automaton and $a, b \in E$ be distant message events.*

- *a* activates *b ($b \notin ?E$) in $q \in Q$, if there exist states $q_a, q_{ab} \in Q$ with $q \xrightarrow{a} q_a \xrightarrow{b} q_{ab}$, but there exists no state $q_b \in Q$ with $q \xrightarrow{b} q_b$.*
- *a* disables *b in $q \in Q$, if there exist states $q_a, q_b \in Q$ with $q \xrightarrow{a} q_a$, $q \xrightarrow{b} q_b$, but there exists no state $q_{ab} \in Q$ with $q_a \xrightarrow{b} q_{ab}$.*
- *Two states $q_1, q_2 \in Q$ are* equivalent *iff $\mathcal{L}([Q, \delta, q_1, F, \mathcal{P}]) = \mathcal{L}([Q, \delta, q_2, F, \mathcal{P}])$.*
- *a and b are* independent *iff, for all states $q \in Q$ holds: a neither activates nor disables b in q and, if $q \xrightarrow{a} q_a \xrightarrow{b} q_{ab}$ and $q \xrightarrow{b} q_b \xrightarrow{a} q_{ba}$, then q_{ab} and q_{ba} are equivalent.*

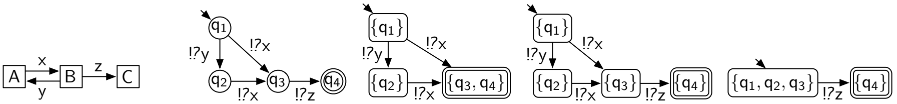
These independence requirements are weaker than the *lossless-join* property proposed in [2] and the *well-informed* property proposed in [8] which both aim at complete realizability only. They are, however, very similar the *autonomous property* [2]. If all distant events are independent and the removal of deadlocking states did not yield an empty automaton, we can finally project the remaining multi-peer automaton to the single-peer automata.

Definition 13 (Peer projection). *Let $A = [Q, \delta, q_0, F, \mathcal{P}]$ be a multi-peer automaton and $[I, O] \in \mathcal{P}$ be a peer implemented by A. For a set of states $S \subseteq Q$, define $closure_{[I,O]}(S) := \{q' \mid q \in S, q \xrightarrow{x_1 \cdots x_n} q', x_i \notin (I \cup O)\}$. Define the* projection *of A to the peer $[I, O]$, denoted $A_{|[I,O]}$, as the single-peer automaton $[Q', \delta', q_0', F', \{[I, O]\}]$ with the initial state $q_0' := closure_{[I,O]}(\{q_0\})$ and Q', δ', and F' inductively defined as follows:*

- *$q_0' \in Q'$.*
- *If $q \in Q'$ with $q_1 \in q$, $(q_1, x, q_2) \in \delta$, and $x \in I \cup O$, then $q' \in Q'$ with $q' := closure_{[I,O]}(q_2) \in Q'$ and $(q, x, q') \in \delta'$. $q' \in F'$ iff $q' \cap F \neq \emptyset$.*

The set $closure_{[I,O]}(S)$ contains all states reachable with a (possibly empty) sequence from a state of S that does not contain an event from $(I \cup O)$. The definition is basically taken from the controllability decision algorithm [4] and was first proposed to be used as a projection algorithm by Decker in [14].

In a final step, we have to "restore" the original message model of the peer implementations that was set to synchronous communication in Def. 9. To this end, we replace each message event "$!?\langle x \rangle$" by "x" (e.g., the event $!?\langle ?a \rangle$ observed by



(a) collaboration (b) choreography (c) peer A (d) peer B (e) peer C

Fig. 6. A choreography and its projection to its peers

the monitor is changed to $?a$ in the peer projection). This step from synchronous to possibly asynchronous is valid, because we ensured that any distant events are independent. Figure 6 shows an example.

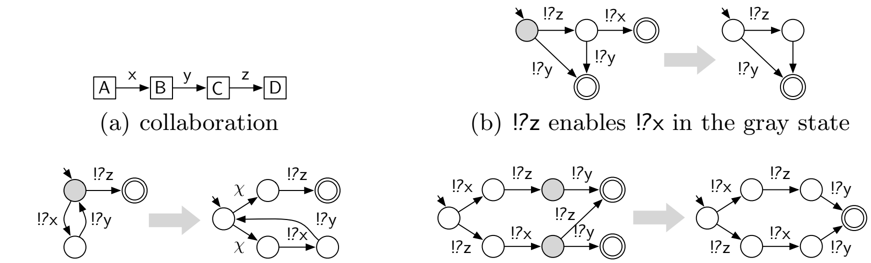
In order to synthesize single-peer automata, potential dependencies between distant message events need to be resolved. The resolution of dependencies is the most important part of the synthesis algorithm for decentralized controllability. Independency can be achieved by removing those edges and states from the automaton that are dependent. In the case of disabling of events, this removal contains nondeterminism: if, for instance, an event a disables an event b in a state q, we can decided whether to remove the a-successor or the b-successor of q [4]. This mutually exclusive deletion yields two different tuples of implementing peers. In the following definition, we introduce a global decision event χ to express the different outcomes of this nondeterminism.

Definition 14 (Resolution of dependency). *Let $A = [Q, \delta, q_0, F, \mathcal{P}]$ be a τ-free multi-peer automaton and $a, b \in E$ be distant message events.*

1. *If a disables b in a state $q \in Q$, then introduce two new states q_a and q_b with $q \xrightarrow{\chi} q_a$, $q \xrightarrow{\chi} q_b$ such that q_a has all outgoing edges of q that are not labeled with b and q_b has all outgoing edges of q that are not labeled with a. Then remove all outgoing edges of q that are not labeled with χ.*
2. *If a enables b in a state $q \in Q$, then delete the state q_{ab} with $q \xrightarrow{a} q_a \xrightarrow{b} q_{ab}$.*
3. *If the states $q_{ab}, q_{ba} \in Q$ with $q \xrightarrow{a} q_a \xrightarrow{b} q_{ab}$ and $q \xrightarrow{b} q_b \xrightarrow{a} q_{ba}$ are not equivalent, then delete q_{ab}, q_{ba} and unite A with the τ-free multi-peer automaton $A' = [Q', \delta', q'_0, F', \mathcal{P}]$ with $\mathcal{L}(A') = \mathcal{L}([Q, \delta, q_{ab}, F, \mathcal{P}]) \cap \mathcal{L}([Q, \delta, q_{ba}, F, \mathcal{P}])$ and add the edges $q_a \xrightarrow{b} q'_0$ and $q_b \xrightarrow{a} q'_0$.*

The first step introduces the global decision events in case of disabling of an event. The second step removes states to avoid the enabling of an event. In the third step, equivalence of states that are reached by different interleavings of events is enforced by intersecting the runs reachable from these states. As we consider regular languages, the automaton having this intersection as language can be constructed easily.

Figures 7(b)–(d) depict examples for each step. To increase legibility, the edges of the monitor services are labeled with the original event "x" instead of the encoded event "$!?\langle x\rangle$". Note that the removal of states and edges can introduce new deadlocks and make other states unreachable from the initial state. Such states need to be removed before projection. A multi-peer automaton is not decentralized controllable if all states are removed.



(a) collaboration

(b) !?z enables !?x in the gray state

(c) !?x disables !?z in the gray state

(d) the gray states are not equivalent

Fig. 7. Examples for the resolution of dependencies

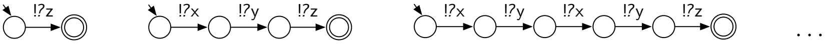


Fig. 8. Resolutions of the global decisions in the automaton depicted in Fig. 7(b)

A multi-peer automaton with global decision events (cf. Fig. 7(c)) implicitly characterizes a set of multi-peer automata in which these decisions have been resolved. Each resolution of these decisions results in a tuple of implementing peers which can be derived using the projection defined in Def. 13. For the example of Fig. 7(c), the global decision is resolved independently each time the initial state is reached. Figure 8 depicts the different resolutions of the global decisions (again, "x" is used as label rather than "$!?\langle x\rangle$"). Each resolution represents a design-time coordination between the peers A and C on how often the !?x!?y loop should be traversed. As the peers A and C cannot communicate with each other, this coordination cannot be done during runtime. The set of all possible implementations distributedly realize the choreography.

With the presented approach, we are able to synthesize single-peer automata that control a given monitor service in a decentralized manner. Using Thm. 1, we can use the same algorithm to synthesize peers for the different realizability notions. It is worthwhile to mention that the approach aims at finding the strongest applicable realizability notion. In case a state need to be deleted due to dependencies (cf. Def. 14), we can derive diagnosis information:

- If a state is deleted by step 2 or 3, the choreography is neither completely realizable nor distributedly realizable.
- If a global decision is introduced by step 1, the choreography is not completely realizable, because the considered events are mutually exclusive.
- If the initial state is removed, the choreography is not partially realizable.

In any case, the respective state and the events that require state deletion can be used to diagnose the choreography and to introduce messages that restore independency. The presented algorithm has been prototypically implemented.[1]

[1] Available at `http://service-technology.org/rebecca`.

5 Asynchronous Communication

Many interaction modeling languages (e.g., WS-CDL or interaction Petri nets) assume atomic and hence synchronous message exchange; that is, the sending and receiving of a message is specified to occur at the same time. Peers realizing such a choreography model inherit this synchronous message model. In implementations, however, asynchronous communication is often preferred over synchronous communication as a "fire and forget" send action is more efficient than a blocking handshaking.

To this end, we studied in [15] how synchronous peers that realize a choreography can be "desynchronized"; that is, atomic message exchange is decoupled to a pair of asynchronous send and receive actions. This desynchronization in turn might introduce deadlocks, and the correction towards deadlock freedom results in refinements of the choreography which require domain information and can hardly be automatized. Fu et al. [16] propose a reverse approach and study "synchronizability" of choreographies — a property under which asynchronous communication can be safely abstracted to synchronous communication. Synchronizability can help to detect problems introduced by asynchronous communication, but is only a sufficient criterion and offers only limited support in resolving these issues.

To avoid both restrictions during the design time of a choreography and a later change of the communication model, we allow to individually define, for each message, whether it should be transfered in an asynchronous or synchronous manner (cf. Def. 2). We claim that the nature of the message transfer is usually known in an early design phase and helps to refine the choreography model.

Unlike related work on collaboration diagrams or conversation protocols, we thereby do not just specify the order in which *send* events occur, but also describe the moment of the respective *receive* events. This is crucial to be able to specify dependencies between asynchronous messages. For instance, one is able to express that a customer must not send an order message to a shop before he received the terms of payment. If modeled synchronously, the shop would be blocked as long as the customer reads the terms of payment.

In addition, the precise specification of message receipt ensures that the message exchange between the peers can be realized with bounded message buffers. This is not only motivated by implementation issues, but also in the fact that unbounded queues would result in an infinite state automaton for which controllability and hence realizability would be undecidable [17].

In Def. 3, we restricted choreographies to only consist of conversations. This does not constrain synchronous message events, but only the asynchronous message events. The following definition manipulates an arbitrary multi-peer automaton such that every run is a conversation; that is, its collaboration language is a choreography. Furthermore, no run will exceed a given message bound k.

Definition 15 (k-bounded peer automaton). *Let $A = [Q, \delta, q_0, F, \mathcal{P}]$ be a peer automaton and $k \in \mathbb{N}$. Define the* k-bounded peer automaton *$A_k := [Q', \delta', q_0', F', \mathcal{P}]$ with $Q' := Q \times Bags_k(M_A)$, $q_0' := [q_0, []]$, $F' := F \times \{[]\}$ and δ' contains exactly the following elements ($B \in Bags_k$):*

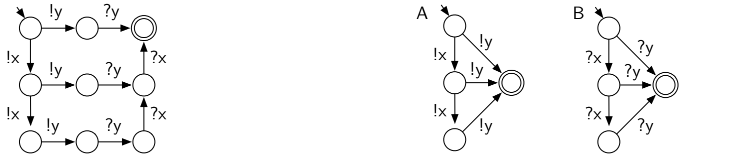
- $\big[[q,B], !?x, [q',B]\big] \in \delta'$ *iff* $q \xrightarrow{!?x} q'$,
- $\big[[q,B], \tau, [q',B]\big] \in \delta'$ *iff* $q \xrightarrow{\tau} q'$,
- $\big[[q,B], !x, [q',B+[x]]\big] \in \delta'$ *iff* $q \xrightarrow{!x} q'$ *and* $B(x) < k$, *and*
- $\big[[q,B+[x]], ?x, [q',B]\big] \in \delta'$ *iff* $q \xrightarrow{?x} q'$.

Thereby, $Bags_k(M_A)$ *denotes the set of multisets such that* $m \in Bags_k(M_A)$ *implies* $m(x) \leq k$ *for all* $x \in M_A$.

The bound k can also be used as a parameter for realizability:

Definition 16 (k-realizability). *Let* C *be a choreography automaton and* $k \in \mathbb{N}$. C *is* (completely/distributedly/partially) k-realizable *iff* C_k *is (completely/distributedly/partially) realizable.*

Figure 9(a) shows the unbounded choreography from Fig. 2(c) is transformed into a 2-bounded choreography which is completely 2-realizable, cf. Fig. 9(b).



(a) 2-bounded choreography (b) completely 2-realizing peers

Fig. 9. Enforcing a message bound to achieve 2-realizability

6 Combining Choreographies and Local Models

There are two approaches to model a choreography. The first approach focuses on the interaction between services and uses message exchange events as basic building blocks. These *interaction models* have already been discussed in Sect. 2. Interaction models are a means to quickly specify a choreography by only modeling the desired observable behavior instead of the local control flow of each peer. With the notion of realizability, these missing local behaviors can then be derived from the choreography. To this end, interaction models are best suited if all peer implementations are unknown. Interaction modeling follows a top-down approach from an abstract global model to concrete peer implementations.

In contrast, the second approach is to specify the choreography implicitly by providing a set of peer implementation and information on their interconnection. As these *interconnected models* specify both the local behavior of the participating services and their interaction, they are close to implementation. Examples of specification languages that follow this modeling style are BPMN and BPEL4Chor [18]. Interconnected models aim at reusing existing peers in new settings. Though first approaches exist to synthesize individual peers [19], this modeling style can only be used in a late stage of development.

However, a setting in which the local behaviors of some peers are completely specified whereas other peers are not specified at all is not supported by neither modeling style. By linking realizability to controllability by transforming a choreography into an orchestration, we left the classical domain of interaction models. Instead, we derived a monitor service that orchestrates the peers. In case some peers are already implemented, they can be composed to the monitor service. This composition then specifies the remaining choreography which can be analyzed for realizability as described in Sect. 4. Figure 10 diagrams this mixed modeling style, in which choreographies and inter-organizational business processes with arbitrary levels of abstraction can be modeled. By combining both classical BPMN constructs together with iBPMN extensions [9] (i.e., modeling processes both inside and and outside pools), this mixed choreography modeling approach can be presented to modelers with a unique graphical representation.

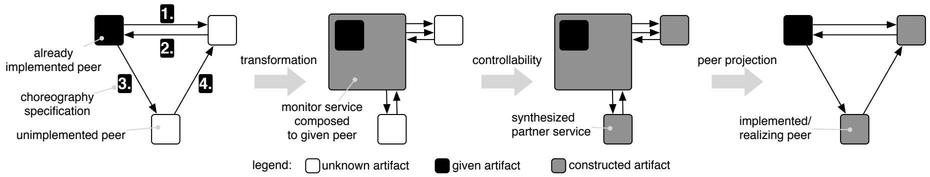


Fig. 10. A mixed approach to combine interaction and interconnected modeling

7 Related Work

This is an extended version of the informal workshop paper [20]. Compared to that paper, the contributions of this paper to partner synthesis, asynchronous communication, and the combination of interaction and interconnected models are original.

Realizability received much attention in recent literature, and was studied for most of the aforementioned interaction modeling languages, see [1] for a survey. Beside the different specification languages, the approaches differ in (i) the expressiveness of the specification language (the main differences concern the support of arbitrary looping) and (ii) the nature of the message exchange (synchronous vs. asynchronous) of the realizing peers. In the following, we classify related approaches into these two groups.

Structural restrictions. Alur et al. [3] present necessary and sufficient criteria to realize a choreography specified by a set of message sequence charts (MSCs) with a set of concurrent automata. Both synchronous and asynchronous communication is supported. Their proposed algorithms are very efficient, but are limited to acyclic choreography specifications since the MSC model used in the paper does not support arbitrary iteration which excludes models such as Fig. 7(c).

In [21], complete and partial realizability of choreographies specified by collaboration diagrams is investigated. The authors express the realizability problem in terms of LOTOS and present a case study conducted with a LOTOS verification tool. Their approach tackle both synchronous and asynchronous communication (using bounded FIFO queues). Collaboration diagrams, however, have only limited support for repetitive behavior (only single events can be iterated and cycles such as in Fig. 7(c) cannot be expressed) and choices (events can be skipped, but complex decisions cannot be modeled). These restrictions also apply to [8] in which sufficient conditions for complete realizability of collaboration diagrams are elaborated.

Communication models. Realizability of conversation protocols by asynchronously communicating Büchi automata is examined in [2]. The authors show decidability of the problem and define a necessary condition for complete realizability. One of the prerequisites, *synchronous compatibility*, heavily restricts asynchronous communication.

Algorithms to check choreographies for partial realizability are discussed in [10]. Both the global and local model are specified in Let's Dance and only atomic message exchanges considered. Decker and Weske [6] study realizability of interaction Petri nets. To the best of our knowledge, it is the only approach in which (complete and partial) realizability is not defined in terms of complete trace equivalence (cf. Def. 5). Instead, the authors require the peer implementations and the choreography to be branching bisimilar. Message exchange specified by interaction Petri nets is, however, inherently synchronous.

Kazhamiakin and Pistore [22] study a variety of communication models and their impact on realizability. They provide an algorithm that finds the "simplest" communication model under which a given choreography can be completely realized. Their approach is limited to complete realizability and gives no diagnosis information in case the choreography cannot be implemented by peers. Furthermore, they fix the communication model for all messages instead of allowing different communication models for each message.

The original contribution of this paper is an automaton framework to specify arbitrary regular choreographies, check for various realizability notations (Theorem 1 states necessary *and* sufficient criteria), and to synthesize peer services that implement as much behavior as possible. Thereby, it is possible to define the message model individually for each message. Additionally, the defined synthesis algorithm provides diagnosis information that can help to fix choreographies towards complete realizability.

8 Conclusion

In this paper, we linked the realizability problem of choreographies to the controllability problem of orchestrations. The close relationship between these problems offers a uniform way to analyze and model arbitrary interacting services. By transforming a choreography specification into a service orchestration, we were able to reuse techniques that were originally proposed to check for controllability. These techniques resulted in a formal framework that allows to specify and analyse choreographies with both synchronous and asynchronous communication. In addition, we refined the existing hierarchy of realizability notions by defining the novel notion of distributed realizability. Finally, we proposed to combine interaction models and interconnected models.

In future work, further consequences of the relationship between controllability and realizability need to be examined. For instance, controllability is used in a number of applications such as test case generation [23] or service mediation [24]. We expect these techniques to be similarly applicable to choreographies.

Acknowledgements. This work is funded by the DFG project "Operating Guidelines for Services" (WO 1466/8-1). The authors wish to thank Stephan Mennicke for his work on the prototype.

References

1. Su, J., Bultan, T., Fu, X., Zhao, X.: Towards a theory of Web service choreographies. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 1–16. Springer, Heidelberg (2008)
2. Fu, X., Bultan, T., Su, J.: Conversation protocols: a formalism for specification and verification of reactive electronic services. Theor. Comput. Sci. 328(1-2), 19–37 (2004)
3. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Trans. Software Eng. 29(7), 623–633 (2003)
4. Wolf, K.: Does my service have partners? In: Jensen, K., van der Aalst, W.M.P. (eds.) Transactions on Petri Nets, Part II. LNCS, vol. 5460, pp. 152–171. Springer, Heidelberg (2009)
5. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM 30(2), 323–342 (1983)
6. Decker, G., Weske, M.: Local enforceability in interaction Petri nets. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 305–319. Springer, Heidelberg (2007)
7. Zaha, J.M., Barros, A.P., Dumas, M., ter Hofstede, A.H.M.: Let's dance: A language for service behavior modeling. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 145–162. Springer, Heidelberg (2006)
8. Bultan, T., Fu, X.: Specification of realizable service conversations using collaboration diagrams. SOCA 2(1), 27–39 (2008)
9. Decker, G., Barros, A.P.: Interaction modeling using BPMN. In: ter Hofstede, A.H.M., Benatallah, B., Paik, H.-Y. (eds.) BPM Workshops 2007. LNCS, vol. 4928, pp. 208–219. Springer, Heidelberg (2008)
10. Zaha, J.M., Dumas, M., ter Hofstede, A.H.M., Barros, A.P., Decker, G.: Service interaction modeling: Bridging global and local views. In: EDOC 2006, pp. 45–55. IEEE Computer Society, Los Alamitos (2006)
11. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting BPEL processes. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 17–32. Springer, Heidelberg (2006)
12. Leymann, F.: The (service) bus: Services penetrate everyday life. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 12–20. Springer, Heidelberg (2005)

13. Kopp, O., Lessen, T.v., Nitzsche, J.: The need for a choreography-aware service bus. In: YR-SOC, 28–34, Imperial College, London (2008)
14. Decker, G.: Realizability of interaction models. In: ZEUS 2009. CEUR Workshop Proceedings, vol. 438, CEUR-WS.org, pp. 55–60 (2009)
15. Decker, G., Barros, A., Kraft, F.M., Lohmann, N.: Non-desynchronizable service choreographies. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 331–346. Springer, Heidelberg (2008)
16. Fu, X., Bultan, T., Su, J.: Synchronizability of conversations among Web services. IEEE Trans. Software Eng. 31(12), 1042–1055 (2005)
17. Massuthe, P., Serebrenik, A., Sidorova, N., Wolf, K.: Can I find a partner? Undecidablity of partner existence for open nets. Inf. Process. Lett. 108(6), 374–378 (2008)
18. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for modeling choreographies. In: ICWS 2007, pp. 296–303. IEEE Computer Society, Los Alamitos (2007)
19. Lohmann, N., Kopp, O., Leymann, F., Reisig, W.: Analyzing BPEL4Chor: Verification and participant synthesis. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 46–60. Springer, Heidelberg (2008)
20. Lohmann, N., Wolf, K.: Realizability is controllability. In: ZEUS 2009. CEUR Workshop Proceedings, vol. 438, CEUR-WS.org, pp. 61–67 (2009)
21. Salaün, G., Bultan, T.: Realizability of choreographies using process algebra encodings. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 167–182. Springer, Heidelberg (2009)
22. Kazhamiakin, R., Pistore, M.: Analysis of realizability conditions for Web service choreographies. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 61–76. Springer, Heidelberg (2006)
23. Kaschner, K., Lohmann, N.: Automatic test case generation for interacting services. In: ICSOC 2008. LNCS, vol. 5472, pp. 66–78. Springer, Heidelberg (2008)
24. Gierds, C., Mooij, A.J., Wolf, K.: Specifying and generating behavioral service adapter based on transformation rules. Preprint CS-02-08, Universität Rostock, Rostock, Germany (2008) (submitted to a journal)

Specification and Verification of Multi-user Data-Driven Web Applications

Monica Marcus

Indiana State University, Math and Computer Science
monica.marcus@yahoo.com

Abstract. We propose a model for multi-user data-driven communicating Web applications. An arbitrary number of users may access the application concurrently through Web sites and Web services. A Web service may have an arbitrary number of instances. The interaction between users and Web application is data-driven. Synchronous communication is done by shared access to the database and global application state. Private information may be stored in a local state. Asynchronous communication is done by message passing. A version of first-order linear time temporal logic (LTL-FO) is proposed to express behavioral properties of Web applications. The model is used to formally specify a significant fragment of an e-business application. Some of its desirable properties are expressed as LTL-FO formulas. We study a decision problem, namely whether the model satisfies an LTL-FO formula. We show the undecidability of the unrestricted verification problem and discuss some restrictions that ensure decidability.

Keywords: business process modeling, decision problem, infinite-state system, model checking, temporal logic.

1 Introduction

Web applications are interactive applications available across the Internet. They are intended for use by humans, through Web sites, and by Web services. A Web service is a self-describing, self-contained software module available via Internet, that executes tasks as service to Web applications and facilitates the communication between them. Web applications are driven by user input and by data organized in large databases. Some examples of Web applications are e-commerce, e-goverment and scientific portals.

The complexity of Web applications easily leads to malfunctioning. To eliminate errors and increase the confidence in the well functioning of Web applications, one may build a formal specification before the implementation. Then it is possible to verify whether the formal specification meets the correctness requirements. A tool for the automatic verification of Web application specifications, called WAVE, was presented in [6]. WAVE may be used to find and correct errors in the formal specification and then to generate the application code consistent with the specification. As formal specification of Web applications, WAVE uses the models presented in [3,4] and [5].

C. Laneve and J. Su (Eds.): WS-FM 2009, LNCS 6194, pp. 128–146, 2010.

In this paper we propose a model of Web applications, called *multi-user data-driven Web application*, in short *DDWA**, that extends both models of [4,5]. A *DDWA** specifies a Web application by modeling the interaction with an arbitrary number of users that may access the application through Web sites and Web services. Users may log on and off arbitrarily. Their identity is not known in advance. Arbitrarily many and not-known-in-advance Web service instances may communicate through asynchronous message passing. Synchronous communication is modeled by allowing shared access to the database and global application state. Private user information may be stored in a local state.

Intuitively, a Web application meets the correctness requirements if all its possible behaviors verify these requirements. The behavior of Web applications is modeled as a sequence of consecutive *global configurations* of *DDWA**, called *run*. A configuration is like a snapshot of the Web application at a particular moment. The next configuration in a run is triggered by user input and determined by the current configuration, results of database queries and received messages.

To exemplify the use of *DDWA** as formal specification we revisit the Bank Loan Web Application example of [5] and show that the *DDWA** specification captures interesting behaviors that cannot be represented by the models of [4,5]. We provide also a formal language to express correctness requirements of Web applications as first-order linear temporal logic (LTL-FO) formulas. We study a decision problem, namely whether all runs of a *DDWA** satisfies an LTL-FO formula. The answer to the decision problem provides an answer to the question whether a Web application meets the correctness requirements. We prove that the unrestricted decision problem is undecidable, and find reasonable restrictions to obtain decidability. We refer informally to the decision problem as the verification problem.

Next we discuss related work. We compare the *DDWA** model with the single-user Web application model of [4] and the composition model proposed in [5]. These are all relational models. A composition involves a bounded number of peers (Web services) interacting asynchronously by message passing. There is no distinction of the type local/global state and only one instance of each peer interacts. There is no straightforward way to express multiple users interactions in the models of [4,5].

The idea that the flow of information between Web services can occur not only via message passing, but also via shared access to first-order logic predicates whose value may change in time (e.g., an inventory database, a reservations database) appears in [2] (see also [10]) where a First-Order Logic Ontology for Web Services (FLOWS) is presented. In [12] a first-order logic (situation calculus) is used to provide semantics for a service description language (predecessor of FLOWS). Service descriptions are translated to Petri Nets, which constitute another type of formal specification. The verified properties are restricted to a subclass of propositional LTL and data values are not explicit.

For a review of Web service and Web application models see e.g. the surveys [9,10]. These models are message-based, the number of users is bounded, and the users' identity is known in advance. Some models include a database, but in a restricted way. A recent work reported in [7] proposes a model of synthesized Web service (SWS) as a uniform formalism to characterize various prior Web service and Web application models, including the relational models of [4,5]. SWS is a data-driven model and allows communication by message passing. The behavior of an SWS is modeled as a tree, allowing parallel processing of database queries, as opposed to a $DDWA^*$ that models behaviors as sequences, usually infinite. The SWS model features neither an *arbitrary* number of users nor communication based on global/local state.

Section 2 presents the $DDWA^*$ model and an example of Web application. Section 3 presents the formal language for property specification and states the verification problem. The undecidability of the verification problem is discussed in Section 4. Section 5 identifies reasonable restrictions that ensure decidability. Section 6 contains some conclusions.

2 Specification of Multi-user Web Applications

In this section we present the multi-user model and use it to specify formally a bank loan Web application.

We model a Web application as a collection of *components*. Each component may specify a Web site and/or a Web services and it is thought of as having multiple "copies", one for each Web application user (human or Web service instance). Informally, a component consists of:

- a database that do not change
- a database called *state* that may change in response to user inputs and received messages. We distinguish between *local state* and *global state*. Local state represents private data (e.g., the contents of the shopping cart). Global state represents data of general interest, not associated with a particular user (e.g., the flight/hotel reservations made so far), and may be accessed and updated by all users. Local state can be accessed and updated by owner only. The database and the global state may be shared by several components facilitating synchronous communication.
- input provided by user when choosing among several options generated by the application, as well as arbitrary input e.g. passwords.
- communication channels facilitating asynchronous message passing between *components*. These are one-way FIFO queues connecting one sender and one receiver. We assume that messages arrive in the same order they were sent.
- a set of rules that specify
  - how the current input choices are generated as a result of querying the database, the current local and global state, the most recent previous input, and the received messages;
  - how the local/global state may be updated in response to user's input and received messages;
  - actions to be taken (e.g pop-up message window, send e-mail), and
  - sending of messages.

The behavior of a Web application as determined by its interaction with users is specified as a set of *runs*. Intuitively, a single-user run is an infinite sequence of steps, called *transitions*. Each transition begins with the user's choice of input and the receiving of messages, if any. This is followed by updates of the local and global state, and the sending of messages. The transition ends with actions. All input options are generated by the system, except for a fixed set of input constants representing specific user information (e.g. name, password, account number, etc.) The user provides values for these constants throughout the run, as requested. The user chooses at most one tuple among the options provided for each input. To model *multi-user runs* we employ the interleaving semantics used to model concurrent systems, as done for instance in [11]. Intuitively, a multi-user run is an interleaving of single-user runs, such that at each point in time only one user performs a transition. Transitions are atomic.

We next formalize the above model as a relational model. We assume a fixed and infinite set $\mathbf{dom}_\infty$ of elements. A *relational schema* is a finite set of relation symbols with associated arities, together with a finite set of constant symbols. A *queue schema* is a finite set of queue symbols with associated arities. We assume three distinct sets of relational, constant, and queue symbols. The arity of a relation/queue symbol R is denoted $ar(R)$. Relation symbols with arity zero are called propositions. A *relational instance* D over a relational schema consists of a finite subset Dom of $\mathbf{dom}_\infty$, and a mapping associating with each relation symbol R of positive arity a finite relation $D(R)$ of the same arity over Dom, to each propositional symbol a truth value, and to each constant symbol c an element $D(c)$ of Dom. A *queue instance* D over a queue schema consists of a finite subset Dom of $\mathbf{dom}_\infty$, and a mapping associating with each queue symbol R a finite sequence $D(R)$ of finite relations over Dom, each of arity $ar(R)$. We denote by $f(D)$, $l(D)$, the first, respectively, last relation in every queue. That is, for a queue symbol R, $f(D)(R)$ ($l(D)(R)$) is the first (last) relation in $D(R)$. An empty instance $\emptyset$ of a relational (queue) schema has all the relations (queues) empty and the propositions false.

We define next the notion of parameterized schema. Parameters are special-purpose variables specifying users' identity as perceived by the Web application. We model the Web application such that every user is assigned an ID, but the ID does not uniquely identify the user. The same ID may be assigned to different users at different times, or the same user may receive a different ID each time the user becomes active. This is consistent with the practice of distinguishing between different sessions of the same user by means of *session keys*. *Correlation IDs* are used to distinguish between different transactions of a Web service (see [13]). Moreover, a user interacting simultaneously with two Web sites may be perceived by the Web application as two users. Parameters are intended to associate parts of a Web application component with users. For instance, the local state is asociated with its owner. A queue is associated with the sender and the receiver.

A *parameterized relational schema* is a relational schema where each symbol is tagged by a parameter. A *parameterized queue schema* is a queue schema where each queue symbol is tagged by two distinct parameters. Parameterized symbols are written $c[u]$, $R[u]$, $Q[u][u']$, for a constant symbol c, relation symbol R, queue symbol Q, and parameters u, u'. If $\mathbf{S}$ is a relational schema, we write $\mathbf{S}[u]$ to refer to the parameterized version of $\mathbf{S}$. Similarly, $\mathbf{Q}[u][u']$ denotes a parameterized version of the queue schema $\mathbf{Q}$. A relational instance D over a parameterized relational schema consists of a finite subset Dom of $\mathbf{dom}_\infty$, a possibly infinite set $\mathcal{ID} \subseteq \mathbf{dom}_\infty$ of IDs, and a mapping associating with each parameterized relation symbol $R[u]$ of positive arity a family $\{D(R)[i] \mid i \in \mathcal{ID}\}$ of finite relations of arity $ar(R)$ over Dom, to each propositional symbol a family of truth values, and to each constant symbol $c[u]$ a family $\{D(c)[i] \mid i \in \mathcal{ID}\}$ of elements of Dom. Likewise, a relational instance D over a parameterized queue schema associates with each queue symbol R a family $\{D(R)[l][j] \mid l, j \in \mathcal{ID}, l \neq j\}$ of finite sequences of finite relations over Dom, each of arity $ar(R)$. For fixed $j \in \mathcal{ID}$ and arbitrary $l \neq j$ we say that $Q[l][j]$ is in-queue and $Q[j][l]$ is out-queue.

We assume familiarity with first-order logic (FO) over relational vocabularies. We adopt here an active domain semantics for FO formulas, as commonly done in database theory (e.g., see [1]).

We define next our model of Web applications as a set of n components $\mathcal{W}_1, \ldots, \mathcal{W}_n$, each specifying a Web site and/or Web service. A component consists of relational and queue schemas, some of which are parameterized, and a set of parameterized rules. We use n distinct parameters $u_1, \ldots, u_n$. The set of natural numbers $\{1, 2, \ldots, n\}$ is denoted $[1 : n]$.

Definition 1. *A* multi-user data-driven Web application *(in short DDWA*) is a set $\{\mathcal{W}_j\}_{1 \leq j \leq n}$ consisting of n components $\mathcal{W}_j = \langle \mathbf{D}_j, \mathbf{G}_j, \mathbf{S}_j[u_j], \mathbf{I}_j[u_j], \mathbf{A}_j[u_j]$, $\{\mathbf{Q}_{lj}[u_l][u_j]\}_{l \in [1:n]-\{j\}}, \{\mathbf{Q}_{jl}[u_j][u_l]\}_{l \in [1:n]-\{j\}}, \mathcal{R}_j, ACT_j\rangle$, where*

- $\mathbf{D}_j$, $\mathbf{G}_j$ *are relational schemas called* database *and* global state *schemas.* $\mathbf{S}_j[u_j]$, $\mathbf{I}_j[u_j]$, $\mathbf{A}_j[u_j]$ *are parameterized relational schemas called* local state*,* input*, and* action *schemas, respectively.*
- *For all distinct $l, j \in [1 : n]$, $\mathbf{Q}_{lj}[u_l][u_j]$ and $\mathbf{Q}_{jl}[u_j][u_l]$ are disjoint parameterized queue schemas. $\mathbf{Q}_{lj}[u_l][u_j]$ is in-queue and $\mathbf{Q}_{jl}[u_j][u_l]$ is out-queue for $\mathcal{W}_j$.*
- *For each j, the schemas $\mathbf{D}_j$, $\mathbf{G}_j$, $\mathbf{S}_j[u_j]$, $\mathbf{I}_j[u_j]$, $\mathbf{A}_j[u_j]$ are disjoint. For $l \neq j$, the schemas $\mathbf{D}_j$, $\mathbf{D}_l$ may share symbols, and so do $\mathbf{G}_j$ and $\mathbf{G}_l$.*
- *We refer to constant symbols in $\mathbf{I}_j$ as input constants and denote them $\mathbf{const}(\mathbf{I_j})$. The constant symbols in $\mathbf{D}_j$ are called database constants. We call $\mathbf{S}_j[u_j] \cup \mathbf{I}_j[u_j] \cup \mathbf{A}_j[u_j]$* the local schema *and its symbols are called* local symbols*. We denote by $\mathbf{Prev}_{\mathbf{I}_j}$ the relational vocabulary $\{prev_R \mid R \in \mathbf{I}_j - \mathbf{const}(\mathbf{I_j})\}$ where $prev_R$ has the same arity as R. The sets of local symbols of distinct components are disjoint.*
- *$\mathcal{R}_j$ is a set of* parameterized rules *using parameterized and unparameterized relation symbols of the schema of $\mathcal{W}_j$ and parameterized queue symbols of the schemas of $\mathcal{W}_1, \ldots, \mathcal{W}_n$, namely:*
  - *for each global state symbol $R \in \mathbf{G}_j$, one, both, or none of the following global state rules:*
    - *an insertion rule $R(\bar{x}) \leftarrow \varphi_R^+[\bar{u}](\bar{x})$*
    - *a deletion rule $\neg R(\bar{x}) \leftarrow \varphi_R^-[\bar{u}](\bar{x})$*

- *for each local state symbol $R \in \mathbf{S}_j$, one, both, or none of the following local state rules:*
  - *an insertion rule $R[u_j](\bar{x}) \leftarrow \varphi_R^+[\bar{u}](\bar{x})$*
  - *a deletion rule $\neg R[u_j](\bar{x}) \leftarrow \varphi_R^-[\bar{u}](\bar{x})$*
- *for each input symbol $R \in \mathbf{I}_j$, an input rule $Options_R[u_j](\bar{x}) \leftarrow \psi_R[\bar{u}](\bar{x})$*
- *for each action symbol $R \in \mathbf{A}_j$, an action rule $R[u_j](\bar{x}) \leftarrow \varphi_R[\bar{u}](\bar{x})$*
- *for each out-queue R in $\{\mathbf{Q}_{jl}\}_{l \in [1:n]-\{j\}}$, a send rule $R[u_j][u_l](\bar{x}) \leftarrow \varphi_R[\bar{u}](\bar{x})$*

where the arity of R is k, $\bar{x}$ is a k-tuple of distinct variables, $Options_R[u_j]$ is a parameterized symbol of arity k, $\psi_R[\bar{u}](\bar{x})$ is a FO formula over schema $\mathbf{D}_j \cup \mathbf{G}_j \cup \mathbf{S}_j[u_j] \cup \mathbf{const}(\mathbf{I_j}[\mathbf{u_j}]) \cup \mathbf{Prev}_{\mathbf{I}_j[u_j]} \cup \{\mathbf{Q}_{lj}[u_l][u_j]\}_{l \in [1:n]-\{j\}}$, with free variables $\bar{x}$, while $\varphi_R^+[\bar{u}](\bar{x})$, $\varphi_R^-[\bar{u}](\bar{x})$ and $\varphi_R[\bar{u}](\bar{x})$ are FO formulas over schema $\mathbf{D}_j \cup \mathbf{G}_j \cup \mathbf{S}_j[u_j] \cup \mathbf{I}_j[u_j] \cup \mathbf{Prev}_{\mathbf{I}_j[u_j]} \cup \{\mathbf{Q}_{lj}[u_l][u_j]\}_{l \in [1:n]-\{j\}}$, with free variables $\bar{x}$. All parameters are free.

– *ACT_j is a global state relation symbol.*

Each component $\mathcal{W}_j$ may be thought of as specifying a "type" of users, namely the humans currently browsing the Web site specified by $\mathcal{W}_j$ and/or the current instances of the Web service specified by $\mathcal{W}_j$. The users will become apparent in the later definitions, by instanciating the parameters with elements of $\mathbf{dom}_\infty$. The relation ACT$_j$ is meant to specify the finite (unbounded) set of current (or active) users of "type" $\mathcal{W}_j$.

The *body of a rule* of $\mathcal{R}_j$ is the formula on the right hand side of the arrow. On the left hand side of the arrow, the *head of the rule* specifies a single relation or queue symbol. At most two parameters may appear in the head because a rule may change either a local relation or an out-queue with respect to j. The local state symbols that appear in the body are tagged by u_j. The body may contain any in-queue symbol with respect to j. Thus a rule of $\mathcal{R}_j$ may specify messages received(sent) by one user of "type" $\mathcal{W}_j$ from (to) other "types" of users. This syntax allows for the definition of a transition (see Definitions 4 and 5) specifying changes due to a single user.

Intuitively, $prev_I$ refers to the most recent non-empty input I. Input constants represent the user's ability to type in arbitrary data (e.g. name, password.) Database constants represent the possibility of presenting sets of predefined values (e.g. menus) for the user to choose (e.g. by clicking on a button or hiperlink). A *DDWA** with one component is called a *simple DDWA**.

We distinguish between *flat* and *nested queues* as done in [5]. A flat queue message contains one tuple only. A nested queue message contains an arbitrary number of tuples. We denote by $\mathbf{Q}_{lj}^f$ ($\mathbf{Q}_{lj}^n$) the flat (nested) queues connecting $\mathcal{W}_l$ and $\mathcal{W}_j$.

A bank loan Web application is modeled in [5] as the composition of four communicating Web services: Loan Officer ($\mathcal{O}$), Loan Manager ($\mathcal{M}$), Credit Reporting Agency ($\mathcal{CR}$), and Applicant Customer ($\mathcal{A}$). The model represents only one instance of each Web service and the communication is done exclusively by message passing. We specify now this Web application as a *DDWA**. The Web services may have arbitrarily many instances. Both $\mathcal{O}$ and $\mathcal{M}$ may have several users, sharing information about the bank customers. $\mathcal{A}$ may have arbitrarily many users. Once a loan officer starts reviewing a loan application, additional dynamic information (regarding, for instance, credit reports) is maintained locally and is not shared with other loan officers. The distinction between local and global state allows the users to interact and view selectively the database. The Web service instances may exchange messages.

Example 1. For convenience, we refer to *Web pages* as done in [4]. Technically, Web pages are represented by local state propositions (e.g. *homeP*, *ackP*.) A user is currently on a Web page iff the corresponding proposition is *true*. Some elements of dom_∞ like "*home*" or "*submitApp*", represent buttons or menu choices on Web pages, others like "*excellent*" or "*poor*" represent database constants. For better readability we write the parameters when they appear in rules only. An out-queue symbol R is written $!R$ and an in-queue symbol R is written $?R$.

The four components $\mathcal{W}_1$, $\mathcal{W}_2$, $\mathcal{W}_3$, $\mathcal{W}_4$ of the *DDWA** specify, respectively, $\mathcal{A}$ and the bank customer Web site, $\mathcal{O}$ and the loan officer Web site, $\mathcal{CR}$, $\mathcal{M}$ and the loan manager Web site. The parameters are denoted a, o, r, m respectively.

We specify first $\mathcal{W}_1$: $\mathbf{D}_1 = \emptyset$, $\mathbf{G}_1 = \{applications(cid, loan)\}$, $\mathbf{S}_1 = \{homeP, ackP\}$, $\mathbf{I}_1 = \{cId, loanType(loan), click(button)\}$, $\mathbf{A}_1 = \{ack\}$, $\mathbf{Q}_{12} = \mathbf{Q}^f_{12} = \{apply(cid, loan)\}$ The queue schemas $\mathbf{Q}_{21}$, $\mathbf{Q}_{13}$, $\mathbf{Q}_{31}$, $\mathbf{Q}_{14}$, $\mathbf{Q}_{41}$ are empty. Some of the parameterized rules of $\mathcal{R}_1$ are:

(1) $homeP[a] \leftarrow \neg homeP[a]$
(2) $Options_{loanType}[a](loan) \leftarrow homeP[a] \wedge (loan = \text{“}home\text{”} \vee loan = \text{“}car\text{”})$
(3) $Options_{click}[a](x) \leftarrow homeP[a] \wedge (x = \text{“}submitAppl\text{”} \vee x = \text{“}status\text{”})$
(4) $!apply[a][o](cid, l) \leftarrow loanType[a](l) \wedge cId[a] = cid \wedge click[a](\text{“}submitAppl\text{”})$
(5) $ack[a] \leftarrow \exists\, l\ homeP[a] \wedge loanType[a](l) \wedge click(\text{“}submitAppl\text{”})$
(6) $ackP[a] \leftarrow homeP[a] \wedge click[a](\text{“}submitAppl\text{”})$
(7) $\neg homeP[a] \leftarrow homeP[a] \wedge click[a](\text{“}submitAppl\text{”})$

Once she is on the home page (rule 1), a bank customer applying for a loan must provide a bank identifier (input constant *cId*) and choose a loan type (input rule 2). A click on a button (rule 3) starts the application process by sending a singleton message $\{(cid, l)\}$ to $\mathcal{O}$ on a flat queue. This is done using the send rule 4 with the queue symbol *apply* in the head. The three inputs in the body are parameterized by a, as they belong to the local schema of $\mathcal{W}_1$. The out-queue symbol in the head is parameterized by a and o to signify the sender "type" $\mathcal{W}_1$ and the receiver "type" $\mathcal{W}_2$. An action rule (5) displays an acknowledgment on a new page (rules 6, 7). The loan applications follow a business process that involves the other three components of the Web application. We will see later that the global state *applications* is shared by $\mathcal{W}_1$ and $\mathcal{W}_2$.

The behavior of a Web application is expressed as an infinite sequence of schema instances. A schema instance consists of all the database, global and local state, input, previous input, action, in-queue and out-queue instances associated with all the active users of the Web application, and represents the configuration of the Web application at a certain moment. Next we formalize the notion of configuration. A *DDWA** has *individual configurations* (consisting of instances of the parameterized local, in-queue and out-queue schemas) and *global configurations* (consisting of instances of the database, the global state, the set of all individual configurations of active users, and the sets of active IDs, for each component $\mathcal{W}_j$).

Let $\mathcal{W}^* = \{\mathcal{W}_j\}_{1 \leq j \leq n}$ be a *DDWA**. Let $\mathcal{ID}_j \subseteq \mathbf{dom}_\infty$ for $j \in [1 : n]$ be sets of IDs, assumed disjoint.

Definition 2. *An* individual $\mathcal{W}^*$-configuration *is a tuple* $\mathcal{K} = \langle i, S[i], I[i], P[i], A[i], \{Q_{lj}[i'][i]\}_{l \in [1:n]-\{j\}, i' \in \mathcal{ID}_l}, \{Q_{jl}[i][i']\}_{l \in [1:n]-\{j\}, i' \in \mathcal{ID}_l}\rangle$ *for some index* $j \in [1 : n]$ *and ID* $i \in \mathcal{ID}_j$. $S[i], I[i], P[i], A[i]$ *are instances over the parameterized schemas* $\mathbf{S}_j[u_j]$, $\mathbf{I}_j[u_j]$, $\mathbf{Prev}_{\mathbf{I}_j[u_j]}$, $\mathbf{A}_j[u_j]$, *respectively. For* $l \in [1 : n] - \{j\}$ *and* $i' \in \mathcal{ID}_l$, $Q_{lj}[i'][i]$ *and* $Q_{jl}[i][i']$ *are instances over the parameterized queue schemas* $\mathbf{Q}_{lj}[u_l][u_j]$, $\mathbf{Q}_{jl}[u_j][u_l]$, *respectively, such that only finitely many of them are non-empty.*

We say that $\mathcal{K}$ *is an individual* $\mathcal{W}^*$*-configuration of* $\mathcal{W}_j$ *with ID* i. $\mathcal{K}$ *communicates with an individual* $\mathcal{W}^*$*-configuration of* $\mathcal{W}_l$ *with ID* i' *if* $Q_{lj}[i'][i]$ *or* $Q_{jl}[i][i']$ *is non-empty, for some* $l \in [1 : n] - \{j\}$. *An* initial *individual* $\mathcal{W}^*$*-configuration has all instances empty, except possibly input and in-queue instances.*

The queue instances in $Q_{lj}[i'][i]$ ($Q_{jl}[i][i']$) specify channels for messages received (sent) by a user with ID i from (to) a user with ID i'. If it is clear from the context we may abbreviate "an individual $\mathcal{W}^*$-configuration of $\mathcal{W}_j$ with ID i" by "an ID i".

Example 2. The tuple $\langle a_1, \{true, false\}, \{\text{“23”}, \{(\text{“}home\text{”})\}, \{(\text{“}submitAppl\text{”})\}\}, \emptyset, \emptyset, \emptyset, \{(\text{“23”}, \text{“}home\text{”})[a_1][o_1]\}\rangle$ is an individual $\mathcal{W}^*$-configuration of $\mathcal{W}_1$ with ID $a_1 \in \mathcal{ID}_1$ that communicates with an ID o_1 of $\mathcal{W}_2$. It specifies a customer currently on the home page of the bank Web site. The customer inputs the bank ID 23 and submits a home loan application. The previous input, action and in-queue instances are empty. The loan officer ID o_1 is chosen arbitrarily from the set of active IDs of $\mathcal{W}_2$, specified by the global configuration of a *DDWA**, to be defined.

Definition 3. *A* global $\mathcal{W}^*$-configuration *is a set* $\{\langle D_j, G_j, \mathcal{C}_j, Act_j\rangle | 1 \leq j \leq n\}$, *where* D_j *is an instance of the database schema* $\mathbf{D}_j$, G_j *is an instance of the global state schema* $\mathbf{G}_j$, $\mathcal{C}_j$ *is a finite set of individual* $\mathcal{W}^*$*-configurations of* $\mathcal{W}_j$ *with distinct IDs, such that* $Act_j \subseteq \mathcal{ID}_j$ *is the set of all IDs in* $\mathcal{C}_j$ *and it is also an instance of* ACT_j. *In addition, for each* $l \in [1 : n] - \{j\}$, $\mathcal{C}_j$ *satisfies the following conditions:*

- *(*) If* $\mathcal{K} \in \mathcal{C}_j$ *has ID* i *and communicates with an ID* i' *of* $\mathcal{W}_l$, *then* $\mathcal{C}_l$ *contains an individual* $\mathcal{W}^*$*-configuration* $\mathcal{K}'$ *of* $\mathcal{W}_l$, *with ID* i', *such that for every queue symbol* $R \in \mathbf{Q}_{jl}$ ($\mathbf{Q}_{lj}$) *the out-queue (in-queue) instance of* R *in* $\mathcal{K}$ *and the in-queue (out-queue) instance of* R *in* $\mathcal{K}'$ *coincide and are tagged by both IDs* i *and* i'.
- *(**) If* R *is a global state (database) relation symbol shared by* $\mathcal{W}_j$ *and* $\mathcal{W}_l$ *then its instances in* G_j *and* G_l (D_j *and* D_l) *coincide.*

*For an individual $\mathcal{W}^*$-configuration $\mathcal{K} \in \mathcal{C}_j$, the tuple $\langle D_j, G_j, \mathcal{K}\rangle$ is called a $\mathcal{W}_j$-*configuration*. An* initial *global $\mathcal{W}^*$-configuration has all global state instances empty, and all the individual $\mathcal{W}^*$-configurations are initial.*

Every queue instance in a global $\mathcal{W}^*$-configuration is both an out-queue in the sender's individual $\mathcal{W}^*$-configuration and an in-queue in the receiver's $\mathcal{W}^*$-individual configuration.

Example 3. The $\mathcal{W}_1$-configuration corresponding to the individual $\mathcal{W}^*$-configuration in Example 2 is $\langle -, G, \langle a_1, \{true, false\}, \{\text{“23”}, \{(\text{“}home\text{”})\}, \{(\text{“}submitAppl\text{”})\}\}, \emptyset, \emptyset, \emptyset, \{(\text{“23”}, \text{“}home\text{”})[a_1][o_1]\}\rangle\rangle$. It has no database instance for the empty schema $\mathbf{D}_1$. The global state instance G is a set of tuples $(cid, loan)$ specifying applications already received.

An atomic transition that updates the global configuration is due to a single user of some component $\mathcal{W}_j$. All the changes appear in this user's W_j-configuration and in the queues of those users (at most one for each $\mathcal{W}_l$, $l \neq j$) communicating with the $\mathcal{W}_j$ user during the transition. The transition relation of a component $\mathcal{W}_j$ defines for every current $\mathcal{W}_j$-configuration its legal successor $\mathcal{W}_j$-configurations, reachable in one atomic step.

Definition 4. *Let $\langle D, G, \mathcal{K}\rangle$ and $\langle D', G', \mathcal{K}'\rangle$ be two $\mathcal{W}_j$-configurations, where $\mathcal{K} = \langle i, S[i], I[i], P[i], A[i], \{Q_{lj}[i'][i]\}_{l\in[1:n]-\{j\}, i'\in\mathcal{ID}_l}, \{Q_{jl}[i][i']\}_{l\in[1:n]-\{j\}, i'\in\mathcal{ID}_l}\rangle$ and $\mathcal{K}' = \langle i, S'[i], I'[i], P'[i], A'[i], \{Q'_{lj}[i'][i]\}_{l\in[1:n]-\{j\}, i'\in\mathcal{ID}_l}, \{Q'_{jl}[i][i']\}_{l\in[1:n]-\{j\}, i'\in\mathcal{ID}_l}\rangle$ for an ID $i \in \mathcal{ID}_j$. We say that $\langle D', G', \mathcal{K}'\rangle$ is a* legal successor *of $\langle D, G, \mathcal{K}\rangle$ in $\mathcal{W}_j$ iff $D = D'$ and for some IDs $i_l \in \mathcal{ID}_l$ for $l \in [1:n] - \{j\}$ the following hold:*

- *For each symbol $R \in \mathbf{I}_j$ of arity $k > 0$, $I'(R) \subseteq \{v\}$ for some $v \in Options_R$, where $Options_R$ is the result of evaluating $\psi_R[\bar{u}](\bar{x})$ on D, G, $S[i]$, $P[i]$, and $\{f(Q_{lj})[i_l][i]\}_{l\in[1:n]-\{j\}}$. For each proposition $R \in \mathbf{I}_j$, $I'(R)$ is a truth value, and for each constant symbol $c \in \mathbf{I}_j$, $I'(c)$ is an element of $\mathbf{dom}_\infty$.*
- *For each symbol $R \in \mathbf{S}_j$, $S'(R)$ is the result of evaluating $(\varphi^+_R[\bar{u}](\bar{x}) \wedge \neg\varphi^-_R[\bar{u}](\bar{x})) \vee (R[u_j](\bar{x}) \wedge \varphi^-_R[\bar{u}](\bar{x}) \wedge \varphi^+_R[\bar{u}](\bar{x})) \vee (R[u_j](\bar{x}) \wedge \neg\varphi^-_R[\bar{u}](\bar{x}) \wedge \neg\varphi^+_R[\bar{u}](\bar{x}))$ on D, G, $S[i]$, $I[i]$, $P[i]$, and $\{f(Q_{lj}[i_l][i])\}_{l\in[1:n]-\{j\}}$ where $\varphi^\epsilon_R[\bar{u}](\bar{x})$ is taken to be false if not provided, $\epsilon \in \{+,-\}$. R remains unchanged if no insertion or deletion rule is specified for it. For each symbol $R \in \mathbf{G}_j$, $G'(R)$ is obtained likewise, except R is not parameterized.*
- *For each symbol $R \in \mathbf{A}_j$, $A'(R)$ is the result of evaluating $\varphi_R[\bar{u}](\bar{x})$ on D, G, $S[i]$, $I[i]$, $P[i]$, and $\{f(Q_{lj})[i_l][i]\}_{l\in[1:n]-\{j\}}$.*
- *For each symbol $prev_R$ in $\mathbf{Prev}_{\mathbf{I_j}}$, $P'(prev_R) = I'(R)$ if $I'(R)$ is non-empty, otherwise $P'(prev_R) = P(R)$.*
- *For each symbol $R \in \mathbf{Q}_{jl}$, let m_R be the result of evaluating $\varphi_R[\bar{u}](\bar{x})$ on D, G, $S[i]$, $I[i]$, $P[i]$, and $\{f(Q_{lj}[i_l][i])\}_{l\in[1:n]-\{j\}}$. If $R \in \mathbf{Q}^n_{jl}$, then $Q'_{jl}(R)[i][i_l]$ is obtained by enqueuing m_R into $Q_{jl}(R)[i][i_l]$. If $R \in \mathbf{Q}^f_{jl}$, then if m_R is non-empty, $Q'_{jl}(R)[i][i_l]$ is obtained by enqueuing into $Q_{jl}(R)[i][i_l]$ a singleton containing a non-deterministically picked tuple $v \in m_R$. If m_R is*

empty, $Q'_{jl}(R)[i][i_l] = Q_{jl}(R)[i][i_l]$ (the queue does not change). For $i' \neq i_l$, $Q'_{jl}(R)[i][i'] = Q_{jl}(R)[i][i']$.

- *For each symbol $R \in \mathbf{Q}_{lj}$, if R is mentioned in the rules $\mathcal{R}_j$, then $Q'_{lj}(R)[i_l][i]$ is obtained by dequeuing the first message from $Q_{lj}(R)[i_l][i]$. Otherwise, $Q'_{lj}(R)[i_l][i] = Q_{lj}(R)[i_l][i]$. For $i' \neq i_l$, $Q'_{lj}(R)[i'][i] = Q_{lj}(R)[i'][i]$.*

Notice that when evaluating a rule body, the parameters $u_1, \ldots, u_n$ are instantiated by the IDs $i_1, \ldots i_{j-1}, i, i_{j+1}, \ldots, i_n$.

Example 4. The rules in Example 1 yield the following successor $\mathcal{W}_1$-configuration to the $\mathcal{W}_1$-configuration in Example 3: $\langle -, G, \langle a_1, \{false, true\}, \emptyset, \{\{(\text{“}home\text{”})\}, \{(\text{“}submitAppl\text{”})\}\}, \{true\}, \emptyset, \{(\text{“23”}, \text{“}home\text{”})[a_1][o_1]\}\rangle\rangle$. It shows: changed Web page, empty input, action *ack* performed, and the singleton message still in the out-queue.

The *transition relation* for $\mathcal{W}^*$ defines for each current global $\mathcal{W}^*$-configuration its legal successor global $\mathcal{W}^*$-configurations, reachable in one atomic step. This step is performed by a $\mathcal{W}_j$ user with an ID in $\mathcal{ID}_j$, for some $j \in [1:n]$. There are three kinds of steps, according to whether the user is active and stays active, or just logs on or off. By firing the $\mathcal{R}_j$ rules tagged by IDs, the global state in $\mathcal{W}_j$, and the individual configuration of the performing ID may change according to Definition 4. The set of active $\mathcal{W}_j$ IDs may also change, if the user (with the performing ID) logs on or off. We assume that in one step, for each pair $\mathcal{W}_j$, $\mathcal{W}_l$ of components, an instance of $\mathcal{W}_j$ may exchange messages with only one instance of $\mathcal{W}_l$, but in subsequent transitions it may exchange messages with other instances of $\mathcal{W}_l$. This assumption is consistent with the general model for concurrency based on interleaving.

Definition 5. *A global $\mathcal{W}^*$-configuration $\{\langle D'_l, G'_l, \mathcal{C}'_l, Act'_l\rangle | 1 \leq l \leq n\}$ is a* legal successor of $\{\langle D_l, G_l, \mathcal{C}_l, Act_l\rangle | 1 \leq l \leq n\}$ in $\mathcal{W}^*$ *iff $D'_l = D_l$ for each $l \in [1:n]$, and for some $j \in [1:n]$, one of the following holds:*

1. [active user step] *$Act'_j = Act_j$ and $\mathcal{C}'_j = \mathcal{C}_j - \{\mathcal{K}\} \cup \{\mathcal{K}'\}$ for individual $\mathcal{W}^*$-configurations $\mathcal{K}, \mathcal{K}' \in \mathcal{C}_j$ with ID $i \in Act_j$, such that $\langle D'_j, G'_j, \mathcal{K}'\rangle$ is a legal successor of $\langle D_j, G_j, \mathcal{K}\rangle$ in $\mathcal{W}_j$, for some IDs $i_l \in Act_l$ for $l \in [1:n] - \{j\}$.*
2. [log on step] *$Act'_j = Act_j \cup \{i\}$ for some non-active ID $i \in \mathcal{ID}_j - Act_j$, $\mathcal{C}'_j = \mathcal{C}_j \cup \{\mathcal{K}\}$, $\mathcal{K}$ is the individual $\mathcal{W}^*$-configuration of $\mathcal{W}_j$ with ID i such that $\langle D'_j, G'_j, \mathcal{K}\rangle$ is a legal successor of $\langle D_j, G_j, \mathcal{K}_0\rangle$ in $\mathcal{W}_j$ for some IDs $i_l \in Act_l$ for $l \in [1:n] - \{j\}$, and $\mathcal{K}_0$ is an initial individual $\mathcal{W}^*$-configuration of $\mathcal{W}_j$.*
3. [log off step] *$Act'_j = Act_j - \{i\}$ for some ID $i \in Act_j$, $\mathcal{C}'_j = \mathcal{C}_j - \{\mathcal{K}\}$, $\mathcal{K}$ is the individual $\mathcal{W}^*$-configuration of $\mathcal{C}_j$ with ID i, and G'_j is the global state in a legal successor of $\langle D_j, G_j, \mathcal{K}\rangle$ in $\mathcal{W}_j$, for some IDs $i_l \in Act_l$ for $l \in [1:n] - \{j\}$.*

In addition, for each $l \in [1:n] - \{j\}$, $Act'_l = Act_l$, $G'_l = G_l$, and the set $\mathcal{C}'_l$ is obtained from $\mathcal{C}_l$ by updating each queue tagged by IDs i, i_l in the individual $\mathcal{W}^*$-configuration of $\mathcal{W}_l$ with ID i_l, with one exception: a log off step does not update the in-queues.

Notice that all $\mathcal{W}_j$ instances share the database $\mathbf{D}_j$ and global state $\mathbf{G}_j$. For $j \neq l$, $\mathbf{D}_j$ and $\mathbf{D}_l$ are not necessarily disjoint, and so are $\mathbf{G}_j$ and $\mathbf{G}_l$. Hence several components may share (part of) the database and global state. Local state, as well as input, action, in-queue and out-queue cannot be shared because they are tagged by the user's id. It is understood that the global state relations shared by two components change accordingly at each transition.

Definition 6. *A* run *of $\mathcal{W}^*$ is an infinite sequence $\rho = \{\rho_t\}_{t \geq 0}$ of global $\mathcal{W}^*$-configurations, such that ρ_0 is initial, and for every $t \geq 0$, ρ_{t+1} is a legal successor of ρ_t in $\mathcal{W}^*$. We denote by D the union of all database instances D_j, for $1 \leq j \leq n$, that appear in every ρ_t. We say that ρ is a run on database D.*

The choice of ID to perform the next step in a $\mathcal{W}^*$ run (and hence the choice of next global $\mathcal{W}^*$-configuration) is non-deterministic. Notice that at each step, if $i_k \in \mathcal{ID}_k$ is the ID that performs the transition, then all the rules in $\mathcal{R}_k$ are simultaneously interpreted over the current database D_k, global state G_k, and individual configuration with ID i_k. The input used in a transition from ρ_i to ρ_{i+1} belongs to ρ_i, while the action "happens" in ρ_{i+1}. If some $\mathcal{ID}_j$ is infinite, there may be infinitely many IDs in a run, even though in each global $\mathcal{W}^*$-configuration there are only finitely many IDs.

Each input constant may be assigned an arbitrary value from dom_∞ only once for each active ID. An input constant may be assigned different values by different IDs in a run. Also, since users may log on and off, an input constant may be assigned different values by the same ID. Database constants have fixed values for the whole run.

Example 5. We specify $\mathcal{O}$ and the loan officer Web site together as a component $\mathcal{W}_2$. The loan officer service assigns non-deterministically a loan officer ID to receive a new loan application, which is immediately stored both in the global state *applications* and in the local state *newAppl* tagged by the parameter *o*. Simultaneously, a message is sent automatically to the credit reporting service, on a flat out-queue:

$$\begin{aligned}&!getRating[o][r](ssn) \leftarrow \\ &\qquad \exists\ cid, loan, name\ ?apply[a][o](cid, loan) \wedge customer(cid, name, ssn)\end{aligned}$$

This message is parameterized by o and r, and contains the applicant customer's SSN obtained by querying the database *customer*. The credit reporting service sends back the credit rating category on the flat queue *rating*, which is in-queue for $\mathcal{W}_2$. The Web application implements a bank policy requiring that loan applications from customers with excellent rating are automatically approved, and those with poor rating are automatically denied. Other credit ratings require further processing. The action rule

$$\begin{aligned}&letter[o](cid, name, loan, dec) \leftarrow \exists\ ssn\ customer(cid, name, ssn) \wedge \\ &\qquad [newAppl[o](cid, loan) \wedge \\ &\qquad (?rating[r][o](ssn, \text{“}excellent\text{”}) \wedge dec = \text{“}approved\text{”} \\ &\qquad \vee ?rating[r][o](ssn, \text{“}poor\text{”}) \wedge dec = \text{“}denied\text{”})] \\ &\qquad\ \vee [applications(cid, loan) \wedge ?decision[m][o](cid, name, dec)]\end{aligned}$$

specifies the sending of letters to applicants. If the credit rating is neither *excellent* nor *poor* then a decision comes from a manager on the flat in-queue *decision*.

Web applications specified using the multi-user model feature interesting behaviors that are not expressible in the previous models [4,5]. For example:

1. The bank loan $DDWA^*$ involves several loan officers and managers processing applications received concurrently from an unlimited number of customers. When a component's rules are fired, the parameters are instantiated by IDs and each rule body is evaluated over local and queue instances tagged by the same IDs.
2. By using both global and parameterized local state relations, as well as parameterized send rules, the multi-user specification can implement a bank policy that requires each loan application to be dealt with by a single loan officer. Once a bank customer applies for a loan, the system chooses non-deterministically a loan officer ID and stores the application data in this officer's local state. Subsequently, this ID is used by all $\mathcal{W}_2$ rules specifying the workflow this loan application goes through.
3. The ID is non-deterministically chosen from among all active loan officer IDs. This makes sense if the bank employs a policy of assigning new loan applications arbitrarily to any loan officer. In practice such a policy may be unfair, unless there is a mechanism in place that keeps track of current work loads. In our high-level specification, global state propositions can be used to specify the availability of loan officers. These propositions may be set to true/false by the loan officers themselves.
4. By allowing the loan officer service and the customer Web site to share the global state *applications*, the customers can keep track of the status of their applications. The local state relation $myAppl(cid, loan, status)$ is added to $\mathbf{S}_1$ and the following state updating rules are added to $\mathcal{R}_1$:

$$
\begin{aligned}
&myAppl[a](cid, l, \text{“}sent\text{”}) \leftarrow loanType[a](l) \wedge cId[a] = cid \wedge click[a](\text{“}submitAppl\text{”})\\
&myAppl[a](cid, l, \text{“}received\text{”}) \leftarrow myAppl[a](cid, l, \text{“}sent\text{”}) \wedge applications(cid, l)\\
&myAppl[a](cid, l, \text{“}processed\text{”}) \leftarrow myAppl[a](cid, l, \text{“}received\text{”}) \wedge\\
&\qquad\qquad\qquad \neg applications(cid, l)
\end{aligned}
$$

An action rule allows customers to view the status of their applications:

$$showStatus[a](cid, l, st) \leftarrow homeP[a] \wedge click[a](\text{“}status\text{”}) \wedge myAppl[a](cid, l, st)$$

Example 6. Some desirable properties of the bank loan Web application are: (1) Every bank customer applying for a loan receives a written answer. (2) Only one loan officer is in charge with reviewing an application. (3) Every applicant customer whose credit rating is neither excellent nor poor requires a manager's decision before a letter is sent to the applicant. (4) A request for credit rating must be sent before a message tagged by the same loan officer ID and containing a credit rating is received. (5) It is never the case that the same loan application is assigned to two loan officers. In Section 3 we show how to express these properties in a formal language.

The model of [5] could be used to specify a bounded number of users (or instances), identified by new database constants. Our model is defined for the more realistic situation of an unbounded number of users. Even though in general this leads to undecidability (see Section 4) thus making algorithmic verification impossible, the model can still be used for the formal specification of complex Web applications, whose correctness could be checked using a theorem prover e.g. PVS [8]. Moreover, algorithmic verification is possible with some restrictions, as shown in Section 5 where Theorem 3 holds for an unbounded number of users of Web services with empty queue schemas, in the presence of a bounded number of users of Web services with non-empty queue schemas. *DDWA** properties allow quantification over ID variables, which is not possible if database constants were used to refer to users.

Users may log on/off at any time. If a message is sent to a user that has logged off, the message stays in the sender's out-queue until the ID becomes active again. As this is not guaranteed to happen, messages may never be received.

3 Linear Time Temporal Properties

We use first-order linear time temporal logic (in short, LTL-FO) to express properties of runs of *DDWA**s. Let $\mathcal{W}^* = \{\mathcal{W}_j\}_{1 \leq j \leq n}$ be a *DDWA**. FO formulas over the schema of $\mathcal{W}^*$ may contain two types of variables: *ID variables* and *non-ID (usual) variables.* The ID variables appear inside square brackets next to a local relation or queue symbol. We assume two disjoint, infinite sets: a set $\mathbf{V}$ of non-ID variables and a set $\mathbf{V}^{id} = \cup_{1 \leq j \leq n} \mathbf{V}^{id}_j$ of ID variables, partitioned into infinite sets of $\mathcal{W}_j$-ID variables. For each $j \in [1:n]$, let C_j be the set of constants in the $\mathcal{W}_j$ schema, and $C = \cup_{1 \leq j \leq n} C_j$. Any occurrence of an input constant $c \in C_j$ in a formula is of the form $c[u]$ for some $u \in \mathbf{V}^{id}_j$. The following are atomic FO formulas: $R[u](\bar{x}), S(\bar{y}), ?P[v][v'](\bar{x}), !P[v][v'](\bar{x}), y_1 = y_2$ where for some $j, l \in [1:n]$, $j \neq l$, $R \in \mathbf{S}_j \cup \mathbf{I}_j \cup \mathbf{A}_j$, $S \in \mathbf{D}_j \cup \mathbf{G}_j$, $P \in \mathbf{Q}_{jl}$, $u, v \in \mathbf{V}^{id}_j \cup C_j$, $v' \in \mathbf{V}^{id}_l \cup C_l$, $\bar{x}, \bar{y}$ are tuples of variables in $\mathbf{V}^{id} \cup \mathbf{V}$ or constants in C, of appropriate arity, and $y_1, y_2 \in \mathbf{V}^{id} \cup \mathbf{V} \cup C$. The FO formulas are obtained as usual from atomic formulas using negation, disjunction and quantifiers. The language LTL-FO is obtained by closing FO under negation, disjunction, and the following formula formation rule: If φ and ψ are formulas, then $\mathbf{X}\varphi$ and $\varphi \mathbf{U} \psi$ are formulas. Free and bound variables are defined in the obvious way. An LTL-FO formula with free ID variables $\bar{u}$ and free non-ID variables $\bar{x}$ is denoted $\varphi[\bar{u}](\bar{x})$. The universal closure of an LTL-FO formula $\varphi[\bar{u}](\bar{x})$ is the formula $\forall \bar{x}[\bar{u}]\varphi[\bar{u}](\bar{x})$. An LTL-FO sentence is the universal closure of an LTL-FO formula. Notice that quantifiers cannot be applied to formulas containing temporal operators, except by taking the universal closure of the entire formula, yielding an LTL-FO sentence.

We define next the semantics of LTL-FO. Let $\rho = \{\rho_t\}_{t \geq 0}$ be a run of $\mathcal{W}^*$. Let $Dom(\rho)$ be the active domain of ρ, which consists of all the elements of dom_∞ occurring in relations or as interpretations of constants in ρ, together with all active IDs of ρ. We define first the satisfaction of an FO formula ψ by a global

$\mathcal{W}^*$-configuration $\rho_t = \{\langle D_j, G_j^t, \mathcal{C}_j^t, Act_j^t\rangle | 1 \leq j \leq n\}$ of ρ. For $1 \leq j \leq n$ and $t \geq 0$, the set $\mathcal{C}_j^t$ consists of individual $\mathcal{W}^*$-configurations of the form $\langle i, S[i], I[i], P[i], A[i], \{Q_{kj}[i'][i]\}_{k\in[1:n]-\{j\}, i'\in\mathcal{ID}_k}, \{Q_{jk}[i][i']\}_{k\in[1:n]-\{j\}, i'\in\mathcal{ID}_k}\rangle$ where $i \in Act_j^t$.

Definition 7. *We associate with ρ_t an FO structure $\mathcal{S}$ over the vocabulary $\mathcal{V} = \cup_{1\leq j\leq n}\mathcal{V}_j \cup \{R_\alpha^* | R \in \mathbf{Q}_{jk}, \alpha \in \{f, l\}, k \in [1:n]-\{j\}, ar(R_\alpha^*) = ar(R)+2\}$, where $\mathcal{V}_j = \mathbf{D}_j \cup \mathbf{G}_j \cup \{ACT_j\} \cup \{c_u | c \in \mathbf{const}(\mathbf{I}_j), u \in V_j^{id}\} \cup \{R^* | R \in (\mathbf{I}_j - \mathbf{const}(\mathbf{I}_j)) \cup \mathbf{prev}_{\mathbf{I}_j} \cup \mathbf{S}_j \cup \mathbf{A}_j, ar(R^*) = ar(R)+1\}$. The domain of $\mathcal{S}$ is $Dom(\rho)$. The symbols of $\mathcal{V}$ are interpreted in $\mathcal{S}$ as follows. For each $1 \leq j \leq n$, $\mathcal{S}(ACT_j) = Act_j^t$ and*

- *for a symbol $R \in \mathbf{D}_j \cup \mathbf{G}_j$, $\mathcal{S}(R) = D_j(R)$, respectively $\mathcal{S}(R) = G_j^t(R)$.*
- *For a symbol $R \in \mathbf{S}_j \cup \mathbf{A}_j$, $\mathcal{S}(R^*) = \cup_{i\in Act_j^t}(\{i\} \times S[i](R))$, respectively $\mathcal{S}(R^*) = \cup_{i\in Act_j^t}(\{i\} \times A[i](R))$.*
- *For a relational symbol $R \in \mathbf{I}_j$, if the transition from ρ_t to ρ_{t+1} is a log on step with input I and ID $i' \notin Act_j^t$, then $\mathcal{S}(R^*) = \cup_{i\in Act_j^t}(\{i\} \times I[i](R)) \cup (\{i'\} \times I(R))$ and $\mathcal{S}(prev_{R^*})$ is empty, otherwise $\mathcal{S}(R^*) = \cup_{i\in Act_j^t}(\{i\} \times I[i](R))$ and $\mathcal{S}(prev_{R^*}) = \cup_{i\in Act_j^t}(\{i\} \times P[i](R))$.*
- *For a constant symbol $c \in \mathbf{I}_j$, if $\mathcal{S}(u) \in Act_j^t$, then $\mathcal{S}(c_u) = I^{\mathcal{S}(u)}(c)$, otherwise $\mathcal{S}(c_u)$ is undefined.*
- *For $k, k' \in [1:n]$, $k \neq k'$, for a queue symbol $R \in \mathbf{Q}_{kk'}$, $\mathcal{S}(R_f^*) = \cup_{i\in Act_k^t} \cup_{i'\in Act_{k'}^t} (\{(i,i')\} \times f(Q_{kk'}[i][i'])(R))$; $\mathcal{S}(R_l^*) = \cup_{i\in Act_k^t} \cup_{i'\in Act_{k'}^t} (\{(i,i')\} \times l(Q_{kk'}[i][i'])(R))$.*

For a variable $y \in \mathbf{V}$, $\mathcal{S}(y) \in Dom(\rho)$. For an ID variable $u \in \mathbf{V}_j^{id}$, $\mathcal{S}(u) \in \mathcal{ID}_j$. Quantifiers range over $Dom(\rho)$ if they bound non-ID variables, and over Act_j^t if they bound $\mathcal{W}_j$-ID variables. Let ψ^* be the FO formula obtained by replacing in ψ each local atom $R[u](\bar{y})$ by $R^*(u, \bar{y})$, each occurrence $c[u]$ of an input constant c, by c_u, each queue atom $?R[u][v](\bar{y})$ by $R_f^*(u, v, \bar{y})$, and each queue atom $!R[u][v](\bar{y})$ by $R_l^*(u, v, \bar{y})$. Then $\rho_t \models \psi$ iff ψ^* is true in $\mathcal{S}$.

The suffix ρ^t satisfies an LTL-FO sentence $\forall\bar{x}\bar{u}\varphi[\bar{u}](\bar{x})$ iff for every valuation ν of $\bar{x}$ on $Dom(\rho)$ and for every valuation μ of $\bar{u}$ on $\mathcal{ID}_1 \times \ldots \times \mathcal{ID}_n$, $\rho^t \models \varphi[\mu(\bar{u})](\nu(\bar{x}))$. The latter is defined inductively on t and on the structure of the formula:

- For an FO sentence ψ, $\rho^t \models \psi$ iff $\rho_t \models \psi$;
- $\rho^t \models \mathbf{X}\psi$ iff $\rho^{t+1} \models \psi$;
- $\rho^t \models \psi_1\mathbf{U}\psi_2$ iff for some $s \geq t$, $\rho^s \models \psi_2$ and for every r, $t \leq r < s$, $\rho^r \models \psi_1$.

We use common temporal operators $\mathbf{F}$ (*eventually*), $\mathbf{G}$ (*always*) and $\mathbf{B}$ (*before*), defined in terms of the others: $\mathbf{F}\psi = true\mathbf{U}\psi$ ("eventually ψ holds"), $\mathbf{G}\psi = \neg\mathbf{F}\neg\psi$ ("φ generally holds"), and $\psi_1\mathbf{B}\psi_2 = \neg(\neg\psi_1\mathbf{U}\neg\psi_2)$ ("ψ_1 holds before ψ_2 fails").

A run ρ of $\mathcal{W}^*$ satisfies $\forall\bar{x}\bar{u}\varphi[\bar{u}](\bar{x})$ iff $\rho^0 \models \forall\bar{x}\bar{u}\varphi[\bar{u}](\bar{x})$. $\mathcal{W}^*$ satisfies $\forall\bar{x}\bar{u}\varphi[\bar{u}](\bar{x})$ iff every run of $\mathcal{W}^*$ satisfies $\forall\bar{x}\bar{u}\varphi[\bar{u}](\bar{x})$.

Example 7. The properties in Example 6 can be expressed as LTL-FO formulas over the schema of the bank loan Web application.

(1) $\forall a, o, cid, l, ssn, nm\ \mathbf{G}\ (p \rightarrow \mathbf{F}\ q)$
(2) $\forall a, o, o', r, r', cid, l, ssn, nm\ \mathbf{G}\ ((p \wedge \mathbf{F}\ s) \rightarrow o = o')$
(3) $\forall a, o, cid, l, ssn, nm, dec\ \mathbf{G}\ (p \rightarrow (t \vee u)\mathbf{B}\ v)$
(4) $\forall o, r, r', ssn, ctg\ \mathbf{G}\ (!getRating[o][r](ssn)\mathbf{B}\ \neg ?rating[r'][o](ssn, ctg))$
(5) $\forall o_1, o_2, a, cid, l\ \mathbf{G}((\mathbf{F}\ w_1 \wedge \mathbf{F}\ w_2) \rightarrow o_1 = o_2)$, where

$p = !apply[a][o](cid, l)\ \wedge customer(cid, nm, ssn)$,
$q = letter[o](cid, nm, l, \text{“}approved\text{”}) \vee letter[o](cid, nm, l, \text{“}denied\text{”})$,
$s = !getRating[o'][r](ssn) \vee !getHistory[o'][r'](ssn)$,
$t = \exists r ?rating[r][o](ssn, \text{“}excellent\text{”}) \vee ?rating[r][o](ssn, \text{“}poor\text{”})$
$u = \exists m\ ?decision[m][o](cid, nm, dec)$,
$v = \neg letter[o](cid, nm, l, dec), w_k = ?apply[a][o_k](cid, l)$ for $k = 1, 2$.

Properties (2), (4), (5) are not expressible in the composition model [5]. In the rest of the paper we study the following decision problem:

Definition 8. *Given a DDWA* $\mathcal{W}^*$ and an LTL-FO sentence $\forall \bar{x}\bar{u}\varphi[\bar{u}](\bar{x})$ over the schema of $\mathcal{W}^*$, the* verification problem *is the problem whether $\mathcal{W}^*$ satisfies $\forall \bar{x}\bar{u}\varphi[\bar{u}](\bar{x})$.*

This is an infinite-state model checking problem [14], because the underlying database and the number of users are not fixed apriori.

4 Undecidability Results

The *ASM*$^+$ transducer defined in [4] is a particular case of a simple *DDWA** with no input constants, where the single user does not log off, and there is no distinction between local and global state. The problem whether an *ASM*$^+$ transducer satisfies an LTL-FO sentence is undecidable [4]. As a corollary, we obtain the following result:

Corollary 1. *It is undecidable whether a DDWA* satisfies an LTL-FO sentence.*

Our *DDWA** model is also an extension of the composition model [5]. To obtain decidability, we have to impose at least the restrictions that ensure decidability for the models of [4,5]. One of these is a restriction called *input-boundedness.* It essentially reduces the range of the quantifiers that appear in the property and in state, send, and action rules to the active domain of the current inputs, previous inputs, and flat queue messages. It is a natural restriction based on the observation that Web applications are driven by user's input and incoming messages. We extend this notion to the multi-user model.

Definition 9. *Let $\mathcal{W} = \{\mathcal{W}_j\}_{1 \leq j \leq n}$ be a DDWA*. The set of* input-bounded FO formulas *over the schema of $\mathcal{W}^*$ is obtained by replacing in the definition of FO the quantification formation rule with the following:*

> *If φ is an input-bounded FO formula, α is a flat queue, current or previous input atom using a symbol from $\mathbf{Q}^f_{jl} \cup \mathbf{I}_j \cup \mathbf{Prev_{I_j}}$ for some $j, l \in [1:n]$, $j \neq l$, $\bar{x} \subseteq free(\alpha) \cap (\mathbf{V} \cup \mathbf{V}^{id})$, and $\bar{x} \cap free(\beta) = \emptyset$ for every state, nested queue, or action atom β in φ, then $\exists \bar{x}(\alpha \wedge \varphi)$ and $\forall \bar{x}(\alpha \rightarrow \varphi)$ are input-bounded FO formulas.*

An LTL-FO sentence over the schema of $\mathcal{W}^*$ is input-bounded iff all of its FO subformulas are input-bounded. $\mathcal{W}^*$ is input-bounded *iff*

- *all formulas in local and global state, action, and send rules into nested queues are input-bounded,*
- *all input rules and send rules into flat queues use $\exists^*$FO formulas in which all state and nested in-queue atoms are ground.*

Example 8. The bank loan application *DDWA** and the properties in Example 7 are input-bounded.

Input-boundedness is necessary to ensure decidability for each of the two models of [4,5]. The composition model of [5] requires also that all queues are bounded and the flat ones are *lossy*. Lossy queues may allow messages to be lost in transit. If every sent message is received, then the queue is *perfect*.

The following theorem shows that input-boundedness and bounded queues do not ensure decidability for the multi-user model.

Theorem 1. *It is undecidable whether a simple input-bounded DDWA* without queue symbols satisfies an input-bounded LTL-FO sentence.*

The proof is by reduction from the halting problem. In the following section we consider additional restrictions to render the verification problem decidable.

5 Decidable Versions of the Verification Problem

In practice, the number of simultaneous users of a Web application is bounded by the servers' capacity, although this may be a big number. This observation suggests to bound the number of active IDs of a *DDWA**. This implies a bound on the number of simultaneous users, but the opposite is not true because a user who repeatedly logs on and off may be assigned a different ID each time. It is often the case that users choose the same values for the input constants, each time they log on a Web site. If the number of users is finite, all these values can be stored in the database, and so they may be modeled as database constants rather than input constants.

It is therefore natural to restrict our attention to *DDWA** without input constants and with a bounded number of active IDs, in addition to the restrictions needed for the models of [4,5]. This ensures decidability of the verification problem.

Theorem 2. *It is decidable whether an input-bounded DDWA* with a bounded number of active IDs, no input constants, bounded queues with lossy flat channels and perfect nested channels, satisfies an input-bounded LTL-FO sentence. The problem is PSPACE complete for schemas with fixed bound on the arity, and EXPSPACE otherwise.*

The proof is by a polynomial time reduction to the verification of an input-bounded LTL-FO sentence by an input-bounded ASM^+ transducer, which is known to be PSPACE complete for schemas with fixed bound on the arity, and EXPSPACE otherwise [4].

Another way to obtain decidability is by allowing an unbounded number of IDs, but further restricting the property language.

Definition 10. A restricted input-bounded LTL-FO formula *is an input-bounded LTL-FO formula of the form $\varphi[\bar{u}](\bar{x})$, with all the ID variables free. A restricted input-bounded LTL-FO sentence is the universal closure of a restricted input-bounded LTL-FO formula.*

All properties in Example 7 are restricted input-bounded, except (3). However, the sentence $\forall a, o, o', r, m, cid, l, nm, ssn, ctg, dec$ $\mathbf{G}([p \wedge \mathbf{G}\neg ?rating[r][o](ssn,$ "*excellent*"$) \wedge \mathbf{G}\neg ?rating[r][o](ssn,$ "*poor*"$) \wedge \neg\mathbf{F}\ ?decision[m][o](cid, nm, dec)]$ $\rightarrow \neg\mathbf{F}\ letter[o'](cid, nm, l, dec))$ is restricted input-bounded, and equivalent to (3) assuming that: all $\mathcal{CR}$ instances provide the same rating for each SSN, all managers take the same decision for each application, and an application is no longer available after a response letter is sent.

Theorem 3. *It is decidable whether an input-bounded DDWA* with no input constants, bounded queues with lossy flat channels and perfect nested channels, a bounded number of active IDs for the components with non-empty queue schema, but an unbounded number of active IDs for the components with empty queue schema, satisfies a restricted input-bounded LTL-FO sentence. The problem is PSPACE complete for schemas with fixed bound on the arity, and EXPSPACE otherwise.*

The proof reduces the problem to the verification of a finite set of so called *pseudoruns*, by extending a method used in [4]. The proof is long, tedious, and it is difficult to provide an intuitive explanation.

6 Conclusions

In this paper we propose a model for interactive, multi-user data-driven and communicating Web applications. We use our model to specify a well known example of e-business application that has interesting behaviours that cannot be captured by the models of [4,5]. A variant of first-order temporal logic is defined to express properties of multi-user data-driven Web applications, some of which are not expressible in the composition model. We study the problem whether the multi-user model satisfies an LTL-FO property. Although this problem is undecidable in general, we identify two sets of restrictions that ensure decidability. In addition to restrictions that ensure decidability for the composition model of [5], we require either a bound on the number of all active IDs, or a more restricted property language while queue-less components may have an unbounded number of active IDs and only components with queues must have a bounded number of active IDs. The e-business application example in Section 2 satisfies these restrictions.

The decidability results for the multi-user model have the same complexity (PSPACE complete) as those obtained for the models of [4,5]. The verification problem considered here is an infinite-state model checking problem. The classic model checking problem for transition systems (finite-state machines or Kripke structures) and propositional LTL (see e.g. [14]) is PSPACE complete. Thus it is rather surprising to obtain the same complexity for models based on the relational ASM^+ transducer and LTL-FO.

The feasability of verification based on the ASM^+ transducer model is demonstrated in [6]. We expect to use the model proposed in this paper to specify and verify a richer class of Web applications. By extending previous models, our work demonstrates the strength of the original model of [4].

Acknowledgment. I thank the anonymous reviewers for useful comments.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
2. SWSF Committee (2005), http://www.w3.org/Submission/SWSF-SWSO/
3. Deutsch, A., Sui, L., Vianu, V.: Specification and verification of data-driven web services. In: PODS Proceedings, pp. 71–82 (2004)
4. Deutsch, A., Sui, L., Vianu, V.: Specification and verification of data-driven web applications. Journal of Computer and Systems Sciences 73(3), 442–474 (2007)
5. Deutsch, A., Sui, L., Vianu, V., Zhou, D.: Verification of communicating data-driven web services. In: PODS Proceedings, pp. 90–99 (2006)
6. Deutsch, A., Sui, L., Vianu, V., Zhou, D.: A system for specification and verification of interactive, data-driven web applications. In: SIGMOD Conference, pp. 772–774 (2006)
7. Fan, W., Geerts, F., Gelade, W., Neven, F., Poggi, A.: Complexity and composition of synthesized web services. In: PODS Proceedings, pp. 231–240 (2008)
8. http://pvs.csl.sri.com/
9. Hull, R., Benedikt, M., Christophides, V., Su, J.: E-services: A look behind the curtain. In: PODS Proceedings, pp. 1–14 (2003)
10. Hull, R., Su, J.: Tools for composite web services: a short overview. SIGMOD Record 34(2), 86–95 (2005)

11. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, Heidelberg (1992)
12. Narayanan, S., McIlraith, S.A.: Simulation, verification, and automated composition of web services. In: Proc. Int. World-Wide Web Conference (2002)
13. Papazoglou, M.P.: Web Services: Principles and Technology. Prentice-Hall, Englewood Cliffs (2008)
14. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Symp. on Logic in Computer Science (1986)

Automated Composition of Nondeterministic Stateful Services

Giuseppe De Giacomo and Fabio Patrizi

Dipartimento di Informatica e Sistemistica
SAPIENZA - Università di Roma
Via Ariosto 25 - 00185 Roma, Italy
{degiacomo,patrizi}@dis.uniroma1.it

Abstract. This paper addresses the automated composition of nondeterministic available services modeled as transition systems. Nondeterminism stems naturally when the results of client-service interactions cannot be foreseen, and calls for specific orchestration strategies able to deal with partial controllability. We show how to build a set of orchestrators, by resorting to a variant of the simulation relation's formal notion, by exploiting recent results on LTL formulas' synthesis and by reducing our technique to the search for a safety game winning strategy. The resulting technique is sound, complete and optimal w.r.t. computational complexity, and generates all possible solutions at once.

1 Introduction

Web services are modular applications that can be described, published, located, invoked and composed over a variety of networks (including the Internet): any piece of code and any application component deployed on a system can be wrapped and transformed into a network-available service, by using standard (XML-based) languages and protocols (e.g., WSDL, SOAP, etc.)- see e.g., [1]. The promise of Web services is to enable the composition of new distributed applications/solutions: when available services cannot satisfy a desired specification, they, or their parts, can be composed and orchestrated in order to realize the specification. Service composition involves two different phases [13]: the *composition synthesis*, where the specification of an orchestrator, which coordinates the available services to fulfill a target service specification, is synthesized, and the *composition deployment*, i.e., the actual implementation of the orchestrator specification in a given technology (such as BPEL) [1]. Here, we focus on the former.

Most of the research on composition synthesis, e.g., [27,8], has considered *atomic* services, essentially abstracting away from their dynamic behavior (a.k.a. *possible conversations*). Notable exceptions are, e.g., [17,7,21,5,14,20,10] where stateful services, and their dynamic behavior, are considered explicitly. A survey on composition synthesis approaches can be found in [13].

[1] http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

C. Laneve and J. Su (Eds.): WS-FM 2009, LNCS 6194, pp. 147–160, 2010.

In this paper, we follow the general approach first proposed in [5], called, in [13], "Roman Model", and recently further investigated in the context of agent behavior composition, in AI [9,25,24]. We address the automatic composition of nondeterministic, partially controllable, available services, modeled as transition systems that capture the possible conversations that services can have with clients. When the result of interactions cannot be foreseen, nondeterminism naturally stems. For instance, think of a service for buying tickets: the service cannot know in advance whether seats are available for a selected performance. In other words, service behavior is partially controllable: a property an orchestrator needs to cope with. We assume that orchestrators can observe available services' states, and hence take advantage of this in choosing how to continue a certain task. This assumption is quite natural in this context [2], as transition systems represent available services' "public" behavior.

Our composition technique is based on the formal notion of simulation relation [18]. It follows the lines drawn in [6,24], in the presence of nondeterminism, which calls for a specific simulation relation's variant, that considers available services' partial controllability. The variant presented here can be proven equivalent to the one in [24].

Our main contribution is relating the service composition problem to the literature on synthesis of reactive systems (cf., e.g., [22]). In particular, we show that the problem can be solved by exploiting safety-games and propose an implementation based on the system TLV. This is a major step toward practical implementation of engines for orchestrator synthesis. Notably, the proposed technique not only is sound, complete, and optimal w.r.t. computational complexity, but also, in a precise formal sense (see later), produces all (infinite) possible solutions at once.

In Section 2, we introduce the formal setting; in Section 3, we develop our technique; in Section 4, we show how the technique can exploit safety games; in Section 5, we propose an implementation based on the system TLV and, finally, in Section 6 we draw some conclusions.

2 The Framework

The framework adopted here is based on [5,9,24], and is sometime referred to as the "Roman Model" [13].

Data box. We assume to have an accessible shared system, called *data box*, which allows client services to store and retrieve shared data. We describe it as a nondeterministic transition system, where *(i)* states represent an abstract finite description of the data content, and *(ii)* transitions represent the execution of operation, including data insertion/deletion or retrieval. Nondeterministic transitions model those operations whose outcome is a-priori unknown.

A data box is a tuple $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$ where:

- $\mathcal{O}$ is the finite set of shared operations, i.e., the *whole* set of operations clients can perform, each of which may or may not affect data box' state;

[2] The reader should observe that also the standard proposal WSDL 2.0 has a similar point of view: the same operation can have multiple output messages (the `out message` and various `outfault messages`), and the client observes how the service behaved only after receiving a specific output message.

- D is the finite set of data box' states;
- $d_0 \in D$ is the initial state;
- $\rho \subseteq D \times \mathcal{O} \times D$ is the transition relation among states: $\langle d, o, d' \rangle \in \rho$, or $d \xrightarrow{o} d'$ in $\mathcal{DB}$, denotes that execution of operation o in state d may lead the data box to a successor state d'.

Available Services. An available service, at each step, offers to its clients a choice of operations, based upon its own and data box' state; the client chooses one of them, and the service executes it, resulting in a new state of the service and a new state of the data box. The available service can take into account data box' influence on available services by putting *guards* on transitions –i.e., conditions on current state of the databox, which restricts the set of transitions that can actually take place.

Formally, an *available service* over a data box $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$ is a tuple $\mathcal{S} = \langle \mathcal{O}, S, s_0, S^f, G, \varrho \rangle$, where:

- $\mathcal{O}$ is the same set of operations as in $\mathcal{O}$;
- S is the finite set of service's states;
- $s_0 \in S$ is the initial state;
- $S^f \subseteq S$ is the set of *final* states, i.e., those where the execution can be legally stopped (if desired);
- G is a set of boolean functions $g : D \to \{true, false\}$ called *guards*;
- $\varrho \subseteq S \times G \times \mathcal{O} \times S$ is the service's transition relation.

When $\langle s, g, o, s' \rangle \in \varrho$, we say that *transition* $s \xrightarrow{g,o} s'$ *is in* $\mathcal{S}$. Given a state $s \in S$, if there exists a transition $s \xrightarrow{g,o} s'$ in $\mathcal{S}$ (for some g and s') and the data box is in a state d such that $g(d) = true$ then operation o is said to be *executable* in s. A transition $s \xrightarrow{g,o} s'$ in $\mathcal{S}$ denotes that s' is a possible successor state of s, when operation o is executed in s, provided $g(d) = true$, d being current data box state.

Available services are, in general, *nondeterministic*, that is, they allow many transitions to take place under execution of a same operation. So, when choosing the operation to execute next, the client of the service cannot be certain of which choices will be available later on, this depending on which transition actually takes place. In other words, nondeterministic behaviors are only *partially controllable*.

We say that a service $\mathcal{S}$ over a data box $\mathcal{DB}$ is *deterministic* iff there is no $\mathcal{DB}$ state $d \in D$ for which there exist, in $\mathcal{S}$, two distinct transitions $s \xrightarrow{g_1,o} s'$ and $s \xrightarrow{g_2,o} s''$ such that $s' \neq s''$ and $g_1(d) = g_2(d) = true$. Notice that given a deterministic service's state and a legal operation in that state, the *unique* next service state is always known. That is, deterministic services are indeed *fully controllable* by selecting operations.

Community and Target Service. A *community* $\mathcal{C}=\{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\}$ is a set containing *(i)* a data box $\mathcal{DB}$ and *(ii)* n nondeterministic available services over $\mathcal{DB}$. In our framework, we also define the so-called *target service*, which is the *deterministic* service one aims at building by properly *composing* available services. A target service has the same form as any other service defined over $\mathcal{DB}$, with the only requirement of being deterministic.

Trace and History. Given $\mathcal{S} = \langle \mathcal{O}, S, s_0, S^f, G, \varrho \rangle$ over data box $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$, a *trace* for $\mathcal{S}$ on $\mathcal{DB}$ is a possibly infinite sequence, alternating configurations and operations, of the form $\langle s^0, d^0 \rangle \xrightarrow{o^1} \langle s^1, d^1 \rangle \xrightarrow{o^2} \cdots$, such that *(i)* $\langle s^0, d^0 \rangle = \langle s_0, d_0 \rangle$, and *(ii)* for all $j > 0$, if $\langle s^j, d^j\rangle \xrightarrow{o^{j+1}} \langle s^{j+1}, d^{j+1}\rangle$, then $s^j \xrightarrow{g,o^{j+1}} s^{j+1}$ in $\mathcal{S}$ with $g(d^j) = \mathit{true}$ for some g, and $d^j \xrightarrow{o^{j+1}} d^{j+1}$ in $\mathcal{DB}$.

Similarly, let $\mathcal{C}$=$\{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\}$ be a community, where $\mathcal{S}_i$=$\langle \mathcal{O}, S_i, s_{i0}, S_i^f, G_i, \varrho_i\rangle$ $(i = 1, \ldots, n)$ and $\mathcal{DB}$ as above. A *community trace* for $\mathcal{C}$ is a possibly infinite sequence of the form $\langle s_1^0, \ldots, s_n^0, d^0\rangle \xrightarrow{o^1,k^1} \langle s_1^1, \ldots, s_n^1, d^1\rangle \xrightarrow{o^2,k^2} \cdots$, such that *(i)* $\langle s_1^0, \ldots, s_n^0, d^0\rangle = \langle s_{10}, \ldots, s_{n0}, d_0\rangle$, and *(ii)* for all $j > 0$, if $\langle s_1^j, \ldots, s_n^j, d^j\rangle \xrightarrow{o^{j+1},k^{j+1}} \langle s_1^{j+1}, \ldots, s_n^{j+1}, d^{j+1}\rangle$, then $s_{k^{j+1}}^j \xrightarrow{g,o^{j+1}} s_{k^{j+1}}^{j+1}$ in $\mathcal{S}_{k^{j+1}}$ with $g(d^j) = \mathit{true}$ for some g, $s_i^{j+1} = s_i^j$ for $i \neq k^{j+1}$, and $d^j \xrightarrow{o^{j+1}} d^{j+1}$ in $\mathcal{DB}$.

We call (community) *history* every finite prefix of a (community) trace ending with a configuration. Given a history h, we denote by $\mathit{last}(h)$ the last configuration, and by $\mathit{length}(h)$ the number of alternations between configurations and operations in h. Notice that the history of length 0 is simply the initial configuration of a trace (which is the same for every trace).

Orchestrator. The *orchestrator* is a component able to activate, stop and resume each of the available services, and select one to perform an executable operation. The orchestrator has *full observability* on available service states, that is, it can keep track (at runtime) of the current state of each available service. Let $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\}$ be a community and $\mathcal{H}$ be the set of its community service histories. An *orchestrator* for a community $\mathcal{C}$ is a function $P : \mathcal{H} \times \mathcal{O} \to \{1, \ldots, n, u\}$ that, given a history $h \in \mathcal{H}$ and an operation $o \in \mathcal{O}$, selects an available service, i.e., returns its index, to which delegate o. Special value u is introduced for technical convenience, to make function P total.

Definition 1. *Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\rangle$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$, where, $\mathcal{S}_i = \langle S_i, s_{i0}, S_i^f, G_i, \varrho_i\rangle$ $(i = t, 1 \ldots, n)$ and $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho\rangle$. Let $P : \mathcal{H} \times \mathcal{O} \to \{1, \ldots, n, u\}$ be an orchestrator for $\mathcal{C}$. Given a trace $\tau = \langle s^0, d^0\rangle \xrightarrow{o^1} \langle s^1, d^1\rangle \xrightarrow{o^2} \cdots$ of $\mathcal{S}_t$ on $\mathcal{DB}$, we say that the orchestrator P realizes the trace τ if and only if:*

- *for all community service histories $h \in \mathcal{H}_\tau$, $P(h, o^{\mathit{length}(h)+1}) \neq u$ and $\mathcal{H}_\tau^\ell \neq \emptyset$ (see below), where $\mathcal{H}_\tau = \bigcup_\ell \mathcal{H}_\tau^\ell$ is a set of community service histories, inductively defined as follows:*
  - *$\mathcal{H}_\tau^0 = \{\langle s_{10}, \ldots, s_{n0}, d_0\rangle\}$;*
  - *$\mathcal{H}_\tau^{j+1}$ is the set of community histories of length $j + 1$ having the form $h' = h \xrightarrow{o^{j+1},k^{j+1}} \langle s_1^{j+1}, \ldots, s_n^{j+1}, d^{j+1}\rangle$ such that:*
    - *$h \in \mathcal{H}_\tau^j$, with $\mathit{last}(h) = \langle s_1^j, \ldots, s_n^j, d^j\rangle$;*
    - *o^{j+1} and d^{j+1} are the operation and the data box state in history of length $j + 1$ obtained from τ.*
    - *$P(h, o^{j+1})$=k, that is, the orchestrator states that operation o^{j+1} in the trace τ after community history h should be executed by available service $\mathcal{S}_k$;*
    - *$s_k^j \xrightarrow{g,o^{j+1}} s_k^{j+1}$ in $\mathcal{S}_k$ with $g(d^j) = \mathit{true}$ for some g, that is, the available service $\mathcal{S}_k$ can evolve according to the history h'.*
    - *$s_i^{j+1} = s_i^j$ for each $i \neq k$*

– *if a configuration* $\langle s_t^\ell, d^\ell \rangle$ *of* τ *is such that* $s_t^\ell \in \mathcal{S}_t^f$*, then every configuration* $\langle s_1^\ell, \ldots, s_n^\ell, d^\ell \rangle = last(h)$*, with* $h \in \mathcal{H}_\tau^\ell$*, is such that* $s_i^\ell \in \mathcal{S}_i^f$*, for* $i = 1, \ldots, n$*.*

Definition 2. *An orchestrator P for* $\mathcal{C}$ *is a* <u>*composition of the target service*</u> $\mathcal{S}_t$ *on data box* $\mathcal{DB}$ *iff it* realizes all traces *of* $\mathcal{S}_t$ *on* $\mathcal{DB}$*.*

Intuitively, the orchestrator realizes a target service if for all target service traces over the data box, at every step, it returns the index of an available service that can actually perform the requested operation. Observe that since available services and data box are nondeterministic, the orchestrator must be always able to execute the next operation, no matter how the activated service and the data box happen to evolve after each step. Finally, note that the orchestrator can observe available services' and data box' states (in fact, the whole community service history so far), in order to decide which available service to select next. This makes orchestrators akin to an advanced form of conditional plans [11].

The Composition Problem. This work addresses the following problem: *given a community* $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\}$ *and a deterministic* target *service* $\mathcal{S}_t$ *over* $\mathcal{DB}$*, synthesize an orchestrator for* $\mathcal{C}$ *which is a composition of* $\mathcal{S}_t$ *on data box* $\mathcal{DB}$*.*

3 Composition via Simulation

Following [6,24], we present a composition technique based on the formal notion of *simulation* [18,12]. Since the devilish nondeterminism of both data box and available services prevents the possibility to use the off-the shelf notion of simulation, a more general variant is needed, called *ND-simulation.*

Definition 3. *Let* $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB} \rangle$ *be a community and* $\mathcal{S}_t$ *a target service over* $\mathcal{DB}$*, where,* $\mathcal{S}_i = \langle S_i, s_{i0}, S_i^f, G_i, \varrho_i \rangle$ $(i = t, 1 \ldots, n)$ *and* $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$*. An* <u>*ND-simulation relation*</u> *of* $\mathcal{S}_t$ *by* $\mathcal{C}$ *is a relation* $R \subseteq S_t \times S_1 \times \ldots \times S_n \times D$ *such that* $\langle s_t, s_1, \ldots, s_n, d \rangle \in R$ *implies:*

1. *if* $s_t \in S_t^f$ *then* $s_i \in S_i^f$*, for* $i = 1, \ldots, n$*;*
2. *for each* $o \in \mathcal{O}$*, there exists a* $k \in \{1, \ldots, n\}$ *such that for all* $\langle s_t, d \rangle \xrightarrow{o} \langle s_t', d' \rangle$ *such that* $s_t \xrightarrow{g,o} s_t'$ *in* $\mathcal{S}_t$ *with* $g(d) = true$ *and* $d \xrightarrow{o} d'$ *in* $\mathcal{DB}$ *, then both the followings hold:*
   (a) there exists a transition $s_k \xrightarrow{g,o} s_k'$ *in* $\mathcal{S}_k$ *with* $g(d) = true$*;*
   (b) for all $s_k \xrightarrow{g,o} s_k'$ *in* S_k *with* $g(d) = true$ *we have that* $\langle s_t', s_1, \ldots, s_k', \ldots, s_n, d' \rangle \in R$*.*

An ND-simulation is essentially a simulation between $\mathcal{S}_t$ and the asynchronous product of the services $\mathcal{S}_i$ in $\mathcal{C}$. With respect to the usual notion of simulation relation, we need to deal with data box $\mathcal{DB}$ in $\mathcal{C}$ that acts as a parameter, and, more importantly, we need to take into account available services' nondeterminism. To this end, we require that *(i)* for each target service's transition an available service k can be selected to perform $\mathcal{S}_t$ labeling operation and *(ii) all possible successor states* (under selected service and current operation) are still included in the ND-simulation relation.

A state s_t *is ND-simulated by* $\langle s_1, \ldots, s_n, d\rangle$ (or $\langle s_1, \ldots, s_n, d\rangle$ *ND-simulates* s_t), denoted $s_t \preceq \langle s_1, \ldots, s_n, d\rangle$, iff there exists an ND-simulation R of $\mathcal{S}_t$ by $\mathcal{C}$ such that $\langle s_t, s_1, \ldots, s_n, d\rangle \in R$. Observe that this is a coinductive definition. As a result, the relation $\preceq$ is itself an ND-simulation, and is in fact the *largest ND-simulation relation*.

Next result shows that checking for the existence of a composition can be reduced to checking whether there exists an ND-simulation relation between the target service and the community, containing their respective initial states.

Theorem 1. *Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\rangle$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$ as above. An orchestrator P for $\mathcal{C}$ that is a composition of target service S_t over $\mathcal{DB}$ exists if and only if $s_{t0} \preceq \langle s_{10}, \ldots, s_{n0}, d_0\rangle$.*

Theorem 1 provides a straightforward method to check for the existence of a composition, namely:

1. compute the largest ND-simulation relation $\preceq$;
2. check whether $\langle s_{t0}, s_{10}, \ldots, s_{n0}, d_0\rangle \in \preceq$.

From the computational point of view, the largest ND-simulation relation $\preceq$ between $\mathcal{S}_t$ and $\mathcal{C}$ can be computed in polynomial time wrt the size of $\mathcal{S}_t$ and $\mathcal{C}$. Since the number of states in $\mathcal{C}$ is exponential in the number of available services n, $\preceq$ can be computed in exponential time. More precisely, it is polynomial wrt the size of $\mathcal{S}_t$, $\mathcal{DB}$ and each service $\mathcal{S}_i$, but exponential in the number of available services n. Thus, observing that the problem is EXPTIME-hard [19], we get that this technique is optimal wrt worst-case complexity.

Once we have computed the ND-simulation, *synthesizing* an orchestrator becomes an easy task. As a matter of fact, there is a well-defined procedure that, given an ND-simulation, builds a finite state program that returns, at each point, the set of available behaviors capable of performing a target-conformant operation. We call such a program *orchestrator generator*, or simply *PG*. Formally:

Definition 4. *Let $\mathcal{C}$=$\langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\rangle$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$ as above. The orchestrator generator (PG) of $\mathcal{C}$ for $\mathcal{S}_t$ is a tuple PG=$\langle \mathcal{O}, \{1, \ldots, n\}, \Sigma, \partial, \omega\rangle$, where:*

1. *$\mathcal{O}$ is the finite set of operations;*
2. *$\{1, \ldots, n\}$ is the set of available behavior indexes;*
3. *$\Sigma = \{\langle s_t, s_1, \ldots, s_n, d\rangle \mid s_t \preceq \langle s_1, \ldots, s_n, d\rangle\}$ is the set of states of PG, formed by the tuples belonging to the largest ND-simulation relation;*
4. *$\partial \subseteq \Sigma \times \mathcal{O} \times \{1, \ldots, n\} \times \Sigma$ is the* transition relation, *where $\langle \sigma, o, k, \sigma'\rangle \in \partial$, or $\sigma \xrightarrow{o,k} \sigma'$* is in *PG, if and only if all of the followings hold:*
   - *$\sigma = \langle s_t, s_1, \ldots, s_k, \ldots, s_n, d\rangle$ and $\sigma' = \langle s'_t, s_1, \ldots, s'_k, \ldots, s_n, d'\rangle$*
   - *$s_t \xrightarrow{g,o} s'_t$ in $\mathcal{S}_t$ with $g(d) = true$;*
   - *there exists a transition $s_k \xrightarrow{g,o} s'_k$ in $\mathcal{S}_k$ with $g(d) = true$;*
   - *for all transitions $s_k \xrightarrow{g,o} s''_k$ in $\mathcal{S}_k$ with $g(d)$=true we have $\langle s'_t, s_1, \ldots, s''_k, \ldots, s_n, d'\rangle \in \Sigma$;*
5. *$\omega : \Sigma \times \mathcal{O} \mapsto 2^{\{1,\ldots,n\}}$ is the* output function, *where:*
   - *$\omega(\sigma, o) = \{k \mid \exists\, \sigma'$ s.t. $\sigma \xrightarrow{o,k} \sigma'$ in PG$\}$.*

Intuitively, *PG* is a finite state transducer that, given an operation o (compliant with the target service), outputs, through ω, the set of *all* available services able to perform o next, according to the largest ND-simulation $\preceq$. Observe that computing *PG* from the relation $\preceq$ is easy, since it involves checking for *local* conditions only.

If there exists a composition of $\mathcal{S}_t$ by $\mathcal{C}$, then $s_{t0} \preceq \langle s_{10}, \ldots, s_{n0}, d_0 \rangle$ and *PG* does include state $\sigma_0 = \langle s_{t0}, s_{10}, \ldots, s_{n0}, d_0 \rangle$. In such case, all the actual orchestrators that are compositions of $\mathcal{S}_t$ by $\mathcal{C}$ can be obtained by just picking up, at each step, one among the services returned by ω. Being, in fact, generated from a given structure (i.e., *PG*), they are called *generated orchestrators*. Prior to provide their formal definition, some preliminary notions are needed.

A *trace for PG* starting from σ^0 is a finite or infinite sequence of the form $\sigma^0 \xrightarrow{o^1,k^1} \sigma^1 \xrightarrow{o^2,k^2} \cdots$, such that $\sigma_j \xrightarrow{o^{j+1},k^{j+1}} \sigma_{j+1}$ is in *PG*, for all j. A *history for PG* starting from state σ^0 is a prefix of a trace starting from state σ^0. By using histories, one can introduce *PG-orchestrators*, which are functions $PGP_{\text{CHOOSE}} : \mathcal{H}_{PG} \times \mathcal{O} \rightarrow \{1, \ldots, n, u\}$ where $\mathcal{H}_{PG}$ is the set of *PG* histories starting from any state in Σ and defined as follows: $PGP_{\text{CHOOSE}}(h_{PG}, o) = \text{CHOOSE}(\omega(last(h_{PG}), o))$, for all $h_{PG} \in \mathcal{H}_{PG}$, where CHOOSE stands for a choice function that chooses one element among those returned by $\omega(last(h_{PG}), o))$.

We can now relate a *PG* to compositions, through the following characterizing theorem.

Theorem 2. *If PG includes the state $\sigma_0 = \langle s_{10}, \ldots, s_{n0}, d_0 \rangle$ then every orchestrator generated by PG is a composition of the target service $\mathcal{S}_t$ by the community $\mathcal{C}$. Moreover, every orchestrator that is a composition of the target service $\mathcal{S}_t$ by the community $\mathcal{C}$ can be generated by PG (which, indeed, includes σ_0).*

Notably, while each specific composition may be an infinite state program, *PG*, which includes them all, is always finite. We conclude the section with an interesting observation. Let us consider the generated orchestrator PGP_{jit}, with CHOOSE resolved at run-time. PGP_{jit} (and *PG* for the matter) can be computed *on-the-fly* by storing only the ND-simulation $\preceq$. Indeed, at each point, the only information we need for the next choice is $\omega(\sigma, o)$ where $\sigma \in \Sigma = \preceq$. Now, in order to compute $\omega(\sigma, o)$ we only need to know $\preceq$.

4 Simulation and Safety Games

In this Section, we show how a service composition problem instance can be encoded into a game structure and how searching for a composition is equivalent to searching for a winning strategy for the corresponding game (cf. [3,4,22]). The main motivation behind this approach is the increasing availability of software systems, such as TLV [23], Lily [15], Anzu [16] or MOCHA [2], which provide *(i)* efficient procedures for strategy computation and *(ii)* convenient languages for representing the problem instance in a modular, intuitive and straightforward way.

4.1 Safety-Game Structures

We specialize the *game structures* proposed in [22] to deal with synthesis problems for invariant properties. Throughout the rest of the paper, we assume to deal with infinite-run TSs, possibly obtained by introducing fake loops, as customary in LTL verification/synthesis.

Starting from [22], we define a *safety-game structure* (or $\Box$-game structure or $\Box$-GS, for short) as a tuple $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$, where:

- $\mathcal{V} = \{v_1, \ldots, v_n\}$ is a finite set of *state* variables, ranging over *finite* domains $V_1, \ldots, V_n$, respectively. $V = V_1, \ldots, V_n$ represents the set of all possible valuations of variables in $\mathcal{V}$. We assume that $\mathcal{V} = \{\mathcal{X}, \mathcal{Y}\}$, i.e., $\mathcal{V}$ is partitioned into sets $\mathcal{X}$ and $\mathcal{Y}$, the former referred to *as set of environment variables* and the latter as *set of system variables*. Let X (resp. Y) be the set of all possible valuations for variables in $\mathcal{X}$ ($\mathcal{Y}$). Then, $\boldsymbol{x} \in X$ ($\boldsymbol{y} \in Y$) is called *environment state* (*system state*). A *game state* $\boldsymbol{s} \in V$ is a complete assignment of values to variables. Without loss of generality, we assume that $\boldsymbol{s} = \langle \boldsymbol{x}, \boldsymbol{y} \rangle \in X \times Y$.
- Θ is a formula representing the initial states of the game. It is a boolean combination of expressions $(v_k = \bar{v}_k)$, where $v_k \in \mathcal{V}$ and $\bar{v}_k \in V_k$ ($k \in \{1, \ldots, n\}$) (partial assignments are allowed). For such formulae, given a state $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in V$, we write $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \models \Theta$ if state s satisfies the assignments specified by Θ.
- $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ is the *environment transition relation* which relates a current (unprimed) game state to a possible next (primed) environment state.
- $\rho_s(\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}')$ is the *system transition relation*, which relates a game state plus a next environment state to a next system state.
- $\Box\varphi$ is a formula representing the invariant property to be guaranteed, where φ has the same form as Θ.

We assume variables in $\mathcal{X}$ (respectively $\mathcal{Y}$) are ordered, so that valuations in X (Y) can be conveniently represented as tuples $\boldsymbol{x} = \langle x_1, \ldots, x_n \rangle$ ($\boldsymbol{y} = \langle y_1, \ldots, y_m \rangle$). In unary tuples, we omit angle brackets when no ambiguity arises.

A game state $\langle \boldsymbol{x}', \boldsymbol{y}' \rangle$ is a *successor* of $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ iff $\rho_e(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}')$ and $\rho_s(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}', \boldsymbol{y}')$. A *play* of G is a maximal sequence of states $\eta : \langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle \langle \boldsymbol{x}_1, \boldsymbol{y}_1 \rangle \cdots$ satisfying *(i)* $\langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle \models \Theta$, and *(ii)* for each $j \geq 0$, $\langle \boldsymbol{x}_{j+1}, \boldsymbol{y}_{j+1} \rangle$ is a successor of $\langle \boldsymbol{x}_j, \boldsymbol{y}_j \rangle$. Given a $\Box$-GS G, in a given state $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ of a game play, the environment chooses an assignment $\boldsymbol{x}' \in X$ such that $\rho_e(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}')$ holds and the system chooses assignment $\boldsymbol{y}' \in Y$ such that $\rho_s(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}', \boldsymbol{y}')$ holds.

A play is said to be *winning for the system* if it is infinite and satisfies the winning condition $\Box\varphi$. Otherwise, it is *winning for the environment*. A *strategy* for the system is a partial function $f : (X \times Y)^+ \times X \to Y$ such that for every $\lambda : \langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle \cdots \langle \boldsymbol{x}_n, \boldsymbol{y}_n \rangle$ and for every $\boldsymbol{x}' \in X$ such that $\rho_e(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{x}')$, $\rho_s(\boldsymbol{x}_n, \boldsymbol{y}_n, \boldsymbol{x}', f(\lambda, \boldsymbol{x}'))$ holds. A play $\eta : \langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle \langle \boldsymbol{x}_1, \boldsymbol{y}_1 \rangle \cdots$ is said to be *compliant* with a strategy f iff for all $i \geq 0$, $f(\langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle \cdots \langle \boldsymbol{x}_i, \boldsymbol{y}_i \rangle, \boldsymbol{x}_{i+1}) = \boldsymbol{y}_{i+1}$. A strategy f is winning for the system from a given state $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ iff all plays starting from $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ and compliant with f are so. When such a strategy exists, $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ is said to be a *winning state* for the system. A $\Box$-GS is said to be *winning for the system* if all initial states are so. Otherwise, it is said to be *winning for the environment*.

Our objective is to encode a composition problem into a □-GS and, then, exploit tools available for the latter to compute the orchestrator generator PG (cf. Section 3). Essentially, as it will be clear soon, one can extract the maximal ND-simulation relation –and, from this, directly compute the PG–, from the maximal set of states that are *winning* for the system. Let us show how such *winning set* can be computed in general on a □-GS. The core of the algorithm is the following operator (cf. [3,22]):

Definition 5. *Let $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$ be a □-GS as above. Given a set $P \subseteq V$ of game states $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$, the set of P's* controllable predecessors *is*

$$\begin{aligned}\pi(P) \doteq \{\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in V \mid \forall\, \boldsymbol{x}'.\rho_e(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}') \rightarrow \\ \exists\, \boldsymbol{y}'.\rho_s(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}', \boldsymbol{y}') \wedge \langle \boldsymbol{x}', \boldsymbol{y}' \rangle \in P\}\end{aligned}$$

Intuitively, $\pi(P)$ is the set of states from which the system can force the play to reach a state in P, no matter how the environment evolves. Based on this, Algorithm 1 computes the set of all system's winning states of a □-GS $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$, as Theorem 3 shows.

Algorithm 1. WIN – Computes system's maximal set of winning states in a □-GS

```
$W := \{\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in V \mid \langle \boldsymbol{x}, \boldsymbol{y} \rangle \models \varphi\}$
repeat
    $W' := W$;
    $W := W \cap \pi(W)$;
until $(W' = W)$
return $W$
```

Theorem 3. *Let $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi \rangle$ be a □-GS as above and W be obtained as in Algorithm 1. Given a state $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in V$, a system's winning strategy f starting from $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ exists iff $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in W$.*

In fact, one can define a system's winning strategy $f(\langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle, \ldots, \langle \boldsymbol{x}_i, \boldsymbol{y}_i \rangle, \boldsymbol{x}) = \boldsymbol{y}$, by picking up, for each $\boldsymbol{x}$ such that $\rho_e(\boldsymbol{x}_i, \boldsymbol{y}_i, \boldsymbol{x})$ holds, any $\langle \boldsymbol{x}, \boldsymbol{y} \rangle \in W$.

4.2 From Composition to Safety Games

In order to encode the composition problem as a □-GS, we need first to individuate which place each abstract component, e.g., target, available services, data box, occupies in the game representation. Conceptually, our goal is to refine an automaton capable of selecting, at each step, one among all the available services, in a way such that the community is always able to satisfy target service requests. So, the orchestrator, i.e., the object of the synthesis, plays as system and, consequently, the other entities, properly combined, form the environment. In addition, according to our purposes, the winning condition requires to satisfy two properties: *(i)* if the target service is in a final state, all community services are in a final state as well; *(ii)* the service selected by the orchestrator is able to perform the action currently requested by the target service.

Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB} \rangle$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$, where, $\mathcal{S}_i = \langle S_i, s_{i0}, S_i^f, G_i, \varrho_i \rangle$ $(i = t, 1 \ldots, n)$ and $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho \rangle$. We derive a □-GS $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_c, \Box\varphi \rangle$, as follows:

- $\mathcal{V} = \{s_t, s_1, \ldots, s_n, d, o, ind\}$, where:
  - s_i ranges over $S_i \cup \{init\}$ ($i = t, 1, \ldots, n$);
  - d ranges over $D \cup \{init\}$;
  - o ranges over $\mathcal{O} \cup \{init\}$;
  - ind ranges over $\{1, \ldots, n\} \cup \{init\}$;

  with an intuitive semantics: each complete valuation of $\mathcal{V}$ represents *(i)* the current state of community (variables $s_1, \ldots, s_n,$), data box (d) and target service (s_t), *(ii)* the operation to be performed next (o) and *(iii)* the available service selected to perform it (ind). Special value $init$ has been introduced for convenience, so as to have fixed initial state;
- $\mathcal{X} = \{s_t, s_1, \ldots, s_n, d, o\}$ is the set of environment variables;
- $\mathcal{Y} = \{ind\}$ is the (singleton) set of system variables;
- $\Theta = (\bigwedge_{i=t,0,\ldots,n}(s_i = init)) \wedge (d = init) \wedge (o = init) \wedge (ind = init)$;
- $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ is defined as follows:
  - $\langle\langle init, \ldots, init\rangle, init, \langle s_t, s_1, \ldots, s_n, d, o\rangle\rangle \in \rho_e$ iff $s_i = s_{i0}$, for $i = t, 1, \ldots,$ n, $d{=}d_0$, and there exists a transition $\langle s_{t0}, g, o, s_t'\rangle \in \varrho_t$ such that $g(d_0){=}true$;
  - if $s_i \neq init$, with $i = t, 1, \ldots, n,$, $d \neq init$, $o \neq init$ and $ind \neq init$ then $\langle\langle s_t, s_1, \ldots, s_n, d, o\rangle, ind, \langle s_t', s_1', \ldots, s_n', d', o'\rangle\rangle \in \rho_e$ iff the followings hold in conjunction:
    1. there exists a transition $s_t \xrightarrow{g,o} s_t'$ in ϱ_t with $g(d) = true$;
    2. either there exists a transition $s_{ind} \xrightarrow{g,o} s_{ind}'$ in ϱ_{ind} with $g(d) = true$ or $s_{ind}' = s_{ind}$ (service wrongly makes no move, and the error violates the safety condition φ, see below);
    3. $s_i = s_i'$, for all $i = 1, \ldots, n$ such that $i \neq ind$;
    4. there exists a transition $d \xrightarrow{o} d'$ in $\mathcal{DB}$;
    5. there exists a transition $s_t' \xrightarrow{g',o'} s_t''$ in ϱ_t for some s_t'', with $g'(d') = true$;
- $\langle\langle s_t, s_1, \ldots, s_n, d, o\rangle, ind, \langle s_t', s_1', \ldots, s_n', d', o'\rangle, ind'\rangle \in \rho_s$ iff $ind' \in \{1, \ldots, n\}$;
- Formula φ is defined depending on current state, operation and service selection as

$$\varphi \doteq \Theta \vee (\bigwedge_{i=1}^{n} \neg fail_i) \wedge (final_t \rightarrow \bigwedge_{i=1}^{n} final_i),$$

  where:
  - $fail_i \doteq (ind = i) \wedge (\bigwedge_{\langle s,g,op,s'\rangle \in \varrho_i}(g(d) = false \vee s_i \neq s \vee op \neq o))$, encodes the fact that service i has been selected but, in its current state, no transition can take place which executes the requested operation;
  - $final_i \doteq \bigvee_{s \in S_i^f}(s_i = s)$ encodes the fact that service $i = t, 1, \ldots, n$ is currently in one of its final states.

We can now show how the so-obtained game structure allows for computing an orchestrator generator. Recall that, in order to define the PG, one needs to build an ND-simulation (see Definition 4). The following Theorem shows that this can be equivalently done by computing the maximal system's set of winning states for G.

Theorem 4. *Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\rangle$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$ where $\mathcal{S}_i = \langle \mathcal{O}, S_i, s_{i0}, S_i^f, G_i, \varrho_i\rangle$ ($i = t, 1 \ldots, n$) and $\mathcal{DB} = \langle \mathcal{O}, D, d_0, \rho\rangle$. From $\mathcal{C}$ and $\mathcal{S}_t$ derive: a $\Box$-GS $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box\varphi\rangle$ as shown above. Let $W \subseteq V$ be the maximal set of system's winning states for G. Then $\langle init, \ldots, init\rangle \in W$ if and only if $s_{t0} \preceq \langle s_{10}, \ldots, s_{n0}, d_0\rangle$.*

Based on this, the following Theorem gives us an actual procedure to build up an orchestrator generator and, hence, all possible compositions.

Theorem 5. *Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\rangle$, $\mathcal{S}_t$ and $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_c, \Box\varphi\rangle$ be as above (hypothesis of Theorem 4). Let W be the system's winning set for G with $\langle init, \ldots, init\rangle \in W$. Then the orchestrator generator $PG = \langle \mathcal{O}, \{1, \ldots, n\}, \Sigma, \partial, \omega\rangle$ of $\mathcal{C}$ for $\mathcal{S}_t$ can be built from W, as follows:*

- *$\mathcal{O}$ is the usual set of operations and $\{1, \ldots, n\}$ the set of available services' indexes;*
- *$\Sigma \subseteq S_t \times S_1 \times \ldots \times S_n \times D$ is such that $\langle s_t, s_1, \ldots, s_n, d\rangle \in \Sigma$ if and only if there exists a game state $\langle s_t, s_1, \ldots, s_n, d, o, ind\rangle \in W$, for some $o \in \mathcal{O}$ and $ind \in \{1, \ldots, n\}$;*
- *$\partial \subseteq (\Sigma \times \mathcal{O} \times \{1, \ldots, n\} \times \Sigma)$ is such that $\langle\langle s_t, s_1, \ldots, s_n, d\rangle, o, k, \langle s'_t, s'_1, \ldots, s'_n, d'\rangle\rangle \in \partial$ if and only if $\langle s_t, s_1, \ldots, s_n, d, o, k\rangle \in W$ and there exist $o' \in \mathcal{O}$ and $k' \in \{1, \ldots, n\}$ such that $\langle\langle s_t, s_1, \ldots, s_n, d, o, k\rangle, \langle s'_1, \ldots, s'_n, s'_t, d', o', k'\rangle\rangle \in \rho_s$;*
- *$\omega : \Sigma \times \mathcal{O} \to 2^{\{1,\ldots,n\}}$ is defined as $\omega(\langle s_1, \ldots, s_n, s_t, d\rangle, o) = \{i \in \{1, \ldots, n\} \mid \langle s_1, \ldots, s_n, s_t, d, o, i\rangle \in W\}$.*

The above theorems show how one can exploit tools from system synthesis for computing all compositions of a given target service. In details, starting from $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\rangle$ and $\mathcal{S}_t$ one can build the corresponding game structure G, then compute the set W and, if it contains G's initial state, use such set to generate the PG. In fact, this last step is not really needed. Indeed, it is not hard to convince oneself that given a current state $\langle s_t, s_1, \ldots, s_n, d\rangle$ and an operation to be executed $o \in \mathcal{O}$, a service selection ind is "good" (i.e, the selected service can actually execute the operation and the whole community can still simulate the target service) if and only if W contains a tuple $\langle s_t, s_1, \ldots, s_n, d, o, ind\rangle$, for some $ind \in \{1, \ldots, n\}$. Consequently, at each step, on the basis of the current state s_t of the target service, the states $s_1, \ldots, s_n$ of available services, the state d of data box, and the operation o requested, one can select a tuple from W, extract the ind component, and use it for next service selection.

Finally, observe that time complexity of Algorithm 1 in polynomial in $|V|$, that is the size of input $\Box$-GS' state space. Since in our encoding $|V|$ is polynomial in $|S_1|, \ldots, |S_n|, |S_t|, |D|$ and exponential in n, we get:

Theorem 6. *Let $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\}$ be a community and $\mathcal{S}_t$ a target service over $\mathcal{DB}$. Checking the existence of compositions by reduction to safety games can be done in polynomial time wrt $|S_1|, \ldots, |S_n|, |S_t|, |D|$ and exponential time in n.*

That is, the technique is actually optimal wrt worst-case time complexity, the composition problem being EXPTIME-hard [19].

5 Using TLV for Computing Compositions

Searching for a winning strategy is a problem solvable by several implemented systems (e.g., [16,15,23]). We focus on TLV [23], the basic concepts being valid for all others.

TLV is a software for verification and synthesis of LTL specifications, based on symbolic manipulation of states, by using Binary Decision Diagrams (BDDs). It takes two inputs: *(i)* a synthesis procedure and *(ii)* an LTL specification, encoded in SMV [23], to be manipulated by the procedure. In particular, we refer to a TLV-BASIC procedure for safety games which takes as input an LTL specification that encodes a □-GS and derives from the system's maximal winning set, if non empty, a structure representing the PG, as shown in Theorem 5. For a detailed description of TLV, TLV-BASIC and SMV, we refer to [23], here introducing some essentials only.

Our approach consists in deriving, from the composition problem specification, i.e., community and target service, the SMV encoding of the respective □-GS, as shown in Section 4.2, then execute TLV against this input and obtain, if the problem is feasible, the respective PG.

Figure 1 shows the basic blocks of a sample encoding for a composition problem with 3 available services. Module `Main` wraps up all other modules and represents the whole game. It consists of two submodules (here declared as `system`), `sys` and `env`, which encode, respectively, the environment and the system in the game structure. Goal formula `good` (i.e., the invariant property) is a combination of subformulae `initial` and `failure` of modules `sys` and `env`, directly obtained from the goal formula in the □-GS representation. Observe that `env` and `sys` evolve synchronously, the former choosing the operation and the latter selecting the service for its execution. The transition relation in module `Sys` encodes an *unconstrained controller*, able to output, at each step, any available service index in the interval $[1, n]$. The synthesis' objective is to restrict such a relation so to obtain a winning strategy.

As for module `Env`, it contains all basic blocks the □-GS environment consists of. Observe that its behavior depends on the value of module `sys`' `index` variable, as prescribed by `Main`. According to SMV semantics, modules `db`, `target`, `s1`, . . . , `s`n execute synchronously. However, each of them can be encoded so to emulate asynchrony, by looping when not selected. In particular, the encoding is such that, at each step, `db`, `target` and only one among `s1`, `s2`, . . . , `s`n move, according to the □-GS description. `Env` behavior is as follows. At each step, the available service selected by the current value of `index`, executes the operation requested by `target`, which is stored in `operation`. All other services loop in their current state. At the same time, `db` moves according to `operation`, `target` selects next operation, according to its specification, and `sys` selects a new service. Note that, in general, there may exist states where the selected service cannot perform the requested operation, due to either operation precondition failure (i.e., `db` state) or service's current state. In such cases, expression `failure` of selected service becomes *true* and, consequently, so does `env.failure`. Avoiding such situations, by properly constraining `sys` transition relation, is exactly the synthesis procedure aim.

```
MODULE Main
VAR
  env: system Env(sys.index);
  sys: system Sys;
DEFINE
  good :=
     (sys.initial & env.initial)|
      !(env.failure);
```

```
MODULE Sys
VAR
  index : 0..3; --num of services, 0 used for init
INIT
  index = 0
TRANS
  case
    index=0 : next(index)!=0;
    index!=0 : next(index)!=0;
  esac
DEFINE
  initial := (index=0);
```

```
MODULE Env(index)
 VAR
  operation  : {start_op,pick,store,play,display_content,free_mem};
  db : Databox(operation);
  target : Target(operation,db.state);
  s1 : Service1(index,operation,db.state);
  s2 : Service2(index,operation,db.state);
  s3 : Service3(index,operation,db.state);
DEFINE
  initial := (db.initial & s1.initial & s2.initial & s3.initial &
                               target.initial & operation=start_op);
  failure := (s1.failure | s2.failure | s3.failure) |
                   (target.final & !(s1.final & s2.final & s3.final));
```

Fig. 1. A TLV sample fragment encoding

6 Conclusions

We presented a new technique for composition of partially controllable available services, which exploits the relationships between *(i)* building a simulation relation and *(ii)* checking invariant properties in temporal-logic-based model checkers and synthesis systems (cf., e.g., [26,4]). We showed that all compositions can be computed at once, as solutions to safety games and developed an implementation for the synthesis system TLV (http://www.cs.nyu.edu/acsys/tlv/ and cf., e.g., [22]). Another option would be to exploit ATL-based verifiers, such as Mocha (http://www.cis.upenn.edu/∼mocha/), which can check game-structures for properties such as invariants, and extract winning strategies for them.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services. In: Concepts, Architectures and Applications, Springer, Heidelberg (2004)
2. Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: MOCHA: Modularity in model checking. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998)
3. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S.S. (eds.) HS 1994. LNCS, vol. 999, pp. 1–20. Springer, Heidelberg (1995)
4. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: IFAC Symposium on System Structure and Control, pp. 469–474. Elsevier, Amsterdam (1998)

5. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic Composition of e-Services that Export their Behavior. In: Proc. of ICSOC 2003, pp. 43–58 (2003)
6. Berardi, D., Cheikh, F., De Giacomo, G., Patrizi, F.: Automatic service composition via simulation. Int. J. Found. Comput. Sci. 19(2), 429–451 (2008)
7. Bultan, T., Fu, X., Hull, R., Su, J.: Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In: Proc. of WWW 2003 (2003)
8. Cardose, J., Sheth, A.P.: Introduction to semantic web services and web process composition. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 1–13. Springer, Heidelberg (2005)
9. De Giacomo, G., Sardiña, S.: Automatic synthesis of new behaviors from a library of available behaviors. In: Proc. of IJCAI 2007, pp. 1866–1871 (2007)
10. Gerede, C.E., Hull, R., Ibarra, O.H., Su, J.: Automated composition of e-services: Lookaheads. In: Proc. of ICSOC 2004 (2004)
11. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kauffman, San Francisco (2004)
12. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proc. of FOCS 1995, pp. 453–462 (1995)
13. Hull, R.: Web services composition: A story of models, automata, and logics. In: Proc. of SCC 2005 (2005)
14. Hull, R., Benedikt, M., Christophides, V., Su, J.: E-Services: a Look Behind the Curtain. In: Proc. of PODS 2003, pp. 1–14 (2003)
15. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: Proc. FMCAD 2006, pp. 117–124 (2006)
16. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A tool for property synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007)
17. McIlraith, S., Son, T.C.: Adapting Golog for programming the semantic web. In: Proc. of KR 2002 (2002)
18. Milner, R.: An algebraic definition of simulation between programs. In: Proc. of IJCAI 1971, pp. 481–489 (1971)
19. Muscholl, A., Walukiewicz, I.: A lower bound on web services composition. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 274–286. Springer, Heidelberg (2007)
20. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: State of the art and research challenges. IEEE Computer 40(11), 38–45 (2007)
21. Pistore, M., Traverso, P., Bertoli, P., Marconi, A.: Automated Synthesis of Composite BPEL4WS Web Services. In: Proc. of ICWS 2005 (2005)
22. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005)
23. Pnueli, A., Shahar, E.: The TLV system and its applications. Technical report, Weizmann Institute (1996)
24. Sardiña, S., De Giacomo, G., Patrizi, F.: Behavior composition in the presence of failure. In: Proceedings of KR 2008 (2008)
25. Sardiña, S., Patrizi, F., De Giacomo, G.: Automatic synthesis of a global behavior from multiple distributed behaviors. In: Proc. of AAAI 2007, pp. 1063–1069 (2007)
26. Vardi, M., Fisler, K.: Bisimulation and model checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 338–341. Springer, Heidelberg (1999)
27. Wu, D., Parsia, B., Sirin, E., Hendler, J., Nau, D.: Automating DAML-S Web Services Composition using SHOP2. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 195–210. Springer, Heidelberg (2003)

Towards Compensation Correctness in Interactive Systems

Cátia Vaz[1] and Carla Ferreira[2]

[1] INESC-ID / DEETC, ISEL, Instituto Politécnico de Lisboa, Portugal
cvaz@cc.isel.ipl.pt
[2] CITI / Departamento de Informática, FCT, Universidade Nova de Lisboa, Portugal
carla.ferreira@di.fct.unl.pt

Abstract. One fundamental idea of service-oriented computing is that applications should be developed by composing already available services. Due to the long running nature of service interactions, a main challenge in service composition is ensuring correctness of failure recovery. In this paper, we use a process calculus suitable for modelling long running transactions with a recovery mechanism based on compensations. Within this setting, we discuss and formally state *correctness criteria* for compensable processes compositions, assuming that each process is correct with respect to failure recovery. Under our theory, we formally interpret *self-healing* compositions, that can detect and recover from failures, as correct compositions of compensable processes.

1 Introduction

Service-oriented computing is an emerging paradigm for creating new services by composing available ones, usually in distributed and heterogeneous environments. This paradigm is particularly suited for describing loosely coupled systems, *i.e.*, systems composed by interacting parts that exchange most information through messages (shared information is minimal). Additionally, in these systems, transactions may last long periods of time. Thus, solutions based on locking are not feasible, contrary to traditional ACID transactions. So, long running transactions and recovery mechanisms based on compensations are used instead.

A main challenge in service composition is ensuring the correctness of failure recovery. In particular, because of the long running nature of transactions, usually it is only possible to partially recover a transaction after a failure. Compensations are activities programmed to recover full or partial executions of transactions, bringing the system again to consistency. A relevant issue is ensuring compensation correctness. Moreover, a notion of compensation correctness is needed for interaction based systems, namely a notion that takes into account the specificities of failure recovery for a given application context.

In this paper, we introduce a notion of compensation correctness within a process calculus suitable for modelling long running transactions, with a recovery mechanism based on compensations. In this setting, we discuss *correctness*

C. Laneve and J. Su (Eds.): WS-FM 2009, LNCS 6194, pp. 161–177, 2010.

criteria for compensable process compositions, assuming that each process is correct with respect to failure recovery. A compensation is said to be correct if its execution has the expected behaviour in the sense that it restores the consistency of the transaction. Since the expected behaviour is dependent on the application context, our model expects a correctness map provided by the programmer, expressing how meaningful interactions can be compensated. Hence, the programmer must provide a set of possible sequences of interactions that compensate each meaningful interaction. Notice that this approach is more general than the insurance by the programmer for cancelling or reversing each action made [2,4,5,6]. In some applications, ensuring transaction consistency will in fact mean that the execution of compensations will revert the effect of each action done before failure. But in other cases, some actions cannot be cancelled, so the programmer will only be interested in approximating the effect of cancellation, bringing the transaction back to consistency.

One of the main contributions of our work is the insurance that composition of correct compensable processes is also a correct compensable process, under reasonable correctness criteria. Our correctness criteria are *stateless*, *i.e.*, we assume that all information needed is exchanged through messages. This is known as contextualisation and exchanged messages describe the state of the overall system.

The developed theory also provides interesting insights on an important issue in the service oriented approach, namely the reliable automated composition of distributed services. In particular, one important challenge is the *self-healing* composition of services, *i.e.*, compositions that automatically detect that some service composition requirements are no longer satisfied and react recovering from the requirement violations. Thus, self-healing implies that when a failure occurs, the system should automatically recover, bringing it back to consistency. Within this setting, we will formally interpret self-healing compositions, relating this concept with the correctness of composition of compensable processes.

In the paper realm, we have chosen to build the calculus upon the core of asynchronous π-calculus, with a notion of transaction scope and other primitives to allow dynamic recovery based on compensations. In our calculus, when transactions fail, they know what interaction context they belong to, *i.e.*, the underlying interaction session. Since this paper aims at tackling the problem of consistency of compensable transactions, we only focus on the interaction among transactions occurring within sessions, not in primitives such as service definition and instantiation. However, the calculus could be extended to include these kind of primitives, in a similar way of the work of Honda *et al.* [9].

In Section 2 we introduce the syntax of the compensating calculus, the labelled transition semantics and a well-formedness criteria. Then, in Section 3, we give some illustrative examples and motivate some key ideas of the correctness criteria. In Section 4 we define the correctness criteria and show the main results. Section 5 concludes with related work and future issues.

2 A Compensating Calculus

In order to reason about a correctness criteria on compensable transactions, we propose a *compensating calculus* for modelling long running transactions with a recovery mechanism based on compensations.

This calculus is inspired on dcπ-calculus [17] and the calculus presented in [11], but is focused on the relevant primitives for reasoning about the correctness of compensations. The language recovery mechanism allows to incrementally update the compensations of transactions, within each interaction. To achieve that, we associate to each input a compensation function that updates the compensation of the transaction upon message reception. We have decided not to allow messages to perform compensation updates since in an asynchronous setting, messages cannot be observed.

A transaction $t[P, Q]_r$ occurs within session r and behaves as process P until an error is notified by an output $\overline{t}$ on the name t of the transaction. In case of failure, which can be either external or internal to the transaction, P is killed and compensation Q is activated and protected against nested external failures. The transaction name t identifies the failure unit. Note that we must know the context session for each transaction, since our correctness notion relies on the analysis of the communicated names within each session.

The calculus allows for nested transactions, but only if it is within different interaction sessions - see ahead. The failure handling occurs in a nested way, *i.e.*, while the abortion of a transaction is silent to its parent, it causes the abortion of all proper subtransactions and the activation of compensations installed either by the transaction or by all of its subtransactions. Notice that the level of granularity of the scope of the interaction sessions is very flexible. In our model, the system designer can choose from defining each two compensable transactions within a different interaction session, to define all the compensable transactions within a unique session.

2.1 Syntax

The syntax of our language relies on: a countable set of *channel names* N, ranged over by $a, b, x, y, a_1, b_1, x_1, y_1 \ldots$; a countable set of *transaction identifiers* T, ranged over by $t, u, t_1, u_1, \ldots$; a countable set of *session names* S, ranged over by $s, r, s_1, r_1, \ldots$; and natural numbers, ranged over by $i, j, k, i_1, j_1, k_1, \ldots$. The sets N, T and S are disjoint and identifiers $v, w, z, v_1, w_1, z_1 \ldots$ are used to refer to elements of both sets N and T when there is no need to distinguish them. The tuple $\boldsymbol{v}$ denotes a sequence $v_1 \cdots v_n$ of such identifiers, for some $n \geq 0$, and $\{\boldsymbol{v}\}$ denotes the set of elements of that sequence.

Definition 1. *The grammar in Figure 1 defines the syntax of processes.*

The calculus includes the core of asynchronous π-calculus processes [15], namely inaction, output, parallel composition and scope restriction. A new primitive is the *transaction scope* $t[P, Q]_r$, that within session r behaves as process P until

$$
\begin{array}{rlll@{\qquad}ll}
P, Q ::= & \mathbf{0} & \text{(Inaction)} & & \mid \sum_{i\in I} a_i(\boldsymbol{x}_i)\lfloor \lambda X_i.Q_i \rfloor.P_i & \text{(Input guarded choice)} \\
\mid & \overline{a}\langle \boldsymbol{v} \rangle & \text{(Output)} & & \mid (P \mid Q) & \text{(Parallel composition)} \\
\mid & X & \text{(Variable)} & & \mid (\nu\, x)\ P & \text{(Restriction)} \\
\mid & \overline{t} & \text{(Failure)} & & \mid t[P, Q]_r & \text{(Transaction scope)} \\
 & & & & \mid \langle P \rangle_r & \text{(Protected block)}
\end{array}
$$

Fig. 1. Syntax of processes

an error is notified by an output $\overline{t}$ on the name t of the transaction. In case of error, P is killed and compensation Q is executed. Error signal $\overline{t}$, that sends a failure message to a transaction identified by t, may come both from the internal process P or from an external process. The *protected block* $\langle P \rangle_r$, that behaves as P within session r, cannot be interrupted even if it occurs in the scope of a failing transaction.

Process P in transaction $t[P, Q]_r$ can update the compensation Q. Compensation update is performed by input prefixes. As said before, we have chosen not to associate compensations with the message sender, since in an asynchronous context there are limited guarantees about the state of the receiver. A compensation update takes the form of a function $\lambda X.Q'$, where process variable X can occur inside process Q'. Applying such a compensation update to compensation Q produces a new compensation $Q'\{{}^{Q}\!/_{X}\}$. Note that Q may not occur in the resulting compensation, or it may occur more than once. Thus the form of input prefix is $a(x)\lfloor \lambda X.Q' \rfloor.P$, which upon reception of message $\overline{a}\langle \boldsymbol{v} \rangle$ updates the compensation with $(\lambda X.Q')\{{}^{\boldsymbol{v}}\!/_{\boldsymbol{x}}\}$ and continues as $P\{{}^{\boldsymbol{v}}\!/_{\boldsymbol{x}}\}$.

The compensation mechanism of the calculus allows for both dynamic generation and static definition of compensations. In fact, if all compensation updates have the form $\lambda X.X$, then the compensation is never changed. We will use id to denote the identity function $\lambda X.X$. The prefix $a(x).P$ can thus be seen as a shortcut for $a(x)\lfloor \mathsf{id} \rfloor.P$.

Since the goal of this paper is to reason about correctness criteria on compensable transactions, we excluded the input guarded replication primitive. The inclusion of this primitive would require the calculus to have primitives for session and transaction initiation, which is not the focus of this paper. Nevertheless our calculus could be extended to include these kind of primitives, following the approach of Honda *et al.* [9]. Such primitives would initiate a new session and ensure the generation of fresh names, such as fresh transaction names. Another alternative for including input guarded replication would be to add a well formed property as used in the work of Lucchi and Mazzara [13], expressing that received names cannot be used as subjects of inputs or of replicated inputs. In both approaches we must ensure that, after each session initiation, the session name is unequivocally identified. This is a requirement for our notion of correctness.

In the following, we denote the channel names, the session names and the transaction names of a process P as $\mathsf{cn}(P)$, $\mathsf{sn}(P)$ and $\mathsf{tn}(P)$, respectively. The names of P, denoted by $\mathsf{n}(P)$, are the union of the three sets.

$$\begin{aligned}
\mathsf{nl}(n,\mathbf{0}) &= \emptyset \\
\mathsf{nl}(n,\overline{t}) &= \emptyset \\
\mathsf{nl}(n,\overline{a}\langle \boldsymbol{v}\rangle) &= \begin{cases} \{a\} \cup \{v \mid v \in \boldsymbol{v}\} & \text{if } n = 0 \\ \emptyset & \text{if } n \neq 0 \end{cases}
\end{aligned}
\qquad
\begin{aligned}
\mathsf{nl}(n,X) &= \emptyset \\
\mathsf{nl}(n,\langle P\rangle_r) &= \mathsf{nl}(n,P) \\
\mathsf{nl}(n,t\,[P,Q]_r) &= \mathsf{nl}(n,P) \cup \mathsf{nl}(n\text{-}1,Q) \\
\mathsf{nl}(n,P \mid Q) &= \mathsf{nl}(n,P) \cup \mathsf{nl}(n,Q) \\
\mathsf{nl}(n,(\nu\,x)\,P) &= \mathsf{nl}(n,P)
\end{aligned}$$

$$\mathsf{nl}(n,\textstyle\sum_{i\in I} a_i(\boldsymbol{x}_i)\lfloor \lambda Y_i.Q_i \rfloor.P_i) = \begin{cases} \cup_{i\in I}(\{a_i\} \cup \mathsf{nl}(n,P_i)) & \text{if } n = 0 \\ \cup_{i\in I}(\mathsf{nl}(n,P_i) \cup \mathsf{nl}(n\text{-}1,Q_i)) & \text{if } n \neq 0 \end{cases}$$

Fig. 2. Channel names of P at level n

2.2 Well-Formedness

For simplicity of the correctness *criteria*, we introduce a well-formedness *criteria* to rule out some wrong processes designs. To this aim, we first introduce some terminology.

A *context* is a process term $C[\bullet]$ which is obtained by replacing in a process an occurrence of $\mathbf{0}$ with a placeholder $\bullet$. Process $C[P]$ is obtained by replacing inside $C[\bullet]$ the $\bullet$ with P. The notion of context can be generalised to n-holes contexts as expected. In particular, generic 2-holes contexts will be denoted by $C[\bullet_1,\bullet_2]$, with $C[P,Q]$ defined as the process obtained by replacing $\bullet_1$ with P and $\bullet_2$ with Q.

Definition 2. *A* session context $C[\bullet]$ *is a context such that the hole occurs within a transaction scope or within a protected block. We denote by C_r a session context that includes a transaction scope or a protected block within session r.*

Definition 3. *Let P be a process and $r, r' \in \mathsf{sn}(P)$ two interaction session names. We write $r \prec_P r'$ if there are two contexts C and C' such that $P = C_r[C'_{r'}[Q]]$, for some process Q.*

Function nl assigns to a natural number n and a process P the channel names that occur at level n in P. Differentiation of names by levels is required to ensure compositional correctness. The goal is the separation, under arbitrary nesting, of normal flow messages from compensation flow messages.

Definition 4. *The function* nl : $\times\, \mathcal{P} \longrightarrow 2^{\mathcal{N}}$*, which gives the set of free channel names of a process P occurring at level n, is defined in Fig. 2.*

The need for this kind of differentiation by level is also pointed out by Carbone *et al.* [7]. However, in contrast to our calculus where compensations are compensable, their exception handlers never fail.

We now define well-formed processes.

Definition 5 (well-formedness). *A compensable process P is* well formed *if the following conditions hold:*

1. *Transaction names are distinct. Different transactions cannot share the same activation name and every failure message is able to activate only a single compensation.*

2. *Communication outside sessions or among distinct sessions is not allowed to install compensations. If two transactions belong to different sessions, their communications cannot be compensated. Also, if the process does not occur within a transaction scope, it cannot install compensations (i.e. compensation updates are* id*).*
3. *relation* $\prec_P$ *is acyclic for all* $s \in \mathsf{sn}(P)$ *(that is,* $\prec_P^+$ *is irreflexive).*
4. *There is no interaction between channel names of different levels, i.e.,* $a \in \mathsf{nl}(n, P) \Rightarrow \forall_{m \neq n}\ a \notin \mathsf{nl}(m, P)$
5. *All bound names are pairwise distinct and disjoint from the set of free names.*

The first property is needed to avoid ambiguity on scope names. The second and third properties are for simplicity of the correctness criteria. Namely, the third property is to avoid processes like $t[k[P, Q]_r, S]_r$ and $t[k[P, Q]_{r'}, S]_r \mid t'[k'[P', Q']_r, S']_{r'}$. The purpose of using levels, namely not allowing interaction between channel names of different levels, is to rule out communication between the normal flow with the compensation flow of a process. Since we allow nesting of compensable transactions, the same idea has to be applied to all levels. The last property is for simplicity of correct compensable processes definition.

2.3 Operational Semantics

The dynamic behaviour of processes is defined by a labelled transition system which takes into account transaction scope behaviour. Upon transaction failure, stored compensations must be activated while preserving all inner protected blocks. Therefore, one has to extract the stored compensations and place them in a protected block, and also preserve already existing protect blocks. The extraction is done by function extr which is defined next.

The definition of function $\mathsf{extr}(P)$ considers a *nested failure* approach, *i.e.*, when a parent transaction is killed, all its subtransactions have to be killed. This is, for instance, the approach of BPEL and others. Notice also that if the compensation is defined for a transaction scope within session r, the failure of the transaction will place the corresponding compensation into a protected block also within session r. The function $\mathsf{extr}(\bullet)$ is defined in Fig. 3.

With respect to bindings, names in $\boldsymbol{x}$ and z are bound in $a_i(\boldsymbol{x}_i)\lfloor \lambda X_i.Q_i \rfloor.P_i$ and in $(\nu\, z)\, P$, respectively. The other names are free. Furthermore, we use the standard notions of free names of processes. We write $\mathsf{bn}(P)$ (respectively $\mathsf{fn}(P)$) for the set of names that are bound (respectively free) in a process P. Bound names can be α-converted as usual. Also, variable X is bound in $\lambda X.Q$. We

$$\begin{array}{rlrl}
\mathsf{extr}(\mathbf{0}) &= \mathbf{0} & \mathsf{extr}(\langle P\rangle_r) &= \langle P\rangle_r \\
\mathsf{extr}(\overline{t}) &= \mathbf{0} & \mathsf{extr}(t\,[P, Q]_r) &= \mathsf{extr}(P) \mid \langle Q\rangle_r \\
\mathsf{extr}(\overline{a}\langle \boldsymbol{v}\rangle) &= \mathbf{0} & \mathsf{extr}(P \mid Q) &= \mathsf{extr}(P) \mid \mathsf{extr}(Q) \\
\mathsf{extr}(\sum_{i\in I} a_i(\boldsymbol{x}_i)\lfloor \lambda Y_i.Q_i \rfloor.P_i) &= \mathbf{0} & \mathsf{extr}((\nu\, x)\, P) &= (\nu\, x)\, \mathsf{extr}(P)
\end{array}$$

Fig. 3. Extraction function with nested failure

$$\begin{array}{lll@{\qquad}lll@{\qquad}lll@{\qquad}lll}
\alpha & ::= & \textit{label tuples} & \alpha_i & ::= & \textit{input} & \alpha_c & ::= & \textit{compensation} & \alpha_o & ::= & \textit{output} \\
& \mid & (r, \alpha_i, \alpha_c) & & \mid & t & & \mid & \lambda X.R & & \mid & (\boldsymbol{w})\overline{a}\langle \boldsymbol{v} \rangle \\
& \mid & (r, \alpha_o) & & \mid & a(\boldsymbol{v}) & & \mid & (\boldsymbol{w})\lambda X.R & & \mid & \overline{t} \\
& & & & \mid & \tau(a) & & & & & \mid & \overline{a}\langle \boldsymbol{v} \rangle
\end{array}$$

Fig. 4. Transition labels

consider only processes with no free variables. As usual, the term $(\nu\,\boldsymbol{x})\,P$ abbreviates $(\nu\,x_1)\,...(\nu\,x_n)\,P$, for some $n \geq 0$. Before presenting the rules of the labelled transition system, we first introduce the transition labels.

Labels α have the syntax and informal meaning defined in Figure 4. Notice that in internal moves, we keep track of the names that are used as subject of a communication. Also, we keep the session names where the interaction has occurred. If the interaction occurs within different sessions, we keep the name of the session that has received the message. These extra information is necessary for defining correct compensable processes. We use $\perp$ whenever an interaction occurs outside a session.

The operational semantics of the language is given in terms of the labelled transition system $(\mathcal{P}, \mathcal{L}, \xrightarrow{\alpha})$, with $\mathcal{L}$ the set of labels within the set $\mathcal{N}$ of names, the set $\mathcal{S}$ of session names and the set $\mathcal{T}$ of transaction identifiers.

Definition 6. *The operational semantics of compensable processes CP is the minimum LTS closed under the rules in Figure 5 (symmetric rules are considered for* L-Par*,* L-Comm*).*

Rule L-Out sends a message and rule L-Fail sends a failure message. Rule L-Inp executes an input-guarded choice. Note that the substitution of the received name is applied both to the continuation and to the compensation to be installed. Note also that labels for inputs (and internal moves) are composed by an extra part, the update to the compensation. Rule L-Par allows one of the components of parallel composition to progress. Rule L-Comm allows communication, and propagates the compensation update coming from the input. As already said, we keep track of the names that are used as subject. Notice that if the communication does not occur within the same interaction session, the compensation update can only be the identity id. This feature is given by condition $r \neq s \Rightarrow R = X$ in the rule L-Comm, *i.e.* there is no real installation of compensations. If we allowed installation between different sessions, we would need more information about the interaction. For instance, we would need information about the communicated names and both the sessions where the message was received and where the message was sent. Such additional information is required to analyse the correctness of both sessions, making the setting more complex and always evolving session merging. Rule L-Res is the classic rule for restriction. Note that session names are not restricted. Rule L-Open allows to extrude bound names. Rule L-Open2 allows to extrude names occurring in the compensation update.

(L-Out)

$$\frac{}{\overline{a}\langle\boldsymbol{v}\rangle \xrightarrow{(\bot,\overline{a}\langle\boldsymbol{v}\rangle)} \mathbf{0}}$$

(L-Inp)

$$\frac{j \in I}{\sum_{i\in I} a_i(\boldsymbol{x}_i)\lfloor\lambda X_i.R_i\rfloor.P_i \xrightarrow{(\bot,a_j(\boldsymbol{v}),\lambda X_j.R_j\{\boldsymbol{v}/\boldsymbol{x}_j\})} P_j\{\boldsymbol{v}/\boldsymbol{x}_j\}}$$

(L-Par)

$$\frac{P \xrightarrow{\alpha} P' \quad \mathsf{bn}(\alpha)\cap\mathsf{fn}(Q)=\emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

(L-Open)

$$\frac{P \xrightarrow{(r,(\boldsymbol{w})\overline{x}\langle\boldsymbol{v}\rangle)} P' \quad z \neq x \quad z \in \{\boldsymbol{v}\}\setminus\{\boldsymbol{w}\}}{(\nu\, z)\, P \xrightarrow{(r,(z\boldsymbol{w})\overline{x}\langle\boldsymbol{v}\rangle)} P'}$$

(L-Fail)

$$\frac{}{\overline{t} \xrightarrow{(\bot,\overline{t})} \mathbf{0}}$$

(L-Open2)

$$\frac{P \xrightarrow{(r,\alpha_i,(\boldsymbol{w})\lambda X.R)} P' \quad (\alpha_i = a(\boldsymbol{v}) \Rightarrow z \notin \mathsf{n}(\alpha_i)) \quad z \in \mathsf{fn}(R)\setminus\{\boldsymbol{w}\}}{(\nu\, z)\, P \xrightarrow{(r,\alpha_i,(z\boldsymbol{w})\lambda X.R)} P'}$$

(L-Comm)

$$\frac{P \xrightarrow{(r,x(\boldsymbol{v}),(\boldsymbol{w})\lambda X.R)} P' \quad Q \xrightarrow{(s,(\boldsymbol{z})\overline{x}\langle\boldsymbol{v}\rangle)} Q' \quad \{\boldsymbol{z}\}\cap\mathsf{fn}(P)=\{\boldsymbol{w}\}\cap\mathsf{fn}(Q)=\emptyset \quad r\neq s \Rightarrow R=X}{P \mid Q \xrightarrow{(r,\tau(x),(\boldsymbol{w})\lambda X.R)} (\nu\,\boldsymbol{z})\,(P' \mid Q')}$$

(L-Scope-out)

$$\frac{P \xrightarrow{(\bot,(\boldsymbol{w})\overline{a}\langle\boldsymbol{v}\rangle)} P' \quad \{\boldsymbol{w}\}\cap(\mathsf{fn}(Q)\cup\{t\})=\emptyset}{t[P,Q]_s \xrightarrow{(s,(\boldsymbol{w})\overline{a}\langle\boldsymbol{v}\rangle)} t[P',Q]_s}$$

(L-Res)

$$\frac{P \xrightarrow{\alpha} P' \quad x \notin \mathsf{n}(\alpha)}{(\nu\, x)\, P \xrightarrow{\alpha} (\nu\, x)\, P'}$$

(L-Scope-in)

$$\frac{P \xrightarrow{(\bot,\alpha_i,(\boldsymbol{z})\lambda X.R)} P' \quad \{\boldsymbol{z}\}\cap(\mathsf{fn}(Q)\cup\{t\})=\emptyset}{t[P,Q]_s \xrightarrow{(s,\alpha_i,\mathsf{id})} (\nu\,\boldsymbol{z})\, t[P',R\{Q/X\}]_s}$$

(L-recover-out)

$$\frac{}{t[P,Q]_s \xrightarrow{(s,t,\mathsf{id})} \mathsf{extr}(P) \mid \langle Q\rangle_s}$$

(L-Scope)

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \in \{(s,\alpha_o),(s,\alpha_i,\alpha_c)\} \quad s \neq \bot}{t[P,Q]_r \xrightarrow{\alpha} t[P',R]_r}$$

(L-Recover-in)

$$\frac{P \xrightarrow{(r,\overline{t})} P'}{t[P,Q]_s \xrightarrow{(s,\tau(t),\mathsf{id})} \mathsf{extr}(P') \mid \langle Q\rangle_s}$$

(L-Block)

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \in \{(s,\alpha_o),(s,\alpha_i,\alpha_c)\} \quad s \neq \bot}{\langle P\rangle_r \xrightarrow{\alpha} \langle P'\rangle_r}$$

(L-Block-s)

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \in \{(\bot,\alpha_o),(\bot,\alpha_i,\alpha_c)\}}{\langle P\rangle_r \xrightarrow{\alpha'} \langle P'\rangle_r \quad \alpha' = \alpha\{r/\bot\}}$$

Fig. 5. LTS for compensable processes

Transaction failures are modelled by L-Recover-out and L-Recover-in. Rule L-Recover-out allows external processes to kill a transaction via a signal t. Notice that the remaining process is composed by two parts: the first one extracted from P, and the second one corresponding to compensation Q, which will be executed inside a protected block. Rule L-Recover-in is similar, but in this case the failure message is internal to the transaction, *i.e.* comes from P. Rule L-Scope-in updates the compensation of a transaction. Rule L-Scope-out allows outputs to go outside transactions, provided that they are not termination signals for the transaction itself. Rule L-Scope is used when input or output

$$\textsf{\textbf{OrderTransaction1}} \overset{\text{def}}{=} \textsf{\textbf{Client}} \mid \textsf{\textbf{Shop1}} \mid \textsf{\textbf{Bank}}$$

$$\textsf{\textbf{Client}} \overset{\text{def}}{=} u\big[\overline{client} \mid (ack.\overline{t} + ack.recp.\overline{okShop}), \mathbf{0}]_{r_0}$$

$$\textsf{\textbf{Shop1}} \overset{\text{def}}{=} t\big[client.(\overline{ack} \mid \overline{initchg}) \mid \textsf{\textbf{Charge}} \mid okShop.\overline{ok}, \mathbf{0}\big]_{r_0}$$

$$\begin{aligned}\textsf{\textbf{Charge}} \overset{\text{def}}{=} c\big[&initchg.\overline{bank} \mid (valid\lfloor \lambda X.refunded \mid X\rfloor.(\overline{recp} \mid ok.\overline{end1}) + invalid.\overline{t}) \mid \\ &ended1\lfloor \lambda X.\mathbf{0}\rfloor, \overline{q}\big]_{r_1}\end{aligned}$$

$$\begin{aligned}\textsf{\textbf{Bank}} \overset{\text{def}}{=} q\big[&bank.(\nu\, y)\,(\overline{y} \mid (y\lfloor \lambda X.\overline{refunded} \mid X\rfloor.\overline{valid} + y.\overline{invalid}) \mid \\ &end1\lfloor \lambda X.\mathbf{0}\rfloor).\overline{ended1}, \mathbf{0}\big]_{r_1}\end{aligned}$$

Fig. 6. Ordering system example

is from an inner transaction scope. Finally, rules L-Block and L-Block-s define the behaviour of a protection block, both when an interaction occurs within different sessions or within its own session, respectively.

We call reduction any transition $P \xrightarrow{(r,\tau(a),\alpha_c)} P'$, with $r \in S$ and $a \in N \cup T$ and α_c a compensation function.

Lemma 1. *Well-formedness is preserved by reductions.*

3 Examples

This section illustrates the expressiveness of the calculus through two Web Services case studies, while motivating the needed for a correctness *criteria*.

3.1 Order Transaction

The first example consists of an ordering system. We can think of it as a web shop that accepts orders from clients. Whenever a client submits an order, the system must take care of payments. The system is modelled as depicted in Figure 6, which for simplicity only considers one client and one order. Notice that the generalisation of this example, where the shop has to interact with several clients, implies the use of a session initiation mechanism, as discussed in Section 2.1.

The Client submits an order to the Shop and waits for order confirmation. The Shop receives the message and tries to charge the Client. The payment is done by the Bank, therefore the Shop sends a message to the Bank and it starts a new interaction session. Notice that the Client may cancel the Shop transaction, causing the execution of the compensation of this transaction. If charging is successfully accomplished, the Shop sends a message to the Client. The Client confirms to the shop the receipt delivery. The Shop informs the Bank that the transaction within the Client has ended, removing all the compensations of the Shop transaction. Then, after the ending notification, the Bank also removes its compensations. Notice that compensations are incrementally built.

$$\textsf{\textbf{OrderTransaction2}} \stackrel{\text{def}}{=} \textsf{\textbf{Client2}} \mid \textsf{\textbf{Shop2}} \mid \textsf{\textbf{Bank}} \mid \textsf{\textbf{Warehouse}}$$

$$\textsf{\textbf{Client2}} \stackrel{\text{def}}{=} u\big[\overline{client} \mid (ack.\bar{t} + ack.(recp.\overline{okShop} \mid delivered.\overline{yesShop})), \mathbf{0}\big]_{r_0}$$

$$\textsf{\textbf{Shop2}} \stackrel{\text{def}}{=} t\big[client.(\overline{ack} \mid \overline{initchg} \mid \overline{initpck}) \mid \textsf{\textbf{Charge}} \mid \textsf{\textbf{Pack}} \mid okShop.\overline{ok} \mid yesShop.\overline{yes}, \mathbf{0}\big]_{r_0}$$

$$\begin{aligned}\textsf{\textbf{Pack}} \stackrel{\text{def}}{=} p\big[&initpck.\overline{pack} \mid (exists\lfloor \lambda X.unpacked \mid X\rfloor.(\overline{delivered} \mid yes.\overline{end2})+\\ &notExists.\bar{t}) \mid ended2\lfloor \lambda X.\mathbf{0}\rfloor\,,\ \bar{q}\big]_{r_2}\end{aligned}$$

$$\begin{aligned}\textsf{\textbf{Warehouse}} \stackrel{\text{def}}{=} q\big[&pack.(\nu\, z)\,(\bar{z} \mid (z\lfloor \lambda X.\overline{unpacked} \mid X\rfloor.\overline{exists} + z.\overline{notExists})\mid\\ &end2\lfloor \lambda X.\mathbf{0}\rfloor).\overline{ended2}\,,\ \mathbf{0}\big]_{r_2}\end{aligned}$$

Fig. 7. Ordering system with unexpected behaviour

For example, when the bank starts to interact with the shop, it installs the compensation $\lambda X.\overline{refunded} \mid X$ and, when it receives the *end1* message, it installs the compensation $\lambda X.\mathbf{0}$. The ability of changing compensations within the execution of the process is an important feature of dynamic installation mechanism. For instance the compensation for the Bank transaction is only removed when the Bank receives a terminating message from the subtransaction Charge, the only transaction that is interacting with it. The installation of $\lambda X.\mathbf{0}$ can be regarded as a transaction commit, since compensations are cleaned and recovery is no longer possible. Notice that the interactions between Client and Shop1 is done within session r_0, and Charge and Bank is within session r_1. Also, communications within transactions belonging to different interaction sessions do not install compensations. In this example, it is natural for the programmer to expect the following behaviour of the system: if the subtransactions of the Shop1 have ended, then compensations are not expected to occur; if the client chooses to cancel Shop1 transaction after the bank has validated the purchase but before the transaction ending, refunding must be processed.

3.2 Order Transaction with Warehouse

This example extends the previous one by adding order packing to the ordering system. The system is modelled as depicted in Figure 7, and similarly to the previous example only considers one client and one order. In this case, after receiving the message from the Client, the Shop starts, within different interaction sessions, two subtransactions, one to charge the Client and another to pack the order. Notice that transactions Charge and Bank are the same as in the previous example. The interaction session of Pack and Warehouse transactions has a similar behaviour to the interaction session of transactions Charge and Bank, but with different actions. In this example, the Client may also cancel the Shop transaction. In this case, the Shop transaction fails and its compensations are executed. However, if subtransaction Pack has been successfully accomplished, it may not be possible to compensate it (after *ended2* message has been communicated). In this case, the behaviour of the system is not the expected one, since the client can get the goods for free. Later we shall see how under our formal framework we can detect such wrong behaviour.

A possible solution for overcoming the unexpected behaviour described above is presented in Figure 8. Later, we shall see how our notion of correctness asserts that this is valid.

$$\textsf{\textbf{OrderTransaction3}} \overset{\text{def}}{=} \textsf{\textbf{Client3}} \mid \textsf{\textbf{Shop3}} \mid \textsf{\textbf{Bank}} \mid \textsf{\textbf{Warehouse}}$$

$$\textsf{\textbf{Client3}} \overset{\text{def}}{=} u\big[\overline{client} \mid (ack.\bar{t} + ack.done), \mathbf{0}\big]_{r_0}$$

$$\textsf{\textbf{Shop3}} \overset{\text{def}}{=} t\Big[client.(\overline{ack} \mid \textsf{\textbf{Charge}} \mid \textsf{\textbf{Pack}} \mid delivered.recp.(\overline{done} \mid \overline{ok} \mid \overline{yes})), \mathbf{0}\Big]_{r_0}$$

Fig. 8. Ordering system with expected behaviour

4 Process Correctness

A programmer expects that communicating programs should realise a correct conversation, even when one of the interacting partners fails due to an unexpected event. Ensuring the correctness of a compensable process is a challenging task. In this section we define a notion of correctness for compensable processes. The proposed notion takes into account that in real world scenarios some actions are not compensable and compensations can be much more than a simple "undo".

Definition 7. *Let P be a process and $s \in \mathcal{L}^n$. We say that P has s as a computation, $P \xrightarrow{s}$, if $s = \alpha_1 ... \alpha_n$ and $P \xrightarrow{\alpha_1} P_1 ... \xrightarrow{\alpha_n} P_n$. We also define $\mathcal{L}^* = \cup_{i \in \mathbb{N}} \mathcal{L}^i$.*

Definition 8. *Let P be a process. We define $L(P) = \{s \in \mathcal{L}^* \mid P \xrightarrow{s}\}$.*

Given $s, s' \in \mathcal{L}^*$, we write $s \prec s'$ whenever the trace s is a subsequence of the trace s'.

Definition 9. *The function* $\mathsf{sn} : \mathcal{L}^* \longrightarrow 2^{\mathcal{N}}$*, which gives the set of session names occurring in a trace, is defined as:*

$$\mathsf{sn}(\epsilon) = \emptyset \qquad \mathsf{sn}((r, \alpha_o).s) = \{r\} \cup \mathsf{sn}(s) \qquad \mathsf{sn}((r, \alpha_i, \alpha_c).s) = \{r\} \cup \mathsf{sn}(s)$$

Definition 10. *The function* $\mathsf{com} : \mathcal{L}^* \times \mathcal{S} \longrightarrow \mathcal{N}^*$*, which maps each trace s to the sequence of communicated names within session r, is defined as:*

$$\begin{array}{rr} \mathsf{com}(\epsilon, r) = \epsilon & \mathsf{com}((r, a(\boldsymbol{v}), \alpha_c).s, r) = \mathsf{com}(s, r) \\ \mathsf{com}((r, \alpha_o).s, r) = \mathsf{com}(s, r) & \mathsf{com}((r, \tau(a), \alpha_c).s, r) = a.\,\mathsf{com}(s, r) \end{array}$$

We require the programmer to provide a *correctness map* that expresses how meaningful interactions can be compensated, *i.e.*, the programmer gives a set of possible finite sequences of interactions that compensate each meaningful interaction. This map and the possible sequences are defined over the set of free names. Notice that the correctness mapping may not be directly equivalent to the compensation pairs that can be extracted from a compensable process.

Definition 11. *A* correctness mapping $\varphi : \mathcal{N} \longrightarrow 2^{\mathcal{N}^*}$ *maps each name $n \in \mathcal{N}$ to a set of sequences of names.*

Consider the previous examples. In *OrderTransaction1*, a feasible correctness map could be defined as: $\varphi(\mathit{valid}) = \{\mathit{refunded}\}$; $\varphi(\mathit{ok}) = \emptyset$; and $\varphi(x) = \{\epsilon\}$ for each $x \in \mathsf{fn}(\mathit{OrderTransaction1})$ such that $x \notin \{\mathit{valid}, \mathit{ok}\}$. Notice that a mapping to set $\{\epsilon\}$ or mapping to the empty set have completely different meanings. Mapping to $\{\epsilon\}$ means that the programmer does not expect to see a compensation trace for that action. However, the mapping to the empty set will mean, as we shall see, that after the communication of *ok*, the programmer is not expecting to see the previous defined compensations. In fact, this is coherent with this example, since after doing *ok* the client was notified with a receipt and confirmed the receipt delivery. Thus, in this situation, it would not make sense to execute any compensation.

However, in the case of example *OrderTransaction2* a feasible correctness map could be defined as: $\varphi(\mathit{valid}) = \{\mathit{refunded}\}$; $\varphi(\mathit{pack}) = \{\mathit{unpacked}\}$; $\varphi(x) = \{\epsilon\}$ for each $x \in \mathsf{fn}(\mathit{OrderTransaction2})$ such that $x \notin \{\mathit{valid}, \mathit{unpacked}\}$. Notice that in this case, it does not make sense to define $\varphi(\mathit{ok}) = \emptyset$ and $\varphi(\mathit{yes}) = \emptyset$, since a client may cancel the transaction after Packing has been successfully accomplished. Thus, under our correctness *criteria*, this process would not be correct.

Definition 12. *Let $\oplus : \mathcal{L}^* \times \mathcal{L}^* \longrightarrow 2^{\mathcal{L}^*}$ be a commutative operator, defined as:*

$$\epsilon \oplus \epsilon = \{\epsilon\} \qquad \epsilon \oplus \alpha = \{\alpha\} \qquad \alpha.s \oplus \beta.r = \alpha.(s \oplus \beta.r) \cup \beta.(s \oplus \beta.r)$$

We further extend $\oplus$ to sets as an associative operator, $\oplus : 2^{\mathcal{L}^*} \times 2^{\mathcal{L}^*} \longrightarrow 2^{\mathcal{L}^*}$, as follows:

$$S \oplus S' = \cup_{(s,s') \in S \times S'} s \oplus s',$$

where $S, S' \subset \mathcal{L}^*$.

The intuition behind the operator $\oplus$ is that, given two traces, we are able to generate their interleaving. Clearly, the interleaving is not unique and thus we may have several different alternative interleaved traces.

Definition 13. *A process is* passive *if it can only perform an input labelled transition as a first possible action (no internal moves or outputs).*

Definition 14. *Let P be a well formed compensable process, φ a correctness mapping and $r \in \mathsf{sn}(P)$. P is* φ-correct with respect to r *if, whenever $P \xrightarrow{s}\xrightarrow{\alpha}\xrightarrow{s'} Q$, with s and s' traces, $\alpha = (r, t, \mathsf{id})$ or $\alpha = (r, \tau(t), \mathsf{id})$, t a failure unit of session r and Q a passive process, exists $s^* \prec s'$ such that* $\mathsf{com}(s^*, r) = \beta_1 ... \beta_n \in \oplus_{i=1}^{m} \varphi(\alpha_i)$ *and $\beta_1, ..., \beta_n \in \mathsf{nl}(k, P)$, with* $\mathsf{com}(s, r) = \alpha_1 ... \alpha_m$ *and $\alpha_1, ..., \alpha_m \in$* $\mathsf{nl}(k-1, P)$*, for $k > 0$.*

$$
\begin{aligned}
s = {} & (r_0, \tau(\mathit{client}), \mathsf{id})\,(r_2, \tau(\mathit{initPack}), \mathsf{id})\,(r_2, \tau(\mathit{pack}), \mathsf{id})\,(r_2, \tau(y), \mathsf{id}) \\
& (r_2, \tau(\mathit{initchg}), \mathsf{id})\,(r_2, \tau(\mathit{bank}), \mathsf{id})\,(r_2, \tau(\mathit{exists}), \mathsf{id})\,(r_0, \tau(\mathit{ack}), \mathsf{id})\,(r_0, \tau(\mathit{delivered}), \mathsf{id}) \\
& (r_0, \tau(\mathit{yesShop}), \mathsf{id})\,(r_2, \tau(\mathit{yes}), \mathsf{id})\,(r_2, \tau(\mathit{ended2}), \mathsf{id})\,(r_1, \tau(y), \mathsf{id}))\,(r_1, \tau(\mathit{invalid}), \mathsf{id}))
\end{aligned}
$$

Fig. 9. Trace

Definition 15. *Let φ be a correctness mapping. A well formed process P is* φ-correct *if it is φ-correct with respect to all $r \in \mathsf{sn}(P)$ sessions.*

We can verify that the feasible consistency map defined previously for process *OrderTransaction2* is not φ-correct. For instance, the trace in Figure 9 is a witness that after the execution of trace s, if a failure occurs, it will not be possible to observe the compensation *unpacked* within session r_2.

In the following, we will present the conditions that should be preserved to ensure that the composition of correct compensable processes is also a correct process. The first condition describes a notion of independence, which is necessary for parallel composition.

Definition 16. *Two processes P and Q are* independent *if $\mathsf{sn}(P) \cap \mathsf{sn}(Q) = \emptyset$, $\mathsf{inp}(P) \cap \mathsf{fn}(P) \cap \mathsf{inp}(Q) \cap \mathsf{fn}(Q) = \emptyset$ and $\mathsf{out}(P) \cap \mathsf{fn}(P) \cap \mathsf{out}(Q) \cap \mathsf{fn}(Q) = \emptyset$.*

Clearly, if some of the above conditions were false, it could not be assured that each trace of $P \mid Q$ would be φ-correct, for a given correctness mapping φ. For instance, non-empty intersection of inputs could raise undesirable internal communications compromising the compensating trace of P or Q. With this definition we can state the following property.

Proposition 1. *Let P and Q be independent compensable processes such that $P \mid Q$ is a well formed process. Then, the parallel composition $P \mid Q$ is φ-correct if both P and Q are φ-correct.*

Proof. Let $r \in \mathsf{sn}(P \mid Q)$ and $P \mid Q \xrightarrow{s}\xrightarrow{\alpha}\xrightarrow{s'} R$, with R a passive process, $\alpha = (r, t, \mathsf{id})$ or $\alpha = (r, \tau(t), \mathsf{id})$, and t a failure unit of r. Since P and Q are independent, we know that $r \notin \mathsf{sn}(P) \cap \mathsf{sn}(Q)$. Let us assume without loss of generality that $r \in \mathsf{sn}(P)$ and let $s_1.\alpha.s'_1 \in L(P)$ be the underlying trace of P within $s.\alpha.s'$. Since P and Q are independent, we know that $\mathsf{com}(s_1.\alpha.s'_1, r) = \mathsf{com}(s.\alpha.s', r)$. Moreover, since P is φ-correct, we know that it exists a trace $s^*_1 \prec s'_1$ such that $\mathsf{com}(s^*_1, r) = \beta_1...\beta_n \in \oplus_{i=1}^{m} \varphi(\alpha_i)$, with $\mathsf{com}(s_1, r) = \alpha_1...\alpha_m$. Then, consider $s^* \prec s'$ such that s^*_1 is the underlying trace of P within s^*. Since P and Q are independent, s^* is such that $\mathsf{com}(s^*, r) = \mathsf{com}(s^*_1, r) = \beta_1...\beta_n \in \oplus_{i=1}^{m} \varphi(\alpha_i)$, with $\mathsf{com}(s, r) = \mathsf{com}(s_1, r) = \alpha_1...\alpha_m$. Therefore, the thesis holds.

Proposition 2. *Let P and Q be independent compensable processes such that $t[P, Q]_r$ is a well formed process. Then, $t[P, Q]_r$ is φ-correct if both P and Q are φ-correct.*

$$\begin{aligned} C_\varphi[\![\bullet]\!] \; ::= \; & \bullet \mid C_\varphi[\![\bullet]\!]|P \mid P|C_\varphi[\![\bullet]\!] \mid \langle C_\varphi[\![\bullet]\!]\rangle_s \mid t\,[C_\varphi[\![\bullet]\!], P]_r \mid \\ & (\nu\, x)\, C_\varphi[\![\bullet]\!] \text{ if } x \notin \mathrm{dom}(\varphi) \cup \mathrm{img}(\varphi) \\ D_\varphi[\![\bullet,\bullet]\!] \; ::= \; & C^1_\varphi[\![\bullet]\!]|C^2_\varphi[\![\bullet]\!] \end{aligned}$$

Fig. 10. φ-safe contexts

Proof. Let $r' \in \mathsf{sn}(P \mid Q) \cup \{r\}$ and $t[P,Q]_r \xrightarrow{s}\xrightarrow{\alpha} S \xrightarrow{s'} R$, with R a passive process, $\alpha = (r, p, \mathsf{id})$ or $\alpha = (r, \tau(p), \mathsf{id})$, and p a failure unit of r'. (1) If $r = r'$, since $t[P,Q]_r$ is a well formed process, $r \notin \mathsf{sn}(P \mid Q)$ and p is a failure unit of session r, *i.e.*, $p = t$. Moreover, it can be easily proved by induction that, if $r \notin \mathsf{sn}(P)$, then $r \notin \mathsf{sn}(s)$, for all $s \in L(P)$. In particular, $\mathsf{com}(s,r) = \epsilon$ and, therefore, the thesis holds choosing $s^* = \epsilon$ accordingly to Definition 14. (2) We have three cases, (2.1) t does not occur in s or s', (2.2) t occurs in s' or (2.3) t occurs in s. (2.1) If t does not occur in s or s', then $r' \in \mathsf{sn}(P)$ and p is a failure unit of r'. Thus, since s is a trace of P and P is φ-correct, the thesis holds. (2.2) If t occurs in s', then t does not occur in s and $r' \in \mathsf{sn}(P)$ with p a failure unit of r'. Since t has occurred within s', we have $t[P,Q]_r \xrightarrow{s}\xrightarrow{\alpha} S \xrightarrow{s'_1} t[P',Q]_r \xrightarrow{\alpha'} \mathsf{extr}(P') \mid \langle Q\rangle_r \xrightarrow{s'_2} R$, with $s' = s'_1.\alpha'.s'_2$ and either $\alpha' = (r,t,\mathsf{id})$ or $\alpha' = (r,\tau(t),\mathsf{id})$. As in the previous case, s is a trace of P and let s'' be the underlying trace of P within $s'_1.\alpha'.s'_2$. By Definition 14 and because P is φ-correct, there is s^* such that $s^* \prec s''$ and $\mathsf{com}(s^*, r') = \beta_1 ... \beta_n \in \oplus_{i=1}^m \varphi(\alpha_i)$, with $\mathsf{com}(s, r') = \alpha_1 ... \alpha_m$. Moreover, $\beta_1, ..., \beta_n \in \mathsf{nl}(k, t[P,Q]_r)$ and $\alpha_1 ... \alpha_m \in \mathsf{nl}(k-1, t[P,Q]_r)$, for some $k > 0$. Thus, $\mathsf{com}(s^*, r')$ occurs at an higher level and is part of the traces of compensations found within P'. Since these compensations are protected, they are not interrupted by t. Because P and Q are independent, $\mathsf{extr}(P')$ and $\langle Q\rangle_r$ are also independent since extr does not introduce names. Thus, there is $s^{**} \prec s'_1.\alpha'.s'_2$ such that $\mathsf{com}(s^{**}, r') = \mathsf{com}(s^*, r')$, and the thesis holds. (2.3) If t occurs in s, then $t[P,Q]_r \xrightarrow{s_1}\xrightarrow{\alpha'} \mathsf{extr}(P') \mid \langle Q\rangle_r \xrightarrow{s_2}\xrightarrow{\alpha} S \xrightarrow{s'} R$, with $s = s_1.\alpha'.s_2$ and either $\alpha' = (r,t,\mathsf{id})$ or $\alpha' = (r,\tau(t),\mathsf{id})$. Again, because P and Q are independent, $\mathsf{extr}(P')$ and $\langle Q\rangle_r$ are also independent since extr does not introduce names. Moreover, $\mathsf{extr}(P')$ is equivalent to trigger a set of failure units within P'. Thus, since P and $\langle Q\rangle$ are φ-correct and independent, by Proposition 1, the thesis holds.

Interactions can happen in different execution contexts. Since all our interactions are binary, we introduce double execution contexts, i.e., two execution contexts that can interact. The grammar in Figure 10 generates the *φ-safe execution contexts* for the correctness mapping φ.

Definition 17. *The grammar in Figure 10 inductively defines φ-safe contexts, denoted by $C_\varphi[\![\bullet]\!]$, and double φ-safe contexts, denoted by $D_\varphi[\![\bullet,\bullet]\!]$.*

The Propositions 3 and 4 shows that for safe contexts with respect to a correctness map φ, the composition of correct processes is also a correct process.

Proposition 3. *Let φ be a correctness mapping, P be φ-correct process and $C_\varphi[\bullet]$ be a safe context such that $C_\varphi[P]$ is a well formed process. If $C_\varphi[\mathbf{0}]$ is φ-correct and independent with respect to P, then $C_\varphi[P]$ is φ-correct.*

Proof. The proof is by induction on the context C_φ and by Propositions 1 and 2.

Proposition 4. *Let φ be a correctness mapping, P and Q be a φ-correct and independent processes , and $D_\varphi[\bullet, \bullet]$ be a safe double context such that $D_\varphi[P, Q]$ is a well formed process. If $D_\varphi[\mathbf{0}, \mathbf{0}]$ is φ-correct and independent with respect to process P and Q, then $D_\varphi[P, Q]$ is φ-correct.*

Proof. The proof is by induction on the context D_φ and by Propositions 1 and 3.

In some cases, we are interested in a more relaxed notion of composition. Consider the example *OrderTransaction3* and the previous correctness mapping φ. We can see that both process $Warehouse \mid Pack$ and $Bank \mid Charge$, within session r_2 and session r_1 respectively, are φ-correct. Also, they are independent processes. However, we cannot apply the previous results to prove that their composition is also φ-correct, because $Pack$ and $Charge$ are subtransactions of $Shop$, but $Warehouse$ and $Bank$ are not. So, the following results generalise the idea of composition in order to extend the correctness result to this kind of generalised composition.

Lemma 2. *Let φ be a correctness mapping, $P \mid Q$ be a φ-correct compensable process, and $C_\varphi[\bullet]$ be a φ-safe context such that $P \mid C_\varphi[Q]$ is a well formed process, $C_\varphi[0]$ is independent of P and $C_\varphi[\bullet]$ does not bind $x \in \mathsf{fn}(P)$. If $C_\varphi[\mathbf{0}]$ is φ-consistent, then $P \mid C_\varphi[Q]$ is φ-consistent.*

Proof. The proof is by induction on context $C_\varphi[\bullet]$.

Theorem 1. *Let φ be a correctness mapping, $P_1 \mid Q_1$ and $P_2 \mid Q_2$ be independent and φ-correct compensable processes, and $D_\varphi[\bullet, \bullet]$ be a safe double context such that $D_\varphi[P_1 \mid P_2, Q_1 \mid Q_2]$ is a well formed process. If $D_\varphi[\mathbf{0}, \mathbf{0}]$ is φ-correct, then $D_\varphi[P_1 \mid P_2, Q_1 \mid Q_2]$ is φ-correct.*

Proof. The proof is by induction on the context D_φ and by Proposition 1 and Lemma 3.

We are now able to interpret the notion self-healing systems [16], *i.e.* systems that can detect and recover from failures, as correct compensable processes. Such a system should perceive that is not operating correctly and make the necessary adjustments to restore itself to consistency. Thus, for this interpretation, it is necessary to express the behaviour the system should have in case of failure. Within our setting, this is done by a correctness mapping. We define self-healing with respect to a correctness mapping as follows.

Definition 18. *Let P be a process and φ a correctness mapping. P is* self-healing with respect to φ *if P is φ-correct.*

Moreover, a *self-healing composition* should be able to automatically detect that some service composition requirements are no longer satisfied by the implementation and react to requirement violations [16]. Our notion of correctness describes the idea of restoring the consistency of these kind of systems. In particular, our results provide a first attempt to build self-healing compositions from existing ones.

From the above results, if the programmer of a system ensures correctness of compensable transactions (or of a self-healing system), then by satisfying the conditions defined above, many kinds of compositions are also correct under our theory.

5 Related Work and Concluding Remarks

We have developed a notion of compensation correctness within a setting for analysing structured compensable transactions. Moreover, in this setting, we discuss correctness criteria for compensable process compositions.

There are other approaches for reasoning about the correctness of compensations. Korth *et al.* [10] have defined compensation soundness in terms of the properties that compensations have to guarantee. The correctness notions are based on the existence of state and state equivalence. Nevertheless, the authors do not provide a formal framework for specifying their definitions. Caires *et al.* [6] proposed a formal framework to reason about correctness of compensating transactions, which is also based on the existence of an appropriate notion of equivalence on system states. Even though their approach supports distributed transactions, the compensable processes do not interact. A different approach is given by Butler *et al.* [5], that proposes a notion of compensation soundness and a stateless equivalence notion. They define a cancellation semantics based on a cancellation function that analysis traces, extracting forward and compensation actions from process traces.

Due to the stateless assumption of the service oriented paradigm, our correctness criteria are not based on the existence of an equivalence notion on states. Hence, the proposed criteria could be used on other paradigms based on a minimal shared knowledge among the interacting parts.

Regarding future work, we plan to develop a type system to guarantee the properties needed to ensure correctness of compensable transactions. It would also be challenging to verify our results in an extended calculus with primitives for instantiation and definition of multiparty asynchronous sessions, taking in account input guarded replication. We plan to investigate the use of our correctness criteria and setting to other calculi with a recovery mechanism based on compensations [1,3,8,12,14].

Acknowledgements. We thank the reviewers for their comments. Cátia Vaz is partially supported by the Portuguese FCT, via SFRH/BD/45572/2008. Research supported by the Project FET-GC II IST-2005-16004 Sensoria.

References

1. Bocchi, L., Laneve, C., Zavattaro, G.: A calculus for long-running transactions. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 124–138. Springer, Heidelberg (2003)
2. Bruni, R., Butler, M.J., Ferreira, C., Hoare, C.A.R., Melgratti, H.C., Montanari, U.: Comparing two approaches to compensable flow composition. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 383–397. Springer, Heidelberg (2005)
3. Bruni, R., Melgratti, H.C., Montanari, U.: Nested commits for mobile calculi: Extending join. In: IFIP TCS, pp. 563–576. Kluwer, Dordrecht (2004)
4. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Palsberg, J., Abadi, M. (eds.) POPL 2005, pp. 209–220. ACM, New York (2005)
5. Butler, M.J., Hoare, C.A.R., Ferreira, C.: A trace semantics for long-running transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) 25 Years Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)
6. Caires, L., Ferreira, C., Vieira, H.T.: A process calculus analysis of compensations. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 87–103. Springer, Heidelberg (2009)
7. Carbone, M., Honda, K., Yoshida, N.: Structured interactional exceptions for session types. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer, Heidelberg (2008)
8. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the interplay between fault handling and request-response service invocations. In: 8th International Conference on Application of Concurrency to System Design, pp. 190–199. IEEE Computer Society, Los Alamitos (2008)
9. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL, pp. 273–284. ACM, New York (2008)
10. Korth, H.F., Levy, E., Silberschatz, A.: A formal approach to recovery by compensating transactions. In: VLDB, pp. 95–106 (1990)
11. Lanese, I., Vaz, C., Ferreira, C.: On the expressive power of primitives for compensation handling. Technical report (2009)
12. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
13. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for ws-bpel. J. Log. Algebr. Program. 70(1), 96–118 (2007)
14. 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)
15. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I and II. Inf. Comput. 100(1), 1–77 (1992)
16. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: a research roadmap. Int. J. Cooperative Inf. Syst. 17(2), 223–255 (2008)
17. Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 201–215. Springer, Heidelberg (2009)

Small Specifications for Tree Update

Philippa Gardner and Mark Wheelhouse

Imperial College London
{pg,mjw03}@doc.ic.ac.uk

Abstract. O'Hearn, Reynolds and Yang introduced Separation Logic to provide modular reasoning about simple, mutable data structures in memory. They were able to construct small specifications of programs, by reasoning about the local parts of memory accessed by programs. Gardner, Calcagno and Zarfaty generalised this work, introducing Context Logic to reason about more complex data structures. In particular, they developed a formal, compositional specification of the Document Object Model, a W3C XML update library. Whilst keeping to the spirit of local reasoning, they were not able to retain small specifications. We introduce Segment Logic, which provides a more fine-grained analysis of the tree structure and yields small specifications. As well as being aesthetically pleasing, small specifications are important for reasoning about concurrent tree update.

1 Introduction

Separation Logic [14], introduced by O'Hearn, Reynolds and Yang, provides modular reasoning about mutable data structures in memory. The idea is to reason about the small, local parts of memory (the footprint) that are accessed by a program. The resulting modular reasoning has been used to notable success for verifying memory safety properties of large C-programs [1], and for reasoning about concurrent imperative programs [13]. Calcagno, Gardner and Zarfaty generalised this work to more complex data structures, such as those found on the Web, by introducing Context Logic for reasoning about arbitrary structured data update [3]. Their original work applied Context Logic reasoning to a simple tree update language. With Smith and Zarfaty, Gardner and Wheelhouse have since applied Context Logic reasoning to the W3C Document Object Model (DOM) [6], a library for in-place XML update [18].

Our goal is to design and formally specify a concurrent XML update language. Such a language will enable web applications to make the most of the dynamic nature of XML. For example, with Wikipedia, users currently copy articles on to their browsers, before updating and returning them to Wikipedia to be integrated with the main site. They cannot view Wikipedia (or a scientific database or information in the Cloud) as a shared XML memory store that can be concurrently updated by many clients, because methods for safely performing such operations are poorly understood. We almost have the technology to develop a safe, formally specified language for concurrent XML update, drawing on our experience experience with sequential DOM [6] and O'Hearn's Concurrent Separation Logic [13]. However, we are missing one ingredient.

In our DOM work, we were not able to provide small specifications for all our DOM programs. In particular, our reasoning for the basic move commands, such as DOM's

C. Laneve and J. Su (Eds.): WS-FM 2009, LNCS 6194, pp. 178–195, 2010.

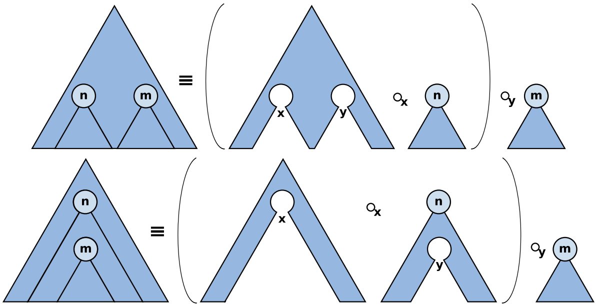


Fig. 1. Splitting up the Working Tree using Multi-holed Contexts

appendChild, used axioms which required a substantial over-approximation of the footprint. Whilst this over-approximation was acceptable for reasoning about sequential programs, it is a serious limitation when reasoning about concurrent programs. In this paper, we solve this limitation, by introducing Segment Logic to provide a more fine-grained analysis of structured data update in general, and tree update in particular. We provide small axioms for all the basic commands of a simple tree update language; it is straightforward to extend our ideas to DOM [6]. Although this paper focuses on a sequential tree update language, we believe it provides the technology necessary for our future work on reasoning about concurrent tree update.

To motivate Segment Logic, consider the DOM command appendChild(n, m) which moves the tree with top node identified by m to be the last child of the tree identified by n. **Fig. 1** indicates how the working tree splits in the two cases where appendChild(n, m) does not fault: it succeeds when n and m are in different parts of the tree and when m is under n; it faults when m is above n. The axiom for appendChild(n, m) using multi-holed Context Logic [2] is [1]:

$$\{(C \circ_\alpha n[c_1]) \circ_\beta m[\mathrm{tree}(c_2)]\}$$
$$\texttt{appendChild}(n, m)$$
$$\{(C \circ_\alpha n[c_1 \otimes m[\mathrm{tree}(c_2)]]) \circ_\beta \varnothing\}$$

The precondition specifies that the working tree can be split into a subtree with top node m, and a tree context with hole variable β ($\boldsymbol{y}$ in **Fig. 1**) satisfying the separating application formula $C \circ_\alpha n[c_1]$. This formula states that the context can be further split into a subcontext with top node n and an unspecified context with hole α ($\boldsymbol{x}$ in **Fig. 1**) given by context variable C. The postcondition states that the tree at m is moved to be the last child of n and is replaced by the empty tree. The surrounding context, denoted by variable C, remains the same.

The problem with this appendChild axiom is that it is not small. The precondition is not the intuitive footprint. The only part of the tree that appendChild(n, m) requires is the tree at m which is being moved, and the tree or context with top node n (actually

[1] In [6], the axiom for appendChild is given using single-holed Context Logic. The multi-holed Context Logic axiom is simpler, but still not suitable for concurrent reasoning.

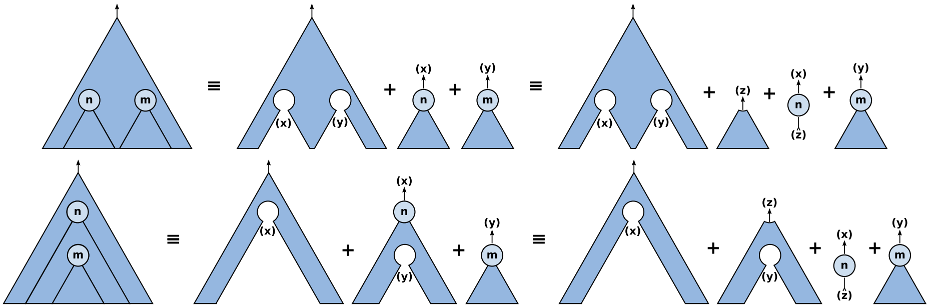


Fig. 2. Splitting up the Working Tree using Tree Segments

node n is enough) whose children are being extended by m. However, our precondition does not just use m and n. It also requires the surrounding context denoted by C. It is possible to put additional constraints on C to insist that the context is minimal. But this is not the point. We need a finer way of analysing the tree in order to capture the intuitive footprint of the command.

Instead of basing our reasoning on multi-holed tree contexts and application, we base our reasoning on *tree segments*. With multi-holed contexts, the working tree is split into a context and subtrees which have lost the information about where they originated from; the application function determines which holes get filled. With segments, the working tree is split into tree segments which still 'know' how to join back together again. As well as unique hole labels, tree segments have unique hole addresses which determine which holes the segments fill. For example, consider **Fig. 2**. In both cases, the working tree is split into a bunch of tree segments. The hole labels (in the holes) and the hole addresses (on the arrows) determine how the tree segments join back up to form the original tree. Notice that hole labels and addresses have brackets around them, denoting that they are bound. In the syntax, we will use a hiding operator $(\boldsymbol{x})$, analogous to the restriction operator of Milner's π-calculus [12].

Moreover, consider the right-hand equalities of **Fig. 2**. In both cases, the tree segment with top node identified by n has been split into just the node n at the same address and fresh hole label z, plus another tree segment at address z which contains the children of node n. We shall see that the node n and the tree with top node m are all that is required to provide the small axiom for `appendChild`. **Fig. 2** thus indicates how we can uniformly separate the minimal data required in order to reason about appendChild. It is possible to take this separation to the extreme, by cutting up the tree structure into a collection of nodes, with the hole labels and addresses showing how the nodes are joined together (a spaghetti of wires analogous to a heap representation). However, this is not how we use the hole information. We only cut up the tree in a minimal way in order to provide the right segment about which to reason.

We introduce Segment Logic for reasoning about our tree segments. It is like Context Logic in that it reasons directly about high-level trees. It is like Separation Logic in that it uses a commutative separating conjunction $*$, rather than the non-commutative separating application of Context Logic. Using Segment Logic, the small axiom for `appendChild`(n, m) is:

$$\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)]\}$$
$$\texttt{appendChild}(n, m)$$
$$\{\alpha{\leftarrow}n[\gamma \otimes m[\mathrm{tree}(c)]] * \beta{\leftarrow}\varnothing_{\mathrm{T}}\}$$

The precondition specifies two tree segments: a node n at address variable α ($\boldsymbol{x}$ in **Fig. 2**) and a complete tree whose top node is m at address β ($\boldsymbol{y}$ in **Fig 2**). The postcondition states that the tree at m moves to be the last child of n and is replaced by the empty tree. The axiom is small, with the precondition capturing the intuitive footprint of `appendChild`(n, m). We can extend the axiom to larger tree segments using the normal separation frame rule, a rule for the hiding quantification, and the rule for logical consequence. In fact, instead of using the hiding quantifier as primitive, we use the basic revelation connective (and a revelation frame rule), the revelation magic wand (the revelation right adjoint), and the fresh label quantification (and a fresh variable elimination rule), inspired by the work of Gabbay and Pitts [5], and Cardelli and Gordon [4]. Interestingly, we shall see that these more primitive constructs are important for describing the weakest preconditions.

2 Tree Update Language

We study a simple, but expressive, high-level tree update language for manipulating finite, ordered, unranked trees, with unique node identifiers for specifying the locations of updates as in DOM. Our tree structures are left intentionally simple. It is straightforward to incorporate (and reason about) additional data such as text data and attributes (see [6]). To simplify our exposition, we work with multi-holed tree contexts [2]. Throughout this paper we use countably infinite and disjoint sets $\mathrm{I}\textsc{d} = \{\boldsymbol{m}, \boldsymbol{n}, ...\}$ for location names and $\mathrm{X} = \{\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, ...\}$ for hole identifiers.

Definition 1 (Tree Contexts). *Multi-holed tree contexts $\boldsymbol{c} \in \mathrm{C}_{\mathrm{I}\textsc{d},\mathrm{X}}$ are defined by:*

tree context	$\boldsymbol{c} ::=$	$\varnothing_{\mathrm{C}}$	*empty tree context*
		$\boldsymbol{x}$	*hole identifier $\boldsymbol{x}$ used as a hole label*
		$\boldsymbol{n}[\boldsymbol{c}]$	*tree context with top node* n
		$\boldsymbol{c} \otimes \boldsymbol{c}$	*composition of tree contexts*

with the restriction that each hole identifier, $\boldsymbol{x} \in \mathrm{X}$, and location name, $\boldsymbol{n} \in \mathrm{I}\textsc{d}$, occur at most once in a tree context c, and subject to an equivalence $c_1 \equiv c_2$ stating that $\otimes$ is associative with identity $\varnothing_{\mathrm{C}}$. The set of hole identifiers that occur in tree context $\boldsymbol{c}$ is denoted $free(\boldsymbol{c})$. A tree context with no context holes (a complete tree) is denoted $\boldsymbol{t}$.

Definition 2 (Context Application). *Context Application is defined as a set of partial functions $ap_{\boldsymbol{x}} : \mathrm{C}_{\mathrm{I}\textsc{d},\mathrm{X}} \times \mathrm{C}_{\mathrm{I}\textsc{d},\mathrm{X}} \rightharpoonup \mathrm{C}_{\mathrm{I}\textsc{d},\mathrm{X}}$ indexed by hole labels $\boldsymbol{x}$:*

$$ap_{\boldsymbol{x}}(\boldsymbol{c}_1, \boldsymbol{c}_2) = \begin{cases} \boldsymbol{c}_1[\boldsymbol{c}_2/\boldsymbol{x}] & \text{if } \boldsymbol{x} \in free(\boldsymbol{c}_1) \text{ and } free(\boldsymbol{c}_1) \cap free(\boldsymbol{c}_2) = \{\} \\ \text{undefined otherwise} \end{cases}$$

We abbreviate $ap_{\mathbf{x}}(\mathbf{c}_1, \mathbf{c}_2)$ by $\mathbf{c}_1 \circ_{\mathbf{x}} \mathbf{c}_2$. We often omit the $\varnothing_{\mathrm{C}}$ leaves from a tree context to make it more readable, writing $\mathbf{n}[\mathbf{m} \otimes \mathbf{p}]$ instead of $\mathbf{n}[\mathbf{m}[\varnothing_{\mathrm{C}}] \otimes \mathbf{p}[\varnothing_{\mathrm{C}}]]$.

Our update language is a high-level, stateful, sequential, imperative language, based on variable assignment and update commands as in DOM. The program state is made up of two components: the working tree which contains all of the nodes we will be manipulating with our programs; and a high-level variable store containing variables for node identifiers.

Definition 3 (Variable Store). *The variable store $\sigma \in \Sigma$ is a finite partial function*

$$\sigma : Var_{\mathrm{ID}} \rightharpoonup_{fin} \mathrm{ID} \cup \{\mathbf{null}\}$$

*mapping location name variables $Var_{\mathrm{ID}} = \{\mathtt{m}, \mathtt{n}, ...\}$ to location names or **null**. We write $\sigma[\mathtt{n} \mapsto \mathbf{n}]$ for the variable store σ overwritten with $\sigma(\mathtt{n}) = \mathbf{n}$.*

To specify location name values, our language uses simple expressions. Location names are specified either with location name variables or the constant ***null***; we forbid direct reference to constant location names other than ***null***. We also require simple Boolean expressions for conditional tests in our language.

Definition 4 (Expressions). *Location name expressions $N \in Exp_{\mathrm{ID}}$ and Boolean expressions $B \in Exp_{\mathbb{B}}$ are defined by:*

$$\begin{array}{ll} N ::= \mathtt{n} \mid \boldsymbol{null} & \qquad \mathtt{n} \in Var_{\mathrm{ID}} \\ B ::= N = N \mid \mathbf{false} \mid B \Rightarrow B & \end{array}$$

The valuation of an expression E in a store σ is written $[\![E]\!]\sigma$ and has the obvious semantics. The classical Boolean connectives **true**, ¬, ∧ and ∨ are derivable.

DOM commands tend to update whole trees although, for example, the DOM commands `getNodeName` and `createNode` manipulate single nodes. Since our reasoning analyses tree segments, we explore a language for manipulating tree segments with primitive commands that update either a single node $\mathbf{n}$ or the whole subtree beneath some node $\mathbf{n}$; the commands for updating whole trees are derivable (Example 1).

Definition 5 (Tree Update Language). *The commands of the tree update language consist of the node update commands $\mathbb{C}_{nodeUp}$, the tree update commands $\mathbb{C}_{treeUp}$ and the standard skip, assignment, local, sequencing, if-then-else and while-do commands:*

$\mathbb{C}_{nodeUp}$ ::=	n′ := `getUp`(n)	*get parent of node n and record it in* n′
	n′ := `getLeft`(n)	*get previous sibling of node n*
	n′ := `getRight`(n)	*get next sibling of node n*
	n′ := `getFirst`(n)	*get first child of node n*
	n′ := `getLast`(n)	*get last child of node n*
	`insertNodeAbove`(n)	*insert a new node above node n*
	`deleteNode`(n)	*delete node n*
	`moveNodeAbove`(n, m)	*move node m above node n*
	`moveNodeLeft`(n, m)	*move node m to the left of node n*
	`moveNodeRight`(n, m)	*move node m to the right of node n*
	`prependNode`(n, m)	*prepend node m to children of node n*
	`appendNode`(n, m)	*append node m to children of node n*

```
ℂtreeUp ::= deleteSubtree(n)     delete subtree (subforest) beneath node n
            moveSubLeft(n, m)    move children of node m to the left of node n
            moveSubRight(n, m)   move children of node m to the right of node n
            prependSub(n, m)     prepend children of node m to children of node n
            appendSub(n, m)      append children of node m to children of node n
```

The set of free variables of a command $\mathbb{C}$ *is denoted free(*$\mathbb{C}$*), and the set of variables modified by* $\mathbb{C}$ *is denoted mod(*$\mathbb{C}$*).*

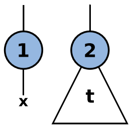
The behavior of these commands should be self-explanatory. The node update commands consist of get commands that return a neighboring node in the tree, a node insertion command that puts a fresh node into the tree above node n (the other node insertion commands are derivable), a delete command that removes a node from the tree, and node move commands that take a node out of the tree and put it in a new position. These node move commands leave the children of the moved node m as children of m's old parent. The tree update commands work on subtrees of an identified node. They consist of a delete command that removes an entire subtree from the tree, and subtree move commands that take a subtree out of the tree and put it in a new position.

These commands are sufficient to express a wide range of tree manipulation, as illustrated by the examples below. Our command set is not minimal: for example, we could derive the `deleteSubtree` command using a combination of get commands, node deletion and recursion. However, we believe the commands chosen provide a natural and expressive tree update language. We give the operational semantics in Section 3 using tree segments, rather than trees or tree contexts, as this simplifies the description of our reasoning in Section 5.

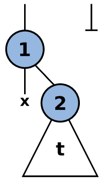
In [8], there are also commands for inserting whole trees. This is achieved by including tree shapes (trees without identifiers) in the variable store. Here we avoid such complications by omitting commands for tree insertion and copying. The reasoning presented here extends simply to these extra commands.

We now give example programs which will be used to illustrate our reasoning in Section 5. In all of these examples, $\sigma(n) = 1$ and $\sigma(m) = 2$.

Example 1 (Move). DOM has the command `appendChild`, whereas our language has `appendNode` and `appendSub`. We implement the standard `appendChild` as:

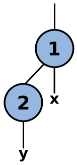


```
appendChild(n, m) ≜ local temp := null in {
                        insertNodeAbove(m) ;
                        temp := getUp(m) ;
                        appendNode(n, m) ;
                        appendSub(m, temp) ;
                        deleteNode(temp)
                    }
```

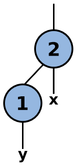


The diagrams illustrate the intuitive effect of the program on the working tree. For the program above not to fault, it requires the node $n = 1$ and the complete tree at node $m = 2$ (the left-hand diagram and the intuitive footprint). The complete tree at m ensures that m is not an ancestor of n. The result of the program is to move the tree at $m = 2$ to be under the node $n = 1$ (the right-hand diagram). Our reasoning captures the intuition illustrated by these diagrams.

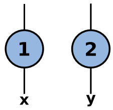
Example 2 (Simple Swap). Our node update commands enable us to define programs that act on arbitrary segments of the tree. For example, consider the program simple(n) which swaps a node n with its first child:



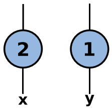
```
simple(n) ≜ local temp := null, first := null in {
              insertNodeAbove(n) ;
              temp := getUp(n) ;
              first := getFirst(n) ;
              moveNodeAbove(first, n) ;
              moveNodeAbove(temp, first) ;
              deleteNode(temp)
            }
```



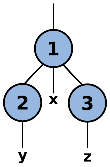
Example 3 (General Swap). The program nodeSwap(n, m) swaps the positions of arbitrary nodes n and m of a tree leaving their subtrees stationary:



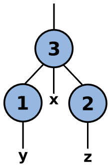
```
nodeSwap(n, m) ≜ local temp := null in {
                   insertNodeAbove(n) ;
                   temp := getUp(n) ;
                   moveNodeAbove(m, n) ;
                   moveNodeAbove(temp, m) ;
                   deleteNode(temp)
                 }
```



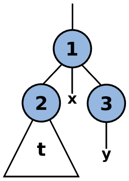
Example 4 (Node Rotate). The program nodeCycle(n) takes the node n, its first and last child, and rotates these nodes with n taking the place of first child, first child taking the place of last child, and last child taking the place of n:



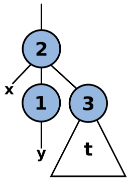
```
nodeCycle(n) ≜ local first := null, last := null in {
                 first := getFirst(n) ;
                 last := getLast(n) ;
                 nodeSwap(n, last) ;
                 nodeSwap(n, first)
               }
```



Example 5 (Combining Move and Node Swap). Consider a simple cyclic list of pictures used, for example, to view properties on an estate agent's web page. It can be implemented as a tree structure, with the root node of the tree containing the ID of the picture currently being displayed, the picture itself being stored beneath the last of its children (under node 3 below), and the other pictures in the list being stored beneath their ID nodes as the rest of the root node's children. The program queuePop(n) cycles the pictures so that the current picture moves to the back of the list and the next picture is displayed. Notice the use of appendChild which includes the complete tree $\boldsymbol{t}$ in the footprint.



```
queuePop(n) ≜ local next := null, info := null in {
                  next := getFirst(n) ;
                  info := getLast(n) ;
                  nodeSwap(n, info) ;
                  nodeSwap(next, info) ;
                  appendChild(next, info)
              }
```



3 Tree Segments

We are not able to provide a small specification of the appendChild command (and the appendSub command) using tree contexts. We are able to provide small specifications for all our tree update commands using tree segments.

Definition 6 (Tree Segments). *Tree segments $\mathbf{s} \in \mathrm{S}_{\mathrm{ID,X}}$ are defined by:*

$$\begin{array}{lrrl} \text{tree segment} & \mathbf{s} ::= & \varnothing_{\mathrm{S}} & \textit{empty tree segment} \\ & & \mathbf{x}{\leftarrow}\mathbf{c} & \textit{tree context } \mathbf{c} \textit{ addressed by hole identifier } \mathbf{x} \\ & & \mathbf{s}+\mathbf{s} & \textit{disjoint union} \\ & & (\mathbf{x})(\mathbf{s}) & \textit{hiding, hole identifier } \mathbf{x} \textit{ bound in tree segment } \mathbf{s} \end{array}$$

with the restriction that each hole identifier $\mathbf{x} \in \mathrm{X}$ occurs free at most once as a hole label and at most once as a hole address in tree segment $\mathbf{s}$, and each location name, $\mathbf{n} \in \mathrm{ID}$, occurs at most once in $\mathbf{s}$. The set $free(\mathbf{s})$ denotes the set of free hole identifiers in $\mathbf{s}$.

With tree contexts, we have the application $(1[\mathbf{x} \otimes 3]) \circ_{\mathbf{x}} 2 = 1[2 \otimes 3]$. The application $\circ_{\mathbf{x}}$ binds $\mathbf{x}$, and declares that hole $\mathbf{x}$ is filled by the argument 2. With tree segments, we have the equivalence $(\mathbf{x})(\mathbf{z}{\leftarrow}1[\mathbf{x} \otimes 3] + \mathbf{x}{\leftarrow}2) \equiv \mathbf{z}{\leftarrow}1[2 \otimes 3]$. In this case, it is the segment $\mathbf{x}{\leftarrow}2$ with address $\mathbf{x}$ that declares that 2 should go into the hole $\mathbf{x}$, and the hiding operator $(\mathbf{x})$ which binds $\mathbf{x}$ in the segment.

Definition 7 (Tree Segment Equivalence). *An equivalence relation $\equiv$ over tree segments is defined by the following axioms and the natural structural rules:*

$$\begin{array}{rcl} \mathbf{s} + \varnothing_{\mathrm{S}} & \equiv & \mathbf{s} \\ (\mathbf{x})(\varnothing_{\mathrm{S}}) & \equiv & \varnothing_{\mathrm{S}} \end{array} \qquad \begin{array}{rcl} \mathbf{s}_1 + \mathbf{s}_2 & \equiv & \mathbf{s}_2 + \mathbf{s}_1 \\ \mathbf{s}_1 + (\mathbf{s}_2 + \mathbf{s}_3) & \equiv & (\mathbf{s}_1 + \mathbf{s}_2) + \mathbf{s}_3 \end{array}$$

$$\begin{array}{rcll} (\mathbf{x})(\mathbf{y})(\mathbf{s}) & \equiv & (\mathbf{y})(\mathbf{x})(\mathbf{s}) & \\ (\mathbf{x})(\mathbf{s}) & \equiv & (\mathbf{y})(\mathbf{s}[\mathbf{y}/\mathbf{x}]) & \textit{if} \quad \mathbf{y} \notin free(\mathbf{s}) \\ (\mathbf{x})(\mathbf{y}{\leftarrow}\mathbf{c} + \mathbf{s}) & \equiv & \mathbf{y}{\leftarrow}\mathbf{c} + (\mathbf{x})(\mathbf{s}) & \textit{if} \quad \mathbf{x} \neq \mathbf{y} \textit{ and } \mathbf{x} \notin free(\mathbf{c}) \\ (\mathbf{x})(\mathbf{y}{\leftarrow}\mathbf{c}_1 + \mathbf{x}{\leftarrow}\mathbf{c}_2) & \equiv & \mathbf{y}{\leftarrow}(\mathbf{c}_1 \circ_{\mathbf{x}} \mathbf{c}_2) & \textit{if} \quad \mathbf{x} \in free(\mathbf{c}_1) \end{array}$$

Most of the axioms involving hiding follow from analogous axioms for the restriction operator of the π-calculus [12]. The last hiding axiom is specific to tree segments. It enables us to pull apart and compress segments, as illustrated in **Fig. 3**. A tree segment is in its compressed form if it cannot be further compressed using this last axiom. A tree segment is well-formed if and only if its compressed form is cycle free; that is, the

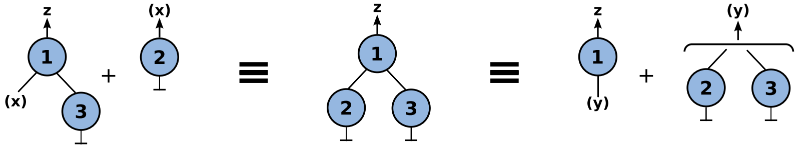


Fig. 3. Equivalent Tree Segments: (x) and (y) denote hidden hole labels and addresses

hole labels and hole addresses are disjoint in its compressed form. We only work with well-formed tree segments in this paper.

Fig. 3 demonstrates a graphical interpretation of segments. The left-hand side of **Fig. 3** will come as no surprise to those familiar with graphical process models: for example, Milner's work on process graphs [11]. Here, the hole identifiers describe the edges of the graph (wires). However, this is not the only use of hole identifiers. Consider the right-hand side of **Fig. 3**. Here, the hole identifier $\boldsymbol{y}$ is used to address multiple edges of a graph. More than this, consider the `appendChild` command in Example 1. The tree segment $z{\leftarrow}1[\boldsymbol{x}] + \boldsymbol{y}{\leftarrow}2[\boldsymbol{t}]$ updates to $z{\leftarrow}1[\boldsymbol{x} \otimes 2[\boldsymbol{t}]] + \boldsymbol{y}{\leftarrow}\varnothing_{\mathrm{C}}$: before update the segment $\boldsymbol{y}{\leftarrow}2[\boldsymbol{t}]$ states that a tree is at address $\boldsymbol{y}$; after update $\boldsymbol{y}{\leftarrow}\varnothing_{\mathrm{C}}$ states that the empty tree is at address $\boldsymbol{y}$. This example illustrates that edge arity is not necessarily preserved by update. Our hole identifiers should therefore not be regarded as describing graph edges. Instead, they describe tree fragments.

Notice that our language manipulates nodes and complete trees. It does not refer to hole identifiers in any way. However, the operational semantics is greatly simplified by using either tree contexts or tree segments. We choose tree segments, as this leads to a simpler interpretation of Hoare triples in Section 5.

Definition 8 (Operational Semantics). *We give the operational semantics for the basic commands of the tree update language in* **Fig. 4** *using an evaluation relation* $\rightsquigarrow$ *relating configuration triples* $\mathbb{C}, \sigma, \boldsymbol{s}$*, terminal states* $\sigma, \boldsymbol{s}$*, and faults, where* $\mathbb{C}$ *is a command,* σ *is a variable store and* $\boldsymbol{s}$ *is a tree segment. The set of variables of a command* $\mathbb{C}$ *is denoted free(*$\mathbb{C}$*) and is contained within the domain of* σ*, denoted dom(*σ*). We omit the standard cases for skip, assignment, local, sequencing, if-then-else and while-do.*

Our style of local Hoare reasoning about programs requires that the commands of our language be local. A command is local if it satisfies two properties, initially introduced in [10], known as the *safety-monotonicity* property and the *frame* property. The *safety-monotonicity* property specifies that, if a command is safe (does not fault) in a given state, then it is safe in a larger state. The *frame* property specifies that, if a command is safe in a given state, then any execution on a larger state can be tracked to an execution on the smaller state. The commands of our language presented here (and the DOM commands [6]) are local. For example, consider the behavior of $n' :=$ `getRight`(n). If the right sibling of n exists, then its identifier is stored at n'. If n is the last child of some parent node (meaning n can never obtain a right sibling via segment composition), then n' stores the value ***null***. However, if the node n is not present in the tree, or n has no right sibling or parent, then the command must fault in order to be local.

$$\frac{\sigma(n) = \boldsymbol{n} \quad \boldsymbol{s} \equiv (\boldsymbol{w},\boldsymbol{x},\boldsymbol{y},z)(\boldsymbol{s}' + \boldsymbol{x}{\leftarrow}\boldsymbol{m}[\boldsymbol{y} \otimes \boldsymbol{n}[\boldsymbol{w}] \otimes z])}{n' := \mathtt{getUp}(n), \sigma, \boldsymbol{s} \rightsquigarrow \sigma[n' \mapsto \boldsymbol{m}], \boldsymbol{s}}$$

$$\frac{\sigma(n) = \boldsymbol{n} \quad \boldsymbol{s} \equiv (\boldsymbol{x},\boldsymbol{y},z)(\boldsymbol{s}' + \boldsymbol{x}{\leftarrow}\boldsymbol{n}[\boldsymbol{y}] \otimes \boldsymbol{m}[z])}{n' := \mathtt{getRight}(n), \sigma, \boldsymbol{s} \rightsquigarrow \sigma[n' \mapsto \boldsymbol{m}], \boldsymbol{s}} \qquad \frac{\sigma(n) = \boldsymbol{n} \quad \boldsymbol{s} \equiv (\boldsymbol{x},\boldsymbol{y},z)(\boldsymbol{s}' + \boldsymbol{x}{\leftarrow}\boldsymbol{m}[z \otimes \boldsymbol{n}[\boldsymbol{y}]])}{n' := \mathtt{getRight}(n), \sigma, \boldsymbol{s} \rightsquigarrow \sigma[n' \mapsto \boldsymbol{null}], \boldsymbol{s}}$$

$$\frac{\sigma(n) = \boldsymbol{n} \quad \boldsymbol{s} \equiv (\boldsymbol{x},\boldsymbol{y},z)(\boldsymbol{s}' + \boldsymbol{x}{\leftarrow}\boldsymbol{n}[\boldsymbol{y} \otimes \boldsymbol{m}[z]])}{n' := \mathtt{getLast}(n), \sigma, \boldsymbol{s} \rightsquigarrow \sigma[n' \mapsto \boldsymbol{m}], \boldsymbol{s}} \qquad \frac{\sigma(n) = \boldsymbol{n} \quad \boldsymbol{s} \equiv (\boldsymbol{x})(\boldsymbol{s}' + \boldsymbol{x}{\leftarrow}\boldsymbol{n}[\varnothing_{\mathrm{C}}])}{n' := \mathtt{getLast}(n), \sigma, \boldsymbol{s} \rightsquigarrow \sigma[n' \mapsto \boldsymbol{null}], \boldsymbol{s}}$$

$$\frac{\begin{gathered}\sigma(n) = \boldsymbol{n} \quad \boldsymbol{s} \equiv (\boldsymbol{x},\boldsymbol{y})(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{n}[\boldsymbol{y}])\\ \boldsymbol{m} \text{ fresh id} \quad \boldsymbol{s}' \equiv (\boldsymbol{x},\boldsymbol{y})(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{m}[\boldsymbol{n}[\boldsymbol{y}]])\end{gathered}}{\mathtt{insertNodeAbove}(n), \sigma, \boldsymbol{s} \rightsquigarrow \sigma, \boldsymbol{s}'} \quad \frac{\begin{gathered}\sigma(n) = \boldsymbol{n} \quad \boldsymbol{s} \equiv (\boldsymbol{x},\boldsymbol{y})(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{n}[\boldsymbol{y}])\\ \boldsymbol{s}' \equiv (\boldsymbol{x},\boldsymbol{y})(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{y})\end{gathered}}{\mathtt{deleteNode}(n), \sigma, \boldsymbol{s} \rightsquigarrow \sigma, \boldsymbol{s}'} \quad \frac{\begin{gathered}\sigma(n) = \boldsymbol{n} \quad \boldsymbol{s} \equiv (\boldsymbol{x})(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{n}[\boldsymbol{t}])\\ \boldsymbol{s}' \equiv (\boldsymbol{x})(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{n}[\varnothing_{\mathrm{C}}])\end{gathered}}{\mathtt{deleteSubtree}(n), \sigma, \boldsymbol{s} \rightsquigarrow \sigma, \boldsymbol{s}'}$$

$$\frac{\begin{gathered}\sigma(n) = \boldsymbol{n} \quad \sigma(m) = \boldsymbol{m}\\ \boldsymbol{s} \equiv (\boldsymbol{w},\boldsymbol{x},\boldsymbol{y},z)(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{n}[z] + \boldsymbol{y}{\leftarrow}\boldsymbol{m}[\boldsymbol{w}])\\ \boldsymbol{s}' \equiv (\boldsymbol{w},\boldsymbol{x},\boldsymbol{y},z)(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{n}[z{\otimes}\boldsymbol{m}] + \boldsymbol{y}{\leftarrow}\boldsymbol{w})\end{gathered}}{\mathtt{appendNode}(n,m), \sigma, \boldsymbol{s} \rightsquigarrow \sigma, \boldsymbol{s}'} \quad \frac{\begin{gathered}\sigma(n) = \boldsymbol{n} \quad \sigma(m) = \boldsymbol{m}\\ \boldsymbol{s} \equiv (\boldsymbol{x},\boldsymbol{y},z)(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{n}[z] + \boldsymbol{y}{\leftarrow}\boldsymbol{m}[\boldsymbol{t}])\\ \boldsymbol{s}' \equiv (\boldsymbol{x},\boldsymbol{y},z)(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{n}[z{\otimes}\boldsymbol{t}] + \boldsymbol{y}{\leftarrow}\boldsymbol{m})\end{gathered}}{\mathtt{appendSub}(n,m), \sigma, \boldsymbol{s} \rightsquigarrow \sigma, \boldsymbol{s}'} \quad \frac{\begin{gathered}\sigma(n) = \boldsymbol{n} \quad \sigma(m) = \boldsymbol{m}\\ \boldsymbol{s} \equiv (\boldsymbol{w},\boldsymbol{x},\boldsymbol{y},z)(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{n}[z] + \boldsymbol{y}{\leftarrow}\boldsymbol{m}[\boldsymbol{w}])\\ \boldsymbol{s}' \equiv (\boldsymbol{w},\boldsymbol{x},\boldsymbol{y},z)(\boldsymbol{s}''+ \boldsymbol{x}{\leftarrow}\boldsymbol{m}[\boldsymbol{n}[z]] + \boldsymbol{y}{\leftarrow}\boldsymbol{w})\end{gathered}}{\mathtt{moveNodeAbove}(n,m), \sigma, \boldsymbol{s} \rightsquigarrow \sigma, \boldsymbol{s}'}$$

For get and move, only some of the cases are given; the other cases are analogous. Our commands fault when the program state does not satisfy any of the preconditions for that command.

Fig. 4. Operational Semantics for the Basic Tree Update Commands

4 Segment Logic

We introduce Segment Logic. First, we present the *logical environment* which contains logical variables for tree contexts, tree segments and hole identifiers. Location name variables have the standard dual role as both program variables and logical variables. They are declared in the variable store, but can be quantified like logical variables.

Definition 9 (Logical Environment). *An environment $e \in \mathrm{E}$ is a set of functions*

$$e : (LVar_{\mathrm{C}} \to \mathrm{C}_{\mathrm{ID},\mathrm{X}}) \times (LVar_{\mathrm{S}} \to \mathrm{S}_{\mathrm{ID},\mathrm{X}}) \times (LVar_{\mathrm{X}} \to \mathrm{X})$$

mapping tree context variables $LVar_{\mathrm{C}} = \{\mathsf{c}, ...\}$ to tree contexts, tree segment variables $LVar_{\mathrm{S}} = \{\mathsf{s}, ...\}$ to tree segments, and hole identifier variables $LVar_{\mathrm{X}} = \{\alpha, \beta, \gamma, \delta ...\}$ to hole identifiers. We write $e[\mathtt{lvar} \mapsto \boldsymbol{val}]$ for e overwritten with $e(\mathtt{lvar}) = \boldsymbol{val}$.

Segment Logic for trees consists of segment formulae and tree formulae. Just as in Separation Logic and Context Logic, segment formulae consist of classical formulae, structural formulae and specific formulae for describing the structure of data (in this case trees). For this paper, we have chosen to use tree formulae in the style of Ambient Logic [4], although we do not see a reason why P_{T} could not be first-order logic formulae for describing trees or even XDuce types [9]. Note that adapting this work to other data structures, such as sequences, just involves changing the tree formulae (or types) to sequence formulae (or types).

Definition 10 (Formulae). *The formulae of Segment Logic for trees consist of the segment formulae P_{S} and tree formulae P_{T} given by:*

$$\begin{array}{lll}
P_{\mathrm{S}}, Q_{\mathrm{S}} ::= P_{\mathrm{S}} \Rightarrow P_{\mathrm{S}} \mid \mathbf{false}_{\mathrm{S}} & P_{\mathrm{T}}, Q_{\mathrm{T}} ::= P_{\mathrm{T}} \Rightarrow P_{\mathrm{T}} \mid \mathbf{false}_{\mathrm{T}} & \textit{Classical} \\
\quad \mid \varnothing_{\mathrm{S}} \mid P_{\mathrm{S}} * P_{\mathrm{S}} \mid P_{\mathrm{S}} -\!\!* P_{\mathrm{S}} \mid \alpha \circledR P_{\mathrm{S}} \mid \alpha -\!\!\circledR P_{\mathrm{S}} & & \textit{Structural} \\
\quad \mid \exists \mathtt{var}.\, P_{\mathrm{S}} \mid \exists \mathtt{lvar}.\, P_{\mathrm{S}} \mid \text{И}\alpha.\, P_{\mathrm{S}} & \quad \mid \exists \mathtt{var}.\, P_{\mathrm{T}} \mid \exists \mathtt{lvar}.\, P_{\mathrm{T}} & \textit{Quantifiers} \\
\quad \mid \alpha \leftarrow P_{\mathrm{T}} & \quad \mid \varnothing_{\mathrm{T}} \mid \alpha \mid n[P_{\mathrm{T}}] \mid P_{\mathrm{T}} \otimes P_{\mathrm{T}} & \textit{Specific} \\
\quad \mid \mathsf{s} \mid \mathsf{B} & \quad \mid \mathsf{c} \mid \mathsf{B} \mid @_{\mathrm{T}}\alpha & \textit{Expression}
\end{array}$$

Let $free(P_{\mathrm{S}})$ and $free(P_{\mathrm{T}})$ denote the appropriate sets of free variables: α is free in $\alpha \circledR P_{\mathrm{S}}$ $\alpha -\!\!\circledR P_{\mathrm{S}}$, $\alpha \leftarrow P_{\mathrm{T}}$ and $@_{\mathrm{T}}\alpha$, bound in $\text{И}\alpha.\, P_{\mathrm{S}}$. $\mathtt{var}$ is a location name variable and $\mathtt{lvar}$ is a logical variable. The binding precedence, strongest first, is: $\otimes$, $\leftarrow$, $*$, $\circledR$, $-\!\!*$, $-\!\!\circledR$, $\Rightarrow$.

The separating connective $*$, its unit $\varnothing_{\mathrm{S}}$ and its right adjoint (the separating magic wand) $-\!\!*$, are structural formulae which are known from the Separation Logic literature: formula $P_{\mathrm{S}} * Q_{\mathrm{S}}$ describes a segment that can be separated into a segment satisfying P_{S} and a disjoint segment satisfying Q_{S}; formula $\varnothing_{\mathrm{S}}$ describes the empty segment; and formula $P_{\mathrm{S}} -\!\!* Q_{\mathrm{S}}$ describes a segment that, whenever it is joined to a segment satisfying P_{S}, results in a segment satisfying Q_{S}.

Before explaining the other structural formulae, we explain the specific segment formulae: the formula $\alpha \leftarrow P_{\mathrm{T}}$ describes a tree segment with hole address given by the value of variable α and tree context satisfying tree formula P_{T}. The tree formulae follow the style of reasoning given by Ambient Logic. For the specific tree formulae, we have $\varnothing_{\mathrm{T}}$ specifying the empty tree context, α specifying a hole label given by the value of variable α, $n[P_{\mathrm{T}}]$ specifying a tree context with top node denoted by node variable n and subtree satisfying P_{T}, and the composition formula $P_{\mathrm{T}} \otimes Q_{\mathrm{T}}$ describing a tree context which can be split into one context satisfying P_{T} and the other disjoint context satisfying Q_{T}. The tree expression formulae include the formula $@_{\mathrm{T}}\alpha$ which describes a tree that contains α free; the analogous formula for tree segments is derivable.

The other structural connectives are the revelation connective, $\circledR$, and its right adjoint, the revelation magic wand $-\!\!\circledR$. As far as we are aware, these connectives have not been used in the local reasoning setting before. Together with the freshness quantifier $\text{И}\alpha$, they have been used in the Ambient Logic [4], following the work of Pitts and Gabbay [5]. The freshness quantifier enables us to pick a completely new hole identifier. The formula $\alpha \circledR P_{\mathrm{S}}$ describes a segment with a top-level hiding binder given by the value of α such that, after the hiding is removed, the remaining segment satisfies P_{S}. Consider the tree segment $z \leftarrow 1[2 \otimes 3] \equiv (\boldsymbol{x})(z \leftarrow 1[\boldsymbol{x} \otimes 3] + \boldsymbol{x} \leftarrow 2)$. It satisfies the formula $\alpha \circledR (\beta \leftarrow 1[\alpha \otimes \mathbf{true}_{\mathrm{T}}] + \alpha \leftarrow 2)$, when $\alpha = \boldsymbol{x}, \beta = z$. The revelation connective $\alpha \circledR$ strips off the hiding binder, disconnecting the tree into the fragments $z \leftarrow 1[\boldsymbol{x} \otimes 3] + \boldsymbol{x} \leftarrow 2$ as illustrated in **Fig. 5**. By contrast, the formula $\alpha -\!\!\circledR P_{\mathrm{S}}$ describes a segment which satisfies P_{S} if it is extended with a hiding binder over the hole identifier stored in variable α. For example, the tree segment $z \leftarrow 1[\boldsymbol{x} \otimes 3] + \boldsymbol{x} \leftarrow 2$ satisfies the formula $\alpha -\!\!\circledR (\beta \leftarrow 1[2 \otimes 3])$ when $\alpha = \boldsymbol{x}$. The revelation magic wand $\alpha -\!\!\circledR$ adds the binder $(\boldsymbol{x})$ to the segment to obtain the tree segment $(\boldsymbol{x})(z \leftarrow 1[\boldsymbol{x} \otimes 3] + \boldsymbol{x} \leftarrow 2) \equiv z \leftarrow 1[2 \otimes 3]$, thus connecting the fragmented tree into the whole tree as illustrated in **Fig. 5**. Analogous to the separating magic wand, we shall see that the revelation magic wand is important for giving the weakest preconditions of commands.

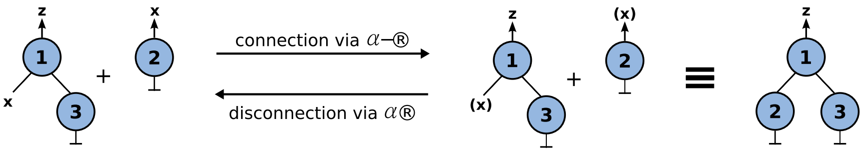


Fig. 5. Connection and Disconnection of Tree Segments

$$
\begin{array}{ll}
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \varnothing_{\mathrm{S}} & \Leftrightarrow \boldsymbol{s} \equiv \varnothing_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} P_{\mathrm{S}} * Q_{\mathrm{S}} & \Leftrightarrow \exists \boldsymbol{s}_1,\boldsymbol{s}_2.\, \boldsymbol{s} \equiv \boldsymbol{s}_1 + \boldsymbol{s}_2 \wedge e,\sigma,\boldsymbol{s}_1 \models_{\mathrm{S}} P_{\mathrm{S}} \wedge e,\sigma,\boldsymbol{s}_2 \models_{\mathrm{S}} Q_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} P_{\mathrm{S}} -\!\!* Q_{\mathrm{S}} & \Leftrightarrow \forall \boldsymbol{s}'.\, e,\sigma,\boldsymbol{s}' \models_{\mathrm{S}} P_{\mathrm{S}} \wedge (\boldsymbol{s}+\boldsymbol{s}')\!\downarrow \Rightarrow e,\sigma,\boldsymbol{s}+\boldsymbol{s}' \models_{\mathrm{S}} Q_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \alpha \circledR P_{\mathrm{S}} & \Leftrightarrow \exists \boldsymbol{x},\boldsymbol{s}'.\, e(\alpha) = \boldsymbol{x} \wedge \boldsymbol{s} \equiv (\boldsymbol{x})(\boldsymbol{s}') \wedge e,\sigma,\boldsymbol{s}' \models_{\mathrm{S}} P_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \alpha \multimap\!\!\circledR\, P_{\mathrm{S}} & \Leftrightarrow \exists \boldsymbol{x},\boldsymbol{s}'.\, e(\alpha) = \boldsymbol{x} \wedge \boldsymbol{s}' \equiv (\boldsymbol{x})(\boldsymbol{s}) \wedge e,\sigma,\boldsymbol{s}' \models_{\mathrm{S}} P_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \text{И}\alpha.\, P_{\mathrm{S}} & \Leftrightarrow \exists \boldsymbol{x}.\, \boldsymbol{x}\#e,\boldsymbol{s} \wedge e[\alpha \mapsto \boldsymbol{x}],\sigma,\boldsymbol{s} \models_{\mathrm{S}} P_{\mathrm{S}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \boldsymbol{\alpha}\leftarrow P_{\mathrm{T}} & \Leftrightarrow \exists \boldsymbol{c},\boldsymbol{x}.\, e(\alpha) = \boldsymbol{x} \wedge \boldsymbol{s} \equiv \boldsymbol{x}\leftarrow\boldsymbol{c} \wedge e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} P_{\mathrm{T}} \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} \mathsf{s} & \Leftrightarrow \boldsymbol{s} \equiv e(\mathsf{s}) \\
e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} B & \Leftrightarrow [\![B]\!]\sigma = \mathbf{true}
\end{array}
$$

$$
\begin{array}{ll}
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} \varnothing_{\mathrm{T}} & \Leftrightarrow \boldsymbol{c} \equiv \varnothing_{\mathrm{C}} \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} \alpha & \Leftrightarrow \boldsymbol{c} \equiv e(\alpha) \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} \boldsymbol{n}[P_{\mathrm{T}}] & \Leftrightarrow \exists \boldsymbol{c}_1.\, \boldsymbol{c} \equiv \sigma(\boldsymbol{n})[\boldsymbol{c}_1] \\
 & \quad \wedge\, e,\sigma,\boldsymbol{c}_1 \models_{\mathrm{T}} P_{\mathrm{T}} \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} P_{\mathrm{T}} \otimes Q_{\mathrm{T}} & \Leftrightarrow \exists \boldsymbol{c}_1,\boldsymbol{c}_2.\, \boldsymbol{c} \equiv \boldsymbol{c}_1 \otimes \boldsymbol{c}_2 \\
 & \quad \wedge\, e,\sigma,\boldsymbol{c}_1 \models_{\mathrm{T}} P_{\mathrm{T}} \\
 & \quad \wedge\, e,\sigma,\boldsymbol{c}_2 \models_{\mathrm{T}} Q_{\mathrm{T}} \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} \mathsf{c} & \Leftrightarrow \boldsymbol{c} \equiv e(\mathsf{c}) \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} B & \Leftrightarrow [\![B]\!]\sigma = \mathbf{true} \\
e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} @_{\mathrm{T}}\alpha & \Leftrightarrow e(\alpha) \in \mathit{free}(\boldsymbol{c})
\end{array}
$$

$(\boldsymbol{s})\!\downarrow$ denotes that $\boldsymbol{s}$ is well formed. $\boldsymbol{x}\#\boldsymbol{s}$ denotes that $\boldsymbol{x}$ is fresh with respect to $\boldsymbol{s}$.
We omit the standard semantics for $P \Rightarrow Q$, **false** and $\exists v.\, P$.

Fig. 6. Satisfaction Relations of Segment Logic for Trees

Definition 11 (Satisfaction Relation). *Given a logical environment e and a variable store σ, the semantics of Segment Logic is given in* **Fig. 6** *by two satisfaction relations $e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} P_{\mathrm{S}}$ and $e,\sigma,\boldsymbol{c} \models_{\mathrm{T}} P_{\mathrm{T}}$ defined on tree segments and tree contexts respectively.*

Definition 12 (Derived Formulae). *The standard classical logic connectives are derived from* **false** *and $\Rightarrow$ as usual, and the following useful formulae are defined:*

$$
\begin{array}{rlrl}
\mathit{tree}(P_{\mathrm{T}}) & \triangleq P_{\mathrm{T}} \wedge \neg\exists\alpha.\, @_{\mathrm{T}}\alpha & @_{\mathrm{S}}\alpha & \triangleq \mathsf{H}\beta.\, (\mathbf{true}_{\mathrm{S}} * (\alpha\leftarrow\beta \vee \beta\leftarrow\alpha)) \\
\mathsf{n} & \triangleq \mathsf{n}[\varnothing_{\mathrm{T}}] & \Diamond P_{\mathrm{S}} & \triangleq \mathbf{true}_{\mathrm{S}} * P_{\mathrm{S}} \\
\circ[P_{\mathrm{T}}] & \triangleq \exists \mathsf{m}.\, \mathsf{m}[P_{\mathrm{T}}] & \mathsf{H}\alpha.\, P_{\mathrm{S}} & \triangleq \text{И}\alpha.\, \alpha \circledR P_{\mathrm{S}}
\end{array}
$$

Formula $\mathit{tree}(P_{\mathrm{T}})$ describes a complete tree satisfying P_{T}. Formula n describes a leaf node identified by n. Formula $\circ[P_{\mathrm{T}}]$ allows us to drop the identifier of a node. Formula $@_{\mathrm{S}}\alpha$ describes a tree segment that contains α free. Formula $\Diamond P_{\mathrm{S}}$ allows us to express that somewhere in the tree segment there is a segment satisfying P_{S}. Finally, formula $\mathsf{H}\alpha.\, P_{\mathrm{S}}$ provides the standard hiding quantification [4] allowing us to quantify over hidden (restricted) labels.

Example 6 (Segment Logic Examples).

(a) The segment formula $\alpha\leftarrow\mathsf{n}[\gamma] * \beta\leftarrow\mathsf{m}[\delta]$ describes a tree segment consisting of a node n with address α and context hole γ, and node m with address β and context hole δ. The variables n and m cannot denote the same node identifier: similarly, α, β cannot denote the same hole identifier; neither can γ, δ.

(b) The segment formula $\alpha\leftarrow\mathsf{n}[\gamma] * \beta\leftarrow\mathsf{m}[\mathrm{tree}(\mathsf{c})]$ describes a tree segment consisting of a single node n at address α and a complete tree (a tree with no holes) with top node m at address β. This formula is the safety precondition for the small axiom of the appendSub(n, m) command. In particular, the formula states that m cannot be an ancestor of n, as n is disjoint from the tree c.

(c) The segment formula $\mathsf{H}\alpha,\beta.\,(\delta{\leftarrow}\mathtt{r}[\alpha\otimes\beta] * \alpha{\leftarrow}\mathtt{n}[\gamma] * \beta{\leftarrow}\mathtt{m}[\mathrm{tree}(\boldsymbol{c})])$ describes a tree segment consisting of a node $\mathtt{r}$ at address δ whose children are given by the holes α and β, and a tree segment satisfying the formula in Example (b). The labels α,β are under the hiding quantification, and hence denote fresh, unequal, identifiers. This formula is equivalent to $\delta{\leftarrow}\mathtt{r}[\mathtt{n}[\gamma]\otimes\mathtt{m}[\mathrm{tree}(\boldsymbol{c})]]$, which states that there is a node $\mathtt{r}$ at address δ whose children are $\mathtt{n}$ and $\mathtt{m}$.

(d) To specify our language, it is enough to work with the hiding quantification. However, to describe the weakest preconditions, we must use revelation. For example, the weakest precondition of the `deleteSubtree`($\mathtt{n}$) command is $\exists \mathtt{c}.\,\text{И}\alpha.\,\alpha\circledR(\,(\alpha{\leftarrow}\mathtt{n}[\varnothing_{\mathrm{T}}] \mathrel{-\!\!*} (\alpha \mathrel{-\!\circledR} P_{\mathrm{S}})) * \alpha{\leftarrow}\mathtt{n}[\mathrm{tree}(\mathtt{c})]\,)$. This formula describes a tree segment which can be separated into a complete tree, with top node $\mathtt{n}$ at a fresh address $\boldsymbol{x}$ denoted by α, and a segment $\boldsymbol{s}$ satisfying $(\alpha{\leftarrow}\mathtt{n}[\varnothing_{\mathrm{T}}]) \mathrel{-\!\!*} (\alpha \mathrel{-\!\circledR} P_{\mathrm{S}})$. The segment $\boldsymbol{s}$, when extended to $(\boldsymbol{x})(\boldsymbol{x}{\leftarrow}\mathtt{n}[\varnothing_{\mathrm{T}}] + \boldsymbol{s})$, satisfies P_{S}.

5 Local Hoare Reasoning

We use Segment Logic to provide local Hoare reasoning about programs written in the language given in Definition 5. First, we give a fault avoiding, partial correctness interpretation of local Hoare triples following [19]. Informally, $\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\}$ means that, when P_{S} holds for a tree segment $\boldsymbol{s}$, then command $\mathbb{C}$ does not fault when run on $\boldsymbol{s}$ and the result, if $\mathbb{C}$ terminates, satisfies Q_{S}.

Definition 13 (Local Hoare Triples). *Recall the evaluation relation* $\rightsquigarrow$ *relating configuration triples* $\mathbb{C},\sigma,\boldsymbol{s}$*, terminal states* $\sigma,\boldsymbol{s}$ *and faults in* **Fig. 4**. *The fault-avoiding partial correctness interpretation of local Hoare Triples is given below:*

$$\begin{array}{l}\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\} \;\Leftrightarrow\; \forall e,\sigma,\boldsymbol{s}.\ \mathit{free}(\mathbb{C}) \subseteq \mathit{dom}(\sigma) \wedge \mathit{free}(P_{\mathrm{S}}) \cup \mathit{free}(Q_{\mathrm{S}}) \subseteq \mathit{dom}(\sigma) \cup \mathit{dom}(e) \\ \qquad\qquad \wedge\, e,\sigma,\boldsymbol{s} \models_{\mathrm{S}} P_{\mathrm{S}} \Rightarrow \mathbb{C},\sigma,\boldsymbol{s} \not\rightsquigarrow \mathit{fault} \wedge \forall\sigma',\boldsymbol{s}'.\, \mathbb{C},\sigma,\boldsymbol{s} \rightsquigarrow \sigma',\boldsymbol{s}' \Rightarrow e,\sigma',\boldsymbol{s}' \models_{\mathrm{S}} Q_{\mathrm{S}}\end{array}$$

$$\begin{array}{rcl}
\{\alpha{\leftarrow}\mathtt{m}[\beta\otimes\mathtt{n}[\delta]\otimes\gamma] \wedge (\mathtt{n}'=\mathtt{n}_0)\} & \mathtt{n}' := \mathtt{getUp}(\mathtt{n}) & \{(\alpha{\leftarrow}\mathtt{m}[\beta\otimes\mathtt{n}[\delta]\otimes\gamma])_{[\mathtt{n}_0/\mathtt{n}']} \wedge (\mathtt{n}'=\mathtt{m})\} \\
\{\alpha{\leftarrow}\mathtt{n}[\delta]\otimes\mathtt{m}[\beta] \wedge (\mathtt{n}'=\mathtt{n}_0)\} & \mathtt{n}' := \mathtt{getRight}(\mathtt{n}) & \{(\alpha{\leftarrow}\mathtt{n}[\delta]\otimes\mathtt{m}[\beta])_{[\mathtt{n}_0/\mathtt{n}']} \wedge (\mathtt{n}'=\mathtt{m})\} \\
\{\alpha{\leftarrow}\mathtt{m}[\beta\otimes\mathtt{n}[\delta]] \wedge (\mathtt{n}'=\mathtt{n}_0)\} & \mathtt{n}' := \mathtt{getRight}(\mathtt{n}) & \{(\alpha{\leftarrow}\mathtt{m}[\beta\otimes\mathtt{n}[\delta]])_{[\mathtt{n}_0/\mathtt{n}']} \wedge (\mathtt{n}'=\boldsymbol{\mathit{null}})\} \\
\{\alpha{\leftarrow}\mathtt{n}[\delta\otimes\mathtt{m}[\beta]] \wedge (\mathtt{n}'=\mathtt{n}_0)\} & \mathtt{n}' := \mathtt{getLast}(\mathtt{n}) & \{(\alpha{\leftarrow}\mathtt{n}[\delta\otimes\mathtt{m}[\beta]])_{[\mathtt{n}_0/\mathtt{n}']} \wedge (\mathtt{n}'=\mathtt{m})\} \\
\{\alpha{\leftarrow}\mathtt{n}[\varnothing_{\mathrm{T}}] \wedge (\mathtt{n}'=\mathtt{n}_0)\} & \mathtt{n}' := \mathtt{getLast}(\mathtt{n}) & \{(\alpha{\leftarrow}\mathtt{n}[\varnothing_{\mathrm{T}}])_{[\mathtt{n}_0/\mathtt{n}']} \wedge (\mathtt{n}'=\boldsymbol{\mathit{null}})\} \\
\{\alpha{\leftarrow}\mathtt{n}[\beta]\} & \mathtt{insertNodeAbove}(\mathtt{n}) & \{\alpha{\leftarrow}\circ[\mathtt{n}[\beta]]\} \\
\{\alpha{\leftarrow}\mathtt{n}[\beta]\} & \mathtt{deleteNode}(\mathtt{n}) & \{\alpha{\leftarrow}\beta\} \\
\{\alpha{\leftarrow}\mathtt{n}[\mathrm{tree}(\mathtt{c})]\} & \mathtt{deleteSubtree}(\mathtt{n}) & \{\alpha{\leftarrow}\mathtt{n}[\varnothing_{\mathrm{T}}]\} \\
\{\alpha{\leftarrow}\mathtt{n}[\gamma] * \beta{\leftarrow}\mathtt{m}[\delta]\} & \mathtt{moveNodeAbove}(\mathtt{n},\mathtt{m}) & \{\alpha{\leftarrow}\mathtt{m}[\mathtt{n}[\gamma]] * \beta{\leftarrow}\delta\} \\
\{\alpha{\leftarrow}\mathtt{n}[\gamma] * \beta{\leftarrow}\mathtt{m}[\delta]\} & \mathtt{appendNode}(\mathtt{n},\mathtt{m}) & \{\alpha{\leftarrow}\mathtt{n}[\gamma\otimes\mathtt{m}[\varnothing_{\mathrm{T}}]] * \beta{\leftarrow}\delta\} \\
\{\alpha{\leftarrow}\mathtt{n}[\gamma] * \beta{\leftarrow}\mathtt{m}[\mathrm{tree}(\mathtt{c})]\} & \mathtt{appendSub}(\mathtt{n},\mathtt{m}) & \{\alpha{\leftarrow}\mathtt{n}[\gamma\otimes\mathrm{tree}(\mathtt{c})] * \beta{\leftarrow}\mathtt{m}[\varnothing_{\mathrm{T}}]\}
\end{array}$$

$\mathtt{n}'$ and $\mathtt{n}_0$ are distinct. The omitted commands have analogous or standard axioms.

Fig. 7. A Selection of the Small Axioms for the Basic Tree Update Commands

Definition 14 (Small Axioms). *The Small Axioms for the basic tree update commands from* **Fig. 4** *are given in* **Fig. 7**.

With Raza, Gardner has developed the formal definitions of footprints and small specifications for abstract local functions using Abstract Separation Logic [15]. It would be interesting to extend this abstract theory to the tree segments and reasoning studied here, and prove that the axioms really are small.

Definition 15 (Inference Rules). *The local reasoning inference rules include the standard Hoare Logic Rules for Sequencing, Consequence, Disjunction, Local Variable, If-Then-Else, While-Do, and the rules for Separation Frame, Revelation Frame, Auxiliary Variable Elimination and Fresh Label Elimination given by:*

Separation Frame:

$$\frac{\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\}}{\{P_{\mathrm{S}} * R_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}} * R_{\mathrm{S}}\}}\quad \mathit{mod}(\mathbb{C}) \cap \mathit{free}(R_{\mathrm{S}}) = \{\}$$

Revelation Frame:

$$\frac{\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\}}{\{\alpha \circledR P_{\mathrm{S}}\}\ \mathbb{C}\ \{\alpha \circledR Q_{\mathrm{S}}\}}$$

Auxiliary Variable Elimination:

$$\frac{\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\}}{\{\exists n.\, P_{\mathrm{S}}\}\ \mathbb{C}\ \{\exists n.\, Q_{\mathrm{S}}\}}\quad n \notin \mathit{free}(\mathbb{C})$$

Fresh Variable Elimination:

$$\frac{\{P_{\mathrm{S}}\}\ \mathbb{C}\ \{Q_{\mathrm{S}}\}}{\{\text{И}\alpha.\, P_{\mathrm{S}}\}\ \mathbb{C}\ \{\text{И}\alpha.\, Q_{\mathrm{S}}\}}$$

Recall that the set of variables modified by a command $\mathbb{C}$ is denoted mod($\mathbb{C}$).

The Separation Frame rule is standard from Separation Logic and allows us to extend the working tree with tree segments that are not used by the command. The Revelation Frame rule is similar. It allows us to add hiding binders to the working tree, since hole identifiers are not used by any of our commands. The Fresh Variable Elimination rule is analogous to the standard Auxiliary Variable Elimination rule. To see how we use these rules, consider again the small axiom of appendSub. We can extend this axiom using our inference rules to obtain:

$$\begin{gathered}\{\mathsf{H}\alpha,\beta.\,(\epsilon{\leftarrow}\mathtt{r}[\alpha \otimes \beta] * \alpha{\leftarrow}\mathtt{n}[\gamma] * \beta{\leftarrow}\mathtt{m}[\mathrm{tree}(\boldsymbol{c})])\}\\ \mathsf{appendSub}(\mathtt{n}, \mathtt{m})\\ \{\mathsf{H}\alpha,\beta.\,(\epsilon{\leftarrow}\mathtt{r}[\alpha \otimes \beta] * \alpha{\leftarrow}\mathtt{n}[\gamma \otimes \mathrm{tree}(\boldsymbol{c})] * \beta{\leftarrow}\mathtt{m}[\varnothing_{\mathrm{T}}])\}\end{gathered}$$

This triple can be simplified using the consequence rule following the discussion in Example 6(c). In this example, it is natural to use the derived hiding quantification. Following the discussion in Example 6(d) we can see that the primitive revelation connectives are, however, necessary to obtain the weakest preconditions, a selection of which are shown in **Fig. 8**.

Theorem 1 (Soundness and Completeness). *The small axioms and inference rules are sound. For straight line code, they are also complete.*

Proof Sketch. Soundness is straightforward to prove. Completeness for straight line code follows from the derivability of the the weakest preconditions (**Fig. 8**) from the small axioms (**Fig. 7**). See the full paper [7] for further details.

$$\begin{array}{rcl}
\{\exists m, n_0.\, \mathsf{H}\alpha,\beta,\gamma,\delta.\, \Diamond\alpha{\leftarrow}m[\beta\otimes n[\delta]\otimes\gamma] \wedge (n'=n_0)\wedge(\alpha,\beta,\gamma,\delta \mathbin{-\!\!\circledR} P_{\mathrm{S}[m/n']})\} & n' := \mathtt{getUp}(n) & \{P_{\mathrm{S}}\}\\
\left\{\exists m, n_0.\, \mathsf{H}\alpha,\beta,\delta.\ \begin{array}{r} \Diamond\alpha{\leftarrow}n[\delta]\otimes m[\beta]\wedge(n'=n_0)\wedge(\alpha,\beta,\delta \mathbin{-\!\!\circledR} P_{\mathrm{S}[m/n']}) \\ \vee\ \Diamond\alpha{\leftarrow}m[\beta\otimes n[\delta]]\wedge(n'=n_0)\wedge(\alpha,\beta,\delta\mathbin{-\!\!\circledR} P_{\mathrm{S}[\boldsymbol{null}/n']})\end{array}\right\} & n' := \mathtt{getRight}(n) & \{P_{\mathrm{S}}\}\\
\left\{\exists m, n_0.\, \mathsf{H}\alpha,\beta,\delta.\ \begin{array}{r} \Diamond\alpha{\leftarrow}n[\delta\otimes m[\beta]]\wedge(n'=n_0)\wedge(\alpha,\beta,\delta \mathbin{-\!\!\circledR} P_{\mathrm{S}[m/n']}) \\ \vee\ \Diamond\alpha{\leftarrow}n[\varnothing_{\mathrm{T}}]\wedge(n'=n_0)\wedge(\alpha\mathbin{-\!\!\circledR} P_{\mathrm{S}[\boldsymbol{null}/n']})\end{array}\right\} & n' := \mathtt{getLast}(n) & \{P_{\mathrm{S}}\}\\
\{\mathsf{H}\alpha,\beta.\,((\alpha{\leftarrow}\circ[n[\beta]] \mathbin{-\!\!*} (\alpha,\beta\mathbin{-\!\!\circledR} P_{\mathrm{S}})) * \alpha{\leftarrow}n[\beta])\} & \mathtt{insertNodeAbove}(n) & \{P_{\mathrm{S}}\}\\
\{\mathsf{H}\alpha,\beta.\,((\alpha{\leftarrow}\beta \mathbin{-\!\!*} (\alpha,\beta\mathbin{-\!\!\circledR} P_{\mathrm{S}})) * \alpha{\leftarrow}n[\beta])\} & \mathtt{deleteNode}(n) & \{P_{\mathrm{S}}\}\\
\{\exists c.\,\mathsf{H}\alpha.\,((\alpha{\leftarrow}n[\varnothing_{\mathrm{T}}] \mathbin{-\!\!*} (\alpha\mathbin{-\!\!\circledR} P_{\mathrm{S}})) * \alpha{\leftarrow}n[\mathrm{tree}(c)])\} & \mathtt{deleteSubtree}(n) & \{P_{\mathrm{S}}\}\\
\{\mathsf{H}\alpha,\beta,\gamma,\delta.\,(((\alpha{\leftarrow}m[n[\gamma]] * \beta{\leftarrow}\delta) \mathbin{-\!\!*} (\alpha,\beta,\gamma,\delta\mathbin{-\!\!\circledR} P_{\mathrm{S}})) * (\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta]))\} & \mathtt{moveNodeAbove}(n,m) & \{P_{\mathrm{S}}\}\\
\{\mathsf{H}\alpha,\beta,\gamma,\delta.\,(((\alpha{\leftarrow}n[\gamma\otimes m[\varnothing_{\mathrm{T}}]] * \beta{\leftarrow}\delta) \mathbin{-\!\!*} (\alpha,\beta,\gamma,\delta\mathbin{-\!\!\circledR} P_{\mathrm{S}})) * (\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta]))\} & \mathtt{appendNode}(n,m) & \{P_{\mathrm{S}}\}\\
\{\exists c.\,\mathsf{H}\alpha,\beta,\gamma.\,(((\alpha{\leftarrow}n[\gamma\otimes \mathrm{tree}(c)] * \beta{\leftarrow}m[\varnothing_{\mathrm{T}}]) \mathbin{-\!\!*} (\alpha,\beta,\gamma\mathbin{-\!\!\circledR} P_{\mathrm{S}})) * (\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)]))\} & \mathtt{appendSub}(n,m) & \{P_{\mathrm{S}}\}
\end{array}$$

Fig. 8. A Selection of the Weakest Preconditions for our Basic Tree Update Commands

Example 7 (Specifying appendChild). In Example 1 we gave the `appendChild` program. The command's specification and its derivation are:

$$\begin{array}{c}
\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)]\}\\
\mathtt{appendChild}(n,m)\\
\{\alpha{\leftarrow}n[\gamma\otimes m[\mathrm{tree}(c)]] * \beta{\leftarrow}\varnothing_{\mathrm{T}}\}
\end{array}$$

$$\begin{array}{l}
\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)]\}\\
\quad \mathtt{local}\ \mathit{temp} := \boldsymbol{null}\ \mathtt{in}\ \{\\
\{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\mathrm{tree}(c)] \wedge (\mathit{temp} = \boldsymbol{null})\}\\
\{\mathsf{H}\delta.\,\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta] * \delta{\leftarrow}\mathrm{tree}(c) \wedge (\mathit{temp} = \boldsymbol{null})\}\\
\qquad \mathtt{insertNodeAbove}(m)\ ;\\
\{\mathsf{H}\delta.\,\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}\circ[m[\delta]] * \delta{\leftarrow}\mathrm{tree}(c) \wedge (\mathit{temp} = \boldsymbol{null})\}\\
\qquad \mathit{temp} := \mathtt{getUp}(m)\ ;\\
\{\mathsf{H}\delta.\,\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}\mathit{temp}[m[\delta]] * \delta{\leftarrow}\mathrm{tree}(c)\}\\
\{\mathsf{H}\epsilon,\delta.\,\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}\mathit{temp}[\epsilon] * \epsilon{\leftarrow}m[\delta] * \delta{\leftarrow}\mathrm{tree}(c)\}\\
\qquad \mathtt{appendNode}(n,m)\ ;\\
\{\mathsf{H}\epsilon,\delta.\,\alpha{\leftarrow}n[\gamma\otimes m[\varnothing_{\mathrm{T}}]] * \beta{\leftarrow}\mathit{temp}[\epsilon] * \epsilon{\leftarrow}\delta * \delta{\leftarrow}\mathrm{tree}(c)\}\\
\{\alpha{\leftarrow}n[\gamma\otimes m[\varnothing_{\mathrm{T}}]] * \beta{\leftarrow}\mathit{temp}[\mathrm{tree}(c)]\}\\
\{\mathsf{H}\epsilon,\delta.\,\alpha{\leftarrow}n[\gamma\otimes\epsilon] * \epsilon{\leftarrow}m[\delta] * \delta{\leftarrow}\varnothing_{\mathrm{T}} * \beta{\leftarrow}\mathit{temp}[\mathrm{tree}(c)]\}\\
\qquad \mathtt{appendSub}(m,\mathit{temp})\ ;\\
\{\mathsf{H}\epsilon,\delta.\,\alpha{\leftarrow}n[\gamma\otimes\epsilon] * \epsilon{\leftarrow}m[\delta\otimes\mathrm{tree}(c)] * \delta{\leftarrow}\varnothing_{\mathrm{T}} * \beta{\leftarrow}\mathit{temp}[\varnothing_{\mathrm{T}}]\}\\
\{\alpha{\leftarrow}n[\gamma\otimes m[\mathrm{tree}(c)]] * \beta{\leftarrow}\mathit{temp}[\varnothing_{\mathrm{T}}]\}\\
\qquad \mathtt{deleteNode}(\mathit{temp})\quad \}\\
\{\alpha{\leftarrow}n[\gamma\otimes m[\mathrm{tree}(c)]] * \beta{\leftarrow}\varnothing_{\mathrm{T}}\}
\end{array}$$

Throughout the proof, we use the rules to separate out the footprint of a command, apply the appropriate small axiom, then use the rules to compress the result back into the original tree.

Example 8 (Specifying Node Manipulation). In Examples 2, 3 and 4 we gave three node manipulation programs: $\mathtt{simple}(n)$, $\mathtt{nodeSwap}(n,m)$ and $\mathtt{nodeCycle}(n)$. The specifications for each of these programs are:

$$\begin{array}{ccc}
\{\alpha{\leftarrow}n[m[\beta]\otimes\gamma]\} & \{\alpha{\leftarrow}n[\gamma] * \beta{\leftarrow}m[\delta]\} & \{\alpha{\leftarrow}n[m[\beta]\otimes\gamma\otimes \mathtt{l}[\delta]]\}\\
\mathtt{simple}(n) & \mathtt{nodeSwap}(n,m) & \mathtt{nodeCycle}(n)\\
\{\alpha{\leftarrow}m[n[\beta]\otimes\gamma]\} & \{\alpha{\leftarrow}m[\gamma] * \beta{\leftarrow}n[\delta]\} & \{\alpha{\leftarrow}\mathtt{l}[n[\beta]\otimes\gamma\otimes m[\delta]]\}
\end{array}$$

The derivations of these specifications are shown in **Fig. 9**.

Example 9 (Specifying queuePop). In a similar way, we can derive the following specification for the queuePop program from Example 5:

$$\begin{array}{c}
\{\alpha{\leftarrow}n[m[\mathrm{tree}(c)]\otimes\gamma\otimes \mathtt{i}[\beta]]\}\\
\mathtt{queuePop}(n)\\
\{\alpha{\leftarrow}m[\gamma\otimes n[\beta]\otimes \mathtt{i}[\mathrm{tree}(c)]]\}
\end{array}$$

simple derivation:

```
$\{\alpha\leftarrow n[m[\beta]\otimes\gamma]\}$
  local $\mathtt{temp}, \mathtt{first} := \textbf{\textit{null}}$ in {
$\left\{\begin{array}{l}\alpha\leftarrow n[m[\beta]\otimes\gamma]\wedge(\mathtt{temp}=\textbf{\textit{null}})\\ \qquad\wedge(\mathtt{first}=\textbf{\textit{null}})\end{array}\right\}$
    insertNodeAbove($n$) ;
$\left\{\begin{array}{l}\alpha\leftarrow\circ[n[m[\beta]\otimes\gamma]]\wedge(\mathtt{temp}=\textbf{\textit{null}})\\ \qquad\wedge(\mathtt{first}=\textbf{\textit{null}})\end{array}\right\}$
    $\mathtt{temp}$ := getUp($n$) ;
$\{\alpha\leftarrow\mathtt{temp}[n[m[\beta]\otimes\gamma]]\wedge(\mathtt{first}=\textbf{\textit{null}})\}$
    $\mathtt{first}$ := getFirst($n$) ;
$\{\alpha\leftarrow\mathtt{temp}[n[\mathtt{first}[\beta]\otimes\gamma]]\wedge(\mathtt{first}=m)\}$
    moveNodeAbove($\mathtt{first}, n$) ;
$\{\alpha\leftarrow\mathtt{temp}[n[\mathtt{first}[\beta]]\otimes\gamma]\wedge(\mathtt{first}=m)\}$
    moveNodeAbove($\mathtt{temp}, \mathtt{first}$) ;
$\{\alpha\leftarrow\mathtt{first}[\mathtt{temp}[n[\beta]\otimes\gamma]]\wedge(\mathtt{first}=m)\}$
    deleteNode($\mathtt{temp}$)   }
$\{\alpha\leftarrow\mathtt{first}[n[\beta]\otimes\gamma]\wedge(\mathtt{first}=m)\}$
$\{\alpha\leftarrow m[n[\beta]\otimes\gamma]\}$
```

nodeSwap derivation:

```
$\{\alpha\leftarrow n[\gamma]*\beta\leftarrow m[\delta]\}$
  local $\mathtt{temp} := \textbf{\textit{null}}$ in {
$\left\{\begin{array}{l}\alpha\leftarrow n[\gamma]*\beta\leftarrow m[\delta]\\ \quad\wedge(\mathtt{temp}=\textbf{\textit{null}})\end{array}\right\}$
    insertNodeAbove($n$) ;
$\left\{\begin{array}{l}\alpha\leftarrow\circ[n[\gamma]]*\beta\leftarrow m[\delta]\\ \qquad\wedge(\mathtt{temp}=\textbf{\textit{null}})\end{array}\right\}$
    $\mathtt{temp}$ := getUp($n$) ;
$\{\alpha\leftarrow\mathtt{temp}[n[\gamma]]*\beta\leftarrow m[\delta]\}$
    moveNodeAbove($m, n$) ;
$\{\alpha\leftarrow\mathtt{temp}[\gamma]*\beta\leftarrow n[m[\delta]]\}$
    moveNodeAbove($\mathtt{temp}, m$) ;
$\{\alpha\leftarrow m[\mathtt{temp}[\gamma]]*\beta\leftarrow n[\delta]\}$
    deleteNode($\mathtt{temp}$)   }
$\{\alpha\leftarrow m[\gamma]*\beta\leftarrow n[\delta]\}$
```

nodeCycle derivation:

```
$\{\alpha\leftarrow n[m[\beta]\otimes\gamma\otimes \mathtt{l}[\delta]]\}$
  local $\mathtt{first}, \mathtt{last} := \textbf{\textit{null}}$ in {
$\left\{\begin{array}{l}\alpha\leftarrow n[m[\beta]\otimes\gamma\otimes \mathtt{l}[\delta]]\\ \wedge(\mathtt{first}=\textbf{\textit{null}})\wedge(\mathtt{last}=\textbf{\textit{null}})\end{array}\right\}$
    $\mathtt{first}$ := getFirst($n$) ;
$\left\{\begin{array}{l}\alpha\leftarrow n[\mathtt{first}[\beta]\otimes\gamma\otimes \mathtt{l}[\delta]]\\ \wedge(\mathtt{first}=m)\wedge(\mathtt{last}=\textbf{\textit{null}})\end{array}\right\}$
    $\mathtt{last}$ := getLast($n$) ;
$\left\{\begin{array}{l}\alpha\leftarrow n[\mathtt{first}[\beta]\otimes\gamma\otimes \mathtt{last}[\delta]]\\ \quad\wedge(\mathtt{first}=m)\wedge(\mathtt{last}=\mathtt{l})\end{array}\right\}$
    nodeSwap($n, \mathtt{last}$) ;
$\left\{\begin{array}{l}\alpha\leftarrow\mathtt{last}[\mathtt{first}[\beta]\otimes\gamma\otimes n[\delta]]\\ \quad\wedge(\mathtt{first}=m)\wedge(\mathtt{last}=\mathtt{l})\end{array}\right\}$
    nodeSwap($n, \mathtt{first}$)   }
$\left\{\begin{array}{l}\alpha\leftarrow\mathtt{last}[n[\beta]\otimes\gamma\otimes \mathtt{first}[\delta]]\\ \quad\wedge(\mathtt{first}=m)\wedge(\mathtt{last}=\mathtt{l})\end{array}\right\}$
$\{\alpha\leftarrow \mathtt{l}[n[\beta]\otimes\gamma\otimes m[\delta]]\}$
```

Fig. 9. Derivations of the Specifications for `simple`, `nodeSwap` and `nodeCycle`

6 Conclusion

We have introduced Segment Logic for reasoning about structured data update in general, and tree update in particular. Using Segment Logic, we have demonstrated that it is possible to give small axioms for tree update commands such as DOM's `appendChild` command. In this paper, we have concentrated on a simple, lightweight tree update language. It is straightforward to transfer the techniques developed here to Featherweight DOM [6]. We do not envisage difficulties with extending the approach to the full DOM specification [16].

A typical Segment Logic proof separates the working tree into segments, identifying the tree segment which corresponds to the footprint of a command. It applies a small axiom to this segment, and compresses the updated fragment back into the original tree. This separation and compression is key to our Segment Logic reasoning. It is not unlike the unfolding and folding of abstract predicates, due to Parkinson and Vafeiadis [17]. The difference is that reasoning using Separation Logic with abstract predicates is implementation dependent: the formula $slist(l, i)$ describes a list l implemented as a singly-linked list with heap address i; the formula $dlist(l, i, j)$ describes a list l implemented as a doubly-linked list with heap addresses i and j. By contrast, reasoning using Segment Logic is implementation independent: the formula $\alpha\leftarrow P$ describes e.g. a list or tree identified and satisfying formula or data type P at abstract address α.

Our next step is to design and formally specify a concurrent XML update language, combining ideas from Featherweight DOM, Concurrent Separation Logic [13] and Segment Logic. We believe that Segment Logic provides us with crucial technology for achieving this goal. For example, consider the program `deleteTree`(m) || `deleteTree`(m), which should succeed if the two trees being called are disjoint. A Segment Logic specification of this program is:

$$\{\alpha \leftarrow n[\text{tree}(c_1)] * \beta \leftarrow m[\text{tree}(c_2)]\}$$
$$\texttt{deleteTree}(n) \parallel \texttt{deleteTree}(m)$$
$$\{\alpha \leftarrow \varnothing_{\mathrm{T}} * \beta \leftarrow \varnothing_{\mathrm{T}}\}$$

Segment Logic allows us to establish such natural disjointness properties, since it combines reasoning directly about the abstract tree structure with using the separating conjunction $*$. Our goal is to extend the update language presented here with parallel composition and critical regions, and adapt the Concurrent Separation Logic reasoning to provide a formal, compositional specification of a concurrent XML update language.

Acknowledgments. We thank Thomas Dinsdale-Young for many interesting discussions regarding this work and Viktor Vafeiadis for the name Segment Logic. Gardner acknowledges support of a Microsoft/RAEng Senior Research Fellowship. Wheelhouse acknowledges support of an EPSRC DTA award.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
2. Calcagno, C., Dinsdale-Young, T., Gardner, P.: Adjunct elimination in context logic for trees. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 255–270. Springer, Heidelberg (2007)
3. Calcagno, C., Gardner, P., Zarfaty, U.: Context logic and tree update. In: POPL, vol. 40, ACM, New York (2005)
4. Cardelli, L., Gordon, A.D.: Ambient logic. Mathematical Structures in Computer Science (in press, 2006)
5. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Aspects of Computing 13 (2002)
6. Gardner, P., Smith, G., Wheelhouse, M., Zarfaty, U.: Local Hoare reasoning about DOM. In: PODS, vol. 27, ACM, New York (2008)
7. Gardner, P., Wheelhouse, M.: Small specifications for tree update (extended version) (2009), `http://www.doc.ic.ac.uk/~mjw03/PersonalWebpage/pdfs/moveFull.pdf`
8. Gardner, P., Zarfaty, U.: Reasoning about high-level tree update and its low-level implementation. Technical Report DTR09-9, Imperial College (2009)
9. Hosoya, H., Pierce, B.C.: Xduce: A statically typed XML processing language. In: TOIT 2003, vol. 3, ACM, New York (2003)
10. Ishtiaq, S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL 2001, vol. 36, ACM, New York (2001)
11. Milner, R.: Pi-nets: A graphical form of π-calculus. In: Sannella, D. (ed.) ESOP 1994. LNCS, vol. 788, Springer, Heidelberg (1994)
12. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes I & II. Information and Computation 100 (1992)
13. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375 (2007)
14. O'Hearn, P.W., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 15. Springer, Heidelberg (2001)
15. Raza, M., Gardner, P.: Footprints in local reasoning. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 201–215. Springer, Heidelberg (2008)

16. Smith, G.: Providing a formal specification for DOM core level 1. PhD Thesis, (to be submitted) (December 2009)
17. Vafeiadis, V.: Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, Cambridge (2008)
18. W3C. Dom: Document object model. W3C recommendation (2005), `http://www.w3.org/DOM/`
19. Yang, H., O'Hearn, P.W.: A semantic basis for local reasoning. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, p. 402. Springer, Heidelberg (2002)

Author Index