Jean-Marie Jacquet
Gian Pietro Picco (E

Coordina
Models
and Lang

LNCS 3454

7th International Conferen
Namur, Belgium, April 200
Proceedings

Lecture Notes in Computer Science 3454

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

Editorial Board

Jean-Marie Jacquet Gian Pietro Picco (Eds.)

Coordination Models and Languages

7th International Conference, COORDINATION 2005
Namur, Belgium, April 20-23, 2005
Proceedings

Springer

Volume Editors

Jean-Marie Jacquet
University of Namur
Institute of Informatics
Rue Grandgagnage 21, 5000 Namur, Belgium
E-mail: jmj@info.fundp.ac.be

Gian Pietro Picco
Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32, 20133 Milan, Italy
E-mail: picco@elet.polimi.it

Library of Congress Control Number: 2005923585

CR Subject Classification (1998): D.2.4, D.2, C.2.4, D.1.3, F.1.2, I.2.11

ISSN 0302-9743
ISBN-10 3-540-25630-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-25630-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11417019 06/3142 5 4 3 2 1 0

Preface

Modern information systems rely increasingly on combining concurrent, distributed, mobile, reconfigurable and heterogenous components. New models, architectures, languages, and verification techniques are therefore necessary to cope with the complexity induced by the demands of today's software development. Coordination languages have emerged as a successful approach, providing abstractions that cleanly separate behavior from communication and therefore increasing modularity, simplifying reasoning, and ultimately enhancing software development.

This volume contains the proceedings of the 7th International Conference on Coordination Models and Languages (Coordination 2005), held at the Institute of Informatics of the University of Namur, Belgium, on April 20–23, 2005. The previous conferences in this series took place in Cesena (Italy), Berlin (Germany), Amsterdam (The Netherlands), Limassol (Cyprus), York (UK), and Pisa (Italy). Building upon the success of these events, Coordination 2005 provided a forum for the community of researchers interested in models, languages, and implementation techniques for coordination and component-based software, as well as applications that exploit them.

The conference attracted 88 submissions from authors all over the world. The Program Committee, consisting of 20 of the most distinguished researchers in the coordination research area, selected 19 papers for presentation on the basis of originality, quality, and relevance to the topics of the conference. Each submission was refereed by three reviewers — four in the case of papers written by a member of the Program Committee. As with previous editions, the paper submission and selection processes were managed entirely electronically. This was accomplished using ConfMan (`www.ifi.uni.no/confman/ABOUT-ConfMan/`), a free Web-based conference management system, and with the invaluable help of Paolo Costa, who installed and customized the system, ensuring its smooth operation.

We are grateful to all the Program Committee members who devoted much effort and time to read and discuss the papers. Moreover, we gratefully acknowledge the help of additional external reviewers, listed later, who reviewed submissions in their areas of expertise.

Finally, we would like to thank the authors of all the submitted papers and the conference attendees, for keeping this research community lively and interactive, and ultimately ensuring the success of this conference series.

February 2005

Jean-Marie Jacquet
Gian Pietro Picco

Organization

Program Co-chairs

Jean-Marie Jacquet	University of Namur, Belgium
Gian Pietro Picco	Politecnico di Milano, Italy

Program Committee

Farhad Arbab	CWI, Amsterdam, The Netherlands
Luca Cardelli	Microsoft Research, Cambridge, UK
Gianluigi Ferrari	University of Pisa, Italy
Paola Inverardi	University of L'Aquila, Italy
Toby Lehman	IBM Almaden Research Center, USA
Ronaldo Menezes	Florida Institute of Technology, USA
Amy L. Murphy	University of Lugano, Switzerland
Andrea Omicini	University of Bologna, Italy
George Papadopoulos	University of Cyprus, Cyprus
Ernesto Pimentel	University of Malaga, Spain
Rosario Pugliese	University of Florence, Italy
Antonio Porto	New University of Lisbon, Portugal
Carolyn Talcott	SRI International, USA
Sebastian Uchitel	Imperial College London, UK
Jan Vitek	Purdue University, USA
Michel Wermelinger	New University of Lisbon, Portugal and The Open University, UK
Herbert Wiklicky	Imperial College London, UK
Alexander Wolf	University of Lugano, Switzerland and University of Colorado, Boulder, USA
Alan Wood	University of York, UK
Gianluigi Zavattaro	University of Bologna, Italy

Steering Committee

Farhad Arbab	CWI, Amsterdam, The Netherlands
Paolo Ciancarini	University of Bologna, Italy
Rocco De Nicola	University of Florence, Italy
Chris Hankin	Imperial College London, UK

George Papadopoulos	University of Cyprus, Cyprus
Antonio Porto	New University of Lisbon, Portugal
Gruia-Catalin Roman	Washington University in Saint Louis, USA
Robert Tolksdorf	Free University of Berlin, Germany
Alexander Wolf	University of Colorado, Boulder, USA

Referees

Marco Aldinucci
Silvia Amaro
Paolo Bellavista
Michele Boreale
Roberto Bruni
Marzia Buscemi
Nadia Busi
Sonia Campa
Carlos Canal
Phil Chan
David G. Clarke
Giovanni Conforti
Maria del Mar Gallardo
Enrico Denti
David F. de Oliveira Costa
Nikolay Diakov
Manuel Diaz
Alessandra Di Pierro
Davide Di Ruscio
Daniele Falassi
Richard Ford
Luca Gardelli
Mauro Gaspari
Stefania Gnesi
Daniele Gorla
Claudio Guidi
Chris Hankin
Dan Hirsch
Jeremy Jacob
Ivan Lanese
Ruggero Lanotte
Alessandro Lapadula
Alberto Lluch-Lafuente
Michele Loreti
Roberto Lucchi
Fabio Mancinelli
Hernan Melgratti
Nicola Mezzetti
Michela Milano
Alberto Montresor
Gianluca Moro
Henry Muccini
Patrizio Pellicione
Alfonso Pierantonio
Cees Pierik
Alessandro Ricci
Bartolome Rubio
Juan Guillen Scholten
Laura Semini
Marius Silaghi
Igor Siveroni
Giuseppe Sollazzo
Ryan Stansifer
Emilio Tuosto
Izura Udzir
Leon W.N. van der Torre
Mirko Viroli
Andrew Wilkinson
Peter Zoeteweij

Table of Contents

A Case Study of Web Services Orchestration

Manuel Mazzara[1] and Sergio Govoni[2]

[1] Department of Computer Science, University of Bologna, Italy
mazzara@cs.unibo.it
[2] Imaging Science and Information Systems Center, Georgetown University, Washington, DC, USA
govoni@isis.imac.georgetown.edu

Abstract. Recently the term Web Services Orchestration has been introduced to address composition and coordination of Web Services. Several languages to describe orchestration for business processes have been presented and many of them use concepts such as long-running transactions and compensations to cope with error handling. WS-BPEL is currently the best suited in this field. However, its complexity hinders rigorous treatment. In this paper we address the notion of orchestration from a formal point of view, with particular attention to transactions and compensations. In particular, we discuss $\texttt{web}\pi_\infty$, an untimed subcalculus of $\texttt{web}\pi$ [15] which is a simple and conservative extension of the π-calculus. We introduce it as a theoretical and foundational model for Web Services coordination. We simplify some semantical and pragmatical aspects, in particular regarding temporization, gaining a better understanding of the fundamental issues. To discuss the usefulness of the language we consider a case study: we formalize an e-commerce transactional scenario drawing on a case presented in our previous work [12].

1 Introduction

The aim of Web Services is to ease and to automate business process collaborations across enterprise boundaries. The core Web Services standards, WSDL [11] and UDDI [26], cover calling services over the Internet and finding them, but they are not enough. Creating collaborative processes requires an additional layer on top of the Web Services protocol stack: this way we can achieve Web Services composition and orchestration. In particular, orchestration is the description of interactions and messages flow between services in the context of a business process [23]. Orchestration is not a new concept; in the past it has been called workflow [28].

1.1 The State of the Art in Orchestration

Three specifications have been introduced to cover orchestration: Web Services Business Process Execution Language (WS-BPEL or BPEL for short) [1] which is the successor of Microsoft XLANG [25, 5] and IBM WSFL [16], together

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 1–16, 2005.

with WS-Coordination (WS-C) [29] and WS-Transaction (WS-T) [30]. BPEL is a workflow-like definition language that allows to describe sophisticated business processes; WS-Coordination and WS-Transaction complement it to provide mechanisms for defining specific standard protocols to be used by transaction processing systems, workflow systems, or other applications that wish to coordinate multiple services. Together, these specifications address connectivity issues that arise when Web Services run on several platforms across organizations.

1.2 Transactions in Web Services

A common business scenario involves multiple parties and different organizations over a time frame. Negotiations, commitments, shipments and errors happen. A business transaction between a manufacturer and its suppliers ends successfully only when parts are delivered to their final destination, and this could be days or weeks after the initial placement of the order.

A transaction completes successfully (*commits*) or it fails (*aborts*) undoing (*roll-backing*) all its past actions. Web services transactions [17] are long-running transactions. As such, they pose several problems. It is not feasible to turn an entire long-running transaction into an ACID transaction, since maintaining isolation for a long time poses performance issues [31]. Roll-backing is also an issue. Undoing many actions after a long time from the start of a transaction entails trashing what could be a vast amount of work.

Since in our scenario a traditional roll-back is not feasible, Web Services orchestration environments provide a *compensation* mechanism which can be executed when the effects of a transaction must be cancelled. What a compensation policy does depends on the application. For example, a customer orders a book from an on-line retailer. The following day, that customer gets a copy of the book elsewhere, then requests the store to withdraw the order. As a compensation, the store can cancel the order, or charge a fee. In any case, in the end the application has reached a state that it considers equivalent to what it was before the transaction started.

The notions of orchestration and compensation require a formal definition. In this paper, we address orchestration with particular attention to web transactions. We introduce $\texttt{web}\pi_\infty$, a subcalculus of $\texttt{web}\pi$ [15] that does not model time, as a simple extension of the π-calculus. As a case study, we discuss and formalize an e-commerce transactional scenario building on a previous one, which we presented in an earlier work [12] using a different algebra, the Event Calculus, which we introduced in [18]. The Event Calculus needed some improvement to make it more readable and easier to use for modelling real-world scenarios. This paper is a step in that direction.

1.3 Related Work

In this paper we mainly refer to BPEL, the most likely candidate to become a standard among workflow-based composition languages. Other languages have

been introduced, among them WS-CDL [14], which claims to be in some relation with the fusion calculus [22].

Other papers discuss formal semantics of compensable activities in this context. [13] is mainly inspired by XLANG; the calculus in [9] is inspired by BP-Beans [10]; the πt-calculus [8] focuses on BizTalk; [6] deals with short-lived transactions in BizTalk; [7] also presents the formal semantics for a hierarchy of transactional calculi with increasing expressiveness.

Some authors believe that time should be introduced both at the model level and at the protocols and implementation levels [15, 3, 2, 4]. XLANG, for instance, provides a notion of *timed transaction* as a special case of long running activity. BPEL uses timers to achieve a similar behavior. This is a very appropriate feature when programming business services which cannot wait forever for the other parties reply.

1.4 Outline

This work is organized as follows. In Section 2 we explain our formal approach to orchestration: extending the π-calculus to include transactions. In Section 3 we discuss this extension with its syntax and semantics, while in Section 4 we discuss an e-commerce transactional scenario to show the strength of the language. Section 5 draws a conclusion.

2 A Formal Approach to Web Services Orchestration

Business process orchestration has to meet several requirements, including providing a way to manage exceptions and transactional integrity [23]. Orchestration languages for Web Services should have the following interesting operations: sequence, parallel, conditional, send to/receive from other Web Services on typed WSDL ports, invocation of Web Services, error handling.

BPEL covers all these aspects. Its current specification, however, is rather involved. A major issue is error handling. BPEL provides three different mechanisms for coping with abnormal situations: *fault handling*, *compensation handling* and *event handling*. [1] Documentation shows ambiguities, in particular when interactions between these mechanisms are required. Therefore it is difficult to use the language, and we want to address this issue.

Our goal is to define a clear model with the smallest set of operators which implement the operations discussed above, and simple to use for application designers. We build on the π-calculus [21, 20, 24], a well known process algebra. It is simple and appropriate for orchestration purposes. It includes: a parallel operator allowing explicit concurrency; a restriction operator allowing compositionality and explicit resource creation; a recursion or a process definition operator allowing Turing completeness; a sequence operator allowing causal relationship

[1] The BPEL event handling mechanism was not designed for error handling only. However, here we use it for this purpose.

between activities; an inaction operator which is just a ground term for inductive definition on sequencing; message passing and in particular name passing operators allowing communication and link mobility.

There is an open debate on the use of π-calculus versus Petri nets in the context of Web Services composition [27]. The main reason here for using the π-calculus for formalization is that the so called *Web Services composition languages*, like XLANG, BPEL and WS-CDL claim to be based on it, and they should therefore allow rigorous mathematical treatment. However, no interesting relation with process algebras has really been proved for any of them, nor an effective tool for analysis and reasoning, either theoretical or software based, has been released. Therefore, we see a gap that needs to be filled, and we want to address the problem of composing services starting directly from the π-calculus.

By itself the π-calculus does not support any transactional mechanism. Programming complex business processes with failure handling in term of message passing only is not reasonable; also, the Web Services environment requires that several operations have transactional properties and be treated as a single logical unit of work when performed within a single business transaction. Below we consider a simple extension of the π-calculus that covers transactions.

3 The Orchestration Calculus $\mathtt{web}\pi_\infty$

The syntax of $\mathtt{web}\pi_\infty$ *processes* relies on countable sets of *names*, ranged over by $x, y, z, u, \cdots$. Tuples of names are written $\widetilde{u}$.

$$
\begin{array}{lll}
P ::= & & \\
 & \mathbf{0} & \text{(nil)} \\
 & \mid \overline{x}\,\langle \widetilde{u} \rangle & \text{(output)} \\
 & \mid x(\widetilde{u}).P & \text{(input)} \\
 & \mid (x)P & \text{(restriction)} \\
 & \mid P \mid P & \text{(parallel composition)} \\
 & \mid A(\tilde{u}) & \text{(process invocation)} \\
 & \mid \langle\!| P \ ; \ P |\!\rangle_x & \text{(transaction)}
\end{array}
$$

We are assuming a set of process constants, ranged over by A, in order to support process definition. A defining equation for a process identifier A is of the form

$$A(\tilde{u}) \stackrel{def}{=} P$$

where each occurrence of A in P has to be guarded, i.e. it is underneath an input prefix. It holds $fn(P) \subseteq \{\tilde{u}\}$ and $\tilde{u}$ is composed by pairwise distinct names.

A process can be the inert process $\mathbf{0}$, an output $\overline{x}\,\langle \widetilde{u} \rangle$ sent on a name x that carries a tuple of names $\widetilde{u}$, an input $x(\widetilde{u}).P$ that consumes a message $\overline{x}\,\langle \widetilde{w} \rangle$ and behaves like $P\{\widetilde{w}/\widetilde{u}\}$, a restriction $(x)P$ that behaves as P except that inputs and messages on x are prohibited, a parallel composition of processes, a process invocation $A(\tilde{u})$ or a transaction $\langle\!| P \ ; \ R |\!\rangle_x$ that behaves as the *body* P until a transaction abort message $\overline{x}\,\langle\rangle$ is received, then it behaves as the *compensation* Q.

Names x in outputs and inputs are called *subjects* of outputs and inputs respectively. It is worth noticing that the syntax of $\texttt{web}\pi_\infty$ processes simply extends the asynchronous π-calculus with the transaction process.

The input $x(\widetilde{u}).P$ and restriction $(x)P$ are binders of names $\widetilde{u}$ and x respectively. The scope of these binders is the processes P. We use the standard notions of α-equivalence, *free* and *bound names* of processes, noted $\texttt{fn}(P)$, $\texttt{bn}(P)$ respectively. In particular

$\texttt{fn}(\langle\!|P\ ;\ R|\!\rangle_x) = \texttt{fn}(P) \cup \texttt{fn}(R) \cup \{x\}$ and α-equivalence equates $(x)(\langle\!|P\ ;\ Q|\!\rangle_x)$ with $(z)(\langle\!|P\{^z/_x\}\ ;\ Q\{^z/_x\}|\!\rangle_z)$;

In the following we let $\tau.P$ be the process $(z)(\overline{z}\,\langle\rangle \mid z().P)$ where $z \notin \texttt{fn}(P)$. $\texttt{web}\pi_\infty$ processes considered in this paper are always *well-formed* according to the following:

Definition 1 (Well-formedness). *Received names cannot be used as subjects of inputs. Formally, in $x(\widetilde{u}).P$ free subjects of inputs in P do not belong to names $\widetilde{u}$.*

This property avoids a situation where different services receive information on the same channel, which is a nonsense in the service oriented paradigm.

3.1 Semantics of the Language

We give the semantics for the language in two steps, following the approach of Milner [19], separating the laws which govern the static relations between processes from the laws which 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 2. *The* structural congruence $\equiv$ *is the least congruence closed with respect to α-renaming, satisfying the abelian monoid laws for parallel (associativity, commutativity and $\mathbf{0}$ as identity), and the following axioms:*

1. *The scope laws:*

$$(u)\mathbf{0} \equiv \mathbf{0}, \qquad (u)(v)P \equiv (v)(u)P,$$
$$P \mid (u)Q \equiv (u)(P \mid Q)\ , \quad \textit{if } u \notin \texttt{fn}(P)$$
$$\langle\!|(z)P\ ;\ Q|\!\rangle_x \equiv (z)\langle\!|P\ ;\ Q|\!\rangle_x\ , \quad \textit{if } z \notin \{x\} \cup \texttt{fn}(Q)$$

2. *The invocation law:*

$$A(\tilde{v}) \equiv P\{\tilde{v}/\tilde{u}\} \qquad \text{if } A(\tilde{u}) \stackrel{def}{=} P$$

3. *The transaction laws:*

$$\langle\!|\mathbf{0}\ ;\ Q|\!\rangle_x \equiv \mathbf{0}$$
$$\langle\!|\langle\!|P\ ;\ Q|\!\rangle_y \mid R\ ;\ R'|\!\rangle_x \equiv \langle\!|P\ ;\ Q|\!\rangle_y \mid \langle\!|R\ ;\ R'|\!\rangle_x$$

4. *The floating law:*

$$\langle\!|\overline{z}\,\langle\widetilde{u}\rangle \mid P\ ;\ Q|\!\rangle_x \equiv \overline{z}\,\langle\widetilde{u}\rangle \mid \langle\!|P\ ;\ Q|\!\rangle_x$$

The scope and invocation laws are standard. Let us discuss transaction and floating laws, which are unusual. The law $\langle\!|\mathbf{0}\ ;\ Q|\!\rangle_x \equiv \mathbf{0}$ defines committed transactions, namely transactions with $\mathbf{0}$ as body. These transactions, being committed, are equivalent to $\mathbf{0}$ and, therefore, cannot fail anymore. The law $\langle\!|\langle\!|P\ ;\ Q|\!\rangle_y \mid R\ ;\ R'|\!\rangle_x \equiv \langle\!|P\ ;\ Q|\!\rangle_y \mid \langle\!|R\ ;\ R'|\!\rangle_x$ moves transactions outside parent transactions, thus flattening the nesting of transactions. Notwithstanding this flattening, parent transactions may still affect children transactions by means of transaction names. The law $\langle\!|\overline{z}\,\langle\widetilde{u}\rangle \mid P\ ;\ R|\!\rangle_x \equiv \overline{z}\,\langle\widetilde{u}\rangle \mid \langle\!|P\ ;\ R|\!\rangle_x$ floats messages outside transactions; it models that 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 transaction boundaries, until it reaches the corresponding input. In case an outer transaction fails, recovery actions for this message may be detailed inside the compensation processes. The dynamic behavior of processes is defined by the reduction relation.

Definition 3. *The* reduction relation $\rightarrow$ *is the least relation satisfying the following axioms and closed with respect to* $\equiv$, $(x)\_\ ,\ \_\,|\_$ *and* $\langle\!|\ \_\ ;\ Q|\!\rangle_x$*:*

$$\overline{x}\,\langle\widetilde{v}\rangle \mid x(\widetilde{u}).P \ \rightarrow\ P\{\widetilde{v}/\widetilde{u}\} \qquad \text{(COM)}$$

$$\overline{x} \mid \langle\!|\textstyle\prod_{i\in I} x_i(\widetilde{u_i}).P_i\ ;\ Q|\!\rangle_x \ \rightarrow\ Q \qquad (I \neq \emptyset) \qquad \text{(FAIL)}$$

Rule (com) is standard in process calculi and models input-output interaction. Rule (fail) models transaction failures: when a transaction abort (a message on a transaction name) is emitted, the corresponding transaction is terminated by garbage collecting the threads (the input processes) in its body and activating the compensation. On the contrary, aborts are not possible if the transaction is already terminated, namely every thread in the body has completed its job.

4 A Case Study

In this section, we discuss an implementation in $\texttt{web}\pi_\infty$ of a classical e-business scenario: a customer attempts to buy a set of items from some providers, using a coordination service exposed by a web portal. Actors involved in this e-business scenario are a *customer*, a *web portal* and a set of *item providers*.

4.1 Participants

The roles who take part in the purchase scenario are the following:

1. a **customer** sends a request to a shopping portal, and waits for a response. The customer can express some constraints: for example, "I want to buy either all items or no one at all". The web portal takes care of implementing policies like this one;
2. a **web portal** tries to fulfill customers' requests and their constraints about the purchase policy. It acts as a coordinator;
3. an **item provider** accepts two kinds of requests from the web portal: a simple browsing of the price-list (read-only), and a purchase request of an item.

The web portal, on behalf of a customer, tries to buy an item from a provider. This could be a failure or success. In case of failure, the web portal is informed, and the item provider forgets everything about the transaction. In case of success, if the request can be fulfilled, the item provider declares that the sale is complete, and it begins the execution of an internal process which simulates the delivery of the item. Meanwhile, the customer can change her mind and tell the item provider, which will *compensate* the relative transaction, i.e. take some actions to establish a safe state. An example of compensation may be charging a fee. This mechanism will be explained more in detail below within the $\texttt{web}\pi_\infty$ specification.

4.2 Constraints

When sending a purchase request, a customer can also specify the behavior that the complete transaction must follow. For example, a customer wants to buy formal attire: a suit, a pair of shoes, a shirt and a tie. A reasonable constraint to impose is that either the shirt and the tie should come together, or none of them, while the suit and the shoes are optional. In our specification, we describe a simplified policy called *all or nothing*. This means that the purchase transaction will be successful only if all sub-transactions will commit, otherwise the purchase will fail. To implement this constraint, the web portal uses the compensation service that the item providers provide.

Buy requests are emitted simultaneously to each item provider, and the web portal gets their outcomes. If each sub-transaction is successful, the web portal informs the customer that its request has been satisfied, otherwise, it compensates any committed sub-transaction.

In our implementation we simplify this scenario. Instead of asking the customer for constraints over an order, we apply a built-in policy. This is fair to pose, because constraints are contained in the coordinator process, and this does not affect the behavior of item providers. It is also very easy to specify different purchase policies, because they are clearly separated from the mechanisms which control them. Further, we also assume that a customer wants to buy two items only from two different sellers.

4.3 Formal Description

We now present a formal description of all participants and how they can be composed in an e-business scenario.

World

$$\begin{aligned}\mathsf{WORLD}() := {} & (a_c)(a_p)(c_1)(p_1)(c_2)(p_2)\\ & (\mathsf{CUSTOMER}(a_c, a_p)\\ & \mid \mathsf{WEB\_PORTAL}(a_c, a_p, c_1, p_1, c_2, p_2)\\ & \mid \mathsf{IP}_1(c_1, p_1)\\ & \mid \mathsf{IP}_2(c_2, p_2))\end{aligned}$$

The process $\mathsf{WORLD}()$ composes the various participants to the scenario; first of all, it creates some global channels, used by the processes to interact together: the channels a_c and a_p are the web portal interfaces exposed to the customers. So, they are passed as arguments both to the $\mathsf{CUSTOMER}(a_c, a_p)$ and to the $\mathsf{WEB\_PORTAL}(a_c, a_p, c_1, p_1, c_2, p_2)$ processes. The first one is used to require a price list, while the second one to emit a purchase order.

The other global channels are the set of pairs c_i and p_i, which are respectively the query and the purchase interface of the i^{th} item provider. Those names are passed as arguments to the $\mathsf{WEB\_PORTAL}(a_c, a_p, c_1, p_1, c_2, p_2)$ and $\mathsf{IP_i}(c_i, p_i)$ processes.

We do not model message loss, because we suppose that reliable protocols are used, which would take care of any transmission error, and we ignore the issue of site crashes. We also assume the world as a closed system, in the sense that $\mathtt{fn}(()WORLD()) = \emptyset$. Because of the dynamic nature of the scenario, this could be regarded as a rather strong assumption. All these aspects could be taken into account in a future evolution of the specification.

Customer

$$\begin{aligned}\mathsf{CUSTOMER}(a_c, a_p) := {} & (\widetilde{q_1})(\widetilde{q_2})(a_r)(a_s)(a_f)\\ & (\overline{a_c}\,\langle \widetilde{q_1}, \widetilde{q_2}, a_r\rangle \mid a_r(\widetilde{l_1}, \widetilde{l_2}).\overline{a_p}\,\langle \widetilde{q_1}, \widetilde{q_2}, a_s, a_f\rangle \mid\\ & a_s().\mathsf{S}() \mid a_f().\mathsf{F}())\end{aligned}$$

The customer process first browses a price list. When it receives an answer, it emits a purchase request, and waits for the outcome. To do this it creates these names: $\widetilde{q_1}$ and $\widetilde{q_2}$, which contain the two item preferences, the channel a_r, which is the restricted reply channel used by the Web Portal to inform the customer about the price list consultation, and the two channels a_s (success) and a_f (failure), which signal respectively the outcome of the purchase transaction. Then the customer process sends the message $\overline{a_c}\,\langle \widetilde{q_1}, \widetilde{q_2}, a_r\rangle$ to the web portal consultation interface. This message carries the items description and the reply channel. This first phase ends with the receipt of the reply message $a_r(\widetilde{l_1}, \widetilde{l_2})$, which carries two names, $\widetilde{l_1}$ and $\widetilde{l_2}$, encoding the features of the requested items, like their availability, the selling price and many others. Basing on this information, the customer process elaborates its orders — which are encoded in $\widetilde{q_1}$ and $\widetilde{q_2}$ — and sends a purchase order $\overline{a_p}\,\langle \widetilde{q_1}, \widetilde{q_2}, a_s, a_f\rangle$ containing the item specifications and

the outcome channels, a_s and a_f. When the customer process receives one of this message, the purchase transaction has completed and it goes on with the appropriate task identified by $\mathsf{S}()$ or $\mathsf{F}()$. Moreover, it is guaranteed that either all the items have been bought, or the appropriate compensations have been emitted.

Web Portal

$$
\begin{aligned}
&\mathsf{WEB\_PORTAL}(a_c, a_p, c_1, p_1, c_2, p_2) := a_c(\widetilde{q}_1, \widetilde{q}_2, a_r).\\
&\qquad (\mathsf{ENGINE}(a_p, c_1, p_1, c_2, p_2, \widetilde{q}_1, \widetilde{q}_2, a_r) \mid\\
&\qquad \mathsf{WEB\_PORTAL}(a_c, a_p, c_1, p_1, c_2, p_2))\\
&\mathsf{ENGINE}(a_p, c_1, p_1, c_2, p_2, \widetilde{q}_1, \widetilde{q}_2, a_r) := \mathsf{QUERY}(c_1, c_2, \widetilde{q}_1, \widetilde{q}_2, a_r) \mid\\
&\qquad \mathsf{PURCHASE}(a_p, p_1, p_2)\\
&\mathsf{QUERY}(c_1, c_2, \widetilde{q}_1, \widetilde{q}_2, a_r) := (r_1)(r_2)(\overline{c_1}\,\langle \widetilde{q}_1, r_1\rangle \mid \overline{c_2}\,\langle \widetilde{q}_2, r_2\rangle \mid\\
&\qquad r_1(\widetilde{l}_1).r_2(\widetilde{l}_2).\overline{a_r}\,\langle \widetilde{l}_1, \widetilde{l}_2\rangle)\\
&\mathsf{PURCHASE}(a_p, p_1, p_2) := a_p(\widetilde{q}_1, \widetilde{q}_2, a_s, a_f).(r_s^1)(r_f^1)(r_s^2)(r_f^2)\\
&\qquad (\overline{p_1}\,\langle \widetilde{q}_1, r_s^1, r_f^1\rangle \mid \overline{p_2}\,\langle \widetilde{q}_2, r_s^2, r_f^2\rangle \mid\\
&\qquad \mathsf{WAIT}(r_s^1, r_f^1, r_s^2, r_f^2, a_s, a_f))
\end{aligned}
$$

The web portal process exposes a service which can be used by a customer to query some distributed price lists, and subsequently to purchase the items. When it receives a request $a_c(\widetilde{q}_1, \widetilde{q}_2, a_r)$, it executes a managing process — $\mathsf{ENGINE}(a_p, c_1, p_1, c_2, p_2, \widetilde{q}_1, \widetilde{q}_2, a_r)$ — and it creates a duplicate, to wait for further requests.

The $\mathsf{ENGINE}(a_p, c_1, p_1, c_2, p_2, \widetilde{q}_1, \widetilde{q}_2, a_r)$ process executes two sub-processes $\mathsf{QUERY}(c_1, c_2, \widetilde{q}_1, \widetilde{q}_2, a_r)$ and $\mathsf{PURCHASE}(a_p, p_1, p_2)$. The first of these subtasks, $\mathsf{QUERY}(c_1, c_2, \widetilde{q}_1, \widetilde{q}_2, a_r)$, receives the consulting channels c_1 and c_2, the customer preferences $\widetilde{q}_1$ and $\widetilde{q}_2$ and the reply channel a_r. It emits in parallel the various price list consultations with the messages $\overline{c_1}\,\langle \widetilde{q}_1, r_1\rangle$ and $\overline{c_2}\,\langle \widetilde{q}_2, r_2\rangle$, which contain the customer preferences and the private channels r_1 and r_2 on which it will wait for a reply. Those replies contain the outcomes of the queries executed on the item provider's databases — encoded with names $\widetilde{l}_1$ and $\widetilde{l}_2$. When the web portal receives them, it forwards them to the customer application with the message $\overline{a_r}\,\langle \widetilde{l}_1, \widetilde{l}_2\rangle$, and it waits for a purchase order on the channel $a_p(\widetilde{q}_1, \widetilde{q}_2, a_s, a_f)$.

The process $\mathsf{PURCHASE}(a_p, p_1, p_2)$ is called with the channel a_p, on which it will wait for the customer's order, and the item providers' channels p_1 and p_2. First, it receives the customer's request $a_p(\widetilde{q}_1, \widetilde{q}_2, a_s, a_f)$, which contains the item specifications and the pair of success/failure channels. At this point, it creates a pair of success/failure reply channels r_s and r_f for each item provider, and emits the purchase requests $\overline{p_1}\,\langle \widetilde{q}_1, r_s^1, r_f^1\rangle$ and $\overline{p_2}\,\langle \widetilde{q}_2, r_s^2, r_f^2\rangle$. When the requests have been emitted, the process $\mathsf{PURCHASE}(a_p, p_1, p_2)$ executes the process $\mathsf{WAIT}(r_s^1, r_f^1, r_s^2, r_f^2, a_s, a_f)$, which will manage the purchase transactions' outcomes.

Waiting Process. The process $\mathsf{WAIT}(r_s^1, r_f^1, r_s^2, r_f^2, a_s, a_f)$ waits for the outcome of the item provider 1 in this way:

$$
\begin{aligned}
\mathsf{WAIT}(r_s^1, r_f^1, r_s^2, r_f^2, a_s, a_f) &:= r_s^1(\widetilde{q}_1, \widetilde{l}_1, t_1).\mathsf{WAIT}_{\mathsf{S},\star}(t_1, r_s^2, r_f^2, a_s, a_f) \\
&\quad \mid r_f^1(\widetilde{q}_1, \widetilde{l}_1).\mathsf{WAIT}_{\mathsf{F},\star}(r_s^2, r_f^2, a_s, a_f) \\
\mathsf{WAIT}_{\mathsf{S},\star}(t_1, r_s^2, r_f^2, a_s, a_f) &:= r_s^2(\widetilde{q}_2, \widetilde{l}_2, t_2).\mathsf{POLICY}_{\mathsf{S},\mathsf{S}}(t_1, t_2, a_s, a_f) \\
&\quad \mid r_f^2(\widetilde{q}_2, \widetilde{l}_2).\mathsf{POLICY}_{\mathsf{S},\mathsf{F}}(t_1, a_s, a_f) \\
\mathsf{WAIT}_{\mathsf{F},\star}(r_s^2, r_f^2, a_s, a_f) &:= r_s^2(\widetilde{q}_2, \widetilde{l}_2, t_2).\mathsf{POLICY}_{\mathsf{F},\mathsf{S}}(t_2, a_s, a_f) \\
&\quad \mid r_f^2(\widetilde{q}_2, \widetilde{l}_2).\mathsf{POLICY}_{\mathsf{F},\mathsf{F}}(a_s, a_f)
\end{aligned}
$$

If the item provider 1 is able to fulfill the order, it emits a message on the input channel $r_s^1(\widetilde{q}_1, \widetilde{l}_1, t_1)$. When the web portal receives this message, the process $\mathsf{WAIT}_{\mathsf{S},\star}(t_1, r_s^2, r_f^2, a_s, a_f)$ can start. This process manages all the cases in which the item provider 1 is successful. On the other hand, if the item provider 1 is not able to fulfill the order, the web portal receives a failure message on the input channel $r_f^1(\widetilde{q}_1, \widetilde{l}_1)$, and the process $\mathsf{WAIT}_{\mathsf{F},\star}(r_s^2, r_f^2, a_s, a_f)$ is executed. This process manages all the cases in which the item provider 1 fails.

The behavior of $\mathsf{WAIT}_{\mathsf{S},\star}(t_1, r_s^2, r_f^2, a_s, a_f)$ and $\mathsf{WAIT}_{\mathsf{F},\star}(r_s^2, r_f^2, a_s, a_f)$ is quite clear: each one waits for the outcome of the item provider 2. When the web portal receives the message, it will be alternatively in one of four possible states, as shown in figure 1.

Policy Process. When all outcome messages have been collected, the web portal is able to take the appropriate actions: this is done by the following processes:

$$
\begin{aligned}
\mathsf{POLICY}_{\mathsf{S},\mathsf{S}}(t_1, t_2, a_s, a_f) &:= \overline{a_s}\,\langle\rangle \\
\mathsf{POLICY}_{\mathsf{S},\mathsf{F}}(t_1, a_s, a_f) &:= \overline{a_f}\,\langle\rangle \mid \overline{t_1}\,\langle\rangle \\
\mathsf{POLICY}_{\mathsf{F},\mathsf{S}}(t_2, a_s, a_f) &:= \overline{a_f}\,\langle\rangle \mid \overline{t_2}\,\langle\rangle \\
\mathsf{POLICY}_{\mathsf{F},\mathsf{F}}(a_s, a_f) &:= \overline{a_f}\,\langle\rangle
\end{aligned}
$$

The first process manages the case in which both of the item providers are successful; in this case, the customer is informed that its purchase order can

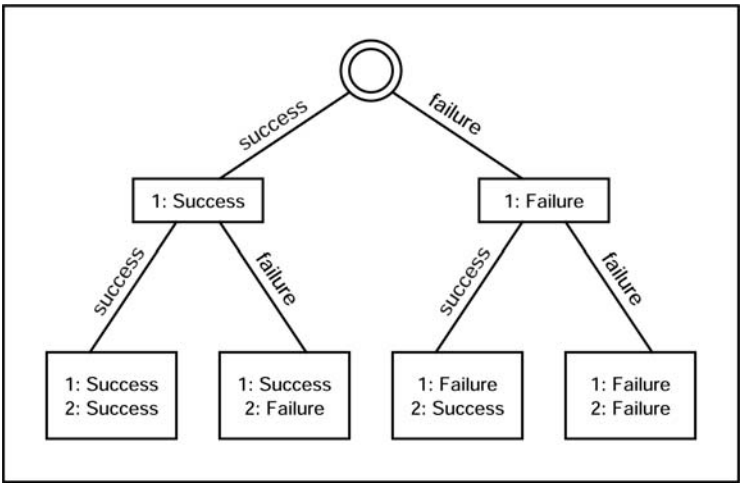


Fig. 1. Tree of Possible Executions

be fulfilled. This process receives the compensation handlers t_1 and t_2 also if it does not use them. This is because, in general, the web portal could implement a policy different from *all or nothing*.

The process $\mathsf{POLICY}_{\mathsf{S,F}}(t_1, a_s, a_f)$ manages the case where the item provider 1 is successful, and the item provider 2 is faulty: to fulfill the constraints imposed by the customer, the transaction is cancelled with the emission of the compensation request $\overline{t_1}\,\langle\rangle$. This way, the web portal implements the *all or nothing* behavior required by the customer. The case where the item provider 1 is faulty while the item provider 2 is successful is simply the dual case. The case where both the item providers are faulty is managed simply by emitting a message on the reply channel a_f, and no compensation is required.

It would be easy to generalize the algorithm to an *at least one* policy. In such a scenario, the web portal would send a success message in all the first three cases, while in the fourth one, it would send a failure message. No compensations would be required.

Item Provider

$$\mathsf{IP_i}(c_i, p_i) := (db_c)(db_p)(\mathsf{CP_i}(c_i, db_c) \mid \mathsf{PP_i}(p_i, db_p) \mid \mathsf{DBP_i}(db_c, db_p))$$

The generic i^{th} item provider receives two names as arguments, c_i and p_i. These names are global, i.e. they have been created by the $\mathsf{WORLD}()$ process. The former represent the item provider interface for the consulting service, while the latter is used to receive a buying order. When the item provider process begins its execution, it creates a pair of channels, which are used to interact with a database process. The channel db_c is used to invoke a price list consultation service exposed by the database; the channel db_p is used to emit a purchase order to the same database. After the creation of these channels, the item provider creates three sub-processes, $\mathsf{CP_i}(c_i, db_c)$, $\mathsf{PP_i}(p_i, db_p)$ and $\mathsf{DBP_i}(db_c, db_p)$. The first two processes manage the consultation and the purchase orders emitted by the customer, while the third one represents a database process.

Consulting Process

$$\begin{aligned}\mathsf{CP_i}(c_i, db_c) := {} & c_i(\widetilde{q_i}, r_i).((odbc)(\overline{db_c}\,\langle \widetilde{q_i}, odbc\rangle \mid odbc(\widetilde{q_i}, \widetilde{l_i}).\overline{r_i}\,\langle \widetilde{q_i}, \widetilde{l_i}\rangle) \\ & \mid \mathsf{CP_i}(c_i, db_c))\end{aligned}$$

$\mathsf{CP_i}(c_i, db_c)$ is a server process which receives price list read requests. It receives two names, c_i and db_c. The first name is the input channel it will listen to for a request, while the second one is the access point for the database querying service. The process $\mathsf{CP_i}(c_i, db_c)$ behaves as follows: when it receives a price check request $c_i(\widetilde{q_i}, r_i)$, containing the customer preferences $\widetilde{q_i}$ and a reply channel r_i, it duplicates itself and begins the price list reading operations. It creates a fresh name, *odbc*, and sends it to the database consulting service with the message $\overline{db_c}\,\langle \widetilde{q_i}, odbc\rangle$, which contains also the customer preferences $\widetilde{q_i}$. Then it waits for an outcome $(odbc(\widetilde{q_i}, \widetilde{l_i}))$ and forwards it to the web portal, using the reply channel $\overline{r_i}\,\langle \widetilde{q_i}, \widetilde{l_i}\rangle$.

Purchase Process

$$\begin{aligned}\mathsf{PP_i}(p_i, db_p) := p_i(\widetilde{q_i}, r_s, r_f).((odbc_s)(odbc_f)(s)(f)(t_i)(\overline{db_p}\,\langle \widetilde{q_i}, odbc_s, odbc_f, s\rangle \\ \mid \{\!|odbc_s(\widetilde{q_i}, \widetilde{l_i}, t).(\overline{f}\,\langle\rangle \mid \overline{r_s}\,\langle \widetilde{q_i}, \widetilde{l_i}, t_i\rangle \mid t_i().\bar{t}\,\langle\rangle)\ ;\ \mathbf{0}|\!\}_s \\ \mid \{\!|odbc_f(\widetilde{q_i}, \widetilde{l_i}).(\overline{s}\,\langle\rangle \mid \overline{r_f}\,\langle \widetilde{q_i}, \widetilde{l_i}\rangle)\ ;\ \mathbf{0}|\!\}_f) \mid \mathsf{PP_i}(p_i, db_p))\end{aligned}$$

The second sub-process created by the item provider $\mathsf{PP_i}(p_i, db_p)$ manages the purchase orders emitted by the web portal on behalf of the customer. When this process runs, it receives two names, p_i and db_p. The first name is the access point for the purchase service exposed by the item provider. The second name represents a private channel shared between the purchase manager and the database process that is used to invoke the purchase service exposed by the database.

The process $\mathsf{PP_i}(p_i, db_p)$ waits for a purchase request on the global channel p_i. The request contains the customer's preferences $\widetilde{q_i}$ and a pair of success/failure reply channels, r_s and r_f. When the process receives this message, it makes a copy of itself and waits for further requests, and begins the purchase managing operations. First it creates two fresh names, $odbc_s$ and $odbc_f$, which are a pair of success/failure reply channels. Then it creates two transactions, s and f, which manage the cases of success and failure of the purchase process. Those names are restricted, together with the name t_i, which will be used by the web portal to compensate a successful purchase transaction. The purchase process emits a request message $\overline{db_p}\,\langle \widetilde{q_i}, odbc_s, odbc_f, s\rangle$, which contains the customer preferences $\widetilde{q_i}$, a pair of success/failure reply channels $odbc_s$ and $odbc_f$ and the name of successful transaction manager, s. Its usefulness is shown below.

After the emission of the purchase request, the process activates the success and the failure transactions. Those transactions share a very similar behavior. Each one listens to the appropriate channel for the database outcome. This means that the transaction s waits for a success message on the $odbc_s$ channel, while the transaction f waits on the $odbc_f$ channel. In both cases, the outcome message brings the customer preferences $\widetilde{q_i}$ and the query result $\widetilde{l_i}$. Moreover, in case of success, the message contains also the name of the database transaction which manages the delivery of the requested item. This name can be used to compensate this activity, as we show below.

When one of the two specular transaction receives the purchase outcome, it triggers the other one. As the two compensation processes are the $\mathbf{0}$ process, this mechanism acts like an explicit garbage collector.[2] After receiving of the outcome, the appropriate transaction forwards it to the web portal. In case of a success, moreover, the reply message contains also a transaction name that can be used to activate the database delivery compensation. Instead of the original name received by the database process, t, a placeholder, t_i, is sent. This forbids a direct access to an internal process — the database — by an external process. In case of success, indeed, the item provider acts as a wrapper for the database

[2] This feature is not really necessary, because the other transaction remains deadlocked on a restricted name, but is useful to show how it is possible to implement a garbage collector with the compensation mechanism provided by the transactions.

compensation mechanism. When the item provider receives a compensation request, it emits the correct signal. The execution of this wrapper process lasts until the delivery operations end. When this happens the clearing signal s is emitted by the database process.

Database Process

$$
\begin{array}{lll}
\mathsf{DBP}_{\mathsf{i}}(db_c, db_p) & := & \mathsf{DBP}^{\mathsf{c}}_{\mathsf{i}}(db_c) \mid \mathsf{DBP}^{\mathsf{p}}_{\mathsf{i}}(db_p) \\
\mathsf{DBP}^{\mathsf{c}}_{\mathsf{i}}(db_c) & := & db_c(\widetilde{q}_i, odbc).((\widetilde{l}_i)(\overline{odbc}\,\langle \widetilde{q}_i, \widetilde{l}_i \rangle) \mid \mathsf{DBP}^{\mathsf{c}}_{\mathsf{i}}(db_c)) \\
\mathsf{DBP}^{\mathsf{p}}_{\mathsf{i}}(db_p) & := & db_p(\widetilde{q}_i, odbc_s, odbc_f, s).(\\
 & & (\widetilde{l}_i)(t)(\overline{odbc_s}\,\langle \widetilde{q}_i, \widetilde{l}_i, t \rangle \mid (\{\!|dlv() \,;\, cmp()|\!\}_t.\overline{s}\,\langle\rangle) \\
 & & \oplus \overline{odbc_f}\,\langle \widetilde{q}_i, \widetilde{l}_i \rangle) \mid \\
 & & \mathsf{DBP}^{\mathsf{p}}_{\mathsf{i}}(db_p))
\end{array}
$$

The third sub-process created by the item provider is $\mathsf{DBP}_{\mathsf{i}}(db_c, db_p)$. This process simulates the behavior of a DBMS. In particular, it exposes two kinds of services: the price list consultation and the purchase order. It receives a pair of private channels db_c and db_p and shares them with the item provider. The former is the access point on which it will wait for a price list consultation, while the latter is used to listen for purchase orders.

Two distinct sub-processes manage the two activities mentioned above. The process $\mathsf{DBP}^{\mathsf{c}}_{\mathsf{i}}(db_c)$ manages the price list consultation. When it receives a request message, it creates a duplicate. The request message carries the customer's preferences $\widetilde{q}_i$ and a reply channel $odbc$. Now, the database simply creates a new name, $\widetilde{l}_i$, which represents the outcome of the query executed on the DBMS, and sends it back to the item provider. This operation simulates a database query, and can never fail; if a query produces no results, its outcome is correctly encoded on the fresh name $\widetilde{l}_i$.

The process $\mathsf{DBP}^{\mathsf{p}}_{\mathsf{i}}(db_p)$ deals with purchase orders, delivery of goods and any compensation requested by the web portal. At first, the process receives a purchase order from the item provider. This request contains the item preferences $\widetilde{q}_i$, a pair of success/failure reply channels $odbc_s$ and $odbc_f$ and a transaction name s. When it receives the request, the process makes a copy of itself, creates a new name $\widetilde{l}_i$, which represents the query outcome, and decides if the customer's request can be fulfilled or it must be rejected. To do so, it uses a constructor called *internal choice*, which is represented with the symbol $\oplus$. This means that only one process is chosen, while the other is simply discharged. This behavior is easily encodable in terms of parallel composition, message passing and restriction only. We introduce this notation just for brevity.

If the database purchase process is not able to fulfill the order, it simply emits a message $\overline{odbc_f}\,\langle \widetilde{q}_i, \widetilde{l}_i \rangle$ on the failure reply channel $odbc_f$, and forgets everything about the transaction. The message contains the customer's preferences $\widetilde{q}_i$ and the outcome of the query, represented by $\widetilde{l}_i$. In case of item availability, the behavior of the database process is more complex. On the successful channel $odbc_s$, it emits a reply message, which contains the customer preferences $\widetilde{q}_i$, the outcome of the query $\widetilde{l}_i$ and the compensation handler t. In parallel with the reply message emission, the database process begins to execute the delivery

operations. From this moment on, the web portal can emit the compensation request while the delivery action is being performed.

5 Conclusion

In this paper we introduced $\texttt{web}\pi_\infty$, a simple extension of the π-calculus with untimed long running transactions. We discussed the notion of orchestration without considering time constraints. This way we focused on information flow, message passing, concurrency and resource mobility, keeping the model small and simple. We motivated the underlying theory we rely on, the π-calculus, in terms of expressiveness and suitability to composition and orchestration purposes. To show the strength of the language we also proposed a formalization of an e-commerce transactional scenario.

This work contributes a simple, concise yet powerful and expressive language, with a solid semantics that allows formal reasoning. The language shows a clear relation with the π-calculus, and the actual encoding of it with the π-calculus is a feasible task, while it would be quite harder to to get such an encoding for XLANG and other Web Services composition languages.

A possible extension of this work could be generalizing the transaction policy and proving constraints satisfaction. Other future developments building on the results achieved in this paper include software tools for static analysis of programs using composition and orchestration. A useful result that could stem from this work could be streamlined definitions of syntax and semantics of web services composition languages, to get a simpler way to model involved transaction behaviors. On a more theoretical side, another research direction could be extending the calculus with a notion of time while keeping it simple. The overall goal we have is to allow for improvement of quality and applicability of real orchestration languages.

Acknowledgments. The authors would like to acknowledge Cosimo Laneve, Enrico Tosi and Andrea Carpineti for their comments and contributions to the paper.

References

1. T. Andrews, F. Curbera et al. Web Service Business Process Execution Language, Working Draft, Version 2.0, 1 December 2004.
2. M. Berger. Basic Theory of Reduction Congruence for Two Timed Asynchronous π-calculi. In *CONCUR'04: Proceedings of the 15th International Conference on Concurrency Theory, LNCS* 3170, pages 115-130, Springer-Verlag, 2004.
3. M. Berger. Towards Abstractions for Distributed Systems. PhD Thesis, Imperial College, London, 2002.
4. M. Berger, K. Honda, The Two-Phase Commit Protocol in an Extended π-Calculus. In *EXPRESS '00: Proceedings of the 7th International Workshop on Expressiveness in Concurrency, ENTCS* 39.1, Elsevier, 2000.

5. Microsoft BizTalk Server. [http://www.microsoft.com/biztalk/default.asp], Microsoft Corporation.
6. R. Bruni, C. Laneve, U. Montanari. Orchestrating Transactions in Join Calculus. In *CONCUR'02: Proceedings of the 13th International Conference on Concurrency Theory*, *LNCS* 2421, pages 321-337, Springer-Verlag, 2003.
7. R. Bruni, H. Melgratti, U. Montanari. Theoretical Foundations for Compensations in Flow Composition Languages. To appear in POPL2005.
8. L. Bocchi, C. Laneve, G. Zavattaro. A Calculus for Long-running Transactions. In *FMOODS'03: Proceedings of th 6th IFIP International Conference on Formal Methods for Open-Object Based Distributed Systems*, *LNCS* 2884, pages 124-138, Springer-Verlag, 2003.
9. M. Butler, C. Ferreira. An Operational Semantics for StAC, a Language for Modelling Long-running Business Transactions. In *COORDINATION'04: Proceedings of the 6th International Conference on Coordination Models and Languages*, *LNCS* 2949, pages 87-104. Springer-Verlag, 2004.
10. M. Chessel, D. Vines, C. Griffin, V. Green, K. Warr. Business Process Beans: System Design and Architecture Document. Technical report.IBM UK Laboratories. January 2001.
11. E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Services Description Language (WSDL 1.1). [www.w3.org/TR/wdsl], W3C, Note 15, 2001.
12. C. Guidi, R. Lucchi, M.Mazzara. A Formal Framework for Web Services Coordination. 3rd International Workshop on Foundations of Coordination Languages and Software Architectures, London 2004.
13. T. Hoare. Long-Running Transactions. Powerpoint presentation [research.microsoft.com/]
14. N. Kavantzas, G. Olsson, J. Mischkinsky, M. Chapman. Web Services Choreography Description Languages. [otn.oracle.com/tech/webservices/ htdocs/spec/cdl_v1.0.pdf]
15. C. Laneve, G. Zavattaro. Foundations of Web Transactions. To appear in FOSSACS 2005.
16. F. Leymann. Web Services Flow Language (WSFL 1.0). [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf], Member IBM Academy of Technology, IBM Software Group, 2001.
17. M. Little. Web Services Transactions: Past, Present and Future. [www.idealliance.org/papers/dx_xml03/ html/abstract/05-02-02.html]
18. M.Mazzara, R.Lucchi. A Framework for Generic Error Handling in Business Processes. First International Workshop on Web Services and Formal Methods (WS-FM), Pisa 2004.
19. R. Milner. Function as Processes. *Mathematical Structures in Computer Science*, 2(2):119-141, 1992.
20. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
21. R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77. Academic Press, 1992.
22. J. Parrow, B. Victor. The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes. In *LICS'98: Proceedings of the 13th Symposium on Logic in Computer Science*, IEEE Computer Society Press.
23. C. Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, October 2003 (Vol.36, No 10), pages 46-52.
24. D. Sangiorgi, D. Walker. *The π-calculus: a Theory of Mobile Processes*, Cambridge University Press, 2001.

25. S. Thatte. XLANG: Web Services for Business Process Design. [http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm], Microsoft Corporation, 2001.
26. Universal Description, Discovery and Integration for Web Services (UDDI) V3 Specification. [http://uddi.org/pubs/uddiv3.htm]
27. W.M.P. van der Aalst. Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype". [mitwww.tm.tue.nl/staff/wvdaalst/publications/pi-hype.pdf]
28. Workflow Management Coalition - http://www.wfmc.org/
29. WS-Coordination Specification [www-106.ibm.com/developerworks/library/ws-coor/]
30. WS-Transaction Specification [www-106.ibm.com/developerworks/webservices/library/ws-transpec/]
31. B. Weikum, G. Vossen. *Transactional Information Systems.* Morgan Kaufmann, 2002.

A Correct Abstract Machine for Safe Ambients*

Daniel Hirschkoff[1], Damien Pous[1], and Davide Sangiorgi[2]

[1] ENS Lyon, France
[2] Università di Bologna, Italy

Abstract. We describe an abstract machine, called GcPan, for the distributed execution of Safe Ambients (SA), a variant of the Ambient Calculus (AC).

Our machine improves over previous proposals for executing AC, or variants of it, mainly through a better management of special agents (*forwarders*), created upon code migration to transmit messages to the target location of the migration.

We establish the correctness of our machine by proving a weak bisimilarity result with a previous abstract machine for SA, and then appealing to the correctness of the latter machine.

More broadly, this study is a contribution towards understanding issues of correctness and optimisations in implementations of distributed languages encompassing mobility.

Introduction

In recent years there has been a growing interest for core calculi encompassing distribution and mobility. In particular, these calculi have been studied as a basis for programming languages. Examples include Join [9], Nomadic Pict [19], Kells [2], Ambients [6], Klaim [16].

In this paper we study issues of correctness and optimisations in implementations of such languages. Although our technical work focuses on Ambient-based calculi, we believe that the techniques can be of interest for the study of other languages: those mentioned above, and more broadly, distributed languages with mobility.

The underlying model of the Ambient calculus is based on the notion of *location*, called ambient. Terms in Ambient-based calculi describe configurations of locations and sub-locations, and computation happens as a consequence of movement of locations. The three primitives for movement allow: an ambient to enter another ambient (In), an ambient to exit another ambient (Out), a process to dissolve an ambient boundary and obtain access to its content (Open).

A few distributed implementations of Ambient-like calculi have appeared [10, 11, 17]. The study of implementations is important to understand the usefulness of the model from a programming language point of view. Such studies have

* Work supported by european FET - Global Computing project Profundis.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 17–32, 2005.

shown that the `open` primitive, the most original one in the Ambient model, is also the most difficult to implement.

Another major difficulty for a distributed implementation of an ambient-like language is that each movement operation involves ambients on different hierarchical levels. For instance, the ambients affected by an `out` operation are the moving ambient and its initial and final parents; before the movement is triggered, they reside on three different levels. In [4, 5] locks are used to synchronise all ambients affected by a move. In a distributed setting, however, this lock-based policy can be expensive. For instance, the serialisations introduced diminish the parallelism of the whole system. In [10] the synchronisations are simulated by means of protocols of asynchronous messages. The abstract machine PAN [11] has two main differences. The first is that the machine executes typed Safe Ambients [13] (SA) rather than untyped Ambients. Typed SA is a variant of the original calculus that eliminates certain forms of interference in ambients, called grave interferences. These arise when an ambient tries to perform two different movement operations at the same time, as for instance $n[\texttt{in}\ h.P \mid \texttt{out}\ n.Q \mid R]$. The second reason for the differences in PAN is the separation

between the logical structure of an ambient system and its physical distribution. Exploiting this, the interpretation of the movement associated to the capabilities is reversed: the movement of the `open` capability is physical, that is, the location of some processes changes, whereas that of `in` and `out` is only logical, that is, some hierarchical dependencies among ambients may change, but not their physical location. Intuitively, IN and OUT reductions are acquisition of access rights, and OPEN is exercise of them.

In PAN, the implementation of OPEN exploits *forwarders* – a common technique in distributed systems – to retransmit messages coming from the inside of an ambient that has been opened. These lead to two major problems:

- *persistence:* along the execution of the PAN, some forwarders may become useless, because they will never receive messages. However, these are never removed, and thus keep occupying resources (very often in examples, the ambients opened are leaves, and opening them introduces useless forwarders).
- *long communication paths:* as a consequence of the opening of several ambients, forwarder chains may be generated, which induce a loss of performance by increasing the number of network messages.

In this paper, we introduce GCPAN, an abstract machine for SA that is more efficient than PAN. The main improvements are achieved through a better management of forwarders, which in the GCPAN enjoy the following properties:

- *finite lifetime:* we are able to predict the number of messages that will be transmitted by a forwarder, so that we can remove the latter once these messages have all been treated;
- *contraction of forwarder chains:* we enrich the machine with a mechanism that allows us to implement a union-find algorithm to keep forwarder chains short, so as to decrease the number of messages exchanged.

The basis of the algorithms we use (e.g., Tarjan's union-find algorithm [18]) are well-known. However, adapting them to Ambient-like calculi requires some care, due to the specific operations proposed by these languages.

We provide a formal description of our machine, and we establish a weak bisimilarity result between Pan and GcPan. We then rely on the correctness of the Pan w.r.t. the operational semantics of SA, proved in [11], to deduce correctness w.r.t. SA.

An original aspect of our analysis w.r.t. the proof in [11] is that we compare two abstract machines, rather than an abstract machine and a calculus. This involves reasoning modulo 'administrative reduction steps' on both sides of the comparison to establish the bisimulation results. However, the fact that, in the GcPan, chains of forwarders are contracted using the union-find algorithm prevents us from setting up a tight correspondence between the two machines. This moreover entails that standard techniques for simplifying proofs of weak bisimilarity results (such as those based on the expansion preorder and up-to techniques) are not applicable. As a consequence, the bisimulation proof in which the two machines are compared is rather long and complex. Still, deriving the correctness w.r.t. SA through a comparison with Pan is simpler than directly proving the correctness of our machine w.r.t. SA. This holds because Pan and GcPan are both abstract machines, with a number of common features.

We believe that our study can also be of interest outside Ambient-based formalisms. For instance, the use of forwarders is common in distributed programming (see e.g. [7, 9]). However, little attention has been given to formal specification and correctness proofs of the algorithms being applied. The formalisation of the management and optimisations of forwarders that we provide and, especially, the corresponding correctness proof should be relevant elsewhere.

Outline of the paper. We present the design principles of the GcPan in Sect. 2. We then give the formal definition of the machine in Sect. 3, and describe the correctness proof in Sect. 4. Sect. 5 gives concluding remarks.

1 The Machine: Design Principles

We introduce the Safe Ambients (SA) calculus [14] and the Pan abstract machine [11]. We then present our ideas to remedy to some inefficiencies of Pan.

1.1 Safe Ambients

The SA calculus is an extension of the Mobile Ambients calculus [6] in which a tighter control of movements is achieved though *co-capabilities*. The four main reduction rules are:

$$
\begin{array}{rcll}
a[\mathtt{in}\ b.P \mid Q] \mid b[\overline{\mathtt{in}}\ b.R \mid S] & \longmapsto & b[a[P \mid Q] \mid R \mid S] & (IN) \\
b[a[\mathtt{out}\ b.P \mid Q] \mid \overline{\mathtt{out}}\ b.R \mid S] & \longmapsto & a[P \mid Q] \mid b[R \mid S] & (OUT) \\
\mathtt{open}\ b.P \mid b[\overline{\mathtt{open}}\ b.Q \mid R] & \longmapsto & P \mid Q \mid R & (OPEN) \\
\langle M \rangle \mid (x)P & \longmapsto & P\{M/x\} & (COM)
\end{array}
$$

Co-capabilities and the use of types (notably those for *single-threadedness*) make it possible to exclude *grave interferences*, that is, interferences among processes that can be regarded as programming errors (as opposed to an expected form of non-determinism). A single-threaded (ST) ambient can engage in at most one external interaction, at any time its local process has only one *thread* (or active capability). In the sequel, when mentioning *well-typed* processes, this will be a reference to the type system of [14]. One of the benefits of the absence of grave interferences is that it is possible to define simpler abstract machines and implementations for ambient-based calculi: some of the synchronisation mechanisms needed to support grave interferences in a distributed setting [10] are not necessary (other possible benefits of SA, concerning types and algebraic theory, are discussed in [14]).

The modifications that yield typed SA have also computational significance. In the Mobile Ambient interaction rules, an ambient may enter, exit, or open another ambient. The latter ambient undergoes the action; it has no control on *when* the action takes place. In SA this is rectified: a movement is triggered only if both participants agree. Further, the modifications do not seem to exclude useful programming examples. In some cases the SA programs can actually be simpler, due to the tighter control over interferences. We refer to [14] for details.

1.2 The Pan

The Pan [11] separates the logical distribution of ambients (the tree structure given by the syntax) from their physical distribution (the actual sites they are running on). An ambient named n is represented as as a *located agent* h: $n[P]_k$, where h is the physical location, k the location of the parent of the ambient, and P is its local process. There can be several ambients named n, but a location h uniquely identifies an ambient. The physical distribution is flat, so that the SA process $a[b[c[] \mid P] \mid d[Q]]$ is represented by the parallel composition (also called *net*) h_1: $a[]_{\texttt{root}} \parallel h_2$: $b[P]_{h_1} \parallel h_3$: $c[]_{h_2} \parallel h_4$: $d[Q]_{h_1}$. For the sake of simplicity, and when this does not lead to confusion, we sometimes use a to refer to the location of an ambient named a.

In the Pan, an ambient has only access to its parent location and to its local process: it does not know its sub-ambients. This simplifies the treatment of ambient interactions: communication between locations boils down to the exchange of asynchronous messages (while manipulating lists of child locations would mean setting many synchronisation points along computation).

In the Pan an ambient interaction is decomposed into three steps: an ambient that wants to move first sends a *request message* to its parent and enters in *wait state*. The father ambient then looks for a valid *match* to this request, and, upon success, sends appropriate *completion messages* back, using the location names contained in the request messages. The scenarios corresponding to the three kinds of movement are depicted in Fig. 1, where white squares (resp. grey squares) represent locations (resp. locations in wait state), and arrows indicate messages.

We remark that, for In and Out moves, the decision is taken by the parent of the moving ambient. Also note that in the Out move, the grandparent, that

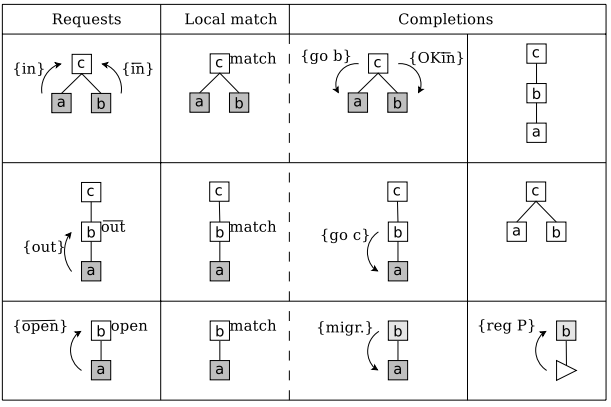


Fig. 1. Simulation of the SA reductions by the PAN

actually receives a new child, does not take part in any interaction: this follows the design of PAN, in which the relation between parent and child 'goes upwards'. Moreover, performing an IN or OUT movement does not trigger any physical migration in the PAN, only the logical distribution of ambients is affected.

On the other hand, in an OPEN move, the code of the process that is local to the ambient being opened (a in Fig. 1) is sent to the parent ambient (via a `reg` message). Indeed, b has no access to its children, and hence it cannot inform them to send their requests to b instead of a. The solution adopted in the PAN is to use *forwarders*: any message reaching a will be routed to b by an agent represented by a triangle in Fig. 1, and denoted by '$h \rhd k$' in the following (h and k being the locations associated respectively to a and b).

The logical structure of the PAN is hence a tree whose nodes are either located ambients or forwarders. Request (resp. completion) messages are transmitted upwards (resp. downwards) along the tree.

The design ideas that we have exposed entail two major drawbacks in the execution of the PAN: persistence of forwarders (even when there are no sub-ambients and therefore no message can reach the forwarder), and long forwarder chains which generate an overload in terms of network traffic.

1.3 The GCPAN

We now explain how we address the problems exposed above, and what influence our choices have on the design of the PAN.

Counters. A forwarder can be thought of as a *service* provided to the children of an opened ambient. Our aim is to be able to bring this service to an end once there are no more children using it. At the same time, we wish to preserve asynchrony in the exchange. For this, GCPAN agents are enriched with a kind of reference counter. Forwarders have a finite lifetime, at the end of which they are garbage collected. The lifetime of a forwarder intuitively corresponds to the number of locations that point to it. A counter is decremented each time a

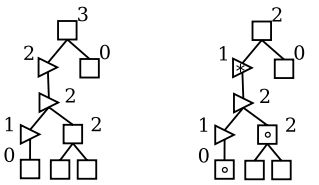


Fig. 2. Depth and local counting

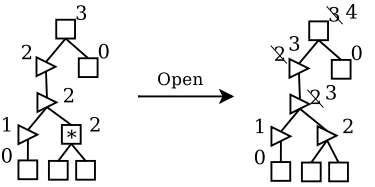


Fig. 3. Problem with depth counting

message is forwarded. If the counter is zero, then the forwarder is a leaf in the logical structure of the net and can safely be removed.

We can think of two ways of associating a lifetime to a forwarder (Fig. 2):

- *(depth counting)* The most natural idea is probably to decorate each located ambient with the number of immediate sub-ambients it has. In doing this, we ignore forwarders, because request messages that are routed via forwarders can only be emitted by located ambients. This solution seems however difficult to implement, due to the asynchrony in the model. This is illustrated by Fig. 3: if the ambient marked '$*$' is opened, the counters along the whole forwarders chain should be updated before any of the children can send a message.
- *(local counting)* In our approach, we only consider the *immediate* children of a location (hence the name *local*), including forwarders. As a consequence, we may well have the situation where several sub-ambients are 'hidden' under a forwarder, so that the counter at a given location has no direct relationship with the number of sub-ambients. The difficulty described above does not arise in this setting: the forwarders chain remains unaffected by the opening, a located ambient becomes a forwarder, and this does not affect the counting.

Synchronisation Problems and Blocked Forwarders. In the local approach, one has to be careful in transmitting request messages. Consider for instance the forwarder marked '$*$' on the right of Fig. 2: each ambient marked with a circle can send a request message. The intermediate forwarder cannot forward directly these two requests, since the '$*$-forwarder' is willing to handle only one message. In the GCPAN, an agent can send only one message to a given forwarder, and whenever this message is sent, the agent commits to relocate itself if the agent it was talking to turns out to be a forwarder.

Implementing this policy is easy for located ambients, that enter a wait state just after emitting a request message. We only have to decorate completion messages with the appropriate information for relocation. For forwarders, we need to devise a similar blocking mechanism: once a forwarder has transmitted a request message, it enters a blocked state and waits for a $\mathsf{go}_{\triangleright}$ completion message, which contains the name of the location to which the next request should be forwarded. Fig. 4 illustrates this (blocked forwarders are represented by reversed, grey triangles): message $\{N\}$ is emitted by the grey ambient, and then routed towards the parent location, which has the effect of blocking forwarders along

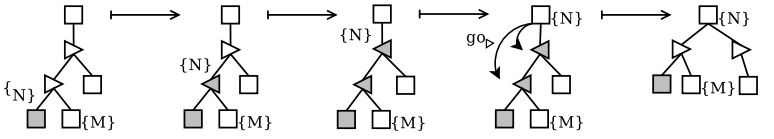


Fig. 4. Relocation of forwarders

the way. When $\{N\}$ reaches the parent ambient, $\mathtt{go}_{\triangleright}$ messages are generated so that forwarders can resume execution, just below the parent ambient. This way, short communication paths between locations are maintained: at the end of the scenario, message $\{M\}$ is closer to its destination, without having been routed yet. The technique we use is based on Tarjan's union-find algorithm [18].

Remark 1 (Communication protocols). We comment on the way messages are transmitted in the GcPan:

- *(race situations)* Having blocked forwarders leads to race situations: consider the scenario of Fig. 4, where messages $\{M\}$ and $\{N\}$ are sent at the bottom of a chain of forwarders. When $\{N\}$ goes through the lowest forwarder, $\{M\}$ has to wait for the arrival of the former at the top of the chain, so that a $\mathtt{go}_{\triangleright}$ message is emitted to rearrange forwarders (following the union-find algorithm). The loss, from $\{M\}$'s point of view, is limited: once $\{N\}$ has entered the parent location, $\{M\}$ can reach the latter in three steps (the $\mathtt{go}_{\triangleright}$ message plus two routing steps).
- *(relocation strategy)* In the GcPan, the ambient that sits at the end of a forwarder chain broadcasts a relocation message ($\mathtt{go}_{\triangleright}$) to all blocked forwarders in the chain. In a previous version of our machine, this message was propagated back along the chain, unblocking the forwarders in a sequential fashion. We prefer the current solution because it brings more asynchrony (race situations introduced a delay of $n + 2$ because the relocation message had to go trough the whole chain in order to unblock all forwarders). On the other hand, request messages carry more information in our approach (we need to record the set of forwarders that have been crossed). However, in practise, we observe that long chains of forwarders are very unlikely to be produced in our machine, thanks to the contraction mechanism we adopt. Consequently, such messages have in most cases a rather limited size.

Updating Counters Along SA Movements. Going back to the GcPan transitions corresponding to the basic SA moves (the *match* transitions of Fig. 1), we need to be able to maintain coherent counters along the three kinds of movement. This is achieved as follows (the names we use correspond to Fig. 1):

In: The overall result of the transition will be that c decrements its counter, and b increments its counter upon reception of the $\mathtt{OK}\overline{\mathtt{in}}$ completion.

Open: counters do not need to be modified.

Out: in the Pan, the match between the capability and the co-capability is done at b, and the grandparent c is not aware of the movement. In the

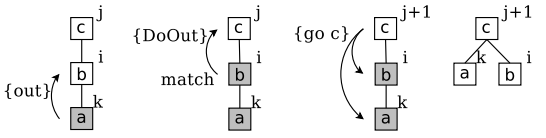


Fig. 5. Counters along an Out move: first approach

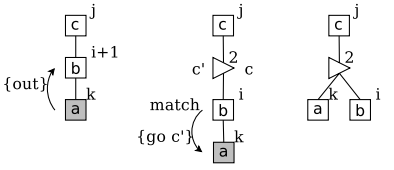


Fig. 6. Counters along an Out move: our approach

GcPan, b decrements its counter, a is unaffected, but, a priori, c has to increment its counter, since it receives a new child, a.

A possibility would be to let b pass the control on the move along to c, that is then in charge of sending the completion messages: this solution is represented in Fig. 5. Adopting this protocol means introducing a new kind of message in the machine (message `DoOut` in Fig. 5, from parent to grandparent), and having two agents in wait state (the child and the parent) while the control is at the grandparent location.

We chose a different solution, that does not use an additional kind of message and in which interaction is more local and asynchronous. It is depicted in Fig. 6: at b, we create a new forwarder that collects the parent (b) and the child (a) under a unique agent, so that the grandparent counter does not need to be updated. It may seem rather counterproductive to add a new form of forwarder creation this way, considering that our goal in designing the GcPan is precisely to erase as many forwarders as possible. We can however observe that:

- the created forwarder has a lifetime of 2, which is short;
- from the point of view of the implementation, the forwarder is created on the parent site, so that the extra communication between the parent and the forwarder will be local.

2 Formal Definition of the Machine

2.1 GcPan Nets

The syntax of the terms of the GcPan (referred to as GcPan *nets*, or simply *nets*) is presented on Table 1. Agents in the GcPan are either located ambients (h^i: $n[P]_k$ is the ambient $n[P]$ running at h, whose parent is located at k), blocked or running forwarders ($h \triangleleft^i$ is a blocked forwarder at h, while $h \triangleright^i k$ is

Table 1. GcPan Syntax

$a, b, m, n, .. \in Names \qquad h, k, .. \in Locations \qquad p, q, .. \in Names \cup Locations$
$i, j, .. \in \mathbb{N} \qquad x, y, .. \in Variables$

Networks:

$$\begin{array}{rll} A := & \mathbf{0} & \text{(empty net)} \\ \mid & Agent & \text{(agent)} \\ \mid & h\{Msg\} & \text{(emission)} \\ \mid & A_1 \parallel A_2 & \text{(composition)} \\ \mid & (\boldsymbol{\nu} p)A & \text{(restriction)} \end{array}$$

$$\begin{array}{rll} Agent := & h \rhd^i k & \text{(forwarder)} \\ \mid & h \lhd^i & \text{(blocked forwarder)} \\ \mid & h^i\colon n[P]_k & \text{(located ambient)} \end{array}$$

$$\begin{array}{rll} Msg := & req/E & \text{(request)} \\ \mid & compl & \text{(completion)} \end{array}$$

$$\begin{array}{rll} req := & \texttt{in}\ n, h & \text{(agent at } h \text{ wants to enter } n) \\ \mid & \overline{\texttt{in}}\ n, h & \text{(agent at } h\text{, named } n\text{, accepts entrance)} \\ \mid & \texttt{out}\ n, h & \text{(agent at } h \text{ wants to leave } n) \\ \mid & \overline{\texttt{open}}\ n, h & \text{(agent at } h\text{, named } n\text{, accepts opening)} \end{array}$$

$$\begin{array}{rll} compl := & \texttt{go}\ h & \text{(request completed, go to } h) \\ \mid & \texttt{go}_{\rhd}\ h & \text{(relocate forwarder to } h) \\ \mid & \texttt{OK}\overline{\texttt{in}}\ h & \text{(request } \overline{\texttt{in}} \text{ completed, go to } h) \\ \mid & \texttt{mig}\ h & \text{(request } \overline{\texttt{open}} \text{ completed, migrate to } h) \\ \mid & \texttt{reg}^s\ P & \text{(add } P \text{ to the local processes)} \end{array}$$

Processes:

$$P := \mathbf{0} \mid P_1 \mid P_2 \mid (x)P \mid (\boldsymbol{\nu} n)P \mid X \mid M.P \mid \mathrm{rec}\, X.P \mid M[P] \mid \mathrm{wait}.P \mid \langle M \rangle \mid \{req\}$$
$$M := x \mid n \mid \overline{\texttt{out}}\ M \mid \texttt{in}\ M \mid \texttt{open}\ M \mid \overline{\texttt{in}}\ M \mid \overline{\texttt{open}}\ M \mid \texttt{out}\ M$$

willing to transmit messages from h to k). In the three cases, the superscript $i \in \mathbb{N}$ represents the value of the agent counter.

E denotes a list of locations. A message of the form $k\{req/E\}$ denotes the request req, located at k, and having been transmitted through the locations contained in E. $k\{req\}$ is an abbreviation for $k\{req/[]\}$, and we write $h :: E$ to denote the list obtained by adding h to E. Reception $((x)P)$ and restriction $((\boldsymbol{\nu} x)P)$ are binders. Given a process P, we let $FL(P)$ stand for the set of free locations of P. An occurrence in a process P is *guarded* if it appears under a prefix or a reception. We suppose that in every process of the form rec $X.P$, all occurrences of X in P are guarded.

Other aspects of the syntax of messages are explained in Subsection 3.2.

The definition of structural congruence, $\equiv$, is mostly standard, and omitted. The only peculiarity is that $\equiv$ does not allow a name restriction to be extruded out of a located ambient in a transparent way: the net $h\colon n[(\boldsymbol{\nu} m)(\texttt{in}\ m)]_k$ is not equivalent to $(\boldsymbol{\nu} m)h\colon n[\texttt{in}\ m]_k$. Such a transformation is handled using reduction, and not as a structural congruence rule, because at the level of implementation, generating names that are fresh even for possibly distant agents involves a nontrivial distributed protocol.

The GcPan (resp. Pan) encoding of a SA process P is written $[\![P]\!]_{gc}$ (resp. $[\![P]\!]$). $[\![P]\!]$ is defined in [11], and $[\![P]\!]_{gc}$ is defined as follows:

Definition 1 (Translation from SA to GcPan). *Given an SA process P, we define:* $[\![P]\!]_{gc} \triangleq \texttt{root}^0\texttt{:}\ \texttt{rootname}[P]_{\texttt{rootparent}}$.

2.2 Reduction Rules

Fig. 7 presents the operational semantics of GcPan nets. The following explanations should help in reading the rules and understanding how they implement the ideas we have discussed above.

Form of the Rules: Rules for emission of request messages and for local reductions have the shape $P \xrightarrow[h:n]{k} P' \gg^i M$, to denote the fact that process P, running in ambient n at location h, may liberate message M and evolve into process P', k being the parent location of h. Integer i decorating $\gg$ records the increment that has to be brought to h's counter (cf. rule Proc-Agent below). $\gg$ is an abbreviation of $\gg^0$. When n, h or k are unimportant, we replace them with '-'. We do the same in the rules for consumption of completion messages, when the parent location of a located ambient is not important.

In rule Local-Com, $P\{x\backslash M\}$ denotes process P in which x is substituted with M. In rule Loc-Rcv, we use the following notations, for $E = [e_1; \ldots; e_i]$: $E\{M\}$ stands for $e_1\{M\} \parallel \ldots \parallel e_i\{M\}$, and $\#E$ is i.

Six Kinds of Rules govern the behaviour of a GcPan net, according to the way SA transitions are implemented in our model.

- Before being able to start interacting, a process might have to allocate new resources for the creation of new names and for the spawning of new ambients: this is handled by the rules for *creation*.
- The translation of a prefixed SA process starts with emitting a request for interaction, which is expressed by the corresponding four rules for *emission of request messages*.
- Request messages are transmitted through forwarders and reach their destination location via the rules for *transmission of request messages*.
- *Local reductions* describe the steps that correspond to SA transitions. Such reductions do the matching between a capability and the corresponding co-capability, and generate completion messages.
  Notation $\ggg$ is introduced similarly to $\gg$, in order to handle the Out movement, that is achieved using rule Proc-Agent'. The subscript k' denotes the source location of the created forwarder (we have to adopt a special treatment for this case because the newly created forwarder is outside the 'active location').
- Some rather standard *inference rules* are used to transform a local reduction into a transition of the whole GcPan net.
  The premises about unguarded ambients insure that all sub-ambients of an ambient are activated as soon as possible (rule New-Locamb), before any local reduction takes place — here we exploit the fact that recursions are guarded, otherwise there could be an infinite number of ambients to create.

Creation

$$[\textsc{New-Locamb}]\quad h^i\colon m[n[P]\,|\,Q]_{h'} \longmapsto h^{i+1}\colon m[Q]_{h'} \parallel (\boldsymbol{\nu} k)(k^0\colon n[P]_h) \quad k \notin FL(P)$$

$$[\textsc{New-Res}]\quad h^i\colon m[(\boldsymbol{\nu} n)P]_k \longmapsto (\boldsymbol{\nu} n)(h^i\colon m[P]_k)$$

Emission of request messages

$$[\textsc{Req-In}]\quad \texttt{in}\ m.P \xrightarrow[h:-]{k} \mathrm{wait}.P \gg k\{\texttt{in}\ m, h\}$$

$$[\textsc{Req-Coin}]\quad \overline{\texttt{in}}\ n.P \xrightarrow[h:n]{k} \mathrm{wait}.P \gg k\{\overline{\texttt{in}}\ n, h\}$$

$$[\textsc{Req-Out}]\quad \texttt{out}\ m.P \xrightarrow[h:-]{k} \mathrm{wait}.P \gg k\{\texttt{out}\ m, h\}$$

$$[\textsc{Req-Coopen}]\quad \overline{\texttt{open}}\ n.P \xrightarrow[h:n]{k} \mathrm{wait}.P \gg k\{\overline{\texttt{open}}\ n, h\}$$

Transmission of request messages

$$[\textsc{Fw-Send}]\quad h \rhd^{i+1} k \parallel h\{req/E\} \longmapsto h \lhd^i \parallel k\{req/h::E\}$$

$$[\textsc{Fw-SendGC}]\quad h \rhd^1 k \parallel h\{req/E\} \longmapsto k\{req/E\}$$

$$[\textsc{Fw-Reloc}]\quad h \lhd^i \parallel h\{\texttt{go}_{\rhd}\ k\} \longmapsto h \rhd^i k$$

$$[\textsc{Loc-Rcv}]\quad h^{i+1}\colon n[P]_k \parallel h\{req/E\} \longmapsto h^{i+\#E}\colon n[P\,|\,\{req\}]_k \parallel E\{\texttt{go}_{\rhd}\ h\}$$

Local reductions

$$[\textsc{Local-Com}]\quad \langle M\rangle\,|\,(x).P \xrightarrow[-:-]{} P\{x\backslash M\} \gg \mathbf{0}$$

$$[\textsc{Local-In}]\quad \{\texttt{in}\ n, h\}\,|\,\{\overline{\texttt{in}}\ n, k\} \xrightarrow[h':-]{} \mathbf{0} \gg^1 h\{\texttt{go}\ k\} \parallel k\{\texttt{OK}\overline{\texttt{in}}\ h'\}$$

$$[\textsc{Local-Out}]\quad \{\texttt{out}\ n, h\}\,|\,\overline{\texttt{out}}\ n.P \xrightarrow[-:n]{} P \ggg_{k'} h\{\texttt{go}\ k'\}$$

$$[\textsc{Local-Open}]\quad \texttt{open}\ n.P\,|\,\{\overline{\texttt{open}}\ n, h\} \xrightarrow[h':-]{} \mathrm{wait}.P \gg^1 h\{\texttt{mig}\ h'\}$$

Inference rules

$$[\textsc{Proc-Agent}]\quad \frac{P \xrightarrow[h:n]{k} P' \gg^s M \qquad Q \text{ has no unguarded ambient}}{h^i\colon n[P\,|\,Q]_k \longmapsto h^{i+s}\colon n[P'\,|\,Q]_k \parallel M}$$

$$[\textsc{Proc-Agent'}]\quad \frac{P \xrightarrow[h:n]{k} P' \ggg_{k'} M \quad Q \text{ has no unguarded ambient}, k' \notin FL(P|Q)}{h^i\colon n[P\,|\,Q]_k \longmapsto (\boldsymbol{\nu} k)'(k' \rhd^2 k \parallel h^i\colon n[P'\,|\,Q]_{k'} \parallel M)}$$

$$[\textsc{Par-Agent}]\quad \frac{A \longmapsto A'}{A \parallel B \longmapsto A' \parallel B} \qquad \frac{A \longmapsto A'}{(\boldsymbol{\nu} p)A \longmapsto (\boldsymbol{\nu} p)A'}\quad [\textsc{Res-Agent}]$$

$$[\textsc{Struct-Cong}]\quad \frac{A \equiv A' \qquad A' \longmapsto A'' \qquad A'' \equiv A'''}{A \longmapsto A'''}$$

Consumption of completion messages

$$[\textsc{Compl-Parent}]\quad h\{\texttt{go}\ k\} \parallel h^i\colon n[P\,|\,\mathrm{wait}.Q]_- \longmapsto h^i\colon n[P\,|\,Q]_k$$

$$[\textsc{Compl-Coin}]\quad h\{\texttt{OK}\overline{\texttt{in}}\ k\} \parallel h^i\colon n[P\,|\,\mathrm{wait}.Q]_- \longmapsto h^{i+1}\colon n[P\,|\,Q]_k$$

$$[\textsc{Compl-Migr}]\quad h\{\texttt{mig}\ k\} \parallel h^{i+1}\colon n[P\,|\,\mathrm{wait}.Q]_- \longmapsto h \rhd^{i+1} k \parallel k\{\texttt{reg}^0\ P\,|\,Q\}$$

$$[\textsc{Compl-Migr'}]\quad h\{\texttt{mig}\ k\} \parallel h^0\colon n[P\,|\,\mathrm{wait}.Q]_- \longmapsto k\{\texttt{reg}^1\ P\,|\,Q\}$$

$$[\textsc{Compl-Reg}]\quad h\{\texttt{reg}^s\ R\} \parallel h^{i+s}\colon n[P\,|\,\mathrm{wait}.Q]_k \longmapsto h^i\colon n[P\,|\,Q\,|\,R]_k$$

Fig. 7. Reduction rules

– The rules for *consumption of completion messages* describe how agents resume computation when they are informed that a movement has occurred.

Counting: Counters have to be kept coherent along the transitions of a net. Intuitively, to understand the counting for an agent located at h, in a given GcPan configuration, we have to consider:

– the number of non waiting ambient locations that are immediate children of h (of the form $k^i\colon n[P]_h$);
– the number of child forwarders ($k \rhd^i h$);
– the number of request messages emitted to h ($h\{req/E\}$);
– the number of completion or relocation messages whose effect will be to increment the number of immediate children of h ($k\{\mathtt{go}\ h\}$, $k\{\mathtt{go}_{\rhd} h\}$, ...).

We explain below how our accounting is preserved along the moves:

In: The two brother ambients taking part in an In move (h and k) are in wait state at the moment when the parent ambient (h') matches the corresponding requests. Ambients in wait state are pending, and hence are not taken into account by the counter of h'. As a consequence, h' has to *increment* its counter in rule Local-In. The role of the completion message $k\{\mathtt{OK}\overline{\mathtt{in}}\ h'\}$ is to bring k under h' (which was its original father in case there was no forwarder between h and h'). Similarly, h, that will receive h' as a new child (message $h'\{\mathtt{go}\ h\}$ and rule Compl-Parent), also increments its counter, upon reception of message $\mathtt{OK}\overline{\mathtt{in}}$ (rule Compl-Coin).
Out: As previously, the intuition is that the parent (h') loses a child (h), and has to decrement its counter, but since this child is in wait state, there is nothing to do. The freshly created forwarder allows us to keep the grandparent counter unaffected: the forwarder hides both parent and child (and hence the value of its counter is set to two).
Open: The opening location (h') increments its counter to take into account the creation of the forwarder (rule Compl-Migr, that lets h, the opened location, react to a `mig` completion message). In the case where the counter of h is null, h has no child: there is no need for such a forwarder, and we avoid creating it (rule Compl-MigrGC). We must be careful, though, to let h' know that it has to undo the increment of its counter, which is achieved using the flag s decorating the `reg` message (rule Compl-Reg).

Forwarders Behaviour is defined by the rules for transmission of request messages. We illustrate these by the following reductions, that show the behaviour of a message carrying request R traversing three forwarders h_1, h_2 and h_3 to reach its *real target*:

$$
\begin{array}{lll}
 & h_1\{R/[]\} \parallel h_1 \rhd^3 h_2 \parallel h_2 \rhd^1 h_3 \parallel h_3 \rhd^4 k \parallel k^2\colon n[P] & \\
\longmapsto & h_1 \lhd^2 \parallel h_2\{R/[h_1]\} \parallel h_2 \rhd^1 h_3 \parallel h_3 \rhd^4 k \parallel k^2\colon n[P] & [\text{Fw-Send}] \\
\longmapsto & h_1 \lhd^2 \parallel h_3\{R/[h_1]\} \parallel h_3 \rhd^4 k \parallel k^2\colon n[P] & [\text{Fw-SendGC}] \\
\longmapsto & h_1 \lhd^2 \parallel h_3 \lhd^3 \parallel k\{R/[h_3{::}h_1]\} \parallel k^2\colon n[P] & [\text{Fw-Send}] \\
\longmapsto & h_1 \lhd^2 \parallel h_3 \lhd^3 \parallel h_3\{\mathtt{go}_{\rhd} k\} \parallel h_1\{\mathtt{go}_{\rhd} k\} \parallel k^3\colon n[P \mid \{R\}] & [\text{Loc-Rcv}] \\
\longmapsto & h_1 \lhd^2 \parallel h_1\{\mathtt{go}_{\rhd} k\} \parallel h_3 \rhd^3 k \parallel k^3\colon n[P \mid \{R\}] & [\text{Fw-Reloc}] \\
\longmapsto & h_1 \rhd^2 k \parallel h_3 \rhd^3 k \parallel k^3\colon n[P \mid \{R\}] & [\text{Fw-Reloc}]
\end{array}
$$

First, the message gets transmitted by forwarder h_1, which decrements its counter, adds its name to the list decorating the message before transmission to h_2, and blocks. In the second step of transmission, since h_2's counter is equal to one, h_2 gets garbage collected, and the message is passed to h_3, which transmits it to k (along the lines of the first step). Then the target location k receives the message, and reacts by broadcasting a $\mathtt{go}_{\triangleright} k$ relocation message to each agent that has been registered in the list decorating the message. k's counter is incremented by the size of this list *minus one*: all forwarders except the uppermost one will become new direct children of the parent location (note that in the case of an empty chain of forwarders, we decrement the counter because the direct child is in wait state, and hence pending). Finally, the blocked forwarders react to the relocation messages by moving to their new location, and resume computation.

3 Correctness of the Machine

We establish the correctness of our machine by showing a weak barbed bisimilarity result with the Pan. Although the overall structure of the proof has similarities with [11], there are important differences. First of all, we compare two abstract machines, rather than a machine and a calculus as in [11]. The correspondence we can make between two configurations of the Pan and the GcPan is fairly coarse (barbed bisimilarity), because the machines route messages and manage forwarders differently.

Also, a few results, that are crucial in the proof for Pan [11] do not hold for GcPan. For instance in Pan, we have

$$(\boldsymbol{\nu} h)(h \rhd k \parallel A) \;\succeq\; A\{k \backslash h\}\,,$$

where $\succeq$ stands for *expansion*, a behavioural preorder that guarantees that, intuitively, if $P \succeq Q$, P exhibits the same behaviour as Q modulo some extra internal computation (expansion is not explicitly mentioned in [11], but the technique is essentially equivalent). This makes it possible, using weak bisimulation up to expansion, to factorise reasoning about forwarders and to considerably reduce the size of the relations needed to establish bisimilarity results.

Unfortunately the corresponding expansion law does not hold in our setting. This is due to the way the union-find algorithm works: rearranging forwarders entails an initial cost, and generates race situations. This cost is later compensated by the fact that messages are transmitted on shorter chains. This kind of delayed improvement cannot be captured using expansion because $P \succeq Q$ if Q is 'better than P' *at every step* (see [12] for a proof of the non-expansion result).

The notion of equivalence we adopt is barbed bisimulation [15], that we denote $\approx$. Here we use it to compare states belonging to different transition systems.

In GcPan the observability predicates $\Downarrow_n$ (where n is any name) are defined as follows. A is *observable at* n means, intuitively, that A contains an agent n that accepts interactions with the external environment. Formally: $A \downarrow_n$ if $A \equiv (\boldsymbol{\nu} \overline{p})\,(\mathtt{root}\colon \mathtt{rootname}[\{M, h\} \mid P]_{\mathtt{rootparent}} \parallel A')$ where $M \in \{\overline{\mathtt{in}}\ n,\ \overline{\mathtt{open}}\ n\}$

and $n \notin \overline{p}$ (here $\overline{p}$ stands for a set of names or localities). Then, using $\Longmapsto$ for the reflexive and transitive closure of $\longmapsto$, we write $A \Downarrow_n$ if $A \Longmapsto \downarrow_n$. In SA and Pan, observability is defined similarly (see [11]). Our main results are:

Theorem 1. *For any well-typed SA process P, we have $[\![P]\!] \approx [\![P]\!]_{gc}$.*

Corollary 1. *Let P be a well-typed SA process, then $[\![P]\!]_{gc} \approx P$.*

Proof: *By [11], we have $[\![P]\!] \approx P$. Theorem 1 allows us to conclude.* ♢

The above corollary implies, for instance, that for all n, $P \Downarrow_n$ iff $[\![P]\!]_{gc} \Downarrow_n$.

For lack of space, we only give the main intuitions behind the proof of Theorem 1 (the reader is referred to [12] for details). The first step is to introduce a notion of *well-formed net*, and to show that it is preserved by reduction. Well-formedness allows us to express which nets are 'reasonable', in particular w.r.t. the destination of messages and the value of counters.

In Pan and GcPan, the routing of messages is deterministic and does not change the bisimilarity class of a net. Therefore, the main idea in introducing the candidate bisimulation relation to establish Theorem 1 is to define a kind of normal form for nets, in which all messages are routed to their destination and the nets in both machines can be compared directly. Based on this, we derive some preliminary lemmas to show that whenever a message is routed to its destination in a given configuration of one of the machines, the other machine can do the same (this might involve some additional transitions in the GcPan, because, as seen above, race conditions may prevent a message from being 'directly routable'). These lemmas are then used in a modular way to construct the bisimulation proof, that amounts to show that by definition, processes related by the candidate bisimulation exhibit the same observables and preserve this property.

4 Final Remarks

Developments of our machine. Besides ST ambients, the other main type for SA processes [14] is that of *immobile* ambients (IM). An immobile ambient is an ambient that can neither move (in or out other ambients), nor be opened ($\overline{\texttt{open}}$ co-capability). Such an ambient is not necessarily single-threaded. We have designed an extension of the GcPan [12] to handle immobile ambients as well.

We have also developed a prototype OCaml implementation of the (extended) GcPan, that is described at [1]. We plan to exploit it to further evaluate the improvements in terms of efficiency brought by our machine.

Related Work. Cardelli [4, 5] has produced the first implementation, called *Ambit*, of an ambient-like language; it is a single-machine implementation of the untyped Ambient calculus, written in Java. The algorithms are based on locks: all the ambients involved in a movement (three ambients for an In or Out movement, two for an Open) have to be locked for the movement to take place.

In [10], a JoCaml implementation of an abstract machine for Mobile Ambients, named AtJ, is presented. In Mobile Ambients, there are no co-capabilities, movements are triggered using only capabilities, and grave interferences arise. These differences enable considerable simplifications in abstract machines for SA (Pan, GcPan) and in their correctness proof — see [11] for a detailed comparison. Other differences are related to the distinction between logical and physical movements: in AtJ physical movements are triggered by the execution of `in` and `out` capabilities, whereas in GcPan only `open` induces physical movement.

[17] presents a distributed abstract machine for the Channel Ambients calculus, a variant of Boxed Ambients [3]. In Channel Ambients the `open` primitive – one of the most challenging primitives for the implementation of Ambient calculi – does not exist (`open` is dropped in favour of a form of inter-ambient communication). Although in the implementation [17] actual movement of code arises as a consequence of movement of ambients, the phenomenon is not reflected in the definition of the Channel Ambient calculus. Therefore, the main problems we have been focusing on do not appear in that setting.

In the Distributed Join calculus [8], migrating join definitions are replaced in the source space with a forwarder, to route local messages to the join definition at its new location. This phenomenon is reminiscent of the execution of Open reductions in our machine.

References

1. GcPan webpage. `http://perso.ens-lyon.fr/damien.pous/gcpan`.
2. P. Bidinger and J.-B. Stefani. The Kell Calculus: Operational Semantics and Type System. In *Proc. of FMOODS'03*, volume 2884 of *LNCS*, pages 109–123. Springer Verlag, 2003.
3. M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *Proc. TACS 2001*, LNCS 2215, pages 38–63. Springer Verlag, 2001.
4. L. Cardelli. Ambit, 1997. `http://www.luca.demon.co.uk/Ambit/Ambit.html`.
5. L. Cardelli. Mobile ambient synchronisation. Technical Report 1997-013, Digital SRC, 1997.
6. L. Cardelli and A. Gordon. Mobile Ambients. In *Proc. of FOSSACS'98*, volume 1378 of *LNCS*, pages 140–155. Springer Verlag, 1998.
7. F. Le Fessant, I. Piumarta, and M. Shapiro. An Implementation for Complete, Asynchronous, Distributed Garbage Collection. In *Proc. of PLDI'98*, ACM Sigplan Notices, pages 152–161, 1998.
8. C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, 1998.
9. C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml: A Language for Concurrent Distributed and Mobile Programming. In *Proc. of Advanced Functional Programming 2002*, volume 2638 of *LNCS*, pages 129–158. Springer Verlag, 2002.
10. C. Fournet, J.-J. Lévy, and A. Schmitt. An asynchronous, distributed implementation of mobile ambients. In *Proc. of IFIP TCS'00*, volume 1872 of *LNCS*, pages 348–364. Springer Verlag, 2000.

11. P. Giannini, D. Sangiorgi, and A. Valente. Safe Ambients: abstract machine and distributed implementation, 2004. submitted; an extended abstract appeared in Proc. ICALP'01, volume 2076 of LNCS, pages 408–420, Springer Verlag.
12. D. Hirschkoff, D. Pous, and D. Sangiorgi. An Efficient Abstract Machine for Safe Ambients. Technical Report 2004–63, LIP – ENS Lyon, 2004.
13. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proc. 27th POPL*. ACM Press, 2000.
14. F. Levi and D. Sangiorgi. Mobile Safe Ambients. *Transactions on Programming Languages and Systems*, 25(1):1–69, 2003. ACM Press.
15. R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. 19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992.
16. R. De Nicola, G.L. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998.
17. A. Phillips, N. Yoshida, and S. Eisenbach. A Distributed Abstract Machine for Boxed Ambient Calculi. In *Proc. of ESOP'04*, LNCS, pages 155–170. Springer Verlag, 2004.
18. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of ACM*, 22(2):215–225, 1975.
19. A. Unyapoth and P. Sewell. Nomadic Pict: Correct Communication Infrastructure for Mobile Computation. In *Proc. of 28th POPL*, pages 116–127. ACM Press, 2001.

A Process Calculus for QoS-Aware Applications*

Rocco De Nicola[1], Gianluigi Ferrari[2], Ugo Montanari[2],
Rosario Pugliese[1], and Emilio Tuosto[2]

[1] Dipartimento di Sistemi e Informatica,
Università di Firenze, Via C. Lombroso 6/17, 50134 Firenze – Italy
{denicola, pugliese}@dsi.unifi.it
[2] Dipartimento di Informatica,
Largo Pontecorvo 1, 56127 Pisa – Italy
{giangi, ugo, etuosto}@di.unipi.it

Abstract. The definition of suitable abstractions and models for identifying, understanding and managing Quality of Service (QoS) constraints is a challenging issue of the Service Oriented Computing paradigm. In this paper we introduce a process calculus where QoS attributes are first class objects. We identify a minimal set of primitives that allow capturing in an abstract way the ability to control and coordinate services in presence of QoS constraints.

1 Introduction

Service Oriented Computing (SOC) [14] has been proposed as an evolutionary paradigm to build wide area distributed systems and applications. In this paradigm, services are the basic building blocks of applications. Services are heterogeneous software components which encapsulate resources and deliver functionalities. Services can be dynamically composed to provide new services, and their interaction is governed in accordance with programmable coordination policies. Examples of SOC architectures are provided by WEB services and GRID services.

The SOC paradigm has to face several challenges like service composition and adaptation, negotiation and agreement, monitoring and security. A key issue of the paradigm is that services must be delivered in accordance with *contracts* that specify both client requirements and service properties. These contracts are usually called Service Level Agreements (SLA). SLA contracts put special emphasis on Quality of Service (QoS) described as a set of non functional properties concerning issues like response time, availability, security, and so on.

The actual metric used for evaluating QoS parameters is heavily dependent on the chosen level of abstraction. For instance, when designing network infrastructures, performance (with some probabilistic guarantees) is the main QoS metric. When describing multimedia applications, visual and audible qualities would be the crucial parameters. Instead, for final users, the perceived QoS is not just a matter of performance

* Work partially supported by EU-FET Project AGILE, EU-FET Project MIKADO, EU-FET Project PROFUNDIS, and MIUR project SP4 Architetture Software ad Alta Qualità di Servizio per Global Computing su Cooperative Wide Area Networks.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 33–48, 2005.

but also involves availability, security, usability of the required services. Moreover, the user would like to have a certain control on QoS parameters in order to customize the invoked services, while network providers would like to have a strict control over services. The resolution of this tension will be inherently dynamic depending on the run-time context.

In our view, it is of fundamental importance to develop formal machineries to describe, compose and relate the variety of QoS parameters. Indeed, the formal treatment of QoS parameters would contribute to the goal of devising robust programming mechanisms and the corresponding reasoning techniques that naturally support the SOC paradigm. In this paper we face this issue by introducing a process calculus where QoS parameters are used to control behaviours, i.e. QoS parameters are first class objects.

The goal of the present paper is to identify a minimal set of constructs that provide an abstract model to control and coordinate services in presence of QoS constraints. This differentiates our proposal from other approaches. In particular, process calculi have been designed to model QoS in terms of performance issues (e.g., the probabilistic π-calculus [15]). Other process calculi have addressed the issues of failures and failure detection [13]. Process calculi equipped with powerful type systems have also been put forward to describe the behavioral aspects of contracts [11, 10, 9].

Some preliminary results towards the direction of this paper can be found in [3, 4]. Cardelli and Davies [3] introduced a calculus which incorporates a notion of communication rate (bandwidth) together with some programming constructs. In [17, 8, 4] a (hyper)graph model to control explicitly QoS attributes has been introduced. The graphical semantics allows us to describe interactions in accordance with the agreed QoS level as optimal paths in the model thus creating a bridge between formal models and the protocols used in practice. Here, we elaborate on [4] with the aim of bridging further the gap between formal theories and the pragmatics of software development.

Fundamental to our approach is the notion of QoS values; a QoS value is a *tuple* of values and each component of the tuple indicates a QoS dimension. The values of the fields can be of different kind, for instance, the value along the latency dimension could be a numerical value but the security values could have the form of sets of capabilities indicating the permissions to perform some operations on given resources, e.g. read or write a file. Compositionality of QoS values is therefore a key element of our approach: the composition of QoS values will be a QoS value as well. Indeed, one might want to build a QoS value based on latency, availability and access rights of a service.

To guarantee compositionality of QoS parameters, we shall require QoS values to be elements of suitable algebraic structures, called *constraint semirings* (c-semirings, for short), consisting of a domain and two operations, the *additive* (+) and the *multiplicative* ($\cdot$) operations, satisfying some properties. The basic idea is that the former is used to select among values and the latter to combine values. C-semirings were originally proposed to describe and program constraints problems [2]. Several semirings have been proposed to model QoS issues. For instance, general algorithms for computing shortest paths in a weighted directed graph are based on the structure of semirings [12]. The modelling of trustness in an ad hoc networks exploits the semiring structure [16]. C-semiring based methods have a unique advantage when problems with multiple QoS

dimensions must be tackled. In fact, it turns out that cartesian products, exponentials and power constructions of c-semirings are c-semirings as well.

Our process calculus, $\mathcal{K}oS$, builds on KLAIM (*Kernel Language for Agent Interaction and Mobility*) [5]. KLAIM is an experimental kernel programming language specifically designed to model and program wide area network applications with located services and mobility. KLAIM naturally supports a *peer-to-peer* programming model where interacting peers (nodes in KLAIM terminology) cooperate to provide common sets of services. $\mathcal{K}oS$ primitives handle QoS values as first class entities. For instance, an overlay network is specified by creating nodes ($node_\kappa\langle t\rangle$) and new links ($s \overset{\kappa}{\frown} t$) and indexing them with the QoS value κ of the operation. Thus, for instance the expression $s \overset{\kappa}{\frown} t$ states that s and t are connected by a link whose QoS parameters are given by κ.

The operational semantics of $\mathcal{K}oS$ ensures that the QoS values are respected during system evolution. Suppose for example that node s would interact by an operation whose QoS value is κ' with node t along the link $s \overset{\kappa}{\frown} t$. This interaction will be allowed provided that the SLA contract of the link is satisfied, namely, $\kappa' \leq \kappa$.

We shall illustrate the expressiveness of the calculus through several examples. This can appear as an exercise in coding a series of linguistic primitives into our calculus notation, but it yields much more because the encodings offer a practical illustration of how to give a precise semantic interpretation of QoS management. Indeed, the main contribution of this paper is the careful investigation of a minimal conceptual model which provides the basis to design programming constructs for SOC. We focus on the precise semantic definition of the calculus because it is a fundamental step to design programming primitives together with methods supporting the correct usages of the primitives and the formal verification of the resulting applications.

The rest of the paper is organized as follows. In the next section we illustrate a motivating example and, in Section 3, we introduce syntax and semantic of $\mathcal{K}oS$. In Section 4, we deal with expressivity issues and in the subsequent one we present a more complex scenario and show how it can be tackled by following our approach.

2 A Motivating Example

Before introducing the formal definition of $\mathcal{K}oS$, we prefer to show its usefulness by modelling a realistic, but simplified, example. Our purpose here is to give a flavour of the underlying programming paradigm. We consider a scenario where n servers provide services to m clients and we focus on balancing the load of the servers. Clients and servers are located on different nodes; a generic client node has address c_i while a generic server node has address s_j. Clients issue requests to servers by spawning process R from their node to a server node. For simplicity, we abstract from the actual structures of QoS values, and we assume that clients and servers "knows" each other and cannot be created dynamically. Adding dynamicity is straightforward.

A generic client node M_i, for $i \in \{1, \ldots, m\}$, is described by the following term:

$$M_i \overset{\text{def}}{=} c_i :: \langle s_1, \kappa_1\rangle \mid \ldots \mid \langle s_n, \kappa_n\rangle \mid !C_\delta.$$

Intuitively, M_i represents a network component with address c_i, containing tuples of the form $\langle s_j, \kappa_j\rangle$, for $j \in \{1, \ldots, n\}$, and running process $!C_\delta$. Each tuple $\langle s_j, \kappa_j\rangle$ represents

the load κ_j of the server s_j that the client perceives, thus the whole set of tuples represents a sort of *directory service* containing the SLA contract with the available servers. Operator ! is the replication operator: $!C_\delta$ represents an unbounded number of concurrent copies of process C_δ. Finally, process C_δ specifies the behaviour of the client and is defined as follows:

$$C_\delta \stackrel{\text{def}}{=} (?u, ?v).\varepsilon_v[R]@u.con_{v\cdot\delta}\langle u\rangle.\langle u, v\cdot\delta\rangle.$$

Initially, the client selects a server by non-deterministically inputting a tuple by means of the operation $(?u, ?v)$. Once the input is executed, variables u and v are instantiated with the server name and its load, respectively. Afterward, the client tries to spawn process R to the selected server u. Execution of $\varepsilon_v[R]@u$ takes place only if a "suitable" link toward u exists. What here is meant for "suitable" is that the load v of the client must not exceed the value on the link. Then, since remote spawning consumes the links traversed during the migration, the client attempts to re-establish a connection with u by executing $con_{v\cdot\delta}\langle u\rangle$. Notice that the operation $con_{v\cdot\delta}\langle u\rangle$ is used by the client to ask for a link with a QoS value increased of a quantity δ. Once the connection has been established, the client updates its SLA view of the servers load by inserting tuple $\langle u, v\cdot\delta\rangle$ into its local directory service.

A generic server N_j, for $j \in \{1, \ldots, n\}$, is described as follows:

$$\begin{array}{rcl} N_j & \stackrel{\text{def}}{=} & s_j :: \langle h\rangle \;\mid\; \langle c_1, \kappa'_1\rangle \;\mid\; \ldots \;\mid\; \langle c_m, \kappa'_m\rangle \;\mid \\ & & !(S\ c_1\ s_j) \;\mid\; \ldots \;\mid\; !(S\ c_m\ s_j). \end{array}$$

Similarly to clients, N_j encapsulates a directory service containing SLA data about the clients. This directory service is formed by tuples of the form $\langle c_i, \kappa'_i\rangle$, for $i \in \{1, \ldots, m\}$, each recording the QoS value κ'_i assigned to the link towards node c_i, and by the current load of the server, represented by a tuple containing a natural number $\langle h\rangle$. For any client c_i there is a *load manager* $S\ c_i\ s_j$ which decides whether a link with c_i can be re-established or not. Process $S\ c\ s$ is written as follows:

$$\begin{array}{rcl} S\ c\ s & \stackrel{\text{def}}{=} & (?l).\langle l\rangle.\mathit{If}_s\ l < \mathit{max} \\ & & \quad \mathit{then}\ (c, ?v).acc_{f(v,l)}\langle c\rangle.\langle c, f(v,l)\rangle. \end{array}$$

The load manager repeatedly acquires the tuple $\langle h\rangle$ (current load) and compares it with the maximum admissible load (max). Then, the process decides whether to accept requests for new connections coming from the client: the link is created only when h is less than max. The QoS value of the new link is computed by a function f and depends on both the old QoS value and the current load.

Finally, we assume that process R representing clients service requests is a sequential process of the form

$$R \stackrel{\text{def}}{=} (?x).\langle x+1\rangle \ldots \mathit{actual\ request} \ldots (?y).\langle y-1\rangle,$$

Namely, R has a prologue and an epilogue which respectively increments and decrements the counter that measures the server load.

3 The Calculus

This section introduces $\mathcal{K}oS$ a calculus that provides a set of basic primitives for modelling and managing QoS values. A $\mathcal{K}oS$ term represents a net made of *nodes* which model places where computations take place or where services can be allocated/accessed. We assume as given a set of nodes $\mathcal{S}$ (ranged over by s, $t, \ldots$) that are connected by *links* representing the middleware infrastructure, i.e., the interactions between two nodes can take place only if they are connected by a sequence of links. Links are weighted by "measures" that represent the QoS value of the connections.

3.1 QoS Values as Constraint Semirings

We assume existence of a set of *QoS values* $\mathcal{C}$, ranged over by κ, that forms a *constraint semiring* [2] (*c-semiring*).

Definition 1 (C-semiring). *An algebraic structure* $\langle A, +, \cdot, \mathbf{0}, \mathbf{1}\rangle$ *is a c-semiring if* A *is a set (*$\mathbf{0}, \mathbf{1} \in A$*), and* $+$ *and* $\cdot$ *are binary operations on* A *that satisfy the following properties:*

- $+$ *(additive operation) is commutative, associative, idempotent,* $\mathbf{0}$ *is its unit element and* $\mathbf{1}$ *is its absorbing element;*
- $\cdot$ *(multiplicative operation) is commutative, associative, distributes over* $+$*,* $\mathbf{1}$ *is its unit element, and* $\mathbf{0}$ *is its absorbing element.*

Operation $+$ induces a partial order on A defined as $a \leq_A b \iff a + b = b$. The minimal element is thus $\mathbf{0}$ and the maximal $\mathbf{1}$. $a \leq_A b$ means that a is more constrained than b.

An example of c-semiring is $\langle \omega, min, +, +\infty, 0\rangle$, where ω is the set of natural numbers, the minimum between natural numbers is the additive operation and the sum over natural numbers is the multiplicative operation. Notice that in this case the partial order induced by the additive operations is the inverse of the ordinary total order on natural numbers. Another example of c-semiring is $\langle \wp(\{A\}), \cup, \cap, \emptyset, A\}\rangle$, where $\wp(A)$ is the powerset of a set A, and $\cup$ and $\cap$ are the usual set union and intersection operations.

$\mathcal{K}oS$ does not take a definite standing on which of the many c-semiring structures to use. The appropriate c-semiring to work with should be chosen, from time to time, depending on the kind of QoS dimensions one intends to model. Below, we introduce some c-semiring structures together with the QoS dimension they handle:

- $\langle \{true, false\}, \vee, \wedge, false, true\rangle$ (boolean): Network and service availability.
- $\langle$Real+$, min, +, +\infty, 0\rangle$ (optimization): Price, propagation delay.
- $\langle$Real+$, max, min, 0, +\infty\rangle$ (max/min): Bandwidth.
- $\langle [0, 1], max, \cdot, 0, 1\rangle$ (probabilistic): Performance and rates.
- $\langle [0, 1], max, min, 0, 1\rangle$ (fuzzy): Performance and rates.
- $\langle 2^N, \cup, \cap, \emptyset, N\rangle$ (set-based, where N is a set): Capabilities and access rights.

C-semiring based methods have a unique advantage when problems with multiple QoS criteria must be tackled. In fact, it turns out that cartesian products, exponentials and power constructions of c-semirings are c-semirings as well.

Table 1. $\mathcal{K}oS$ Syntax

N, M	::=	NETS
	$\mathbf{0}$	*Empty net*
	$\mid\ s :: P$	*Located Process*
	$\mid\ s \overset{\kappa}{\frown} t$	*Link*
	$\mid\ (\nu\, s)N$	*Node restriction*
	$\mid\ N \parallel M$	*Net composition*
P, Q	::=	PROCESSES
	$\mathbf{0}$	*Null process*
	$\mid\ \gamma.P$	*Action prefixing*
	$\mid\ (\nu\, s)P$	*Restriction*
	$\mid\ P \mid Q$	*Parallel process*
	$\mid\ !P$	*Iteration*
γ	::=	PREFIXES
	$node_\kappa\langle t\rangle$	*Node creation*
	$\mid\ con_\kappa\langle t\rangle$	*Connection request*
	$\mid\ acc_\kappa\langle t\rangle$	*Connection acceptance*
	$\mid\ (T)$	*Input*
	$\mid\ \langle v_1, \ldots, v_n\rangle$	*Output*
	$\mid\ \varepsilon_\kappa[P]@t$	*Remote process spawning*
T	::= $\varepsilon \mid v \mid ?x \mid \neg v \mid T, T$	INPUT TEMPLATES

3.2 Syntax

The syntax of $\mathcal{K}oS$ is presented in Table 1. Other than the existence of $\mathcal{C}$, existence of a set of *names* $\mathcal{N}$ (ranged over by r, s and t) is assumed. First-class *values*, ranged over by u and v, can be either QoS values or names.

The syntax for nets permits the (possibly empty) parallel composition of *located processes* and *links*. A located process $s :: P$ consists of a name s, called the *address* of P, and the process P running at s. A *link* $s \overset{\kappa}{\frown} t$ states that s and t are connected by a link whose QoS value is κ. The net $(\nu\, s)N$ is a net that declares s as restricted in N, which is the scope of the restriction.

The syntax for processes is standard. The symbol $\mathbf{0}$ overloads the symbol for empty nets; however, the contexts will clarify whether it refers to processes or nets. Prefixes γ encompass actions for

- creating a node ($node_\kappa\langle t\rangle$) or a connection to/from another node ($con_\kappa\langle t\rangle$, $acc_\kappa\langle t\rangle$),
- exchanging tuples of values ((T) and $\langle v_1, \ldots, v_n\rangle$),
- remotely spawning a process ($\varepsilon_\kappa[P]@t$).

Links are oriented, indeed $s \overset{\kappa}{\frown} t$ allows a process to be spawned from s to t but not the viceversa. The creation of new links is obtained by synchronising actions $con_\kappa\langle t\rangle$ and $acc_{\kappa'}\langle s\rangle$ performed at s and t, respectively.

Communication involves exchange of tuples (i.e. finite sequences) of values that are retrieved via pattern matching. Input prefixes use *templates* T, namely finite sequences of values or placeholders (written as $?x$). Execution of an output prefix causes gener-

ation of a tuple of values $v_1, \ldots, v_n$. Both the empty template and the empty tuple are denoted by ε. Hereafter, we let $\mathfrak{t}$ range over tuples of values and, given a template T and a tuple $\mathfrak{t}$, we let T_i and $\mathfrak{t}_i$ denote the i-th element of T and $\mathfrak{t}$, respectively.

The placeholder $?x$ binds the occurrences of x in the rest of the template, namely, in $?x, T$, the scope of $?x$ is T. The set $\mathrm{bn}(T)$ collects the names bound in T while $\mathrm{fn}(T)$ denotes the names having free occurrences in T; their definitions are standard. We consider as equivalent those templates that differ only for renaming of bound names. The template $\neg v$ tests for inequality, namely, it requires the matching tuple to contain a value different from v (see Definition 7). The only binders of the calculus are the placeholder $?x$ and the node restriction $\nu\, s$. Note that node names might be QoS values (e.g., for specifying access rights), hence, we write $\mathrm{fn}(\kappa)$ to denote the names appearing in κ. Moreover, we require that QoS values do not bind node names, therefore, $\mathrm{bn}(\kappa)$ is empty, for any QoS value κ. We formally define *free* and *bound* names of nets and processes as follows. In the following we write $\mathrm{fn}(\_, \_)$ (resp. $\mathrm{bn}(\_, \_)$) as an abbreviation for $\mathrm{fn}(\_) \cup \mathrm{fn}(\_)$ ($\mathrm{bn}(\_) \cup \mathrm{bn}(\_)$, respectively).

Definition 2 (Free and bound names). *The free names of prefix actions are defined as expected:* $\mathrm{fn}(\gamma) = \mathrm{fn}(\kappa) \cup \{s\}$, *if* $\gamma \in \{node_\kappa\langle s\rangle, con_\kappa\langle s\rangle, acc_\kappa\langle s\rangle\}$, $\mathrm{fn}((T)) = \mathrm{fn}(T)$, $\mathrm{fn}(\langle v_1, \ldots, v_n\rangle) = \mathrm{fn}(v_1) \cup \ldots \cup \mathrm{fn}(v_n)$ *and* $\mathrm{fn}(\varepsilon_\kappa[P]@s) = \mathrm{fn}(\kappa, P) \cup \{s\}$. *Bound names of* γ *are defined similarly, e.g.,* $\mathrm{bn}((T)) = \mathrm{bn}(T)$ *and* $\mathrm{bn}(\varepsilon_\kappa[P]@s) = \mathrm{bn}(P)$ *(while in the remaining cases is the empty set).*

The sets $\mathrm{fn}(\_)$ *and* $\mathrm{bn}(\_)$ *of free and bound names of processes and nets are defined accordingly. The only non-standard case is that for links where we let* $\mathrm{fn}(r \overset{\kappa}{\frown} s) = \mathrm{fn}(\kappa) \cup \{s, r\}$ *and* $\mathrm{bn}(r \overset{\kappa}{\frown} s) = \emptyset$.

As usual, processes or nets obtained by α-converting bound names are considered equivalent. Moreover, we assume the following structural congruence laws.

Definition 3 (Structural congruence). *The relation* $\equiv_P \subseteq P \times P$ *is the least equivalence relation on processes (containing* α*-conversion and) satisfying the following axioms:*

- $(P, \mid, \mathbf{0})$ *is a commutative monoid;*
- $!P \equiv_P P \mid !P$.

The relation $\equiv \subseteq N \times N$ *is the least equivalence relation on nets (containing* α*-conversion and) satisfying the following axioms:*

- $(N, \parallel, \mathbf{0})$ *is a commutative monoid;*
- *if* $P \equiv_P Q$ *then* $s :: P \equiv s :: Q$;
- $s :: P \mid Q \equiv s :: P \parallel s :: Q$;
- $s :: (\nu\, t)P \equiv (\nu\, t)(s :: P)$, *if* $t \neq s$;
- $(\nu\, s)(N \parallel M) \equiv N \parallel (\nu\, s)M$, *if* $s \notin \mathrm{fn}(N)$;
- $(\nu\, s)(\nu\, t)N \equiv (\nu\, t)(\nu\, s)N$.

The last axiom of Definition 3 states that the order of the restrictions is irrelevant, hence we can write $(\nu\, s_1, \ldots, s_n)N$ instead of $(\nu\, s_1) \ldots (\nu\, s_n)N$.

3.3 Semantics

We define the operational semantics of $\mathcal{K}oS$ by means of a labelled transition system that describes the evolution of nets. In the semantic clauses, it is useful to define a function that, given a net N, yields the names that are used as node addresses in the net.

Definition 4 (Addresses). *Let* addr *be the function given by:*

$$\mathtt{addr}(N) = \begin{cases} \emptyset, & N = \mathbf{0} \vee N = s \overset{\kappa}{\frown} t \\ \{s\}, & M = s :: P \\ \mathtt{addr}(M) \setminus \{s\}, & N = (\nu\, s)M \\ \mathtt{addr}(N_1) \cup \mathtt{addr}(N_2), & N = N_1 \parallel N_2. \end{cases}$$

Notice that $\mathtt{addr}(N) \subseteq \mathrm{fn}(N)$, but not necessarily $\mathtt{addr}(N) = \mathrm{fn}(N)$, for instance if $N = s :: \langle t \rangle.\mathbf{0}$ then $\mathrm{fn}(N) = \{s, t\}$ while $\mathtt{addr}(N) = \{s\}$. Basically, $\mathtt{addr}(N)$ collects those free names of N that effectively occur in N as address of some node.

Definition 5 (Localized Actions). *Let γ be a prefix, then the* localized prefix $\gamma@s$ *is defined as follows:*

$$\gamma@s = \begin{cases} s\ \varepsilon^{s}_{\kappa}\langle P \rangle@t & \text{if } \gamma = \varepsilon_{\kappa}\langle P \rangle@t \\ s\ \gamma & \text{otherwise} \end{cases}$$

The syntax of localized actions α *is given below:*

$$\alpha ::= \gamma@s \quad | \quad s\ link\ t \quad | \quad \tau$$

We let $\mathrm{fn}(\gamma@s) = \mathrm{fn}(\gamma) \cup \{s\}$ and $\mathrm{bn}(\gamma@s) = \mathrm{bn}(\gamma)$.

Definition 6 (Nets semantics). *The operational semantics of nets is given by the relation $\twoheadrightarrow \subseteq N \times (\alpha \times C) \times N$. Relation $\twoheadrightarrow$ is defined by the rules in Table 2 and the following standard rules:*

$$(\textsc{res})\ \frac{N \overset{\tau}{\underset{\kappa}{\twoheadrightarrow}} M}{(\nu\, s)N \overset{\tau}{\underset{\kappa}{\twoheadrightarrow}} (\nu\, s)M} \qquad (\textsc{str})\ \frac{N \equiv N' \overset{\alpha}{\underset{\kappa}{\twoheadrightarrow}} M' \equiv M}{N \overset{\alpha}{\underset{\kappa}{\twoheadrightarrow}} M}$$

$$(\textsc{par})\ \frac{N \overset{\alpha}{\underset{\kappa}{\twoheadrightarrow}} N'}{N \parallel M \overset{\alpha}{\underset{\kappa}{\twoheadrightarrow}} N' \parallel M}\ \text{if}\ \begin{cases} \mathrm{bn}(\alpha) \cap \mathrm{fn}(M) = \emptyset \quad \wedge \\ (\mathtt{addr}(N') \setminus \mathtt{addr}(N)) \cap \mathtt{addr}(M) = \emptyset \end{cases}$$

Intuitively, $N \overset{\alpha}{\underset{\kappa}{\twoheadrightarrow}} M$ states that the net N can perform the transition α to M by exposing the QoS value κ. Clearly, all local transitions (communications, node or link creations) have unitary QoS value, while the only non-trivial QoS values appear on the transitions that spawn processes or show the presence of links. Let us give more detailed comments on the rules in Table 2.

Rule (LINK) states that a link within a net disappears once it has been used. These transitions are used in the premises of rules (ROUTE) and (LAND) for establishing a path between two nodes such that a remote evaluation can take place.

Table 2. Network semantics

(LINK) $$s \overset{\kappa}{\frown} t \xrightarrow[\kappa]{s \text{ link } t} \mathbf{0}$$

(PREF) $$s :: \gamma.P \xrightarrow[\mathbf{1}]{\gamma@s} s :: P, \gamma \notin \{node_\kappa\langle t\rangle, con_\kappa\langle s\rangle, acc_\kappa\langle s\rangle\}$$

(NODE) $$s :: node_\kappa\langle t\rangle.P \xrightarrow[\mathbf{1}]{node\langle t\rangle} s :: P \parallel s \overset{\kappa}{\frown} t \parallel t :: \mathbf{0}, s \neq t$$

(CON) $$\frac{N \xrightarrow[\mathbf{1}]{s\ con_\kappa\langle t\rangle} N' \qquad M \xrightarrow[\mathbf{1}]{t\ acc_{\kappa'}\langle s\rangle} M'}{N \parallel M \xrightarrow[\mathbf{1}]{\tau} N' \parallel M' \parallel s \overset{\kappa}{\frown} t}\ \kappa \leq \kappa'$$

(LEVAL) $$s :: \varepsilon_\kappa[Q]@s.P \xrightarrow[\mathbf{1}]{\tau} s :: P \parallel s :: Q$$

(ROUTE) $$\frac{N \xrightarrow[\kappa']{r\ \varepsilon^s_\kappa\langle P\rangle@t} N' \qquad M \xrightarrow[\kappa'']{r\ link\ r'} M' \qquad \kappa' \cdot \kappa'' \leq \kappa}{N \parallel M \xrightarrow[\kappa' \cdot \kappa'']{r'\ \varepsilon^s_\kappa\langle P\rangle@t} N' \parallel M'}, t \neq r'$$

(LAND) $$\frac{N \xrightarrow[\kappa']{r\ \varepsilon^s_\kappa\langle P\rangle@t} N' \qquad M \xrightarrow[\kappa'']{r\ link\ t} M' \qquad \kappa' \cdot \kappa'' \leq \kappa}{N \parallel M \xrightarrow[\kappa' \cdot \kappa'']{\tau} N' \parallel M' \parallel t :: P}$$

(COMM) $$\frac{N \xrightarrow[\mathbf{1}]{s\ (T)} N' \qquad M \xrightarrow[\mathbf{1}]{s\ \mathfrak{t}} M' \qquad \bowtie (T, \mathfrak{t}) = \sigma}{N \parallel M \xrightarrow[\mathbf{1}]{\tau} N'\sigma \parallel M'}$$

Rule (PREF) accounts for action prefixing; node creation, however, deserves a specific treatment that is defined in rule (NODE). The side condition of (PREF) also states that no link from s to itself can be created. Indeed, we assume that transitions that involve only the local node have unitary QoS value and are always enabled.

Rule (NODE) allows a process allocated at s to use a name t as the address of a new node and to create a new link from s to t exposing the QoS value κ. The side condition of (PAR) prevents that new nodes (and links) are created by using addresses of existing nodes.

Rule (CON) adds a new link between two existing addresses s and t; the link is created only if the processes at s and t satisfy the SLA contract. More precisely, the accepting

node t is willing to connect only to those nodes that declare a QoS value lower than κ'. If this condition holds, a new link is added to the net, such link has the QoS value exposed by s. One can think of s as asking for the connection with *at least* some characteristics expressed by κ and t establishes the connection only when it can enforce the requirement of s, namely $\kappa \leq \kappa'$.

Rule (LEVAL) states that the local spawning of a process is always enabled while rules (ROUTE) and (LAND) control process migration and require more detailed explanations. A remote spawning action $\varepsilon_\kappa[P]@t$ consists of the migrating process P, the arrival node t and a QoS value κ expressing that P must be routed on a path exposing a QoS value[1] at most κ. Differently from the local spawning of processes, remote spawning is not always possible, it is indeed mandatory that the net contains a path of links from the starting node s to the arrival node t. Moreover, the SLA contract of the path between s and t must not exceed the value κ that the spawner has declared. Notice that this semantically describes the SLA agreement on the mobility of processes. This is formally achieved by rules (ROUTE) and (LAND). More specifically, rule (ROUTE) states that, if the migrating process can go through an intermediate node r and a link from r to a node $r' \neq t$ exists, the QoS value κ' of the partial path from s to r composed with the value κ'' of the link from r to r' must be lower than κ. If this is the case, a transition can be inferred stating that P, spawned from s, can go through r' exposing the QoS value $\kappa' \cdot \kappa''$. Rule (LAND) is similar to (ROUTE) but describes the last hop of P, namely when the target node t is reached. In this case, P is spawned at t, provided that the QoS value of the whole path that has been found is lower than κ.

Rule (COMM) establishes that a synchronization takes place provided that sender and receiver are allocated at the same node and that the template and the tuple match according to the definition below. Hereafter, we use σ to denote a substitution, i.e. a map from names to names and QoS values, and $\sigma[\sigma']$ to denote the composition of substitutions, i.e. the substitution σ'' defined as follows: $\sigma''(x) = \sigma'(x)$ if $x \in \mathrm{dom}(\sigma')$, $\sigma''(x) = \sigma(x)$ if $x \in \mathrm{dom}(\sigma) - \mathrm{dom}(\sigma')$.

Definition 7 (Pattern matching). *A template T and a tuple* $\mathfrak{t}$ match *when the following function is defined*

$$\bowtie(T, \mathfrak{t}) = \begin{cases} \varepsilon & \text{if } (T = \varepsilon \ \wedge \ \mathfrak{t} = \varepsilon) \vee (T = v \ \wedge \ \mathfrak{t} = v) \\ \varepsilon & \text{if } T = \neg v \ \wedge \ \mathfrak{t} = v' \ \wedge \ v \neq v' \\ \{^v/_x\} & \text{if } T = ?x \ \wedge \ \mathfrak{t} = v \\ \sigma[\sigma'] & \text{if } T = F, T' \ \wedge \ \mathfrak{t} = v, \mathfrak{t}' \wedge \ \bowtie(F, v) = \sigma \ \wedge \ \bowtie(T'\sigma, \mathfrak{t}') = \sigma' \end{cases}$$

where the application of a substitution to a template, $T\sigma$, is defined as follows:

$$T\sigma = \begin{cases} \varepsilon & \text{if } T = \varepsilon \\ v, T'\sigma & \text{if } T = x, T' \ \wedge \ \sigma(x) = v \\ x, T'\sigma & \text{if } T = x, T' \ \wedge \ x \notin \mathrm{dom}(\sigma) \\ ?x, T'\sigma\{^x/_x\} & \text{if } T = ?x, T'. \end{cases}$$

Under the conditions of (COMM), the substitution $\bowtie(T, \mathfrak{t})$ is applied to the receiver. Note that $\bowtie$ may not be defined, for instance $\bowtie(\neg s, s)$ does not yield any substitution and, therefore, the match in such a case does not hold.

[1] The QoS value of a path $s_0 \overset{\kappa_1}{\frown} s_1 \ldots s_{n-1} \overset{\kappa_n}{\frown} s_n$ is defined as $\kappa_1 \cdot \ldots \cdot \kappa_n$.

4 Examples

In this section we present some specification examples. To make the presentation more readable let us introduce some notational conventions. First, we avoid writing trailing **0** processes, second, we write $\varepsilon[P]@r$ instead of $\varepsilon_{\mathbf{1}}[P]@r$ and similarly for $node_{\mathbf{1}}\langle t\rangle$, $con_{\mathbf{1}}\langle t\rangle$ and $acc_{\mathbf{1}}\langle t\rangle$.

Boolean expressions Booleans are encoded as processes that allocate a pair of names to a node:

$$\begin{aligned} True\ r &\stackrel{\text{def}}{=} (\nu\, t)\varepsilon[\langle t, t\rangle]@r \\ False\ r &\stackrel{\text{def}}{=} (\nu\, f, f')\varepsilon[\langle f, f'\rangle]@r. \end{aligned}$$

The truth and the falsity are tested by checking that the names in a pair are equal or different, respectively. The following process tests for the equality of two names:

$$Test\ x\ y\ r \stackrel{\text{def}}{=} (\nu\, t)(node\langle t\rangle.\varepsilon[Eval\ y\ r \ \mid\ \langle x\rangle]@t),$$

where $Eval\ y\ r \stackrel{\text{def}}{=} (y).True\ r \ \mid\ (\neg y).False\ r$. Process *Test* spawns the tuple $\langle x\rangle$ and the *Eval* process onto a newly generated node so that the first or the second component of *Eval* have exclusive access to $\langle x\rangle$. Notice that only one of the components can consume the tuple, indeed, either $x = y$ (and only the pattern (y) matches $\langle x\rangle$) or $x \neq y$ (and only the pattern $(\neg y)$ matches $\langle x\rangle$). Finally, *True* or *False* allocates on node r the truth value corresponding to evaluation of $x = y$. Assuming the encoding of booleans, we can represent standard control structures such as `if-then-else` and `while`.

The encoding of boolean values is indeed an example of a standard programming metaphor for finding and handling services. Assume that we want to describe a *look-up* mechanism for discovering distributed services. For instance, the *web services* technology allows deploying new services by gluing together those that have been published. Web service composition, however, requires a look-up phase where the available service must be discovered. In the boolean example, processes *True* and *False* are the services that have been published and composed together to provide the *Test* service. Notice that the look-up phase does not require the knowledge of the service name but only that of the "schema" of the service. For instance, whenever a new "true" service is published it suffice to generate a new name and use it for building the "schema" for the true service (i.e., a pair of two equal names).

Public, private, permanent and stable links. Links in $\mathcal{K}oS$ are public entities: when available they can be exploited by all processes. Consider the following $\mathcal{K}oS$ net:

$$N \stackrel{\text{def}}{=} s :: \varepsilon_3[P]@t \ \parallel\ s \overset{1}{\frown} r \ \parallel\ r :: con_2\langle t\rangle.\varepsilon_2[Q]@t \ \parallel\ t :: acc_2\langle r\rangle,$$

where QoS values are the c-semiring of natural numbers. Net N has three nodes s, r and t and, initially, only s and r are connected by a link with QoS value 1. Node s is trying to spawn P on t which is not possible because there is no path from s to t. Node r is willing to spawn a process Q on t, as well; however, r is aware that a link must be

first created. Node t simply accepts requests for establishing a link from r. Initially, it is only possible to synchronize $con_2\langle t\rangle$ and $acc_2\langle r\rangle$ which, by applying rule (CON) leads to

$$N' \stackrel{\text{def}}{=} s :: \varepsilon_3[P]@t \quad \| \quad s \overset{1}{\frown} r \quad \| \quad r :: \varepsilon_2[Q]@t \quad \| \quad r \overset{2}{\frown} t \quad \| \quad t :: \mathbf{0}.$$

Now, applying rules (PREF), (LINK) and (LAND) we derive

$$N' \xrightarrow[2]{\tau} s :: \varepsilon_3[P]@t \quad \| \quad s \overset{1}{\frown} r \quad \| \quad r :: \mathbf{0} \quad \| \quad t :: Q.$$

Notice that the link between r and t is consumed by the migration of Q hence P cannot reach t. However, N' can also evolve differently, in fact, both the two spawning actions are enabled, because the creation of the link between r and t has also provided a path from s to t exposing the QoS value 3. Hence, by rules (PREF), (LINK), (ROUTE) and (LAND) we can also derive

$$N' \xrightarrow[3]{\tau} s :: \mathbf{0} \quad \| \quad r :: \varepsilon_2[Q]@t \quad \| \quad t :: P.$$

Noteworthy, the migration of P prevents Q to be spawned because the link created by r has been used by P.

In general, this kind of interference should be avoided and this can be done in $\mathcal{K}oS$ by expressing *private links* which can be specified by exploiting the properties of c-semirings. The intuition is that the use of a link is allowed only whether the traversing process has the appropriate "rights". If we represent access rights as sets of names, then a process must "know" all the names needed for traversing the link. For instance, consider the following net:

$$s :: \varepsilon_{\{r,s\}}[P]@t \parallel s \overset{\{r\}}{\frown} s',$$

process P can traverse the link $s \overset{\{r\}}{\frown} s'$ because it "knows" r, that is the only name required to traverse the link. Noteworthy, P could not traverse $s \overset{\{r,u\}}{\frown} s'$ because it does not expose name u.

We consider the c-semiring $\mathcal{R} = \langle \wp_{\text{fin}}(\mathcal{S}) \cup \{\mathcal{S}\}, glb, \cup, \mathcal{S}, \emptyset\rangle$ to represent access rights (recall that $\mathcal{S}$ is the set of sites). It is straightforward to prove that $\mathcal{R}$ is a c-semiring; moreover, the order induced by the additive operation of $\mathcal{R}$ is the inverse of the set inclusion (i.e., $X \leq Y \iff Y \subseteq X$).

Therefore, a private link between the nodes s and t can be specified as

$$(\nu\, p)(s :: P \parallel s \overset{\{p\}}{\frown} t \parallel t :: Q),$$

indeed, in order to pass through link $s \overset{\{p\}}{\frown} t$, a process must exhibit the "password" p. The knowledge of p is handled by enlarging the scope of the restriction and communicating it.

We conclude by illustrating how one could implement *permanent* links, i.e. links that are always available, by exploiting replication:

$$s :: !con_\kappa\langle t\rangle \parallel t :: !acc_{\kappa'}\langle s\rangle$$

A slight variation are *stable* links, which are links existing until a given condition is satisfied.

$$Stable_s\ G\ t \stackrel{\text{def}}{=} !con_\kappa\langle t\rangle \quad | \quad \varepsilon[While\ G\ do\ acc_\kappa\langle s\rangle\ od\ \mathbf{0}]@t$$

Cryptography. By exploiting private links, $\mathcal{K}oS$ can encode standard encryption/decryption mechanisms usually adopted for expressing security protocols in process calculi (see e.g. [1]). Consider the following net:

$$(\nu\, k, s_k)(i :: P \parallel i \overset{\{k\}}{\frown} s_k \parallel s_k :: M \parallel s_k \overset{\{k\}}{\frown} r \parallel r :: Q), \tag{1}$$

and assume that the only links from/to s_k are those appearing in (1). Net (1) aims at representing the initiator i and the responder r of a protocol that share a key k. According to (1) a key is modelled by means of a pair made of a name and a node which roughly speaking contains those messages that are encrypted with k.

The intuition is that encrypting corresponds to allocating a message on s_k while decrypting corresponds to the possibility of "jumping" on s_k and reading a message or, in other words, to the knowledge of k for traversing links $i \overset{\{k\}}{\frown} s_k$ or $s_k \overset{\{k\}}{\frown} r$.

5 Composing Overlay Networks

We consider a scenario where a service is replicated over the nodes of an overlay network and can be invoked trough a *unique* handler H that manages the requests of the clients. This kind of architectures is adopted from Internet Service Providers (ISP) that offer dial-up connection to end-users (EU). In this case a telecommunication company (TC) handles the phone overlay networks. The EU connects to the "nearest" ISP server by dialing a single (country-wide) number. The TC takes care of dispatching the call to the closest ISP server on the overlay network. There are (at least) two possible way of connecting the EU and the ISP server. Either the TC establishes a direct connection between the EU and the ISP, or the TC act as a gateway between the phone overline network and the ISP overlay network. Both solutions can be easily expressed in $\mathcal{K}oS$ in the logical architecture of the system: the handler H manages the requests (e.g., controls the access rights of the client), looks for a suitable server, and forwards the request, while trying to balance the load of any replica of the server. Hence, the request of a client C might not be forwarded to the "best" server from the client's point of view. In this case, H provides another server to C, however, the client may or may not commit to use it.

The simplest way to model this composed overlay network is to assume that the link between C and H have QoS values expressing the access rights of C. When a server s meeting both the request of C and the load constraints is found, H replies to C and tells s to accept a (private) link from C. Hereafter, we assume that h is the node address of H. We detail the client first:

$$\begin{aligned} C\ \kappa\ pr\ c \overset{\text{def}}{=}\ & (\nu\, r)(\ \varepsilon_\kappa[\langle \text{“connect”}, c, r\rangle]@h. \\ & \quad (r, ?s, ?pr', ?p). \\ & \quad If_c\ pr' < pr \\ & \quad\quad then\ con_{\{r,p\}}\langle s\rangle.\varepsilon_{\{p,r\}}[R]@s \\ & \quad\quad else\ con_{\{r,p\}}\langle s\rangle.\varepsilon_{\{p,r\}}[\langle r, \text{“to-much”}\rangle]@s). \end{aligned}$$

Process C requests H to find a server and waits for the response. The request contains c, the node address of C, and a private name r. Name r can be thought of as the unique

marker of the request so that only C will acquire data corresponding to request r. Possibly, H returns a response (marked with r) containing the server address s, the price pr', and the password p for the private link. Finally, C establishes a private link with s and, depending on the price pr' required by the server, either raises its request R ('then' branch) or notifies s that the service is too expensive ('else' branch).

The definition of H requires the following auxiliary processes.

$$Rd(T) \stackrel{\text{def}}{=} (T).\langle \natural_T \rangle$$
$$Lt_s\ r\ i \stackrel{\text{def}}{=} Rd(r, ?j).If_s\ j \leq i\ then\ True\ else\ False$$

Process $Rd(T)$ looks for a tuple matching T and immediately re-generates the consumed tuple; this is denoted by $\natural_T$ which is obtained from T by removing all the '?' occurring in its placeholders. Then, process $Lt_s\ r\ i$, interpreting $\langle r, v \rangle$ as a "cell" having address r and containing value v, reads the value in r and establishes if it is less than/equal to i.

$$\begin{aligned}
H \stackrel{\text{def}}{=} !(\ & (\text{“connect”}, ?x, ?r).\langle r, 1 \rangle \\
& While_h\ Lt_h\ r\ \text{nserv} \\
& \quad do \\
& \quad\quad (r, ?i).Rd\ (pref(x, i), ?l, ?pr.) \\
& \quad\quad If_h\ l \geq \max \\
& \quad\quad\quad then\ \langle r, i + 1 \rangle \\
& \quad\quad\quad else \\
& \quad\quad\quad\quad (\nu\ p)(\varepsilon[\langle \text{“newlink”}, x, r, p \rangle]@pref(x, i). \\
& \quad\quad\quad\quad\quad\quad \varepsilon[\langle r, pref(x, i), pr, p \rangle]@x. \\
& \quad\quad\quad\quad\quad\quad (x, ?v).acc_{f(v,l)}\langle x \rangle.\langle x, f(v, l) \rangle. \\
& \quad\quad\quad\quad\quad\quad (pref(x, i), ?l, ?pr).\langle pref(x, i), l + 1, pr \rangle) \\
& \quad od\ \varepsilon[\langle r, \text{“no-server-available”} \rangle]@x).
\end{aligned}$$

Process H is continuously listening for a connection request. Once such a request is issued from a client at x, H starts scanning the server list (nserv is the number of servers). For each server s, node h contains a tuple $\langle s, l, pr \rangle$ where l is an estimation of the load of s and pr is the price for using s. Also, for each client x, h maintains a tuple $\langle x, \kappa \rangle$ that reports the connection between H and x (as done in Section 2). Moreover, H uses a function $pref$ that, given the client address x and the index i, yields the i-th server "preferred" by the client. At the i-th iteration of the while loop, H reads the information of the i-th server preferred by x and, if the load of such a server is too high, the cycle is repeated provided that more servers are left ('then' branch); otherwise, a password p for a private link is generated and communicated to both x and the selected server. The server will accept a private link creation from x so that the client owning the password p can perform a request at s. Finally, H re-establish a link with x according to the new load of the servers by exploiting function f and reflecting this changes in the tuple corresponding to x (i.e., $\langle x, f(v, l) \rangle$), as in Section 2. Indeed, the mechanism of load balancing is the one defined in Section 2, the only difference being that now H is the unique handler that manages the connections with the clients.

Given H and C, the servers must simply wait for a connection request (issued from H) and establish the private connection with the client:

$$S\ s \stackrel{\text{def}}{=} !(\ (\text{"newlink"}, ?x, ?r, ?p).acc_{\{r,p\}}\langle x\rangle.\text{....wait \& execute....}\ \varepsilon[(s, ?l, ?pr).\langle s, l-1, pr\rangle)]@h)).$$

Once the request has been served, S simply updates the load of s.

A net where C, H and S work can be defined as follows.

$$\begin{array}{c} \|_{i=1,\ldots,m}\ x_i :: C\ \kappa_i\ pr_i\ x_i \quad | \quad !acc\langle h\rangle \quad | \quad con_{\kappa_i'}\langle h\rangle \\ \|_{i=1,\ldots,m}\ h :: !con\langle x_i\rangle \parallel h :: H \parallel \\ \parallel (\nu\, s_1, \ldots, s_n)(\ \|_{j=1,\ldots,n}\ h :: !con\langle s_j\rangle \quad | \quad \langle s_j, l_j, pr_j\rangle \parallel s_j :: S\ ) \end{array}$$

where $\|_{i=1,\ldots,m} N_i$ shortens $N_1 \parallel \ldots \parallel N_n$.

The other solution touched upon at the beginning of this section can be achieved by exploiting the possibility offered by $\mathcal{K}oS$ of "connecting" links to form paths between nodes. More precisely, instead of connecting directly the client's node x and (the node of) the selected server s, we can connect h and s so that the client's request at s is routed through h.

6 Conclusion

We have formally defined $\mathcal{K}oS$ a process calculus that provides basic primitives to describe QoS requirements of distributed applications. We demonstrated the applicability of the approach by specifying some expressive case studies.

Our research program is to provide a solid foundation to drive the design of languages and middleware having application-oriented QoS mechanisms. The work reported here is a preliminary step in this direction. In terms of calculus design, the current definition of $\mathcal{K}oS$ assumes that links are the basic construct to manage QoS interactions and cooperation. This is a reasonable assumption for several cases. For instance, in this paper we handled the QoS composition between different overlay networks by suitable links. However, one could interpret QoS composition of overlay networks in a more general sense than adding suitable links. An interesting challenge for future research is to extend $\mathcal{K}oS$ with more general mechanisms for composing overlay networks than simple parallel composition via links.

There are a number of ways in which our setting can be extended. For instance, it would be interesting to develop type systems which would allow determining QoS properties of processes. We plan to extend types for access control of [7, 6] to deal with QoS attributes. In particular, it would be interesting to exploit such types to capture the notion of contract. Another direction for future research is developing observational semantics for $\mathcal{K}oS$ based on the idea of observing QoS values. These abstract theories could permit reasoning on $\mathcal{K}oS$ nets and comparing them on the basis of the perceived QoS values.

References

1. M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, January 1999.
2. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, March 1997.
3. L. Cardelli and D. Rowan. Service combinators for web computing. *Software Engineering*, 25(3):309–316, 1999.
4. R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A Formal Basis for Reasoning on Programmable QoS. In N. Dershowitz, editor, *International Symposium on Verification – Theory and Practice – Honoring Zohar Manna's 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 436 – 479. Springer-Verlag, 2003.
5. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE/ACM Transactions on Networking*, 24(5):315–330, 1998.
6. R. De Nicola, G. Ferrari, and R. Pugliese. Programming access control: The KLAIM experience. In *International Conference in Concurrency Theory*, Lecture Notes in Computer Science. Springer-Verlag, 2000.
7. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, June 2000.
8. G. Ferrari, U. Montanari, and E. Tuosto. Graph-based Models of Internetworking Systems. In T. Aichernig, Bernhard K. Maibaum, editor, *Formal Methods at the Crossroads: from Panaces to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 242 – 266. Springer-Verlag, 2003.
9. A. Igarashi and N. Kobayashi. A generic type system for the Pi-calculus. *Theoretical Computer Science*, 311(1–3):121–163, Jan. 2004.
10. N. Kobayashi. Type Systems for Concurrent Processes: From Deadlock-Freedom to Livelock-Freedom, Time-Boundedness. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, Proceedings of the International IFIP Conference TCS 2000 (Sendai, Japan)*, volume 1872 of *Lecture Notes in Computer Science*, pages 365–389. IFIP, Springer-Verlag, Aug. 2000.
11. G. Meredith and S. Bjorg. Service-Oriented Computing: Contracts and Types. *Communications of the ACM*, 46(10):41 – 47, October 2003.
12. M. Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata Languages and Combinatorics*, 7(3):321–350, 2002.
13. U. Nestmann and R. Fuzzati. Unreliable failure detectors with operational semantics. In *Proc.ASIAN 2003*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
14. M. Papazouglou and D. Georgakopoulos. Special issue on service oriented computing. *Communications of the ACM*, 46(10), 2003.
15. C. Priami. Stochatic π-calculus. *The Computer Journal*, 38(6):578–589, 1995.
16. G. Theodorakopoulos and J. Baras. Trust Evaluation in AdHoc Networks. In *WiSe '04: Proceedings of the 2004 ACM workshop on Wireless security*, pages 1–10. ACM Press, 2004.
17. E. Tuosto. *Non-Functional Aspects of Wide Area Network Programming*. PhD thesis, Dipartimento di Informatica, Università di Pisa, May 2003. TD-8/03.

Abstract Interpretation-Based Verification of Non-functional Requirements

Agostino Cortesi[1,⋆] and Francesco Logozzo[2,⋆⋆]

[1] Università Ca' Foscari di Venezia, I-30170 Venezia (Italy)
cortesi@dsi.unive.it
[2] École Polytechnique, F-91128 Palaiseau cedex (France)
Francesco.Logozzo@Polytechnique.fr

Abstract. The paper investigates a formal approach to the verification of non functional software requirements, e.g. portability, time and space efficiency, dependability/robustness. The key-idea is the notion of observable, i.e., an abstraction of the concrete semantics when focusing on a behavioral property of interest. By applying an abstract interpretation-based static analysis of the source program, and by a suitable choice of abstract domains, it is possible to design formal and effective tools for non-functional requirements validation.

1 Introduction

Abstract interpretation [10] is a theory of semantics approximation for computing conservative over-approximations of dynamic properties of programs. It has been successfully applied to infer run-time properties useful for debugging (e.g., type inference [7, 28]), code optimization (e.g., compile-time garbage collection [22]), program transformation (e.g., partial evaluation [25], parallelization [36]), and program correctness proofs (e.g., safety [20], termination [5], cryptographic protocol analysis [33], proof of absence of run-time errors [3], semantic tattooing/watermarking [13]).

As pointed out in [30], there is still a large variety of tasks in the software engineering process that could greatly benefit from techniques akin to static program analysis, because of their firm theoretical foundations and mechanical nature.

In particular, as observed by [26], during the development of large-scale software systems, effective and efficient management of customer and user requirements is one of the most crucial, but unfortunately also least understood issues. Problems in the requirements are typically not recognized until late in the development process, where negative impacts are substantial and cost for correction has grown large. Even worse, problems in the requirements may go undetected

⋆ Partially supported by MIUR FIRB grant n.RBAU018RCZ and by MIUR PRIN'04 grant n.2004013015.

⋆⋆ This work was conceived when the author was visiting Ca' Foscari.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 49–62, 2005.

through the development process, resulting in software systems not meeting customers and users expectations, especially when the coordination with other components is an issue. Therefore, methods and frameworks helping software developers to better manage software requirements are of great interest for component based software.

In this paper, we are interested to investigate the impact of Abstract Interpretation theory in the formalization and automatic verification of Non-Functional Software Requirements, as they seem not adequately covered by most requirements engineering methods ([27], pag. 194). Non functional requirements can be defined as restrictions or constraints on the behavior of a system service [35]. Different classifications have been proposed in the literature [4, 16, 15], though their specification may give rise to troubles both in their elicitation and management, and in the validation process.

In fact, this work originated from a quite naive question: "*what do we mean when we say that a program is portable on a different architecture?*". In [17] a software is said portable if it can run in different environments. It is clear that it is assumed not only that it runs, but that it runs the same way. And it is also clear that if we require that the behavior is exactly the same, portability to different systems (e.g., from a PC to a PDA, or from an OS to another) can almost never be reached. This means that implicit assumptions are obviously made about the properties to be preserved, and about the ones that might be simply disregarded. In other words, portability needs to be parameterized on some specific properties of interest, i.e. it assumes a suitable abstraction of the software behavior. The same holds also for other product non-functional requirements, like space and time efficiency, dependability, robustness, usability, etc. It is clear that, in this context, the main features of abstract interpretation theory, namely modularity, modulability, and effectiveness may then become very valuable.

The main contributions of the paper can be summarized as follows:

- We extend the usual abstract interpretation notions to the deal with systems, i.e. programs + architectures.
- We show that a significant set of product qualities (non functional requirements) can be formally expressed in terms of abstraction of the concrete semantics when focusing on a behavioral property of interest. This yields an unifying view of product non-functional requirements.
- We show how existing tools for automatic verification can be re-used in this setting to support requirements validation; their practicality directly depends on the complexity of the abstract domains.

The advantage of this approach with respect to previous attempts of modelling software requirements, e.g. by using Milner's Calculus of Communicating Systems [19] or formal methods like Z [24] or B [1, 2] is twofold: (i) the soundness of the approach is guaranteed by the general abstract interpretation theory, and (ii) the automatic validation process can be easily tuned according to the desired granularity of the abstraction.

As far as we know, this is the first attempt to apply Abstract Interpretation theory to the treatment of non-functional software requirements. These semi-

nal results can be seen as a partial contribution towards the achievement of a more challenging objective: to integrate formal analysis by abstract interpretation in the full software development process, from the initial specifications to the ultimate program development [9].

Paper Structure. In Section 2, the concrete semantics of a simple imperative language is introduced to instantiate our framework. In Section 3, the core abstract interpretation theory is extended to deal with program and architecture abstractions. In Section 4 we show how to instantiate our framework on a suite of non-functional product requirements. Section 5 concludes the paper.

2 Operational Semantics of a Core Imperative Language with Exceptions

In order to illustrate the results of this paper, we instantiate our framework with a core imperative language with exceptions and a core architecture. The results can be easily generalized to more complex languages and architectures. We give the syntax, the transition relations and the trace semantics of systems, composed by architectures and a programs.

2.1 Syntax

In this paper setting an architecture is a tuple $\langle bits, Op, stdio, stdout\rangle$, where *bits* is the number of bits used to store integer numbers, *Op* is a set of functions implementing basic arithmetic operations, *stdio* is the input stream (e.g., the keyboard) and *stdout* is the output stream (e.g., the screen). The input stream has a method *next* that returns immediately the next value in the stream, and the output stream has a method *add* to put a pair $\langle v, c\rangle$, i.e., a value v with a color c. We assume that if an arithmetic error occurs in the application of an operation $op \in Op$ (e.g., an overflow or a division by zero), then the exception `ExcMath` is raised.

The syntax of programs is specified by the following grammar:

$$\begin{aligned}
\mathtt{C} ::= {} & \mathtt{skip} \mid \mathtt{x} = \mathtt{E} \mid \mathtt{C}_1; \mathtt{C}_2 \mid \mathtt{if}(\mathtt{E} != 0)\, \mathtt{C}_1\ \mathtt{else}\ \mathtt{C}_2 \mid \mathtt{while}\ (\mathtt{E} != 0)\ \mathtt{C} \\
& \mathtt{write}(\mathtt{x}, \mathtt{col}) \mid \mathtt{throw}\ \mathtt{Exc} \mid \mathtt{try}\ \mathtt{C}_1\ \mathtt{catch}(\mathtt{Exc})\ \mathtt{C}_2 \\
\mathtt{E} ::= {} & k \mid \mathtt{read} \mid \mathtt{E}_1 + \mathtt{E}_2 \mid \mathtt{E}_1 - \mathtt{E}_2 \mid \mathtt{E}_1 * \mathtt{E}_2 \mid \mathtt{E}_1 / \mathtt{E}_2
\end{aligned}$$

where `x` and `col` belong to a given set `Var` of variables, `Exc` belongs to a given set `Exceptions` of exceptions (including the arithmetic ones) and k is (the *syntactic* representation of) an integer number.

A system is a pair $\langle \mathtt{A}, \mathtt{C}\rangle$, where `A` is an architecture and `C` is a program.

2.2 Semantics

The semantics of a system is described in operational style. We assume that the only available type is that of architecture-representable natural numbers:

$$\frac{\underline{k}\in\mathbb{N}_{bits}}{\langle k,\sigma\rangle \xrightarrow{\mathrm{E}} k} \qquad \frac{\underline{k}\notin\mathbb{N}_{bits}}{\langle k,\sigma\rangle \xrightarrow{\mathrm{E}} \langle \texttt{ExcMath},\sigma\rangle} \qquad \frac{\texttt{A}.stdio.next=v}{\langle \texttt{read},\sigma\rangle \xrightarrow{\mathrm{E}} \langle v,\sigma\rangle}$$

$$\frac{\langle \mathtt{E}_1,\sigma\rangle \xrightarrow{\mathrm{E}} \langle v_1,\sigma\rangle \;\; \langle \mathtt{E}_2,\sigma\rangle \xrightarrow{\mathrm{E}} \langle v_2,\sigma\rangle \;\; v_1,v_2\neq\texttt{ExcMath} \;\; \texttt{A}.op(v_1,v_2)=v\neq\texttt{ExcMath}}{\langle \mathtt{E}_1\, op \mathtt{E}_2,\sigma\rangle \xrightarrow{\mathrm{E}} \langle v,\sigma\rangle}$$

$$\frac{\langle \mathtt{E}_1,\sigma\rangle \xrightarrow{\mathrm{E}} \langle v_1,\sigma\rangle \;\; \langle \mathtt{E}_2,\sigma\rangle \xrightarrow{\mathrm{E}} \langle v_2,\sigma\rangle \;\; v_1,v_2\neq\texttt{ExcMath} \;\; \texttt{A}.op(v_1,v_2)=\texttt{ExcMath}}{\langle \mathtt{E}_1\, op \mathtt{E}_2,\sigma\rangle \xrightarrow{\mathrm{E}} \langle \texttt{ExcMath},\sigma\rangle}$$

$$\frac{\langle \mathtt{E}_1,\sigma\rangle \xrightarrow{\mathrm{E}} \langle v_1,\sigma\rangle \;\; \langle \mathtt{E}_2,\sigma\rangle \xrightarrow{\mathrm{E}} \langle v_2,\sigma\rangle \;\; (v_1=\texttt{ExcMath})\; or\; (v_2=\texttt{ExcMath})}{\langle \mathtt{E}_1\, op\, \mathtt{E}_2,\sigma\rangle \xrightarrow{\mathrm{E}} \langle \texttt{ExcMath},\sigma\rangle}$$

Fig. 1. The transition relation for expressions

$\mathbb{N}_{bits} = \{0, , \ldots 2^{bits} - 1\}$. Given the *syntactic* representation k of a number, $\underline{k}$ is the *semantic* correspondent. For instance, $\underline{\texttt{0xFFFF}} = 65535$ so that $\underline{\texttt{0xFFFF}} \notin \mathbb{N}_8$. An environment is a partial map from variables to representable integers: $\mathsf{Env} = [\mathtt{Var} \to \mathbb{N}_{bits}]$. If a variable $\mathtt{x}$ is not defined in a state σ, we denote that by $\sigma(\mathtt{x}) = \Omega$. A state is either a command to execute in a given environment, or an environment, or an exception raised within an environment. Formally: $\Sigma = \mathtt{C} \times \mathsf{Env} \cup \mathsf{Env} \cup \mathtt{Exceptions} \times \mathsf{Env}$.

The transition relations for expressions and programs are defined by structural induction, and they are depicted in Fig. 1 and Fig. 2. It is worth noting that the transition rules are parameterized by the underlying architecture (e.g., the raising of an overflow exception depends on $\mathbb{N}_{bits}$).

Let Σ^* denote the set of finite traces on Σ, and let $S_0 \subseteq \Sigma$ be a set of initial states. With a slight abuse of notation, we refer to a state as a trace of unitary length. The partial-traces semantics [12] of a system is then expressed as a least fixpoint over the complete boolean lattice $\langle \mathcal{P}(\Sigma^*), \subseteq\rangle$ as follows:

$$\mathbb{s}[\![\langle \mathtt{A}, \mathtt{C}\rangle]\!](S_0) = \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda X.\; S_0 \cup \{\sigma_0 \ldots \sigma_n\sigma_{n+1} \mid \sigma_0 \ldots \sigma_n \in X,\; \sigma_n \to \sigma_{n+1}\}.$$

3 Abstracting Systems = Programs + Architectures

Abstract interpretation [10] is a general theory of approximation which formalizes the idea that the semantics of a program can be more or less precise depending on the considered observation level. In this section we revise some basic concepts, and we extend them to deal with composed systems.

In the abstract interpretation terminology, $\langle \mathcal{P}(\Sigma^*), \subseteq\rangle$ is the *concrete domain*, its elements are semantic properties, and the order $\subseteq$ stands for the logical implication. As a consequence, the most precise property about the behavior of a system is the semantics $\mathbb{s}[\![\langle \mathtt{A}, \mathtt{C}\rangle]\!]$, called the *concrete* semantics [10]. Set of traces are approximated are represented by suitable abstract elements, which capture interesting properties while disregarding other execution properties that

$$\frac{}{\langle \texttt{skip},\sigma\rangle \longrightarrow \sigma} \qquad \frac{\langle \texttt{E},\sigma\rangle \xrightarrow{\texttt{E}} \langle v,\sigma\rangle \;\; v \neq \texttt{ExcMath}}{\langle \texttt{x=E},\sigma\rangle \longrightarrow \sigma[x \mapsto v]} \qquad \frac{\langle \texttt{E},\sigma\rangle \xrightarrow{\texttt{E}} \langle \texttt{ExcMath},\sigma\rangle}{\langle \texttt{x=E},\sigma\rangle \longrightarrow \langle \texttt{ExcMath},\sigma\rangle}$$

$$\frac{\langle \texttt{C}_1,\sigma\rangle \longrightarrow \sigma'}{\langle \texttt{C}_1\texttt{;C}_2,\sigma\rangle \longrightarrow \langle \texttt{C}_2,\sigma'\rangle} \qquad \frac{\langle \texttt{C}_1,\sigma\rangle \longrightarrow \langle \texttt{Exc},\sigma\rangle}{\langle \texttt{C}_1\texttt{;C}_2,\sigma\rangle \longrightarrow \langle \texttt{Exc},\sigma\rangle}$$

$$\frac{\langle \texttt{E},\sigma\rangle \xrightarrow{\texttt{E}} \langle \underline{k},\sigma\rangle \;\; \underline{k} \neq 0}{\langle \texttt{if(E!=0)C}_1 \texttt{ else C}_2,\sigma\rangle \longrightarrow \langle \texttt{C}_1,\sigma\rangle} \qquad \frac{\langle \texttt{E},\sigma\rangle \xrightarrow{\texttt{E}} \langle 0,\sigma\rangle}{\langle \texttt{if(E!=0)C}_1 \texttt{ else C}_2,\sigma\rangle \longrightarrow \langle \texttt{C}_2,\sigma\rangle}$$

$$\frac{\langle \texttt{E},\sigma\rangle \xrightarrow{\texttt{E}} \langle \texttt{ExcMath},\sigma\rangle}{\langle \texttt{if(E!=0)C}_1 \texttt{ else C}_2,\sigma\rangle \longrightarrow \langle \texttt{ExcMath},\sigma\rangle}$$

$$\frac{\langle \texttt{E},\sigma\rangle \xrightarrow{\texttt{E}} \langle \underline{k},\sigma\rangle \;\; \underline{k} \neq 0}{\langle \texttt{while(E!=0) C},\sigma\rangle \longrightarrow \langle \texttt{C;while(E!=0) C},\sigma\rangle} \qquad \frac{\langle \texttt{E},\sigma\rangle \xrightarrow{\texttt{E}} \langle 0,\sigma\rangle}{\langle \texttt{while(E!=0) C},\sigma\rangle \longrightarrow \sigma}$$

$$\frac{\langle \texttt{E},\sigma\rangle \xrightarrow{\texttt{E}} \langle \texttt{ExcMath},\sigma\rangle}{\langle \texttt{while(E!=0) C},\sigma\rangle \longrightarrow \langle \texttt{ExcMath},\sigma\rangle}$$

$$\frac{\texttt{A}.\mathit{stdout}.\mathit{add}(\sigma(\texttt{x}),\sigma(\texttt{col}))}{\langle \texttt{write(x,col)},\sigma\rangle \longrightarrow \sigma} \qquad \frac{\texttt{Exc} \in \texttt{Exceptions}}{\langle \texttt{throw Exc},\sigma\rangle \longrightarrow \langle \texttt{Exc},\sigma\rangle}$$

$$\frac{\langle \texttt{C}_1,\sigma\rangle \longrightarrow \sigma'}{\langle \texttt{try C}_1 \texttt{ catch(Exc) C}_2,\sigma\rangle \longrightarrow \sigma'} \qquad \frac{\langle \texttt{C}_1,\sigma\rangle \longrightarrow \langle \texttt{Exc},\sigma'\rangle}{\langle \texttt{try C}_1 \texttt{ catch(Exc) C}_2,\sigma\rangle \longrightarrow \langle \texttt{C}_2,\sigma'\rangle}$$

$$\frac{\langle \texttt{C}_1,\sigma\rangle \longrightarrow \langle \texttt{Exc}',\sigma'\rangle \;\; \texttt{Exc}' \neq \texttt{Exc}}{\langle \texttt{try C}_1 \texttt{ catch(Exc) C}_2,\sigma\rangle \longrightarrow \langle \texttt{Exc}',\sigma'\rangle}$$

Fig. 2. The transition relations for programs

are out of the scope of interest. Abstract properties (or elements) belong to an *abstract domain of observables*, $\bar{\mathsf{D}}$, and they are ordered according to $\bar{\sqsubseteq}$, the abstract counterpart for logical implication. In this work we assume that $\langle \bar{\mathsf{D}}, \bar{\sqsubseteq} \rangle$ is a complete lattice.

The correspondence between the concrete and the abstract semantic domains is given by a pair of monotonic functions $\langle \alpha, \gamma \rangle$. The function $\alpha \in [\mathcal{P}(\Sigma^*) \to \bar{\mathsf{D}}]$, called the abstraction function, formalizes the notion of the abstraction, and $\alpha(T)$ represents the *best* approximation in $\bar{\mathsf{D}}$ of the set of traces T (wrt the order in $\bar{\mathsf{D}}$). If $\alpha(T) \bar{\sqsubseteq} \bar{\mathsf{p}}$ then $\bar{\mathsf{p}}$ is also a correct, although less precise, abstract approximation of T. On the other hand, the function $\gamma \in [\bar{\mathsf{D}} \to \mathcal{P}(\Sigma^*)]$, called the concretization function, returns the set of traces that are captured by an abstract property $\bar{\mathsf{p}}$. The abstraction and concretization functions must satisfy the following property:

$$\forall T \in \mathcal{P}(\Sigma^*). \forall \bar{\mathsf{d}} \in \bar{\mathsf{D}}.\ \alpha(T) \;\bar{\sqsubseteq}\; \bar{\mathsf{d}} \iff T \subseteq \gamma(\bar{\mathsf{d}}),$$

in such a case, we say that $\langle \alpha, \gamma \rangle$ form a Galois connection between the concrete and the abstract domains. We write is as

$$\langle \mathcal{P}(\Sigma^*), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{\mathsf{D}}, \bar{\sqsubseteq} \rangle. \tag{1}$$

The abstract semantics of a system, $\bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!]$, is defined over an abstract domain that is linked to the concrete domain by a Galois connection. It must satisfy the soundness criterion, [10]:

$$\forall S_0 \subseteq \Sigma.\ \alpha(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](S_0)) \ \bar{\sqsubseteq}\ \bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](\alpha(S_0)).$$

The soundness criterion above imposes that, when the properties encoded by a given abstract domain are considered, the abstract semantics $\bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!]$ captures all the behaviors of $\langle \mathtt{A}, \mathtt{C} \rangle$. As a consequence, given a specification of a system $\langle \mathtt{A}, \mathtt{C} \rangle$ expressed as an abstract property $\bar{\mathsf{p}}$, if $\bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](\alpha(S_0)) \bar{\sqsubseteq} \bar{\mathsf{p}}$, by the soundness criterion and by the transitivity of $\bar{\sqsubseteq}$, we have that

$$\alpha(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](S_0)) \bar{\sqsubseteq} \bar{\mathsf{p}}.$$

This means that $\langle \mathtt{A}, \mathtt{C} \rangle$ respects the specification $\bar{\mathsf{p}}$.

In the following, we instantiate the abstract domain and $\bar{\mathsf{p}}$ in order to reflect non-functional requirements of systems and we show how well-known static analyses can be re-used in this enhanced context for the automatic verification of such properties.

4 Application: Non-functional Requirement Analysis

Non-functional software requirements are requirements which are not directly concerned with the specific functions delivered by the system [35]. They may relate to emergent system properties such as reliability, response time and store occupancy. Alternatively, they may define constraints on the system like the data representation used in system interfaces.

The 'IEEE-Std 830 - 1993' [23] presents a comprehensive list of non-functional requirements. In the following we will focus on a few of such requirements, namely *portability*, *efficiency*, *robustness* and *usability*. The approach can be extended to cope with other non-functional requirements.

In this section, we show (i) how such requirements admit a rigorous formalization, unlike, e.g., what stated in [27–§8.2], (ii) how, by a suitable choice of abstract domains, existing tools can be re-used to verify such requirements, and (iii) the effectiveness of the approach on a public-domain static analyzer [8].

4.1 Portability

Informal Definition. According to [17], a software "*is portable if it can run on different environments*". The term *environment* may refer to a hardware platform or a software environment. Analogously, another widespread textbook, [31], defines portability as "*the ease of transferring software products to various hardware and software environments*". The first observation is that the two definitions implicitly link the requirement to unspecified software metrics. Furthermore, as

any natural-based language specifications, they are intrinsically ambiguous. For instance, the word "*run*" can be read as just the possibility of recompiling and executing the software on different system, but also as the request that some behavioral properties of the software are preserved in different platforms.

Formal Definition. We specify portability as a property of the execution of a program that is preserved when it is ported on different architectures. This means that up to a certain property of interest, the behavior of a software is the same on a different architecture.

Definition 1 (Portability). *Let us consider a program* $\mathtt{C}$*, an architecture* $\mathtt{A}$ *and a Galois connection* $\langle \mathcal{P}(\Sigma^*), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{\mathsf{D}}, \bar{\sqsubseteq} \rangle$*. We say that* $\mathtt{C}$*, developed on* $\mathtt{A}$*, is portable on the architecture* $\mathtt{B}$ *w.r.t. the observable domain* $\bar{\mathsf{D}}$*, if*

$$\forall S_0 \subseteq \Sigma.\ \alpha(\mathbb{s}[\![\langle \mathtt{B}, \mathtt{C} \rangle]\!](S_0)) \ \bar{\sqsubseteq}\ \alpha(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](S_0)).$$

Abstraction. A class property one is interested to keep unchanged among different porting of the software is the behavior w.r.t. arithmetic overflow. For instance, the violation of such a property in porting the control software on a different architecture was at the origin of the Arianne V crash [29].

Arithmetic overflow can be checked by using numerical abstract domains, e.g., [10, 14, 32]. In such domains the range of the values assumed by a variable can be constrained so that it can be checked against the largest representable number in a given architecture.

Example 1 (Portability). Let us consider the program $\mathtt{C}$ in Fig. 3(a), and let us consider an architecture $\mathtt{A}$ such that $\mathtt{A}.\mathit{bits} = 32$. We can use the Intervals abstract domain [10], and the public-domain static analyzer [8] to infer that $\bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](\mathtt{i} \mapsto [-\infty, +\infty]) = [1, 2^{16}]$, and as 2^{16} is representable on a 32 bit architecture, then program $\mathtt{C}$ does not cause any arithmetic overflow. As a consequence, by the soundness of the static analysis (guaranteed by abstract interpretation theory), we can safely infer that the program is portable to any architecture in which 2^{16} is representable (this is not the case in a 16 bits architecture).

4.2 Efficiency

Informal Definition. In the existing literature, efficiency "*refers to how economically the software utilizes the resources of the computer*" [17], or it is "*the ability of a software system to place as few demands as possible on hardware resources, such as processor time or space occupied*" [31]. Once again, such definitions suffer from the ambiguity of the natural language,e.g., it is not clear if when verifying efficiency requirements the underlying architecture must be considered or not, or if space and time requirements must be considered independently or not.

```
C ≜ i = 1;
    while (2^16-i != 0)
        i = i*4
```

(a) C, a program not portable on 16 bits architectures

```
C' ≜ i = 1;
     while (2^16-i != 0)
         i = i+2
```

(b) C', a non-efficient program

```
D ≜ try
        i = ?;
        if(i !=0) c = i / 0
        else throw Err
    catch(Err)
        c = 0;
    write(c,255)
```

(c) D, a robust program

```
E ≜ x = ?; r = ?; g = ?; b = ?
    if(r+g-1!=0)
      col = 2^r + 2^g + 2^b
     else col = 0;
    write(x,col)
```

(d) E, a program usable by daltonians

Fig. 3. Four programs on which we verify non-functional prequirements

Formal Definition. Efficiency can be formally defined as an abstraction of the execution traces of a program. As such behavior depends on the underlying architecture, our definition explicitly mentions the architecture in which the program is executed. Efficiency requirements can be specified by natural numbers, standing, for instance, for the number of processor cycles or the size of the heap. As a consequence our abstract domain will be set of natural numbers with the usual total order, $\langle \mathbb{N}, \leq \rangle$.

We distinguish between efficiency in time and space. The first one corresponds to the length of a trace, i.e. the number of transitions for executing the system, and the second one to the size of the environment, i.e. the maximum quantity of memory allocated during program execution. It is worth noting that the following definitions are well-formed as we consider partial execution traces, i.e., (possible infinite) sets of finite traces. Recall that Ω denotes an uninitialized variable.

Definition 2 (Time Efficiency). *Let* C *be a program,* A *an architecture,* length $\in [\mathcal{P}(\Sigma^*) \to \mathbb{N}]$ *be the length of a trace, and* $\langle \mathcal{P}(\Sigma^*), \subseteq \rangle \underset{\alpha_t}{\overset{\gamma_t}{\leftrightarrows}} \langle \mathbb{N}, \leq \rangle$ *a Galois connection where*

$$\alpha_t = \lambda T.\ \sup(\{\text{length}(\tau) \mid \tau \in T\})$$
$$\gamma_t = \lambda n.\ \{\tau \in \mathcal{P}(\Sigma^*) \mid \text{length}(\tau) \leq n\}.$$

We say that the system $\langle \mathtt{A}, \mathtt{C} \rangle$ *respects the time requirement* k *if*

$$\forall S_0 \subseteq \Sigma.\ \alpha_t(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](S_0)) \leq k.$$

Definition 3 (Space Efficiency). *Let* $\mathtt{C}$ *be a program,* $\mathtt{A}$ *an architecture,* $\text{size} \in [\mathcal{P}(\Sigma) \to \mathbb{N}]$ *be the function defined as*

$$\text{size} = \lambda\sigma.\ \#\{\mathtt{x} \in \mathtt{Vars} \mid \sigma(\mathtt{x}) \neq \Omega\},$$

and $\langle \mathcal{P}(\Sigma^*), \subseteq \rangle \underset{\alpha_s}{\overset{\gamma_s}{\leftrightarrows}} \langle \mathbb{N}, \leq \rangle$ *a Galois connection where*

$$\alpha_s = \lambda T.\ \max_{\tau \in T}\{\text{size}(\sigma) \mid \sigma \in \tau\}$$
$$\gamma_s = \lambda n.\ \{\tau \in \mathcal{P}(\Sigma^*) \mid \forall \sigma \in \tau.\ \text{size}(\sigma) \leq n\}$$

We say that the system $\langle \mathtt{A}, \mathtt{C} \rangle$ *respects the space requirement* k *if*

$$\forall S_0 \subseteq \Sigma.\ \alpha_s(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](S_0)) \leq k.$$

Abstractions. In order to automatically verify time requirements, we must find an upper bound to the number of transitions performed during the execution of a system. Once again, we can do it by using a numerical abstract domain. In fact, we can endow a concrete state σ with a (hidden) variable `time`, to be incremented at each transition [18]. Then, the values taken by `time` will be upper-approximated in the numerical domain, say by $\overline{\mathtt{time}}$, so that the verification boils to check that $\overline{\mathtt{time}} \leq k$. In the same way, the verification of space requirements can be obtained by abstracting a state with the number of variables different from Ω it contains. The approach can be generalized to more complex languages, e.g., a language with recursive functions. In this case, the stack will be approximated by its height.

In our approach, verification of time and space efficiency requirements can be easily combined by considering the reduced product of the two abstract domains [10].

Example 2 (Efficiency). Let us consider the programs `C` and `C′` in Fig. 3, an architecture `A`, where the multiplication is a primitive operation, and an architecture `A′` where the multiplication is implemented as a sequence of additions, e.g., $\mathtt{i} = \mathtt{i} * 4$ becomes $\mathtt{i} = \mathtt{i} + \mathtt{i}; \mathtt{i} = \mathtt{i} + \mathtt{i}$. Using the analyzer described in [8], we can infer:

$$\bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](\langle \mathtt{i} \mapsto [-\infty, +\infty], \mathtt{time} \mapsto \underline{0} \rangle) = \langle \mathtt{i} \mapsto [1, 2^{16}], \mathtt{time} \mapsto [0, 9] \rangle$$
$$\bar{\mathbb{s}}[\![\langle \mathtt{A}', \mathtt{C} \rangle]\!](\langle \mathtt{i} \mapsto [-\infty, +\infty], \mathtt{time} \mapsto \underline{0} \rangle) = \langle \mathtt{i} \mapsto [1, 2^{16}], \mathtt{time} \mapsto [0, 25] \rangle,$$
$$\bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{C}' \rangle]\!](\langle \mathtt{i} \mapsto [-\infty, +\infty], \mathtt{time} \mapsto \underline{0} \rangle) = \langle \mathtt{i} \mapsto [0, 2^{16}], \mathtt{time} \mapsto [0, 32769] \rangle.$$

Observe that the results above can be used for comparing different programs on different architectures.

4.3 Robustness

Informal Definition. Robustness, or dependability, for [17] is "*the ability of a program to behave reasonably, even in circumstances that were not anticipated*

in the specifications", for [31] is "*the ability of software systems to react appropriately to abnormal conditions*", and for [27] is "*the time to restart after failure*". Once again, the three definitions are not rigorous enough: the first definition does not specify what is a reasonable behavior, the second one does not specify what is an abnormal condition, and the latter has implicit the strong assumption that all possible failures are considered.

Formal Definition. A software is robust, if any exception raised during its execution, in any architecture and with any initial state, is caught by some exception handler. We recall that exceptions can be raised either by the architecture, e.g., division-by-zero, or by the software itself. As a consequence, a robust program never terminates in an exceptional state.

Definition 4 (Robustness). *Let* `C` *be a program, and let* $\langle \mathcal{P}(\Sigma^*), \subseteq \rangle \underset{\alpha_d}{\overset{\gamma_d}{\leftrightarrows}} \langle \mathcal{P}(\Sigma), \subseteq \rangle$ *be a Galois connection where*

$$\alpha_d = \lambda T.\ \{\sigma_n \mid \sigma_0 \dots \sigma_n \in T\}$$
$$\gamma_d = \lambda S.\ \{\sigma_0 \dots \sigma_{n-1}\sigma_n \mid \forall i \in [0, n-1].\sigma \in \Sigma \wedge \sigma_n \in S\}.$$

We say that a system is robust if for all the architectures `A`*,*

$$\forall S_0 \in \mathcal{P}(\Sigma).\ \alpha_d(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](S_0)) \cap \mathtt{Exceptions} \times \mathtt{Env} = \emptyset.$$

Abstraction. Robustness can be checked either by considering an abstract domain for inferring the uncaught exceptions [34], or by considering an abstract domain for reachability analysis [8]. In the first case, a program is robust if the analysis reports that no exception can be raised; in the latter, a program is robust if the analysis reports that the lines of code that may raise an exception (e.g., with a `throw` statement) are never reached.

Example 3 (Robustness). Let us consider the program `D` of Fig. 3(c). An interval analysis determines that when the true-branch of the `if` statement is taken, `i` is different from zero, so that the `MathErr` exception cannot be raised. In the other case, the exception `Err` is raised and then it is also caught. As a consequence, `D` is robust w.r.t. the chosen abstraction.

4.4 Usability

Informal Definition. The definition of usability is probably the most contrived one. The definition in [17] says that "*software system is usable [...] if its human users find it easy to use*", whereas [31] talks about ease of use as "*the ease with which people of various backgrounds [...] can learn to use software*" and [27] defines it in function of other, undefined, basic concepts as "*learnability, satisfaction, memorability*".

Formal Definition. In our setting, usability is a abstraction of the output stream that is preserved when a given property, depending on the particular user, is considered. For instance, an abstraction that considers the colors of the output characters can be used to verify if a system is usable for daltonians. We need some auxiliary definitions. Output streams belong to the set $\mathtt{Stdout}$. Given a state $\sigma \in \Sigma$, the function $\mathrm{out} \in [\Sigma \to \mathtt{Stdout}]$ is such that $\mathrm{out}(\sigma)$ is the output stream in the state σ.

Definition 5 (Usability). *Let* $\mathtt{C}$ *be a program,* $\mathtt{A}$ *an architecture, let* $\langle \mathcal{P}(\Sigma^*), \subseteq \rangle \underset{\alpha_\Sigma}{\overset{\gamma_\Sigma}{\leftrightarrows}} \langle \mathcal{P}(\mathrm{Stdout}), \subseteq \rangle$ *be a Galois connection where*

$$\alpha_\Sigma = \lambda T.\ \{\mathrm{out}(\sigma) \in \Sigma \mid \exists \tau \in T.\ \sigma \in T\}$$
$$\gamma_\Sigma = \lambda O.\ \{\tau \in \Sigma^* \mid \forall \sigma \in \tau.\ \exists o \in O.\ \mathrm{out}(\sigma) = o\},$$

let $\langle \mathcal{P}(\mathrm{Stdout}), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{\mathsf{D}}, \bar{\sqsubseteq} \rangle$ *be a Galois connection, and let* $\bar{\mathsf{p}} \in \bar{\mathsf{D}}$. *We say that the system* $\langle \mathtt{A}, \mathtt{C} \rangle$ *is usable w.r.t. the observable* $\bar{\mathsf{p}}$ *if*

$$\forall S_0.\ \alpha(\alpha_\Sigma(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!])(S_0)) \ \bar{\sqsubseteq}\ \bar{\mathsf{p}}.$$

Abstract Domains. The definition above can be instantiated to consider the usability of a system for daltonians, i.e., people afflicted by red/green color blindness. In fact, the colors of the output stream can be abstracted in order to collapse together colors indistinguishable by daltonians. As colors are represented by integers in the RGB color system, numerical abstract domains can be used to automatically check properties on colors.

Example 4 (Usability). Let us consider the program $\mathtt{E}$ in Fig. 3(d), an architecture where the input stream is a sequence of 0/1 digits, and colors are represented as in RGB schema using 3 bits, i.e. colors range between 0 (black) and 7 (white). Using the static analyzer of [8] instantiated with the Intervals abstract domain, and refined with trace partitioning [21], one infers that

$$\bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{E} \rangle]\!](\langle \mathtt{x} \mapsto [0,1], \mathtt{r}, \mathtt{g}, \mathtt{b} \mapsto [0,1] \rangle)$$
$$= (\langle \mathtt{x} \mapsto [0,1], \mathtt{r}, \mathtt{g}, \mathtt{b} \mapsto [0,1], \mathtt{col} \mapsto [0,1] \cup [6,7] \rangle),$$

so that as $\mathtt{col}$ is always in the set of the colors distinguishable by daltonians (i.e. { black, blue, yellow, white}), $\mathtt{E}$ respects the usability specification.

4.5 Other Non-functional Requirements

We showed how four typical non-functional requirements can be encapsulated in our framework. This approach based on preservation of a property up to a given observation, can be easily generalized to other product non-functional requirements. For instance, *upgrade* means that when a new program $\mathtt{N}$, replaces a program $\mathtt{O}$ on a given architecture $\mathtt{A}$, then the observed behavior is preserved: $\alpha(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{N} \rangle]\!]) \bar{\sqsubseteq} \alpha(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{O} \rangle]\!])$. Similarly, if *compatibility* is a property specified by an abstract element $\bar{\mathsf{c}}$, then we say that two programs $\mathtt{P}$ and $\mathtt{P}'$ are compatible w.r.t. $\bar{\mathsf{c}}$ if $\alpha(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{P} \rangle]\!]) \bar{\sqsubseteq} \bar{\mathsf{c}}$ and $\alpha(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{P}' \rangle]\!]) \bar{\sqsubseteq} \bar{\mathsf{c}}$.

5 Conclusions and Future Work

In this paper, we faced the issue of applying Abstract Interpretation theory in order to model non functional software requirements and to support their automatic validation.

Recent very encouraging experiences show that abstract interpretation-based static program analysis can be made efficient and precise enough to formally verify a class of properties for a family of large programs with few or no false alarms, also in case of critical embedded systems [3]. We strongly believe that also the treatment of non functional requirements can well fit in this picture.

Two research directions seem particularly promising, in this respect: (i) the design of a library of sophisticated abstract domains for non functional requirements validation, and (ii) the automatic derivation of metrics associated to the domain of observables. In the first case, as observed in Example 2, the precision of the analysis can be greatly improved by a suitable choice of operators on domains (e.g., reduced product [11], and open product [6]). In the second case, it would be interesting to study *abstract metrics*, i.e. metrics tunable with respect to a given observable. In fact, any (even infinite) finite-height domain of observables can be associated with at least two metrics, by considering as distance between two abstract properties

$$\rho_1(d_1, d_2) = \min\{\text{length}(d_i, d_1 \sqcup d_2) \mid i \in \{1, 2\}\}$$
$$\rho_2(d_1, d_2) = \min\{\text{length}(d_i, d_1 \sqcap d_2) \mid i \in \{1, 2\}\}$$

where *length* returns the length of the path in the domain $\bar{\mathsf{D}}$. ρ_1 computes the lack of precision with respect to the element that represents the union of the information the two elements contain, while ρ_2 does the same with respect to the element that keeps the common information.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. J.-R. Abrial. B# : Toward a synthesis between Z and B. In Didier Bert, Jonathan P. Bowen, S. King, and M. Waldn, editors, *ZB'2003 – Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 168–177, Turku, Finland, June 2003. Springer.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the 2003 ACM Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM Press, June 2003.
4. B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, pages 1266–41, December 1976.
5. J. Brauburger. Automatic termination analysis for partial functions using polynomial orderings. *Lecture Notes in Computer Science*, 1302:330–344, 1997. In P. Van Hentenryck, editor, Proc.4 th Int. Symp.SAS 97, Paris.

6. A. Cortesi, B. Le Charlier, and P. van Hentenryck. Combinations of abstract domains for logic programming. In *21th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'94, ACM-Press, New York*, pages 227–239, 1994.
7. P. Cousot. Types as abstract interpretations, invited paper. In *24th ACM Symposium on Principles of Programming Languages (POPL '97)*, pages 316–331. ACM Press, January 1997.
8. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
9. P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, January 1977.
11. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '79)*, pages 269–282. ACM Press, 1979.
12. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, pages 178–190. ACM Press, New York, NY, January 2002.
13. P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–185, Venice, Italy, January 14-16 2004. ACM Press, New York, NY.
14. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '78)*, pages 84–97. ACM Press, 1978.
15. A. Davis. *Software Requirements: Objects, Functions and States*. Prentice Hall, 1992.
16. M. S. Deutsch and R. R. Willis. *Software Quality Engineering*. Prentice-Hall, 1988.
17. C. Ghezzi, Jazayeri M., and D. Mandrioli. *Foundamentals of Software Engineering*. Prentice Hall, 2nd edition, 2003.
18. N. Halbwachs. Determination automatique de relations lineaires verifees par les variables d'un programme. *These de 3eme cycle d'informatique, Universite scientifique et medicale de Grenoble*, March 1979.
19. N. Halbwachs. Non-functional requirements in the software development process. *Software Quality*, 5(4):285–294, 1995.
20. N. Halbwachs. About synchronous programming and abstract interpretation. *Science of Computer Programming*, 31(1):75–89, May 1998.
21. M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Proceedings of the Static Analysis Symposium (SAS '98)*, volume 1503 of *Lectures Notes in Computer Science*, pages 200–215. Springer-Verlag, 1998.
22. S. Hughes. Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation*, 2(4):483–509, 1992.

23. IEEE. *IEEE Recommended Practice for Software Requirement Specification*, 1988.
24. J.M.Spivey. *The Z notation*. Prentice Hall, 1992.
25. N. D. Jones. Combining abstract interpretation and partial evaluation. In Pascal Van Hentenryck, editor, *Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*, pages 396–405. Springer-Verlag, 1997. In P. Van Hentenryck, editor, Proc.4 th Int. Symp.SAS 97, Paris.
26. J. Karlsson. Managing software requirements using quality function deployment. *Software Quality Control*, 6(4):311–326, 1997.
27. G. Kotonya and I. Sommerville. *Requirements Engineering - Processes and Techniques*. Wiley, 1998.
28. D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, October 1994.
29. P. Lacan, J. N. Monfort, L.V.Q. Ribal, A. Deutsch, and A. Gonthier. The software reliability verification process: The ariane 5 example. In *Proceedings DASIA 98 DAta Systems In Aerospace, Athens, GR. ESA Publications*, 1998.
30. D. Le Métayer. Program analysis for software engineering: New applications, new requirements, new tools. *ACM Computing Surveys*, 28(4es):167, December 1996.
31. B. Meyer. *Object-Oriented Software Construction*. Professional Technical Reference. Prentice Hall, 2nd edition, 1997.
32. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. `http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf`.
33. D. Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 47(2–3):177–202, May/June 2003.
34. F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, bo 2000.
35. I. Sommerville. *Software Engineering*. Addison Wesley, 6th edition, 2000.
36. K. R. Traub, D. E. Culler, and K. E. Schauser. Global analysis for partitioning nonstrict programs into sequential threads. *ACM LISP Pointers*, 5(1):324–334, 1992. Proceedings of the 1992 ACM Conference on LISP and Functional Programming.

Coordination Systems in Role-Based Adaptive Software

Alan Colman and Jun Han

Faculty of Information and Communication Technologies,
Swinburne University of Technology,
Melbourne, Victoria, Australia
{acolman, jhan}@swin.edu.au

Abstract. Software systems are becoming more open, distributed, pervasive, and connected. In such systems, the relationships between loosely-coupled application elements become non-deterministic. Coordination can be viewed as a way of making such loosely coupled systems more adaptable. In this paper we show how coordination-systems, which are analogous to nervous systems, can be defined independently from the functional systems they regulate. Such coordination systems are a network of organisers and contracts. We show how the contracts that make up the coordination-system can be used to monitor, regulate and configure the interactions between clusters of software entities called roles. Management and functional levels of contracts are defined. Management contracts regulate the flow of control through the roles. Functional contracts allow the specification of performance conditions. These contracts bind clusters of roles into self-managed composites — each composite with its own organiser role. The organiser roles can control, create, abrogate and reassign contracts. Adaptive systems are built from a recursive structure of such self-managed composites. The network of organiser roles and the contracts they control constitute a coordination-system that is a *separate concern* to the functional system. Association aspects are suggested as a mechanism to implement such coordination-systems.

1 Introduction

As software systems become more open, distributed, pervasive, and connected, such systems need to be able to adapt to their dynamic environments. One approach to building adaptable and adaptive systems is to construct them of loosely coupled elements. These elements are dynamically coordinated to meet changing goals and environmental demands. This paper proposes that coordination functions be implemented as a separate sub-system. This coordination-system can be described and controlled independently from the sub-systems that interact directly with the application domain. This approach is analogous to the coordination-systems that exist both in living things and in man-made organisations. In the realm of biology, the nervous system can be viewed as a system that, in part, coordinates the respiratory, circulatory, and digestive systems. Similarly, the management structure or financial system in a manufacturing business can also be described at a separate level of abstraction from the functional processes that transform labour and material into products.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 63–78, 2005.

The aim of this paper is to show how such coordination/control systems can be imposed on top of functional software, to better monitor, regulate, and coordinate those systems in dynamic environments. The structure of these coordination-systems can be defined as a *separate concern* from the functional systems they coordinate.

Coordination-systems have a mapping to the state of the underlying functional system. The type(s) of abstraction on which the coordination-system is built will depend on the variables that need to be controlled in order to maintain the system's viability in its environment. In terms of a biological analogy, a controlled variable is the level of oxygen supply to the cells. In a management system, the variable might be the amount of funds in the bank. In computerised systems, such control variables could be computational or communication resources; or environmental variables.

This paper is structured as follows: Section 2 gives an overview of a schema for coordination systems based on roles, along with a motivating example. Section 3 examines *contracts* between roles, and the organiser-roles that create, monitor and control contracts. In Section 4 we introduce the concept of adaptive coordination systems built from these organiser roles and contracts. Coordination systems enable the functional systems to maintain their viability in dynamic environments and to respond to changes in non-functional requirements (NFRs). Section 5 discusses the implementation of coordination systems using association aspects. Section 6 briefly looks at related work and Section 7 concludes.

2 Coordination as the Control of Interactions Between Roles

The ROAD (Role-oriented adaptive design) framework presented in this paper extends work on object-oriented role and associative modelling in [1-4]. In this framework, the elements that are being coordinated are *roles* played by *objects*. A role satisfies responsibilities to the system as a whole. Roles are first-class entities that can be added to, and removed from, objects. Kristensen [3] provides a definition of roles that is based on the distinction between intrinsic and extrinsic members (methods and data) of an object. Intrinsic members provide the *core* functionality of the object, while extrinsic members contain the functionality of the role. In our view, this 'core functionality' is the situated computational and communication *capabilities* of the object.

The ROAD approach to creating adaptive software systems is based on the distinction between two types of role – *functional* roles and *management* roles. *Functional roles* (or more properly *domain*-function roles) are focused on first-order goals — on achieving the desired application-domain output. Functional roles constitute the *process* as opposed to the *control* of the system. In ROAD these functional roles are decoupled; they do not directly reference each other. Functional roles are associated by contracts that mediate the interactions between them. The creation and monitoring of these contracts is the responsibility of a type of management role — *organiser-role*s.

To help us discuss the coordination of roles we will consider an example of a highly simplified business department that makes Widgets and employs Employees with different skills to make them. In such a business organisation an employee can

perform a number of roles, sometimes simultaneously. Employees (objects) can perform the roles of Foreman, ThingyMaker, DooverMaker and Assembler (who assembles thingies and doovers into widgets). The Foreman's role is to supervise ThingyMakers, DooverMakers, and Assemblers and to allocate work to them. The WidgetDept Organiser role is responsible for creating the bindings between roles and the objects that play them, and for creating the associations between the various functional roles.

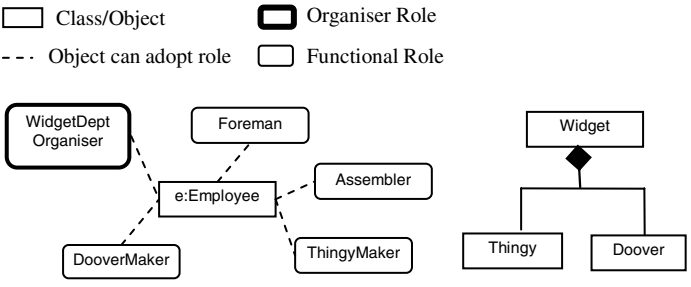


Fig. 1. Roles and objects in a Widget Department

In a purely object-oriented analysis only the domain function is modelled. For example, in the above department the various roles might be modelled as sub-class specialisations of the Employee class. However, in an *open* system we cannot assume that all objects of type Employee have the same *capabilities*. In a real-world manufacturing department some ThingyMakers would be more capable than others (e.g. faster, more accurate, more adaptable etc.). In the ROAD schema, we identify this difference by separating roles from objects, where the situated objects provide the performance capability to the purely domain-function roles. Object-role pairs may be running in different computational and communication contexts. These contexts affect the relative performance of roles (e.g. they may be faster, more reliable, better connected, more costly etc.). As well as the variation in computational contexts, the relationship between the Widget department and its environment may also vary. For example, orders flowing into the department to make new widgets might exceed the capacity of the department to manufacture them. The organiser role may have to reconfigure the relationships between the functional roles, or else create extra object-roles in order to meet the increased demand.

In the Fig. 2 below, an organiser-role (CR) creates, monitors and controls the contracts (C1, C2) between functional roles (F1, F2, F3). The domain of organiser-roles is the system itself rather than the problem-domain. Each organiser role is responsible for a cluster of functional roles. We will call these regulated clusters of roles "self-managed composites" because each composite attempts to maintain a homeostatic relationship with its environment and other composites. We use the word *composite* rather than *component* because the roles in the composite are not necessarily encapsulated in a single package (although they may be). In terms of a management analogy, a self-managed composite in a business organisation would be

a department (e.g. manufacturing department). Such managed composites perform a definable domain function, and can themselves be part of higher-level composites. A role-based organisation is built from a recursive structure of self-managed composites. This structure is coordinated through a network that connects the organiser roles of each of the composites. The network of organiser roles and the contracts they control constitute the *coordination-system*. An example schema of a coordination-system's topology is illustrated in Fig. 2 below.

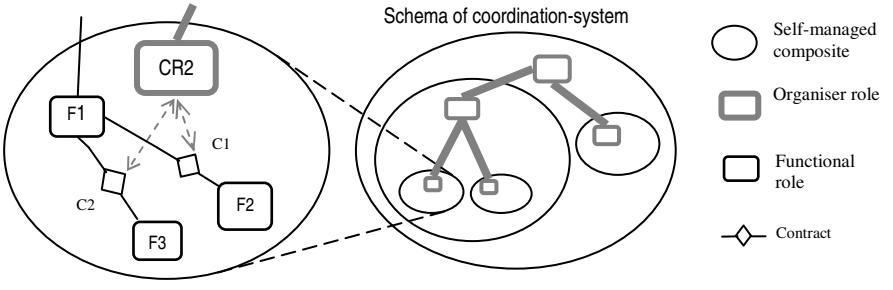


Fig. 2. Self-managed Composite with a Coordination-system

In following sections we show how a coordination-system, built from contracts and organisers, can be imposed on a cluster of functional roles to enable the system to better cope with uncertain environments and changing NFRs.

3 Controlling Interactions Using Contracts

In the first part of this section we summarise our previous work [5] on using contracts to manage interactions between roles. This approach distinguishes contracts at the functional and management level. In the latter part of this section, we extend this work by introducing the concept of management-level protocols that control interaction between roles, and then discuss what is needed to specify contracts at the functional level.

ROAD contracts [5] are association classes that express the obligations of the contract parties to each other. The *form* (type) of a contract sets out the mutual obligations and interactions between classes of party (e.g. vendor and purchaser). A contract is *instantiated* when values are put against the variables in the contract *schedule* (e.g. the vendor and purchaser are named, the date of commencement is agreed etc.) and the contract is signed. Contracts also have an *execution state* that indicates the level of fulfilment of the various clauses of the contract. This dynamic information is needed to ensure that the terms of the contract are being met. It includes information on the state of the relationship between the parties, and the state of any interaction defined by the protocols. ROAD contracts store this dynamic state

information in the contract itself. The contract itself can enforce the terms of the contract by controlling the interactions between the parties.

Contracts between functional roles often share common characteristics. In particular, the *control* aspects of the contract can be abstracted. Using our example of a WidgetMaking department, the control-management relationship between a Foreman and an Assembler could be characterised as a Supervisor-Subordinate relationship. Similarly, a Foreman-ThingyMaker relationship would also be characterised a Supervisor-Subordinate relationship. Rules control the interactions between *operational-management* roles such as a supervisor and a subordinate. For example a supervisor can tell a subordinate to do a work related task but a subordinate *cannot* tell a supervisor what to do. Other types of management contract include auditor – auditee; peer – peer; and predecessor – successor in supply-chain and production-lines.

In ROAD, these abstracted control aspects of the relationships between roles are encapsulated in a *management contract*. Domain-function-level contracts ("functional contracts" for short) inherit control relationships from these management contracts. The conceptual relationships between functional and management contracts, and the respective roles they associate, are illustrated in Fig. 3 below.

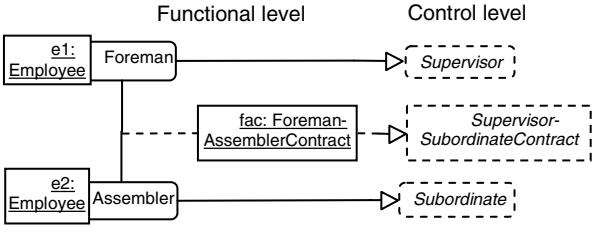


Fig. 3. Functional and management contracts

As can been seen from Fig. 3 above, `fac` is an *instance* of a `Foreman-AssemblerContract`. The contract is an association class between the Foreman and the Assembler roles. The `ForemanAssemblerContract` inherits the form of its control relationships from the `SupervisorSubordinateContract` by virtue of the fact that the Foreman plays a Supervisor role in relation to the Assembler's Subordinate role.

Fig. 4 below illustrates an organisational structure for our Widget department, based on the Bureaucracy pattern [6] and built using contracts. In order to simplify the diagram, functional contracts have been drawn as diamonds. The structure, which is similar to a business organisation chart, is still abstract because objects have not yet been assigned to roles. Note that the Foreman plays *both* Supervisor and Subordinate roles in the organisational structure.

In the next two sections we look at management and functional contracts in more detail. *Management contracts* specify the type of communication acts and protocols that are permissible between the two parties. *Functional contracts* specify the performance obligations of each party to the other.

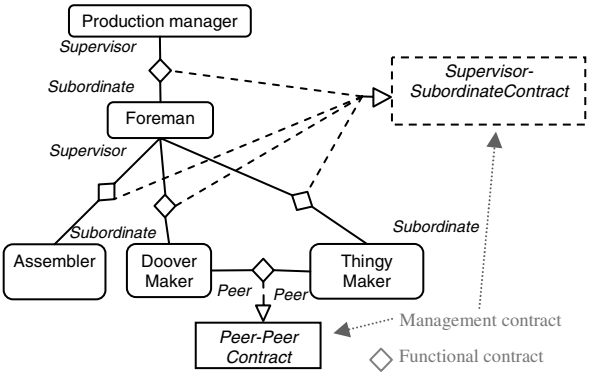


Fig. 4. Domain specific abstract organisation bound by contracts

3.1 Contracts at the Management Level

Contracts at the management level of abstraction *restrict* the types of method that roles can invoke in each other. From our example above, the Supervisor-Subordinate operational-management role association restricts interactions between the objects playing the ThingyMaker and the Foreman to certain *types* of interaction. For example, a ThingyMaker cannot tell its Foreman Supervisor what to do. Furthermore, these contracts only *allow* interactions between particular instances of object-roles. For example, the method `ThingyMaker.setProductionTarget()` can only be invoked by the ThingyMaker's own Foreman.

Control-Communication Acts. The control communication in management contracts can be defined in terms of *control-communication act* (CCA) primitives. These performatives abstract the *control* aspects of the communication from the functional aspects. In [5], we defined a simple set of CCAs for direct and indirect control and for information passing. As an example, Table 1 below defines a set of primitives suitable to a hierarchical organisatiop. Unlike the communication act primitives in agent communication languages such as FIPA-ACL [7], CCAs express only *control* relationships between the two parties bound in a contract, rather than the types of communication between two independent agents. Note that indirect CCAs carry a reference to a resource `r`. The set below is not logically complete. For instance, it does not capture a referential command relationship (A tells B to tell C to do something), but it is sufficient to allow us to define a number of contracts between

Table 1. Example of Control-Communication Act Primitives

Type of communication	Communication control acts
Direct Control	DO, SET_GOAL
Indirect Control	RESOURCE_ALLOCATE(r), RESOURCE_REQUEST(r)
Information	ACCEPT, REJECT, INFORM, QUERY, ACKNOWLEDGE

operational-management roles. From these contracts we can create organisational structures.

In terms of the primitives we defined above, Supervisors can initiate some types of interaction and Subordinates others. Initial CCAs for these roles are:

Supervisor initiated: DO, SET_GOAL, INFORM, QUERY, RESOURCE_ALLOCATE
Subordinate initiated: INFORM, QUERY, RES_REQUEST

Other forms of management contract such as Peer-Peer would have different sets of valid initial CCAs for each party.

CCA Protocols. Protocols can be defined from CCA primitives. These Control Protocols (CPs) are sequence patterns of CCAs that terminate when the task is achieved. They are at the same level of abstraction as CCAs. Only the form of communication between the parties is represented. There is no information about the content of the task.

There are a limited number of protocols that form sensible interactions. It is possible to represent these protocols as strings of CCAs between initiator and respondent. To do this we will encode the CCAs as single letters so that complete protocols can be represented as individual strings. The codes for the above CCAs (plus "no response") are defined as follows:

Table 2. CCA short hand codes

D	DO	I	INFORM	X	NO RESPONSE
G	SETGOAL	Q	QUERY	K	ACKNOWLEDGE
L	RESOURCE_ALLOCATE	A	ACCEPT		
S	RESOURCE_REQUEST	R	REJECT		

For further clarity we can apply the convention that initiator CCAs are capitalised, and respondent CCAs are in lower case. For example, the string "`Da`" indicates that the initiating party asks the respondent to do something, and that the respondent subsequently accepts. Control protocol clauses can also be represented by finite state machines (FSMs). These FSMs keep track of the conversation between two contracted parties. Clauses can have as a goal the *maintenance* of a state or the *achievement* of a state. In the case of maintenance clauses, successful transactions of the protocol will result in a return to a 'ready' state. The successful completion of achievement clauses will result in a 'completed' state for that clause. Where an interaction is following a protocol in which a response is expected but not forthcoming after a specified time, the protocol may specify that `n` retires are permitted before the clause is violated. The nodes in Fig. 5 below represent CCAs issued by either the initiator or the respondent in the transaction of a particular clause in the contract. The letters in the nodes are a short hand for CCAs, as defined in Table 2 above (initiator CCAs in capitals, respondent CCAs in lower case). The FSM for the Supervisor-Subordinate contract *valid* protocols starting with a Supervisor D is illustrated below.

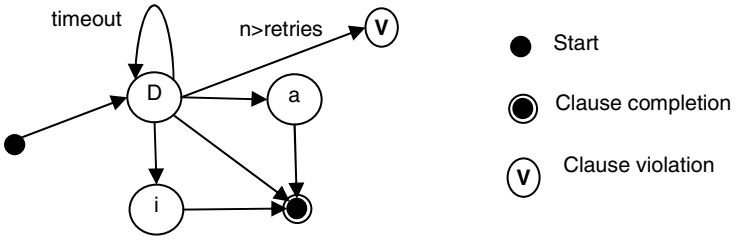


Fig. 5. Valid CCA protocol initiated with a Supervisor DO CCA

The form of our Supervisor-Subordinate management contract has been summarised in Table 3 below. The valid protocol sequences are expressed as strings as defined above. The management contract clause will be violated if the sequence of interactions does not follow one of these strings. For each protocol clause where a response is expected, values for the response timeout and the number of permissible retires would need to be specified (this is done at the functional level of the contract). Other types of management contract will have different sets of permissible protocols. For example, the protocol "DaR" is presumably acceptable in a Peer-Peer management contract where the "a" is a conditional accept. If management contracts are to enforce CCA protocols, they need to keep track of the state of communication between the roles that are party to the contract. This implies that there must be an instance of a contract for every association between roles.

Table 3. Example form of an operational-management contract

Management Contract	
Name	`Supervisor-Subordinate`
Party A	`Supervisor`
Party B	`Subordinate`
A initiated clauses	`D; Da; Di; I; Ik; Qi; G; Ga; Gi; L; La; ...`
B initiated clauses	`I; Ik; Qi; Sa; Sr; Si ; ...`

Management contracts are very limited in that they only monitor or enforce the *form* of the communication between the parties — there is no domain content apparent at the management control level of abstraction. Values for timeouts (in the event of no response), and values for the number of retries that are permitted, only make practical sense in relation to a domain specific function. Such values, along with the identity of the parties to the contract and other performance requirements for clauses and the contract as a whole, are provided by functional contracts. *Functional* contracts specialise abstract *management* contracts. It is the functional contract rather than the management contract that is instantiated. When a functional role in an organisational structure is bound to an operational-management role using such a contact, all functional role method invocations and responses between the parties are associated with CCA primitives.

3.2 Contracts at the Functional Level

Contracts, at the abstraction level of domain function (functional contracts), add schedule details to the form of contract defined by its management contract. They also define extra clauses that are needed to govern the domain-specific interactions between the functional roles. Instances of functional contracts, such as a Foreman-ThingyMaker contract, include a completed contract schedule as well as clauses relating to protocols, performance conditions and contract breach. They are specified as follows:

Schedule details such as the names of the parties to the contract and any temporal constraints on the contract – commencement date, duration etc – are specified. The operational-management roles in the management contract are specialised with functional roles. For example, the supervisor party is specialised as a foreman.

The rules for communication that are defined in the management contract protocols are made concrete. CCAs are mapped to the method signatures in the functional roles (see the section below on implementation). For example, the method `do_make-Thingy()` of the ThingyMaker `tm` would be matched to the DO CCA. This means a supervisor, contracted to `tm`, can invoke this method. If a *control* protocol is to be enforced for an interaction, a valid protocol string is assigned to the interaction, and values are specified for retries and timeouts. As long as the protocol is followed during the interaction, the contract clause will not be violated. Protocol sequences of particular *domain-function* interactions which need to be followed by the parties [8]might also be specified, as in [9]. For example, a foreman might be required to allocate the resources (thingy parts) to a thingyMaker, before it can ask the thingyMaker to make a thingy.

Clauses relating to any preconditions, post-conditions and invariants for the interactions are specified. These conditions are similar to those defined in the design-by-contract (DBC) approach [10], where such conditions are aimed at ensuring the correct functioning of the software. Consequently, these DBC conditions themselves cannot be changed. In functional-level contracts these conditions *can* be varied (provided the variation does not contravene correctness constraints). This allows variable NFRs to be expressed as conditions of the contract. For example, if there are costs associated with the performance of a function such as making a thingy, then the contract might specify the acceptable limit of those costs. Any performance conditions also are specified. For example, a foreman might specify the maximum time allowed for a contracted thingyMaker to produce a thingy. These NFRs reside in, and are enforced by, the contract rather than the component itself.

Finally, clauses relating to clause violation and what constitutes a breach of contract are specified. Some clause violations "go to the heart" of the contract and violation of the critical clause leads to automatic breach of the contract — the contract throws an exception. Other clauses may not be as critical to the contract and will be monitored. The contract contains metrics to measure the performance of the clauses: for example 'average time to make a widget'. Such metrics are monitored by the organiser role, which may then choose to abrogate the contract if the performance is unacceptable – and, for example, if another role-object is available to perform the

function. There may also be remedies for clause violation. If a clause is violated it may be permissible to renegotiate the contract e.g. to a different service level.

During execution, a contract itself monitors the interactions between the roles. The contract will prevent unauthorised or invalid interactions and monitor all interactions in order to maintain the state of execution of its clauses. The contract also keeps the state of any performance metrics updated. If a clause is violated, the contract informs the organiser role that controls it. Contracts may also be actively monitored by their organiser roles. We examine the interactions between contracts and organiser roles in the next section.

4 The Coordination-System

A coordination-system is constructed of contracts and the organisers that control them. Every organiser role is responsible for a cluster of roles and contracts. Organisers have three main functions: Firstly, organisers control and monitor their contracts. An organiser can instantiate, change, abrogate and reassign its contracts. Organisers monitor the contracts in their self-managed composite for under-performance. Secondly, organisers can reconfigure the contracts to try to remediate any underperformance that results from perturbations or changing requirements. Any such reorganisation must maintain the composite's viability. For example, if the structure is based on a Bureaucracy pattern [6], the organiser must ensure proper chains-of-responsibility (i.e. supervisor-subordinate chains) are maintained to preserve the functional flow-of-control. The organiser also creates role-object bindings [11] (the discussion of such bindings is beyond the scope of this paper).

Thirdly, organisers are the nodes of the coordination-system network. They interpret regulatory control messages that flow through this network and translate these messages into contract clauses. The coordination-system is a hierarchy in which non-functional performance requirements flow down, and information on the performance of the managed composites flow up. We will call these two regulatory control-message flows, respectively, *performance/constraint-propagation* and *performance-monitoring*.

The structure and adaptive behaviour of a coordination-system will be illustrated by looking at a ThingyMaking team within our Widget making department (WMD). The relationship between functional requirements and NFRs is illustrated with a production scheduling problem. We need to keep in mind that in an open system, the time taken to execute a function may vary or come at a cost.

Performance and Constraint Propagation. Performance requirements pass down the hierarchy of organiser roles to alter the performance requirements of the contracts. In our Manufacturing department the Production Manager receives (from above) orders for Widgets. It determines the priority of the orders, and passes these on to the Foreman (as determined by the contract `C1`). To fulfil its obligations under the `C1` contract, the foreman must organise the production of thingy orders within a specified timeframe. For example, the management level of contract `C1` allows the Production-Manager pm to invoke the Foreman `do_thingyOrder(…)` method. The contract `C1`,

with additional advice it receives from the organiser role (pm), can add performance requirements and constraints. For example, the contract may require that thingies be made within certain time constraints, or that certain resource costs not be exceeded.

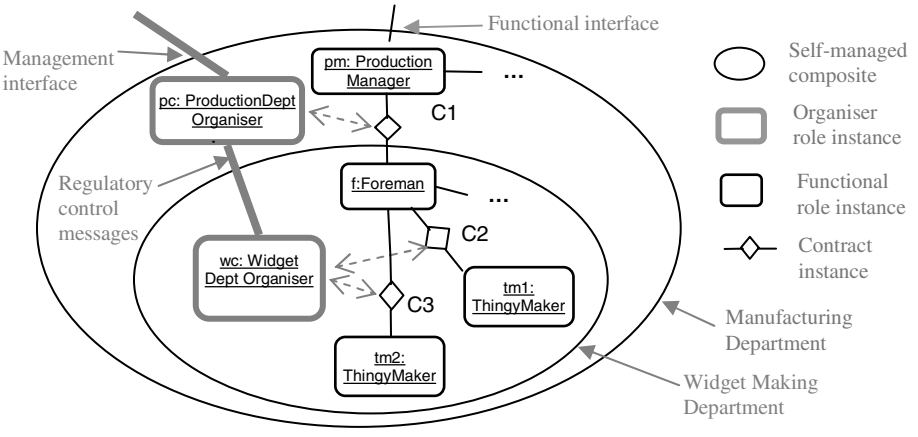


Fig. 6. Role-instance diagram of self-managed composites

The Foreman f in turn allocates work to, among others, the thingyMakers (tm1 and tm2). The Foreman can do this under the terms of the Foreman-ThingyMaker contract (instances C2 and C3) by invoking ThingyMaker's do_makeThingy() method. While the contracts C2 and C3 have the same form, these instances of the Foreman-ThingyMaker contract have different performance characteristics written into their respective contract schedules. Suppose the role-player object attached to tm1 can make 10 thingies per hour while the role-player object attached to tm2 can only make 5 thingies per hour. Because the Foreman f is party to contracts C2 and C3, it can use the performance capacity information in the schedule in deciding to whom the work should be allocated.

Performance and constraint information can be about either the *actual* capacity or the *required* capacity. The *actual*-performance and constraint information is held by the contracts themselves. For example, contracts C2 and C3 contain information on the respective thingy-making-abilities of their contracted thingyMaker role-objects. The source of this actual-performance information could be from a specification provided by the builder of the component/system, or it could be derived from the organiser role (e.g. the WidgetDeptOrganiser wc) monitoring the situated performance of the role-objects (see the next section).

Required-performance information on goals and constraints is transmitted through the organiser roles. The organiser role (e.g. wc) receives NFRs from the organiser above it in the coordination hierarchy (e.g. pc), then interprets these into performance requirements for the contracts it controls.

Performance Monitoring and Breach Escalation. In addition to its responsibility for characterising the actual-performance of role-objects mentioned above, the

organiser role can monitor the contracts to see if they are meeting their performance requirements. Changing requirements, environment or computational contexts can lead to the violation of performance clauses in the contract. If the *actual* capacity (as expressed in the contracts) falls below the *required* capacity, then the organiser must attempt to reconfigure the composite by altering the existing contracts, reassigning roles to more capable objects or creating new contracts and roles.

If this reconfiguration is beyond the capability of the composite organiser, the organiser must inform the role above it in the coordination system hierarchy. Contract breach can occur if the violation(s) is severe enough. If a breach occurs (or if the organiser has the intelligence to predict a breach from the actual and required performance information), the composite organiser needs to reconfigure the contracts and roles. In the figure above, if wc detects the thingyMakers it controls are, or will be, over-loaded, it may ask `pc` for resources to get more thingyMakers.

Such escalation can be viewed as an organised form of exception-handling where control-messages flow up through the *coordination*-system, before error-messages flow back up the *functional*-system's flow of control. Just as an animal detecting a threat will increase its adrenaline levels to stimulate the heart rate for flight or fight, so the coordination- system detects stress on the system then changes the parameters of contracts (or reorganises them) in an attempt to avert system failure.

The performance parameters that an organiser could monitor include the rate of the flow of data resources through the system, the state of communication networks or the computational loads on the objects performing the various roles. The organiser's ability to successfully reconfigure the composite will depend on its ability to sense the performance parameters from either the contracts or the environment, its ability to reason about the causes of underperformance, and its ability to implement effective re-organisations. The discussion of such adaptive feedback loops is beyond the scope of this paper.

5 Implementing Coordination-Systems Using Association Aspects

If a coordination-system is to be implemented as a separate concern, the contracts and organiser roles that constitute the system must be able to be defined separately both at conceptual and code levels. A coordination-system *cross-cuts* the program structure defined by the functional-roles and classes. This section briefly describes how we have implemented contracts using association aspects. A more comprehensive description can be found in [5] and [12].

Aspect-oriented methods and languages seek to maintain the modularity of separate cross-cutting concerns in the design and program structures. The AspectJ [13] extension to Java allows the programmer to define *pointcuts* that pick out certain *join points* (well-defined points in the program flow). An *advice* is code that is executed when a join point that matches a pointcut is reached. *Aspects* encapsulate such pointcuts and advices. These units of modularity can model various cross-cutting concerns.

Implementing contracts as aspects enables interactions to be monitored and controlled. As implemented in [5], method signatures follow an arbitrary naming convention that indicate their CCA type (e.g DO CAA method names start with `do_`). At compile time, CCA pointcuts are pattern matched against methods so that contract advice can be woven into the functional code. Such advice can prohibit or permit interaction. Advice can be inserted before and after method invocations, allowing the state of the contract to be updated and the performance of the contract to be measured. Pointcuts also allow the execution context of the invoking and target objects to be exposed. Thus the state of the objects that participate in the contracts can also be accessed. This enables pre and post conditions to be monitored and enforced.

Aspects, as currently implemented in AspectJ, do not easily represent the behavioural associations between objects [14]. Current implementations of AspectJ provide *per-object* aspects. These can be used to associate a unique aspect instance to either the executing object (*perthis*) or the target object (*pertarget*). When an advice execution is triggered in an object, the system looks up the aspect instance associated with that object and executes that instance. This allows the aspect to maintain a unique state for each object, but *not* for associations of objects.

Sakurai et al. [15] have implemented an extension to the AspectJ compiler to handle *association-aspects.* Association-aspects are declared with a `perobjects` modifier that takes, as an argument, a tuple of objects. Instances of these association-aspects are suitable for implementing management and functional contracts. A contract can be created as an aspect instance that associates a group of objects. In previous work [5], we have demonstrated the implementation of contracts that control the communication between roles. Due to space constraints here, our notated Java code that demonstrates the creation, revocation, and reassignment of contracts is available at [12]. The example code at [12] also shows how to intercept various CCA-type method-calls between various authorised and unauthorised parties; how to permit, modify or prohibit the execution of the interaction; and how to keep a contract FSM updated.

The use of aspects to implement contracts allows functional code to be developed independently from the contracts that associate them. The only dependency in the functional code is that method signatures need to follow a CCA naming convention. However, there is a limitation in using AspectJ to create contracts: both the coordination code and the functional code must be compiled together in order for weaving to occur. This means new classes cannot be added dynamically without recompilation. Other approaches, such as load-time weaving of byte-code, might prove effective in addressing this limitation.

6 Related Work

ROAD extends work on role and associative modelling in [1-4,16]. Kendall [2] has shown how aspect-oriented approaches can be used to introduce role-behaviour to objects. Roles are encapsulated in aspects that are woven into the class structure. While these role-oriented approaches decouple the class structure, they do not

explicitly define a coordination-system using management contracts. They are primarily concerned with role-object bindings rather than role associations.

The coordination model outlined here adopts a control-oriented [17] architectural approach, primarily focused on adaptivity rather than synchronisation. It has many similarities and some major differences with work by Andrade, Wermelinger and colleagues [18-20]. Both approaches represent contracts as first-class entities, and both use a layered architecture. In [19,20] the layers are Computation, Coordination and Configuration ('3C'). This is broadly similar to ROAD's four layer architecture (Computational-object, Functional-role, Management-contract, Organisation) with 3C's Computation layer similar to ROAD's Object and Functional role layers. 3C's contracts are method-centric rather than role-association-centric. They define a single interaction sequence that might involve many parties, whereas ROAD contracts are currently limited to two roles and many involve many types of interaction. Both approaches use contracts to model unstable aspects of the system, but 3C's focus is on business rules whereas ROAD focuses on performance variability. In 3C, there is no concept of a coordination network through which regulatory control messages pass.

The concept of a CCA in this paper is derived from the concept of a *communication act* in agent communication languages such as FIPA-ACL [7]. CCAs, as defined here, are much more restricted in their extent. CCAs deal only with control communication, and do not have to take intentionality of the other parties into account [21]. Work on roles has also been undertaken in multi-agent systems (MAS) [22-24]. In particular, [21] extends the concept of a role model to an organisational model. MAS systems, however, rely on components that have deliberative capability and more autonomy than the objects and roles discussed here. These agents negotiate interactions with other agents to achieve system level goals. These negotiations occur within a more amorphous structure than is defined here.

Like our approach, control-theoretic architectures separate control from functional processes [21]. Such systems are designed to maintain system viability during anticipated environmental perturbation, but they cannot be considered adaptive. Recent work on intelligent control [25] adds an adaptive loop on top of the operational control loop. This is similar to our concept of organiser roles that control the structure of the organization through manipulating contracts. In control-theoretic approaches there is no concept of role or object-role binding. Such systems are not structurally adaptive.

7 Conclusion

Separating management concerns from functional concerns can make systems more adaptive. In this paper we have introduced a framework for creating coordination-systems that control the interactions between functional roles. These coordination systems can be developed independently and the imposed on functional systems. They are built from a hierarchy of organiser roles that control the contracts between functional roles. ROAD contracts have management and domain function levels. Management contracts specify the type of communication acts and protocols that are permissible between the two parties. Functional contracts specify, among other things,

the performance obligations. Abstracting management-contract aspects makes possible, through contract inheritance, the reuse of their communication-control capability in many types of organisational structure.

There are a number of aspects of this framework that need further development. The set of CCAs that defined our example protocol is not complete and somewhat arbitrarily defined. This informality may suffice if operational-management contracts are only application or domain specific. However, if CCAs are to be generalised, a more rigorous approach may be needed. The UML 2.0 Superstructure Specification [26] provides a list of *primitive actions* which may provide the basis for a more formal definition of CCAs. Alternatively, agent communication languages such as [7] may provide the basis of a more rigorous definition. In this paper we have only briefly addressed the nature of organiser roles. While many features of such roles will be domain-specific, there are general principles of organisational viability that need to be elaborated for organiser roles. The discussion of indirect control is also under-developed. In our example, the resource allocation clause gave permission to the Superordinate to access any subtype of Resource. In practice, different roles are likely to have access to different resources. It follows that we need to develop some scheme of resource ownership or access rights. We also assume that the interfaces of the functional roles are compatible if they are to enter into contracts. Issues of functional compatibility and component composition need to be addressed.

References

[1] Bäumer, D., Riehle, D., *Siberski*, W., and Wulf, M. "Role Object" in *Pattern languages of program design 4*, eds. Harrison, Foote, and Rohnert, H. Addison-Wesley, 2000, pp. 15-32.

[2] Kendall, E. A., "Role Modelling for Agents System Analysis, Design and Implementation" *First International Symposium on Agent Systems and Applications IEEE CS Press*, 1999

[3] Kristensen, B. B. and Osterbye, K., "Roles: Conceptual Abstraction Theory & Practical Language Issues" *Special Issue of TAPOS on Subjectivity in Object-Oriented Systems*, 1996

[4] Lee, J. S. and Bae, D. H., "An enhanced role model for alleviating the role-binding anomaly" *Software: practice and experience*, vol.32, 2002, pp. 1317-1344.

[5] Colman, A. and Han, J., "Operational management contracts for adaptive software organisation," *Proc. Australian Software Engineering Conference (ASWEC 2005),* 2005.

[6] Riehle, D. "Bureaucracy" in *Pattern Languages of Program Design 3*, eds. Martin, Riehle, and Buschmann. Reading, Massachusetts: Addison-Wesley, 1998, pp. 163-186.

[7] The Foundation for Physical Intelligent Agents, *FIPA Communicative Act Library Specification* http://www.fipa.org/specs/fipa00037/, 2002, *last accessed 27 Aug 2004*

[8] Bracciali, A., Brogi, A., and Canal, C., "Dynamically Adapting the Behaviour of Software Components," *Proc. Coordination'02 LNCS 2315,* York, UK, 2002.

[9] Han, J. and Ker, K. K., "Ensuring Compatible Interactions within Component-based Software Systems" *Proc.10th Asia-Pacific Software Engineering Conference(APSEC)* 2003.

[10] Meyer, B. *Object-oriented software construction*, New York: Prentice-Hall, 1988.

[11] Colman, A. and Han, J., "Organizational abstractions for adaptive systems," *Proceedings of the 38th Hawaii International Conference of System Sciences,* Hawaii, USA, 2005.
[12] Colman, A. and Han, J., “Implementation of Contracts using Association Aspects”, SUT Report, SUTICT-TR2005.04/SUT.CeCSES-TR007 www.it.swin.edu.au/centres/CeCSES, 2005.
[13] Eclipse Foundation, *AspectJ* http://eclipse.org/aspectj/, 2004, *last accessed 7 Oct 2004*
[14] Sullivan, K., Gu, L., and Cai, Y., "Non-modularity in aspect-oriented languages: integration as a crosscutting concern for *AspectJ*," *Proc. of the 1st international conference on Aspect-oriented software development, AOSD 02,* Enschede, The Netherlands, 2002.
[15] Sakurai, K., Masuharat, H., Ubayashi, N., Matsuura, S., and Komiya, S., "Association Aspects," *Proc. of the Aspect-Oriented Software Development '04,* Lancaster U.K, 2004.
[16] Kendall, E. A., "Role model designs and implementations with aspect-oriented programming." *Proc. Object-Oriented Systems, Languages, and Applications,* 1999.
[17] Arbab, F., "What Do You Mean, Coordination? " *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, vol.March, 1998
[18] Andrade, L., Fiadeiro, J. L., Gouveia, J., Koutsoukos, G., Lopes, A., and Wermelinger, M., "Patterns for coordination ," *Coordination '00 LNCS 1906,* pp. 317-322, 2000.
[19] Wermelinger, M., Fiadeiro, J. L., Andrade, L., Koutsoukos, G., and Gouveia, J., "Separation of Core Concerns: Computation, Coordination, and Configuration," *Workshop on Advanced Separation of Concerns in Object-Oriented Systems, OOPSLA*, 2001.
[20] Andrade, L., Fiadeiro, J. L., Gouveia, J., and Koutsoukos, G., "Separating computation, coordination aand configuration" *Journal of Software Maintenance and Evolution: Research and Practice* , vol.14(5) , 2002, pp. 353-369 .
[21] Zambonelli, F., Jennings, N. R., and Wooldridge, M., "Developing multiagent systems: The Gaia methodology" *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol.12(3) , 2003, pp. 317-370 .
[22] Juan, T., Pearce, A., and Sterling, L., "ROADMAP: extending the Gaia methodology for complex open systems" *Proceedings of the first international joint conference on Autonomous agents and multiagent systems, Bologna, Italy, ACM*, 2002, pp. 3-10.
[23] Odell, J., Parunak, H. V. D., Brueckner, S., and Sauter, J., "Changing Roles: Dynamic Role Assignment" *Journal of Object Technology, ETH Zurich*, vol.2(5) , 2003, pp. 77-86.
[24] Zambonelli, F., Jennings, N. R., and Wooldridge, M. J., "Organisational Abstractions for the Analysis and Design of Multi-Agent Systems," *Workshop on Agent-oriented Software Engineering ICSE 2000,* 2000.
[25] Herring, S. and Kaplan, C., "Viable Systems: The Control Paradigm for Software Architecture Revisited" *Australian Software Engineering Conference*, 2000, pp. 97-105.
[26] Object Management Group, *UML 2.0 Superstructure (Final Adopted specification)* http://www.uml.org/#UML2.0, 2004, *last accessed 13 Oct 2004*

Coordination with Multicapabilities

Nur Izura Udzir[1,2], Alan M. Wood[1], and Jeremy L. Jacob[1]

[1] Department of Computer Science, University of York, UK
{izura, wood, Jeremy.Jacob}@cs.york.ac.uk
[2] Department of Computer Science,
Universiti Putra Malaysia, Malaysia

Abstract. In the context of open distributed systems, the ability to coordinate the agents coupled with the possibility to control the actions they perform is important. As open systems need to be scalable, capabilities may provide the best-fit solution to overcome the problems caused by the loosely controlled coordination of LINDA-like systems. Acting as a 'ticket', capabilities can be given to the chosen agents, granting them different privileges over different kinds of data—thus providing the system with a finer control on objects' visibility to agents. One drawback of capabilities is that they can only be applied to named objects—something that is not universally applicable in LINDA since, in contrast to tuple-spaces, tuples are nameless. This paper demonstrates how the advantages of capabilities can be extended to tuples, with the introduction of *multicapabilities*, which generalise capabilities to collections of objects.

1 Introduction

Coordination is essential in open systems, where agents and active objects are free to join and leave the system at any time, i.e. they need not be defined prior to starting the infrastructure ('middleware'). The discussion in this paper is based on the tuple-space (TS), or LINDA model [9, 3, 8] as an open distributed system. The shared data space coordination paradigm is a popular alternative to the conventional point-to-point communication approaches. LINDA is distinguished by its temporal and spatial separation properties, as well as its independence from any computation language or machine architecture—essential properties for coordination in open systems. Now a mature technology, research in TS-based coordination is providing general-purpose data spaces to create efficient large-scale implementations of open distributed multi-component systems.

There is no doubt of LINDA's power for coordination in an open, heterogeneous environment. However, in order to profit from the advantages of open and flexible coordination mechanisms, a number of challenging practical problems need to be addressed. These have been noted by many authors in the past, and several solutions have been proposed, all imposing varying degrees of additional control by the system. Unfortunately, getting the optimum balance betweenflexibility and tighter control is difficult, and many of the proposed solutions lose the principal advantages that LINDA-like systems have over many other models.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 79–93, 2005.

It is useful, however, to provide a finer control over the agents' interactions and coordination, than is available in the 'classic' LINDA model. This is particularly challenging in a decentralised, and distributed environment, while at the same time maintaining the flexibility inherent in open systems. One aspect of having a finer control is to be able to restrict what methods an agent is allowed to invoke on an object. Earlier work on coordination using object attributes [17] demonstrated a simple solution to control agents' access on objects in the system without resorting to any complex cryptographic security approach. Together with access control lists (ACLs), this earlier paper also discussed the advantages of capability-based control [6, 12] in distributed environment.

The concept of capabilities is not new. Although various capability systems have been developed over the years, and is still an active research in some areas such as in object-based systems, it does not enjoy the same popularity in the TS-based systems.[1] It seems that capability-based coordination has yet to demonstrate a significant impact to improve LINDA-like coordination in the open distributed environment.

It has been established that capabilities offer more flexibility than ACLs, thus making them more attractive for open systems. However, unlike ACLs, capabilities must refer to single named objects. In the LINDA context, TSs are uniquely identifiable—therefore, they can be referenced by capabilities—whereas tuples are anonymous: they can only be referred to using associative matching. We propose a solution to this problem by using *multicapabilities*, as will be elaborated later in the paper.

1.1 The TS Coordination Model

The LINDA coordination model [9] used as the platform for discussion in this paper promotes *generative* communication where agents interact by 'generating' data (an ordered collection of typed values called a *tuple*) into a shared data space known as a *tuple-space* (TS). A tuple can be retrieved, destructively or otherwise, from the TS by specifying a *template* whose pattern matches the tuple. The associative matching retrieval is *non-deterministic*: a retrieving agent may get *any tuple* that matches its template; and a tuple may be given to *any agent* specifying a matching template.

The basic primitives to enable agent interactions defined in the model are: `out` (to write a tuple into a TS), `rd` (to read a copy of the tuple that matches the template), and `in` (a destructive version of `rd`). Both `rd` and `in` block if no matching tuple is available. Non-blocking versions of these primitives, `inp` and `rdp` were originally introduced. However, the definition of these primitives was ill-formed, until the proposal of a principled semantics of `inp` [11].

[1] At least, none have been proposed as a 'pure' capability system, as capabilities are combined with other security techniques.

1.2 Capability

A capability [14] is an unforgeable 'ticket' given to an agent that specifies which kind of actions on a certain object are permitted to the holder of the capability. We can define capabilities in a more general way: as 'visibility' filters to create a more refined control over agents' actions on objects in the system.

It is well-known that capabilities are more suitable for open distributed systems as they themselves are *distributed* in the sense that:

- the controlling attributes are held by the agents, rather than being attached to objects, thus putting no storage overhead on the objects.
- verification is made by the kernel upon the presentation of the capability by the agent, without the need to search any list. Therefore, verification time is constant in capability-based systems.
- the decision to grant a capability to an agent is the responsibility of some holder of the capability, not the kernel[2]. The kernel only generates and checks the capabilities.
- capabilities can be transferred from one agent to another. This is a form of 'distribution' as, subject to certain constraints, any agent (not necessarily the object creator) possessing the capability can pass a copy of it to another agent.

In addition, a capability mechanism also supports the *flexibility* inherent in distributed systems: it accommodates user-defined rights, not restricted to those fixed by the system, thus allowing them to be dynamically changed; and its 'domain flexibility' feature allows agents to join and leave the system simply by requesting (and possessing) appropriate capabilities to access the objects, as opposed to having to modify numerous lists attached to each object relevant to the agents' execution. Despite the advantages of implementing capabilities, this approach, however, is more complicated compared to ACLs. Nevertheless, due to its suitability for open distributed systems, and partly inspired by this challenge, we pursue this area of research to obtain a more refined control in such an environment.

2 Multicapabilities

Multicapabilities extend capabilities for a collection (multiset) of 'things', rather than the traditional notion of a (uni)capability referring to a single *named* object. Whereas a permission in a uni-capability allows its holder to operate in a certain way on the *object* it refers to, a permission in a multicapability allows the operation on an *element* within the group, not on the entire group referred to by the multicapability. In the LINDA context, multicapabilities extend capabilities to

[2] The term 'kernel' in this paper refers to the underlying distributed mechanism that controls all operations in the system. It represents the totality of the LINDA 'middleware'.

apply to *nameless* tuples without jeopardising the associative matching and the non-deterministic properties of the TS model. This enables various operations to be performed on the tuples as they can now be referenced.

Tuples are classified by their multicapabilities. Two tuples with the same pattern (possibly having the same, or different values) may be referred to by two different multicapabilities, while the same multicapability can refer to (a collection of) many different tuples (with any value). To illustrate this, let us consider a simple example where an agent, A, wishes to `out` a tuple of an integer and a character, $\langle$`?int,?char`$\rangle$. Before referring to a class of tuples, either for output or retrieval, it must first request an appropriate multicapability from the kernel. It will be given a multicapability[3] for the template (with full rights), to which the kernel will assign a unique multicapability name, e.g. α:

$$c1 = [\alpha, \langle ?int, ?char \rangle, P]$$

where P is the set of all permissions, granting the agent full rights to perform all operations on the template (i.e. group) of tuples. For the purpose of this discussion, let us assume that $P = \{i, r, o\}$, where i, r and o represent permissions to perform the operations `in`, `rd`, and `out`, respectively.

If A requests another multicapability for a similar template from the kernel, it will receive a new multicapability, different from $c1$:

$$c2 = [\beta, \langle ?int, ?char \rangle, P].$$

Now assume another agent, B, requests a multicapability for a similar template from the kernel, and receives a multicapability which it stores in $c3$:

$$c3 = [\gamma, \langle ?int, ?char \rangle, P].$$

Using the multicapabilities they possess, the agents can perform operations (within those permitted by P), e.g. writing tuples into a TS (Fig. 1).

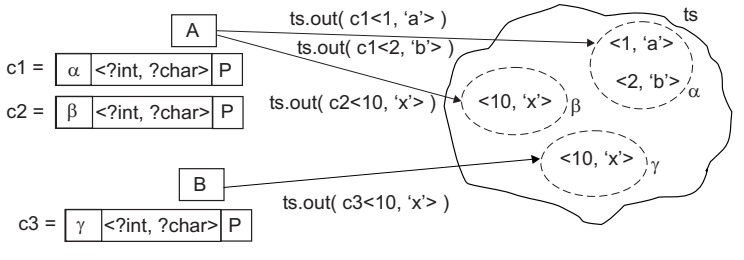


Fig. 1. TS operations using multicapabilities

Even though the templates are similar in pattern—a sequence of an integer followed by a character—they are essentially different, distinguished by the

[3] A more formal definition of the multicapability structure will be given in Section 2.1.

multicapabilities used to create (`out`) them. The result of any input operation performed by the agents will be limited to only the tuple(s) within the specified multicapability group. For example, a `rd` operation for ⟨`?int,?char`⟩ performed by A using $c1$ will retrieve tuple ⟨`1,'a'`⟩ or ⟨`2,'b'`⟩, but not any of those in other regions.

Likewise, if B performs an input operation

```
ts.in( c3⟨?int,'a'⟩ );
```

the operation will block—the template ⟨`?int,'a'`⟩ of multicapability $c3$ does not match tuple ⟨`1,'a'`⟩ of multicapability $c1$, even though ⟨`?int,'a'`⟩ and ⟨`1,'a'`⟩ are of the same pattern.

Therefore, multicapabilities can provide a template-based partitioning facility, thus enabling certain operations to be performed on a tuple of a specific group, but not on one of another group, even though both groups have the same template.

2.1 Basic Structure

A multicapability refers to a group of objects of a certain template, or pattern. In the LINDA context, a template is defined as sequence of types and/or scalars. A type can be modelled as the set of values it contains, and so $Template \in \mathbb{P}(Value)$. We define a multicapability as a structure consisting of three parts: u, a unique tag or identifier which acts as a reference to a collection of objects; t, a template of the objects; and p which denotes the set of actions permitted on the collection. In object-oriented terms, this set corresponds to a sub-interface of the methods in the objects' class. Hence, a multicapability c can be written as $[u, t, p]$. In the case of (uni)capabilities, the capability for a *single* object o is also the 'handle' to the object, with permission p attached, $[o, p]$.

A capability represents the permission given to the holder of the capability to perform some action on an object, and the capability must be presented to the kernel for verification before the attempted action is allowed to be carried out. Having that permission would mean that the action is valid for the target object. If an agent attempts to perform an action, a on object o, using capability $c = [o, p]$, it will be allowed to take place (*succeed*) if $a \in p$, otherwise an error will be thrown (*fails*). Likewise, in the case of multicapability, e.g. $c = [u, t, p]$: an action a with a value v is allowed to take place if a is within the permission set p of c, and v matches the template t of the multicapability region identified by u. Therefore, we define a function *allowed* where

$$allowed([u, t, p]) = \{a.v | a \in p \wedge v \textbf{ match } \langle u, t \rangle\}$$

In LINDA, there are four possibilities if the agent does not possess a valid right for a: the action either blocks, throws an exception, or fails. In the case of an `out` action, there is an additional possibility: the action may simply return, as in the original semantic of `out`. If the implementation supports *exception*, then an exception is raised (by the host computational language) and the agent can resume

its execution. Unfortunately, the implication of this is that the orthogonality of the coordination and computational language is broken [4].

Blocked input/output operations affect the system differently. Superficially, the agent cannot detect whether an input operation is blocked because of either (1) the unavailability of a matching tuple, or (2) the permission being denied due to invalid (multi)capability. Thus, from the user's point of view, there is no difference in the behaviour of the system. A *blocked output* operation, however, would be noticeable as a fundamental alteration in the semantics of `out` has been forced [17]—`out` never blocks in the standard LINDA model. In both input and output operations, if the operation blocks due to an invalid capability—which essentially means that it "blocks until the agent possesses the required capability", and it is obvious that it cannot 'possess' the capability simply by waiting there—it will block forever. In order to obtain the capability, the operation must first return to the agent, who will send a request (and wait) for the appropriate capability before retrying the operation.

There has to be a mechanism to handle exceptions or failure in both input and output operations. The practical solution to this is to use the deadlock breaking `inp`/`rdp` [11], or some other exception handling mechanism to inform the agent that the given operation is not allowed.

It should be reiterated that we view the permissions in a (multi)capability as a set of names of methods that the holder is allowed to invoke, and so are not limited to access control, but represent a more general concept: *visibility*—an agent cannot see a method that is not listed in the (multi)capability it holds. As the term 'object' can refer to anything that is appropriate to the system, the set of rights may well include any appropriate, even user-defined, operations.

The basic structure of our model is based on the following rules:

1. Every TS and tuple operation requires a (multi)capability. Our model requires that each agent performing an operation on a tuple must obtain a capability to access the TS where the tuple resides (or to be written to), as well as the multicapability to operate on the tuple. Therefore, prior to performing an operation, all agents must first request:
   - a (uni)capability for the target TS, and
   - a multicapability for a specified template.
2. Every request for a new (multi)capability returns a new, unique (multi)capability (even for identical patterns), with full rights.
3. All agents have the full capability (with full rights) for the universal TS (UTS), i.e. a default space that exists throughout the life-span of the system, and that is (publicly) accessible by all agents in the system. This unicapability only represents the permission to perform operations on the contents of UTS in general, but multicapabilities for tuples are required in order to operate on these tuples (See Rule 1).
4. Each agent has a default universal capability for (multi)capabilities, $cc[\langle ?cap \rangle]$ with full rights, where *cap* is a (multi)capability type, to enable capabilities and multicapabilities to be passed among the agents. For brevity, the set of rights for *cc* is not shown as it has been established that *cc* represents full-rights.

The capability data (unique id, reference/template, and permissions) are assumed to be securely encapsulated in the (multi)capability and only interpretable by the kernel when the multicapability is presented for verification. To avoid confusion, it is important *not* to see multicapabilities as 'tagging' tuples—this leads to an ACL-like view. Rather, it is better to view tuples being 'grouped' into regions specified by the multicapabilities referring to them. Each region then (virtually) exists in every TS.

2.2 Operations on Multicapabilities

A multicapability may be copied in order to be given to other agents. Each copy would have the same unique tag of the original's, thus referring to the same group of objects. However, the template and permissions in a (copy of a) multicapability can be reduced to allow agents more control over the visibility of the objects they created to other agents. Reducing *permissions* in a multicapability is a fairly straightforward operation: the permissions $p2$ in the reduced copy of a multicapability should not exceed those in the original multicapability ($p1$), i.e. $p2 \subseteq p1$. The *template* of a copy of a multicapability can be the exact copy of the original template, or specialised to a sub-type of the same pattern.

We define two operators for this purpose: $-$ and @, where $c - s$ is capability c *without* the permissions in s, and $c@s$ is c *with only* the permissions in s that c also possesses. Even though the two operators serve similar purposes, they are separately defined for convenience, for instance, it is easier to express a sub-multicapability with only two (out of ten) permissions granted using @, rather than using $-$.

Let $c = [u, t, p]$ be a multicapability, and s a set of permissions where $s \subseteq p$, then

$$\_ - \_ \in Mulc \times \mathbb{P}(P) \longrightarrow Mulc \qquad \_@\_ \in Mulc \times \mathbb{P}(P) \longrightarrow Mulc$$
$$[u, t, p] - s = [u, t, p \backslash s] \qquad [u, t, p]@s = [u, t, p \cap s]$$

For example, consider a multicapability for a template of an integer and a character, $c = [\alpha, \langle ?int, ?char \rangle, \{i, r, o\}]$. The expression `c - {i}` creates a sub-multicapability of c with all the permissions of c except $\{i\}$. This expression is equivalent to `c @ {r,o}` which has all the permissions of c restricted to $\{r, o\}$. Another dimension of 'reduction' is to specialise the template by fixing the values of its element(s). For example, the template in a copy of c, c' say, may be ⟨`?int, ?char`⟩, or the value of the first element is fixed to a certain integer value, and/or a fixed character value for the second element. For example, the expression `c'`⟨`?int,'y'`⟩ will limit the number of possible matches to only those tuples with the character `'y'` as its second element.

It should be emphasised that there is *no way* of adding permissions to a multicapability, nor to expand the scope of the template in the multicapability. For instance, if the template in the original multicapability is ⟨`?int,'x'`⟩, then a copy of it can never have a template with the second element set to `?char` as this will generalise the template, and would defeat the purpose of the use of capabilities, which is to obtain a finely controlled environment in open distributed

systems by incorporating capabilities to limit objects' visibility to agents in the system. Therefore, the *derived-from* relation is:

$$[u, t, p] \preceq [v, s, q] \equiv u = v \\ \wedge\ t \leq s \\ \wedge\ p \subseteq q$$

where $t \leq s$ have the meaning of $\forall i \in 1..n \bullet t_i \in s_i \vee t_i = s_i$, and i is the index of an element in a template; n is the length of the template, i.e. the maximum number of fields in a template allowed by the system.[4]

Derivation contributes to finer control over the objects in the system as it provides a means to have different versions of a multicapability (with different restrictions) referring to tuples of the same template.

Example: Suppose that an agent A wishes to `out` a tuple $\langle$`1,2`$\rangle$. Before doing so, it has to request an appropriate multicapability from the kernel, and is given a multicapability for two integers with full rights, to be stored in variable $c1$: $[\alpha_1, \langle ?int, ?int\rangle, \{i, r, o\}]$. A may then make a modified copy of the multicapability if necessary, by 'reducing' the template and the list of rights, before writing the tuple (in the form of the newly derived multicapability) into a TS.

Agent B wants to read a tuple that matches the template $\langle$`?int,2`$\rangle$. If it requests a new multicapability for the template, it will receive a unique multicapability, different from the one given to A. This means that it can never retrieve any tuple produced by A (or any other producer, for that matter) with its newly acquired multicapability. If B wishes to read the tuple `out`'ed by A, it must use the same multicapability (or a derivation) as the tuple's. One way of doing so is to obtain it from A: since every agent automatically gets a default cc, A can pass the tuple containing the said multicapability to B via `ts`.

The following code excerpts illustrate the agents' operations where A `outs` the tuple $\langle$`1,2`$\rangle$ and the restricted multicapability $[\alpha_2, \langle ?int, ?int\rangle, \{r, o\}]$; and B retrieves the necessary multicapability before reading a two-integer tuple using the multicapability.

Agent A:

```
// Request new multicapability
c1 = cc.newcap( <?int,?int> );
// c1 now is [α1,<?int,?int>,{i,r,o}]
c1_1 = c1 - {i};       // reduce permission
// c1_1 holds [α2,<?int,?int>, {r,o}]
ts.out( c1<1,2> );     // write tuple into ts
ts.out( cc<c1_1> );    // write capability for the tuple
```

[4] Note that a more general and expressive condition based on *sub-typing* is possible. However, for simplicity we only consider the relation based on templates here.

Agent B:

```
// Retrieve the multicapability
cap1 = ts.in( cc<?cap> );
// Assuming cap1 now holds [α2,<?int,?int>,{r,o}]
data = ts.rd( cap1<?int,2> );
```

Prior to performing an action, the kernel will verify that the intended action is valid based on the multicapability presented by the agent. To `out` a tuple, A has to present the multicapability $c1$ for verification, and since the `out` permission in its set of rights verifies that the action is valid, the action is allowed to proceed. Note that A can `out` a tuple of $c1$ or $c1\_1$: both multicapabilities refer to the same collection of tuples.

After obtaining $c1\_1$, B can specialise the values (in the template) to suit its need. As $c1\_1$ is a sub-capability of $c1$, it still refers to the same object(s) as $c1$, but with restricted permissions.

2.3 Combining Multicapabilities

In this paper, we also present preliminary work on combining multicapabilities: operations which take one or two multicapabilities and produce a new multicapability. We are developing a calculus of multicapabilities by investigating what operations may be sensibly defined for combining them. For example, three obvious operations that might be performed on two multicapabilities are the union, intersection, and relative negation.

Naturally it is sensible to only allow the combination of those multicapabilities with a similar pattern that belong to the same agent. The multicapability produced as a result of these operations may be stored in another capability variable.

Union. Assuming the function *allowed* given in section 2.1, the *union* of two multicapabilities c and d can be defined by

$$allowed(c \cup d) = allowed(c) \cup allowed(d).$$

The expression produces a multicapability referring to the templates of either c or d, and represents permissions if *either* c or d (or both) grants that permission. If both multicapabilities refer to the *same* collection of tuples, i.e. $c_u = d_u$,[5] then

$$c \cup d = [c_u, c_t, (c_p \cup d_p)].$$

To further elaborate on these operations, let us consider the following multicapabilities:

$$c1 = [\alpha, \langle ?int, ?char \rangle, \{i, r, o\}]$$
$$c2 = [\beta, \langle 3, ?char \rangle, \{r, o\}]$$

[5] We shall use c_u, c_t, and c_p hereafter to refer to the unique tag u, the template t, and the permission set p of multicapability c.

If $c1$ and $c2$ refer to the same group, writing a combined tuple of these multicapabilities, e.g.

$$\texttt{ts.out((c1} \cup \texttt{c2)}\langle \texttt{1,'a'} \rangle \texttt{);}$$

produces the tuple into the group referred to by both multicapabilities. It should be emphasised that the semantics allow the tuple to be of either of the two templates—which is reasonable as the agent *does* possess these multicapabilities. The union operation merely gives it an added advantage. The produced tuple can be retrieved by any agent possessing (copies of) either $c1$ or $c2$, and not necessarily both multicapabilities. However, as the tuples are written using multicapabilities of different rights, then any agent who has $c1$ (or both multicapabilities) can `in` or `rd` the tuple, while those with $c2$ can only `rd` it.

A read operation using the union of $c1$ and $c2$, would operate on the tuples that match the templates $\langle$`3,'a'`$\rangle$ or $\langle$`?int,?char`$\rangle$ from the region of $c1$ or $c2$.

If the multicapabilities refer to *different* collections of objects, then the operation should be a *disjoint union*. Uniting (conjoining) the templates and the permission sets would be semantically incorrect, as the permission sets cannot be merged—even though they contain method(s) with the same name, these methods are of different signatures, e.g. method m in (signature) c is not the same as method m in d: they are applicable to different collections of objects. Therefore, a disjoint union of multicapabilities referring to two different groups is defined as

$$c \cup d = [c_u, c_t, c_p] \sqcap [d_u, d_t, d_p].$$

This means that the intended action will be performed on *either* group non-deterministically chosen by the kernel.

Let g and h be two different groups of objects referred to by c and d, respectively. To perform the disjoint union of c and d, would mean that the system will (non-deterministically) choose one of the groups, before performing the action a on an object in the said group. Assuming the group selected is g, the action succeeds if $a \in c_p$, and fails otherwise. In LINDA, there is another richer possibility: if $a \notin c_p$, instead of failing, the action blocks (on g), but the system may allow the action to be performed on the other group, i.e. h. If the action succeeds this time around, upon completion, the previously blocked action will be broken.

The union operation allows actions permitted by any of the two multicapabilities to be performed on either of the two templates for the same collection of tuples. However, if the union involves different collections of tuples, the action will only be permitted if it is allowed by the permission set of the group non-deterministically selected by the kernel.

Intersection. Generally, an *intersection* of two multicapabilities c and d is defined by

$$allowed(c \cap d) = allowed(c) \cap allowed(d)$$

which is a multicapability referring to the 'lesser' (i.e. the less generic) template of the two, and represents permissions only if *both* c and d grants that permission. If c and d refer to the *same* group of objects, then

$$c \cap d = [c_u, (c_t \cap d_t), (c_p \cap d_p)].$$

Otherwise, if c and d refer to *different* groups, then

$$c \cap d = [(c_u, d_u), (c_t \cap d_t), (c_p \cap d_p)]$$

where (c_u, d_u) implies that the action will be performed on *both* regions simultaneously. As the permission sets are intersected,

An `out` operation using $(c1 \cap c2)$ will write a tuple which can only be of the template $\langle$`3,?char`$\rangle$ which is 'less' than $\langle$`?int,?char`$\rangle$. The tuple is written into the region referred to by both multicapabilities, if $c1_u = c2_u$; otherwise, if the multicapabilities refer to different regions, then it is written into the 'shared' (i.e. intersecting) section of the two multicapabilities, which is tagged with $(c1_u, c2_u)$. This tuple can only be retrieved by an agent that possesses both multicapabilities. In the previous example of $c1$ and $c2$, the kernel will only allow the tuple to be `rd`—the only permission granted by both multicapabilities.

A read operation using $(c1 \cap c2)$ will operate on tuples of the more restricted version of the two templates that exist in both groups, which can be regarded as the shared region of $c1$ and $c2$.

Therefore, intersecting two multicapabilities allows actions limited to only those permitted by both multicapabilities to be performed on the less generic of the two templates.

Negation. Another conceivable operation is the *relative negation* of multicapabilities, which is defined by

$$allowed(c - d) = allowed(c) - allowed(d).$$

The produced multicapability has a permission *only if* c grants that permission, but d does not, and operates on tuples of the template of c, except those of the d template.

$$c - d = [c_u, (c_t \cap \neg d_t), (c_p \backslash d_p)]$$

For example, let $c3$ be another multicapability as an addition to the previously defined $c1$ and $c2$,

$$c3 = [\gamma, \langle 3, ?char \rangle, \{i\}].$$

An output operation using the negation expression $(c1 - c3)$ will write a tuple of template $\langle$`?int,?char`$\rangle$, excluding any tuple whose first element is `3`. This tuple will be written in the $c1$ region, and can only be retrieved by agents holding a copy of $c1$; it is not accessible with $c3$.

Similarly, an input operation using the expression $(c1 - c3)$ will operate on tuples in the $c1$ region, while selectively disregarding any tuple that matches the template of $c3$.

If we allow negated permissions, then

$$c - d = c \cap \neg d$$
$$\forall\, t, p \bullet c_t \geq d_t \wedge c_p \supseteq d_p$$

The constraint is to avoid such cases where a completely empty multicapability being negated to get a universal multicapability.

The relative negation allows actions permitted by the multicapability on the left-hand side of the operator, except those in the right-side multicapability to be performed on any of tuple that matches the left-side template, but not the other.

The above operations are the most obviously applicable given the fundamental properties of a multicapability. There are certainly others that might usefully be defined, and so part of the further development is to identify a *sufficient* set of operations. However, a minimal set, if one can be defined, may not be the most practical since no account has yet been taken of the *implementability* of the operations. For instance, the union of two capabilities is trivially implemented, whereas the intersection is much more intricate—consider the result of an arbitrary sequence of unions and intersections of such expressions.

This paper focuses on identifying *conceivably useful* operations which can be performed on multicapabilities (and capabilities), and work on the *feasible implementation* is ongoing.

3 Related Work

Although LINDA has become a well known coordination model, as an alternative to the conventional communication paradigms, it is sometimes considered as 'too open and too flexible', leaving it vulnerable to accidental, or even malicious, manipulation. Much work has been carried out to impose more control, at the expense of its most attractive feature—flexibility. Capabilities, on the other hand, provide the mechanism to finely control a system without losing the flexibility essential in open environment. Examples of early (and somewhat limited) attempts at improving LINDA using a capability-like mechanism is Pinakis's Joyce-LINDA [15], which uses public-key encryption to support capabilities, and Law-Governed LINDA (LGL) [13]. Unfortunately, the capability values in Joyce-LINDA cannot be matched by formals in templates, thus hindering its pass through TS. Introducing an additional special data type to enable the matching process only complicates things. The notion of capability-based control in LGL is somewhat restricted: capabilities are required in order to send a message to another agent, but none is necessary to receive one. This means that a message can only be sent if the sender has the capability for the target agent. Therefore, there is a possibility of indefinite waiting, since these agents might not be aware of the sender's need to acquire the necessary capability.

SECOS [1] provides a facility to define 'views', using a rather complex, two-level cryptographic scheme: all tuple fields are locked with a key, and each field must be locked with a different key. Unfortunately, the problem with using a

key-related control is that it is not possible to discriminate the rights for certain operations: if an agent possesses a key to a tuple, there is no way of restricting the permission, e.g. for a read-only, but not remove, operation. Indeed, their can-match-anything 'empty' template, which can be useful for garbage collection, for example, may be exploited by any agent to remove any tuple. Regarding the distribution of capabilities, unlike our model that allows (subject to the transitivity rights) the capabilities distribution by any agent holding a copy of the capability, the key distribution in the SecOS's scheme is the responsibility of the tuple's originator, which, though more controlled, can become an onerous task, especially in a large open system.

Another system that uses a capability-based security policy with keys is Lana [2]. The messages (i.e. tuples) on their (TS-based) associative 'message boards' are locked with keys, not capabilities—capabilities are used for remote method invocations. Locked with a key, the reference to (the capability for) the object being called is placed on the message board.

An interesting example of more recent work is VLOS [5], a distributed operating system based on tuple-spaces. Capabilities are required to create new TSs, new field types, as well as type signatures, i.e. unique groupings of field types which are unique to a particular TS, to make up a tuple type. A capability for an object in VLOS consists of a unique id and the type of the object (i.e. whether it is a TS, a field type, a type signature, or other objects), the name of the TS (of which the object is a member), a set of rights, and a cryptographic hash function to minimise forgery. Like ours, a capability in VLOS acts as a 'handle' to the object it is associated with. Our multicapabilities, however, are more flexible in the sense that they are not associated with a TS—they can be used with any TS as long as the user possesses the (uni)capability for the target TS. Furthermore, the combination calculus of (multi) capabilities gives an added advantage for our model to maintain control while being flexible.

The work closest to ours is μKLAIM [10]. Based on the KLAIM language [7], μKLAIM uses its type system to enforce access control. Unlike our capability model that uses dynamic checking, μKLAIM relies on both static and dynamic checking. A μKLAIM capability is a pair that represents the operation allowed on a pattern of the matching (target) tuple. Our multicapability, on the other hand, is a triple that provides extra features of partitioning via tags, to limit and control agents' access, as well as the combination calculus to further enrich the model.

In the systems mentioned above, capabilities are mainly discussed as an access control mechanism for security purposes, some with the assistance of cryptography, as in [15, 1, 2]. μKLAIM for instance, emphasises on security policies: an agent may have 'knowledge' of a location name even though it does not have a capability to it. We are concentrating on the functional properties of capabilities as 'visibility' filters—agents can only know about objects for which they hold a capability, and the capability makes visible a sub-set of the operations available for that object's type. This scheme enables a refined control over agents' actions

(not limited to access control only) on objects in the system, and facilitates certain aspects of coordination, such as resource management [16].

4 Conclusions and Future Work

Capabilities represent the information on 'who knows about what operation on a certain object'. The more the kernel knows of the system's behaviour, the better, more optimised coordination can be achieved, thus increasing the system's efficiency. The extra information, supplied by the capabilities given to the agents, can provide the facility to create a finer level of control in distributed systems.

As capabilities can only be applied to uniquely identifiable objects, such as TSs, we have proposed the new concept of multicapabilities which extends capabilities to apply to a group of un-named objects to accommodate tuples. A multicapability consists of a unique tag to differentiate between different capabilities for the same template (and which in addition aids in its unforgeability), a reference to a group of tuples, and a set of rights to control the actions permitted to be performed on an object in the said group. The set of rights need not be limited to input and output operations, but may include any sensible, even user-defined, operations that are appropriate to the system.

Some capability combination operations have been introduced—namely the set-like union, intersection, and negation operations—to provide further mechanisms towards achieving a finely controlled system. We are still investigating the sensible 'combining' operations, and currently developing proper relevant semantics.

It is known that one of the disadvantages of capabilities (and multicapabilities) is that they are difficult to revoke. A solution to this problem is to incorporate indirect (multi)capability objects: the (multi)capability held by an agent do not directly 'point' to an object, but instead refers to the indirection object, which in turn points to the object. Deleting an indirection object enables a (multi)capability to be permanently revoked. As an option to deletion, we are studying the idea of selective temporary revocation using indirection objects in which the corresponding sub-interfaces can be 'turned off and on' to provide a finer control in the system. When an indirection object is turned off, any access (including `out`) to the object will block until it is switched back on.

Indirection objects can become a filter for a group of multicapability and its derivations: a derived multicapability with identical or restricted rights will point to the same indirection object as its super-multicapability; whereas creating (requesting for) a new multicapability (even of the same template) will automatically create a different indirection object. Deleting the former will revoke the said multicapability along with its sub-multicapabilities.

On a final note, we view a capability as not merely access control, but in more general terms as *visibility* control. Visibility can represent security. Although this paper does not address the issue of security, it is indeed a crucial problem when dealing with agents with intelligence and autonomy, particularly those involved in some sort of confidential and sensitive business transactions or other critical applications.

Acknowledgements. The authors are grateful to the anonymous reviewers for the helpful comments on the previous version of this paper. Nur Izura Udzir is supported by the Public Services Department of Malaysia. Presentation of this paper is partly funded by Microsoft Research Ltd, UK.

References

1. C. Bryce and M. Oriol and J. Vitek: A Coordination Model for Agents Based on Secure Spaces. In Proc. 3rd International Conference on Coordination Models and Languages (Coordination'99), LNCS 1594. Springer-Verlag, Berlin Hiedelberg (1999) 4–20
2. C. Bryce, C. Razafimahefa, M. Pawlak: Lana: An Approach to Programming Autonomous Systems. In ECOOP 2002, LNCS 2374. Springer-Verlag, Berlin Hiedelberg (2002) 281–308
3. N. Carriero, D. Gelernter: How to Write a Parallel Program: A Guide to the Perplexed. ACM Computing Surveys, 21(3):323–357 (1989)
4. N. Carriero, D. Gelernter: Coordination Languages and Their Significance. Communication of the ACM, 35(2):97–107 (1992)
5. V-L. Chung, C. S. McDonald: The Development of a Distributed Capability System for VLOS. Australian Computer Science Communications, 24(3):57–64 (2002)
6. J. B. Dennis, E. C. van Horn: Programming Semantics for Multiprogrammed Computations. Communication of the ACM, 9(3):143–154 (1966)
7. R. de Nicola, G. L. Ferrari, R. Pugliese: KLAIM: A Kernel Language for Agents Interaction and Mobility. IEEE Trans. on Software Engineering, 24(5):315–330 (1998)
8. E. Freeman, S. Hupfer, K. Arnold: JavaSpaces: Principles, Patterns, and Practice. The Jini Technology Series. Addison-Wesley (1999)
9. D. Gelernter: Generative Communication in Linda. ACM Transactions on Programming Languages and Systems, 7(1):80–112 (1985)
10. D. Gorla, R. Pugliese: Enforcing Security Policies via Types. In Proc. 1st Int. Conf. on Security in Pervasive Computing (SPC'03), LNCS 2802. Springer-Verlag, Berlin Hiedelberg (2003) 88–103
11. J. L. Jacob, A. Wood: A Principled Semantics for `inp`. In Coordination Models and Languages, LNCS 1906. Springer-Verlag, Berlin Hiedelberg (2000) 51–66
12. H. M. Levy: Capability-Based Computer Systems. Digital Press (1984)
13. N. H. Minsky, J. Leichter: Law Governed Linda as a Coordination Model. In Object-Based Models and Languages for Concurrent Systems, LNCS 924. Springer-Verlag, Berlin Hiedelberg (1995) 125–146
14. G. Nutt: Operating System: A Modern Perspective (2nd). Addison-Wesley (2002)
15. J. Pinakis: Providing Directed Communication in Linda. In Proc. 15th Australian Computer Science Conference (1995) 731–743
16. N. I. Udzir, A. Wood: Multicapabilities for Distributed Resource Management in Open Systems. In Proc. IASTED Int. Conference on Parallel and Distributed Computing and Systems (PDCS 2004). ACTA Press (2004)
17. A. Wood: Coordination with Attributes. In Coordination Languages and Models, LNCS 1594. Springer-Verlag, Berlin Hiedelberg (1999) 21–36

Delegation Modeling with Paradigm

Luuk Groenewegen[1], Niels van Kampenhout[1], and Erik de Vink[1,2]

[1] LIACS, Leiden University
[2] Dept of Math. and Comp. Sc., Technische Universiteit Eindhoven
luuk@liacs.nl
nielsvankampenhout@wanadoo.nl
evink@win.tue.nl

Abstract. Within one model, behavioural consistency of its constituents is often problematic. Within UML such horizontal behavioural consistency between the objects of a concrete model, is particularly needed in the context of dynamic patterns. Here, we investigate delegation, which is fundamental to patterns that separate the locality of receiving a request, and one or more localities actually handling it. We specify delegation by means of the coordination language Paradigm. In particular, we present some variants of delegation in the context of a broker pattern and clarify how the Paradigm notions are the basis for understanding a solution as well as for adapting it to deal with other dynamic features.

1 Introduction

Software architectures are the major instrument to handle the size and complexity of today's software systems. Moreover, within the context of a business architecture, they pinpoint the software system's embedding in the non-digital world. Typically, an architecture consists of a number of components related via specific links. Components express certain aspects that contribute to the functionality of the system or the organization as a whole. Interaction among components is directed via their interfaces. To stress this even more, components are usually considered stateless. In the architectural description one abstracts away from the internal dynamics of a component in order not to clutter up the overall view. See, e.g., [19, 8]. Nevertheless, some dynamics survive in architectural descriptions, e.g. via protocols and protocol roles and other global dynamics, as these are relevant for dynamic consistency between components.

The problem of dynamic consistency between components constituting an architecture is, as yet, far from being solved. Even within the UML [3, 9], where the underlying, detailed dynamics of objects constituting a model contribute additional information to base dynamic consistency on, the problem of dynamic consistency is comparably far from being solved. Clarification of this problem situation is the more pressing, as increasingly often patterns are being used (both as means of design [10] and for business processes [5]) for consistently organizing and reorganizing the dynamics of the model's constituents.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 94–108, 2005.

Often, when modeling a software system with the UML, the use-cases act as 'glue logic' for the information carried by the respective description methods. Similarly, sequence diagrams restricting the dynamics to interactive steps only, concentrate on 'gluing'. This is not only a matter of style: the concrete behaviour, as captured by the use-case or the sequence diagrams, has its consequences for the interaction of the components involved. More frequently rule than exception, the relevant information goes beyond the interface. Some of the internal dynamics of the component must be revealed in order to assess the correctness of the cooperation of a component and its software or non-software environment.

In order to judge the global behaviour of the system, the local behaviour of components is pivotal. For example, the interaction of a component should not be in conflict with its internal dynamics. This is consistency between different levels of description, more or less similar to Küster's vertical consistency (cf. [7, 16]). Also, the component should act in compatibility with the components from its surroundings. This is consistency between different model constituents, on the same level of description, suitably chosen to reflect the relevant collaboration; here, Küster's notion of horizontal consistency is more appropriate. Typically, such questions of consistency arise when components play multiple roles in multiple protocols that overlap in time. See, e.g., [17, 18].

The modeling technique that we propose for aligning global and local behaviour is Paradigm [6, 20, 12]. In Paradigm the coordination among a manager and its employees is the prime concern. It does so by relating the local behaviour of the manager to the global behaviour of the employees, the latter being decorated with just that little information that is necessary to maintain consistency of the system. In this way, via a manager, Paradigm addresses horizontal consistency between the manager's employees. Furthermore, via its special notions of subprocess and trap, Paradigm guarantees vertical consistency within an employee between its detailed and global behaviour. In the present paper we report on an application of Paradigm in business process modeling for a non-hierarchical organization. The example deals with delegation. Delegation, i.e. separating the components for starting behaviour and the component(s) continuing it, is behind many patterns [10, 5], thus requiring horizontal consistency between them.

Below, Section 2 introduces, informally, the key ingredients of Paradigm. A first description of the delegation example is covered in Section 3. An alternative model is presented in Section 4. Some other variants are discussed in Section 5. Finally, Section 6 wraps up with some concluding remarks.

2 Paradigm

Paradigm is a coordination specification language, concentrating on expressing behaviour and behaviour influencing. In this section we present Paradigm briefly and informally. Operational semantics of Paradigm have been presented in [12] and [11]. It is stressed that in the present paper Paradigm models do not require that coordination is organized in a strictly hierarchical manner.

Paradigm uses the notion of a process, together with a state-transition-diagram-like visualization for it. Usually, a process expresses a constituent's behaviour on the detailed level, corresponding to its inner, hidden behaviour. See, e.g., Figure 1 for an example visualization as a directed graph: nodes are states and directed edges are transitions between two states.

For the behavioural description of a constituent on a more global level, Paradigm uses two additional notions: subprocess and trap. Whereas a process specifies all possible behaviours of a constituent —in Figure 1 called $\texttt{Client}(i)$— a subprocess (of a process) expresses a phase of that behaviour: a (temporary) restriction of that behaviour, relevant in the context of some collaboration between constituents. A trap of a subprocess, being a subset of the subprocess states, reflects a final, irrevocable stage of the subprocess: within a subprocess, a trap of it cannot be left once entered. So, a trap can serve as a kind of commit or acknowledge within the collaboration, e.g. declaring the subprocess behaviour has proceeded far enough to be changed from the (current) behaviour restriction into a suitable next one.

A partition then divides the full process behaviour into a set of subprocesses with their traps. Figure 4 gives a visualization of a partition of $\texttt{Client}(i)$ into 3 subprocesses. The relevant traps are drawn as polygons surrounding the states a trap consists of. We formalize these notions in the next definition.

Definition 2.1

(a) A process or STD S is a pair $\langle \texttt{ST}, \texttt{TS} \rangle$. Here $\texttt{ST}$ is called the set of states, or also the state space; $\texttt{TS} \subseteq \texttt{ST} \times \texttt{ST}$ is the set of transitions. We write $x \to x'$ in case $(x, x') \in \texttt{TS}$.

(b) A subprocess of S is a process $\langle \texttt{st}, \texttt{ts} \rangle$ such that $\texttt{st} \subseteq \texttt{ST}$ and $\texttt{ts} \subseteq \{ (x, x') \in \texttt{TS} \mid x, x' \in \texttt{st} \}$. A trap t of a subprocess $s = \langle \texttt{st}, \texttt{ts} \rangle$ is a nonempty set of states $t \subseteq \texttt{st}$ such that $x \in t$ and $x \to x' \in \texttt{ts}$ imply that $x' \in t$. If $t = \texttt{st}$, the trap is called trivial.

(c) Let $s = \langle \texttt{st}, \texttt{ts} \rangle$ and $s' = \langle \texttt{st}', \texttt{ts}' \rangle$ be two subprocesses of the same process. A trap t of s is called a connecting trap from s to s' if the states belonging to the trap t are states in s' as well, i.e., $t \subseteq \texttt{st}'$.

(d) A partition $\{ (s_i, t_i) \mid i \in I \}$ of a process $S = \langle \texttt{ST}, \texttt{TS} \rangle$ is a set of subprocesses $s_i = \langle \texttt{st}_i, \texttt{ts}_i \rangle$ with traps t_i such that $\texttt{ST} = \bigcup_{i \in I} \texttt{st}_i$ and $\texttt{TS} = \bigcup_{i \in I} \texttt{ts}_i$.

Although not explicitly defined, a global behaviour for a constituent, see, e.g., Figure 5, can be formulated in terms of a sequence of subprocesses glued together by means of a connecting trap. All phases occurring in such a sequence come from the same partition; we therefore say about such a global behaviour, it occurs on the level of that partition. Note that for a connecting trap all states in it belong to both subprocesses involved. For this paper we restrict ourselves to a single trap of any subprocess connecting it to a next subprocess.

The formal structure on which these semantics are defined (cf. [11]) are tuples of configurations, one per process. A configuration looks as follows:

$$[\, s_i, \langle S_{ij} \rangle_{j=1}^{m(i)} \,]_{i=1}^{n}$$

It consists of the local state s_i of the process P_i and a sequence of $m(i)$ subprocesses S_{ij}, one for each partition π_{ij} of the process. The local state belongs to the detailed behaviour of the process whereas a subprocess belongs to the global behaviour of the process on the level of one of its partitions. Thus, for each process and its partitions the configuration gives the current state and the current subprocesses. Transitions in the various coordinates are governed by so-called consistency rules. The general format of a consistency rule is

$$\begin{aligned} &\texttt{ProcP} : \texttt{state\_a} \to \texttt{state\_b} \ * \\ &\quad \texttt{ProcQ}_1[\texttt{PART}_1] : \texttt{SubProc}_1 \to \texttt{SubProc}'_1, \\ &\quad \ldots \\ &\quad \texttt{ProcQ}_n[\texttt{PART}_n] : \texttt{SubProc}_n \to \texttt{SubProc}'_n \end{aligned} \tag{2.1}$$

Here $\texttt{state\_a} \to \texttt{state\_b}$ is a ProcP transition, $\texttt{PART}_i$ is a partition of process $\texttt{ProcQ}_i$ and $\texttt{SubProc}_i \to \texttt{SubProc}'_i$ is a transition in the global behaviour or transfer on the level of partition $\texttt{PART}_i$, requiring the various connecting traps have been entered. Via a consistency rule, a combined transition occurs consisting of a state transition and zero or more subprocess changes. In the presence of the consistency rule (2.1) the process ProcP is called manager of the processes $\texttt{ProcQ}_1, \ldots, \texttt{ProcQ}_n$. The latter processes are called employees of ProcP. So, an employee has at least one partition and, therefore, global behaviour.

If a process has one or more partitions, the semantics guarantee, a state change in the process only happens if that transition belongs to each current subprocess of the process. In other words, for an employee process the detailed transitions are consistent in all partitions with the current subprocesses for that process. The global transitions correspond to a detailed state transition in some manager process. Such a global transition can only happen if the traps of the relevant subprocesses have been reached. Informally, a manager prescribes new subprocesses to some of its employees by making a suitable state transition; similarly, an employee, by entering a suitable trap, allows a manager to prescribe a new subprocess to it. In other words, a global transition is consistent with the connecting trap that has actually been entered.

In the present setting based on the operational model of [11], in contrast to the operational semantics given in [12], we allow an employee to have more than one manager, even with respect to the same partition. This forms the basis for delegation. Even more extremely, an employee can be its own manager. This is self-management, which can be very useful in combination with delegation.

3 Delegation I

In this section, we consider a delegation example where n clients are served by m servers. For simplicity, all clients behave the same; similarly, all servers behave the same. A broker selects a client in round-robin order and assigns a server to it when necessary. This server is subsequently responsible for handling the needs of the client.

A client can state its interest in a service by 'approaching the desk'. When the needs of the client are clear –possibly after some interaction with the broker, not modeled here– a server is selected by the broker to handle the client's request. After this delegation, the broker continues its activities. The server takes care of the clients it has been assigned to in a round-robin fashion. Once the client is being served, it releases the service by getting satisfied. The server does not inform the broker that it has become available for serving client i, but the broker will conclude so, if needed, when it sees this client at its desk again.

A formal description of the above in Paradigm involves three process types: client, broker, server. A client process is given by the state-transition diagram of Figure 1. It consists of a cycle of 5 states, viz. `no_needs`, `at_desk`, `need_clear`, `service` and `satisfied`, that are subsequently visited. The state `no_needs` is considered to be the starting state of the process. We distinguish n client processes named `Client`$(1), \ldots,$ `Client`(n). For presentational reasons we assume in the pictures below the number n to be equal to 5.

Each client process has a partition named `STATUS` that has the three subprocesses `WithoutService`, `Orienting` and `UnderService` given in Figure 4. The three subprocesses together describe the global or coarse-grained behaviour of the client process as pictured in Figure 5. It simply cycles through its three subprocesses.

The trap `asking` of the subprocess `WithoutService` comprises the local states `at_desk` and `need_clear`. If a client process has entered this trap, i.e. has control in one of the two local states mentioned, it signals that it is ready for moving to a next phase. The traps of the subprocesses `Orienting` and `UnderService` are likewise. When residing in state `need_clear` or in either of the two states `satisfied` and `no_needs`, respectively, the corresponding phase has reached its final stage and the client process is ready to be transferred (in its single `STATUS` partition).

The state-transition diagram of the broker process is given in Figure 2. The broker process checks all the client processes and mediates service on their desire. The broker process has no partition.

The state-transition diagram of the m server processes have a similar shape as the broker process. We assume that a server will check in a round-robin fashion whether a client has been assigned to it, see Figure 3. A server process has n partitions called `CLIENT`(i), one per client. Each `CLIENT`(i) partition has two subprocesses, `Assigned` and `NotAssigned`., see Figure 6. Thus, the current subprocesses of a server process together indicate, out of 2^n possibilities, the server's status: for each client whether it is to be served or not. See Figure 7 for the global behaviour of a server process in one of its n partitions.

Next, we have to describe the coordination of the n client processes, the broker process and the m server processes. This is done via the so-called consistency rules in Table 1. For the concrete case here, we explain the mechanism of a consistency rule as described abstractly in the previous section. E.g., the consistency rule $(B2)$ of the broker process

Table 1. Consistency rules I

$(B1)$	$\texttt{Broker} : \texttt{check}(i) \rightarrow \texttt{mediate}(i) *$ $\texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \rightarrow \texttt{Orienting}$
$(B2)$	$\texttt{Broker} : \texttt{mediate}(i) \rightarrow \texttt{check}(i+1) *$ $\texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{Orienting},$ $\texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{NotAssigned} \rightarrow \texttt{Assigned}$
$(B3)$	$\texttt{Broker} : \texttt{check}(i) \rightarrow \texttt{check}(i+1) *$ $\texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \not\rightarrow$
$(C1)$	$\texttt{Client}(i) : \texttt{no\_needs} \rightarrow \texttt{at\_desk}$
$(C2)$	$\texttt{Client}(i) : \texttt{at\_desk} \rightarrow \texttt{need\_clear}$
$(C3)$	$\texttt{Client}(i) : \texttt{need\_clear} \rightarrow \texttt{service}$
$(C4)$	$\texttt{Client}(i) : \texttt{service} \rightarrow \texttt{satisfied}$
$(C5)$	$\texttt{Client}(i) : \texttt{satisfied} \rightarrow \texttt{no\_needs}$
$(S1)$	$\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{serve}(i) *$ $\texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{UnderService},$ $\texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{Assigned} \rightarrow \texttt{NotAssigned}$
$(S2)$	$\texttt{Server}(j) : \texttt{serve}(i) \rightarrow \texttt{check}(i+1) *$ $\texttt{Client}(i)[\texttt{STATUS}] : \texttt{UnderService} \rightarrow \texttt{WithoutService}$
$(S3)$	$\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{check}(i+1)$

$$\begin{aligned}&\texttt{Broker} : \texttt{mediate}(i) \rightarrow \texttt{check}(i+1) *\\ &\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{Orienting},\\ &\quad \texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{NotAssigned} \rightarrow \texttt{Assigned}\end{aligned}$$

expresses a transfer dependent on three conditions: (i) the broker resides in its local state $\texttt{mediate}(i)$; (ii) the i-th client process, in its single partition STATUS, has reached the trap of subprocess Orienting; (iii) the j-th server processes has reached in its partition $\texttt{CLIENT}(i)$ the trap of subprocess NotAssigned. The effect of the transfer is also threefold: (i) the broker process will move to its local state $\texttt{check}(i+1)$; (ii) the i-th client processes will continue adhering to the subprocess Orienting in its partition STATUS (though in fact it cannot do anything); (iii) the j-th server will adhere to the subprocess Assigned in its partition $\texttt{CLIENT}(i)$.

From the point of view of designing the coordination for the client-broker-server system the consistency rules of Table 1 can be interpreted as follows: The rule $(B1)$ allows for a local transition of the broker process in the local state $\texttt{check}(i)$ provided that the i-th client process has reached the trap asking of its subprocess WithoutService. So, the current state of client i is either at_desk or need_clear. The broker will move to the local state $\texttt{mediate}(i)$ to see what are the needs of the client; the client changes, on the level of the partition STATUS, from subprocess WithoutService to subprocess Orienting.

The rule $(B2)$ illustrates the coordination of three processes. If the broker is mediating service for the i-th client, i.e. it resides in the local state $\texttt{mediate}(i)$, and the needs of this client have become clear, i.e. client i has reached the trap serverClear of the Orienting subprocess that consists of the local state

`need_clear`, and the j-th server has not been assigned to this client, i.e. it is prescribed the subprocess `NotAssigned` in the partition for the i-th client, then the ($B2$) rule can fire. The choice of the particular server is non-deterministic. The broker moves to the local state `client`$(i+1)$ as it considers its involvement with the i-th client to be finished for the moment. This has been delegated to the j-th server. The ith client is left in the subprocess `Orienting` waiting for service. The j-th server is notified to take care, at the appropriate time, of client i as it now follows the subprocess `Assigned` for this client.

The consistency rule ($B3$) is an instance of the negative rule format. It expresses that the broker can make a local transition from state `check`(i) to `check`$(i+1)$ in case the i-th client does not reside in the trap `asking` of the subprocess `WithoutService`. Note that the non-determinacy of moving either to state `mediate`(i) or to state `check`$(i+1)$ for the broker process in state `check`(i) is resolved by the i-th client (and, strictly speaking, also involving the server processes). We claim, an equivalent Paradigm model without negative rules can be constructed as well, an issue not treated here.

The consistency rules of the client processes are rather simple. As the clients have not been assigned coordination tasks, their local transitions are unconditional, but for the overall requirement that the transitions belong to the current subprocess of the partition `STATUS`.

The consistency rule ($S1$) of the server is similar to the rule ($B1$) of the broker process. The rule covers the case where the server j has been delegated coordination of client i by the broker. Here, we also see a case of self-management: the server process will transfer itself from its subprocess `Assigned` to the subprocess `NotAssigned`. This way, the server will be available for the broker for assignment to client i again, when this client returns to the desk asking for brokerage of another service. By the delegation, the broker is leviated from keeping track of the precise stage of the clients and of the availability of the servers. Based on rule ($S2$), server j will only move from state `serve`(i) to state `check`$(i+1)$ when client i has reached the trap `ready` of its subprocess `UnderService`. The server then transfers the client to the subprocess `WithoutService`. The local transition of the j-th server from state `client`(i) to state `client`$(i+1)$ has no side-conditions in rule ($S3$). However, the transition is only possible if, on the level of partition `CLIENT`(i), the server's current subprocess is `NotAssigned`. Thus, the broker resolves the non-determinacy of server j in state `client`(i). If the server is assigned, it will only have the transition to its state `serve`(i) based

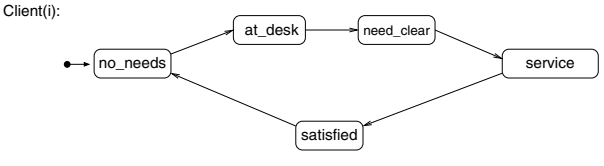


Fig. 1. Client STD

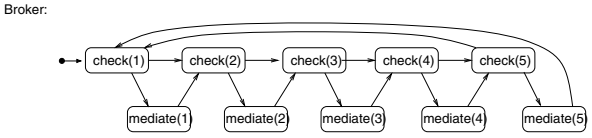


Fig. 2. Broker STD

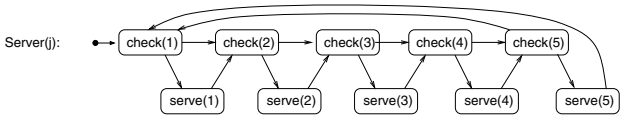


Fig. 3. Server STD

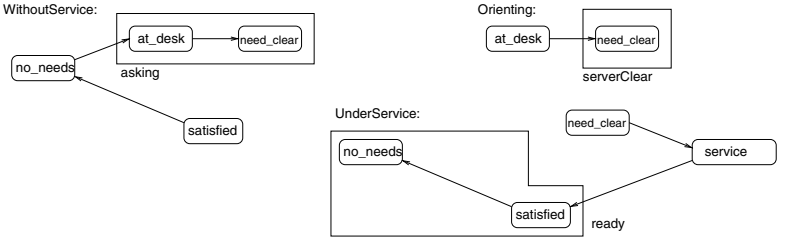


Fig. 4. Subprocesses of the Client process for partition `STATUS`

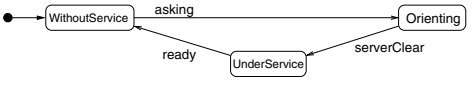


Fig. 5. Global behaviour of the Client process on the level of partition `STATUS`

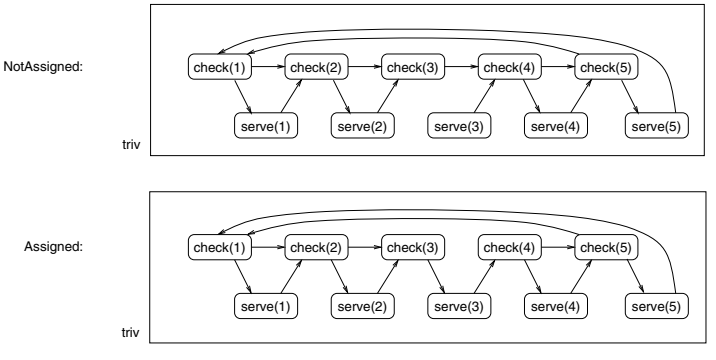


Fig. 6. Subprocesses of a Server process for partition `CLIENT`(3)

on rule ($S2$) as an option; if the server is not assigned, it can only move to state `client`$(i+1)$ by rule ($S3$).

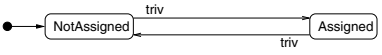


Fig. 7. Global behaviour of a Server process on the level of partition `CLIENT`(i)

4 Delegation II

As in the previous section, we have three process types: client, broker, server. Only the broker is slightly different, see Figure 8, as it has additional loops in its states `check`(i). Furthermore, we consider the same configuration of n client processes, 1 broker process and m server processes. See Figure 1 and Figure 3 for the other two process types.

The small difference of the broker has to do with the different details of the delegation. In the previous section, the broker delegated the actual service of a client to a server, without being informed explicitly about the precise beginning of such service. In the current section we let the broker be informed about such a beginning to serve `Client`(i) by `Server`(j). This enables the broker to withdraw the assignment of `Server`(j) to `Client`(i). So now it is the broker who changes subprocess `Assigned` into `NotAssigned`, instead of `Server`(j) doing it. So the (partial) delegation of coordinating `Server`(j)'s global behaviour on the level of partition `CLIENT`(i) does no longer exist: the broker does the complete coordination of this and similar global behaviours. This has the following consequences for the Paradigm model. Partition `STATUS` and the global behavior for it remain unchanged, see Figure 4 and 5. The servers remain unchanged, see Figure 3, but their partitions `CLIENT`$(1), \ldots,$ `CLIENT`(n) are rather different, see Figure 9.

Their traps `idle` and `busy` are apparently nontrivial. Trap `idle`, being very large, expresses that the server can do anything but starting to serve `client`(i). So, a new assignment of this very client can happen when needed. Trap `busy` is a small one, expressing that service can be started and completely given, but it cannot be terminated, so the client is not really released - although it can continue as far as state `no_needs`. The slightly adapted global behaviour is given in Figure 10. The coordination of the various detailed and global behaviours is described by the consistency rules in Table 2 (rules for Broker and Server processes only).

The differences between the rule set from Table 2 compared to those from Table 1 exactly reflect, on the basis of the new Paradigm model, the new coordination details. The delegation by the broker towards the individual servers of controlling a part of their global behaviour on the level of their partition `CLIENT`(i), is no longer there. Moreover, the delegation by the broker towards the individual servers of controlling a part of the global behaviours of the various clients on the level of partition `STATUS` is changed such that in the new situation any server explicitly informs the broker when it starts or finishes such a delegated task. The consistency rules changed to this aim, are as follows. Rule ($B4$) is added to guarantee the transition from partition `Assigned` to `NotAssigned`, which is no longer the responsibility of a server. Note that only after such a global transition, the corresponding server can release the particular client it is

Table 2. Consistency rules II

$(B1)$	$\texttt{Broker} : \texttt{check}(i) \rightarrow \texttt{mediate}(i) *$ $\texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \rightarrow \texttt{Orienting}$
$(B2)$	$\texttt{Broker} : \texttt{mediate}(i) \rightarrow \texttt{check}(i+1) *$ $\texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{Orienting},$ $\texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{NotAssigned} \rightarrow \texttt{Assigned}$
$(B3)$	$\texttt{Broker} : \texttt{check}(i) \rightarrow \texttt{check}(i+1) *$ $\texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \not\rightarrow$
$(B4)$	$\texttt{Broker} : \texttt{check}(i) \rightarrow \texttt{check}(i) *$ $\texttt{Server}(j)[\texttt{CLIENT}(k)] : \texttt{Assigned} \rightarrow \texttt{NotAssigned}$
$(S1)$	$\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{serve}(i) *$ $\texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{UnderService}$
$(S2)$	$\texttt{Server}(j) : \texttt{serve}(i) \rightarrow \texttt{check}(i+1) *$ $\texttt{Client}(i)[\texttt{STATUS}] : \texttt{UnderService} \rightarrow \texttt{WithoutService}$
$(S3)$	$\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{check}(i+1)$

serving. Rule $(S1)$ has been simplified, as the global transition from a subprocess `Assigned` to `NotAssigned` is taken care of by the broker. The explicit informing by a server to the broker when it starts or finishes its delegated task, occurs with the (detailed) transition in rules $(S1)$ and $(S2)$. Thus, a server enters its trap `busy` by rule $(S1)$ or its trap `idle` by rule $(S2)$. It is on the basis of a server having entered such a trap, the broker applies rule $(B2)$ or $(B4)$. The other rules do not change.

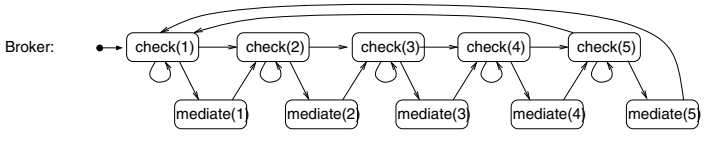


Fig. 8. Broker STD II

5 Variations

In this section we illustrate some more flexibility of Paradigm. We discuss three variations on the delegation example of Sections 3 and 4. We describe how one can add other processes in a clean way. First, by addition of a tool that is coordinated by the servers as manager; second, by extension of the configuration with a maintainer that coordinates the servers as its employees. As a third variation, we consider a refinement of the broker in its assignment of servers based on a parameter mechanism.

5.1 Adding an Employee Process

We consider the case where the servers share some resources that are needed for the servicing of clients. We add two tools, `Tool`(1) and `Tool`(2): the one

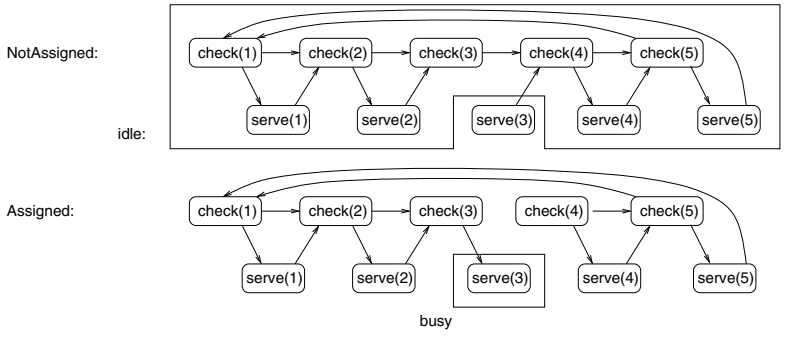


Fig. 9. Subprocesses of a Server for partition `CLIENT`(3)

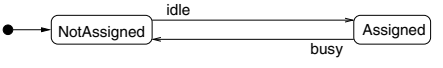


Fig. 10. Global behaviour of a Server on the level of partition `CLIENT`(i)

shared amongst the odd-numbered servers, the other shared amongst the even-numbered servers. Each tool will have one partition named `AVAILABILITY` representing its availability, being either `Released` or `Taken`. The state-transition diagram and subprocesses are pictured in Figure 11.

The tool alternates between its two states. Servers of the same parity are all managing the corresponding tool in the same partition. When a tool has been released, as signaled by reaching the trap `toBeTaken` of subprocess `Released`, the server can take the tool. The tool is then transferred to its subprocess `Taken`. When the tool has reached its state `occupied`, i.e. the trap `toBeReleased` of subprocess `Taken`, the server can use the tool at its leisure. The server then releases the tool by transferring it to the subprocess `Released`, so that it can move to its local state `free`, where it can be taken again. The transfer of tool subprocesses thus maps 1-1 on the transitions `check`(i) $\rightarrow$ `serve`(i) and `serve`(i) $\rightarrow$ `check`($i+1$).

In order to mix the coordination of the tools by the server and the existing client-broker-server dynamics, we add the signaling of traps and transfer of subprocesses of the tools to the consistency rules of Table 1 of the servers. The rules for the broker and client remain the same. The new rules for the two tool processes are simple as the tool processes have no manager role. See Table 3.

5.2 Adding a Manager Process

In the previous subsection the management of a tool was done by a collection of servers. Therefore, the consistency rules of the servers were adapted to cope with the new situation. Next, we show how to extend the system by the addition of a

Table 3. Consistency rules for the Server and Tool processes

$(S1)$	$\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{serve}(i) *$
	$\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{UnderService},$
	$\quad \texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{Assigned} \rightarrow \texttt{NotAssigned},$
	$\quad \texttt{Tool}(j \bmod 2)[\texttt{AVAILABILITY}] : \texttt{Released} \rightarrow \texttt{Taken}$
$(S2)$	$\texttt{Server}(j) : \texttt{serve}(i) \rightarrow \texttt{check}(i+1) *$
	$\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{UnderService} \rightarrow \texttt{WithoutService},$
	$\quad \texttt{Tool}(j \bmod 2)[\texttt{AVAILABILITY}] : \texttt{Taken} \rightarrow \texttt{Released}$
$(S3)$	$\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{check}(i+1)$
$(T1)$	$\texttt{Tool}(k) : \texttt{free} \rightarrow \texttt{occupied}$
$(T2)$	$\texttt{Tool}(k) : \texttt{occupied} \rightarrow \texttt{free}$

process that manages some existing ones. We introduce a maintenance process that influences the dynamics of the servers. The state-transition diagram of the `Maintainer` process is given in Figure 12. (Again, for reasons of presentation, we choose in the figure the number of servers equal to 5 too.) The maintainer in its starting state `no_maint` selects non-deterministically one of the servers. If the selected server is servicing a client, it can finish this. Then the server is brought under maintenance; it resumes servicing as soon as the maintainer process returns to its initial position.

As, with respect to the design choices made here, the maintenance issues are orthogonal to the original dynamics, we simply add a new partition for the servers. This is partition `MAINTENANCE` with subprocesses as in Figure 13: subprocess `OutOfService` with trap `stalled` only allows to finish the current service (a graceful interrupt) and the subprocess `Running` with the trivial trap allows all behaviour.

The consistency rules for the `Maintainer` process are not surprising, see Table 4. Note, as the trap used is trivial, rule $(M1)$ is not biased to any of the servers. Any server process can be interrupted for maintenance, based on the maintainer's decision only.

Table 4. Consistency rules for the Maintainer process

$(M1)$	$\texttt{Maintainer} : \texttt{no\_maint} \rightarrow \texttt{maint}(j) *$
	$\quad \texttt{Server}(j)[\texttt{MAINTENANCE}] : \texttt{Running} \rightarrow \texttt{OutOfService}$
$(M2)$	$\texttt{Maintainer} : \texttt{maint}(j) \rightarrow \texttt{no\_maint} *$
	$\quad \texttt{Server}(j)[\texttt{MAINTENANCE}] : \texttt{OutOfService} \rightarrow \texttt{Running}$

5.3 Parameter-Based Refinement

Our last variation shows how load balancing or history-based allocation can be handled in Paradigm. A process can be decorated with a parameter representing the local variables or data of the process. The parameter mechanism

Table 5. Consistency rules for the Broker process with parameter

$$
\begin{array}{ll}
(B1) & \texttt{Broker}\{H\} : \texttt{check}(i) \to \texttt{mediate}(i) * \\
 & \quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \to \texttt{Orienting} \\
(B2a) & \texttt{Broker}\{H\} : \texttt{mediate}(i) \to \texttt{check}(i+1) * \qquad \text{if } (i,j) \in H \\
 & \quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \to \texttt{Orienting}, \\
 & \quad \texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{NotAssigned} \to \texttt{Assigned} \\
(B2b) & \texttt{Broker}\{H\} : \texttt{mediate}(i) \to \texttt{check}(i+1) * \qquad \text{if } \nexists k\colon (i,k) \in H \\
 & \quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \to \texttt{Orienting}, \\
 & \quad \texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{NotAssigned} \to \texttt{Assigned} \\
 & \quad \texttt{Broker}\{H\} \Longrightarrow \texttt{Broker}\{H + (i,j)\} \\
(B3) & \texttt{Broker}\{H\} : \texttt{check}(i) \to \texttt{check}(i+1) * \\
 & \quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \not\to
\end{array}
$$

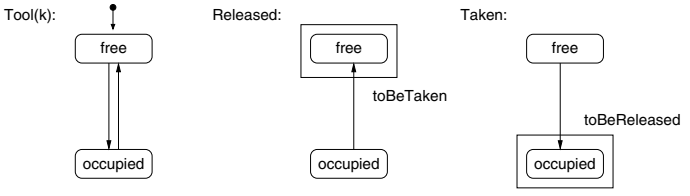


Fig. 11. Tool STD and subprocesses of the Tool process

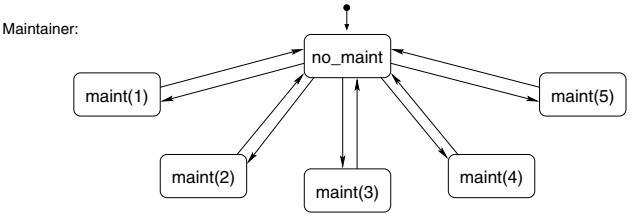


Fig. 12. Maintainer STD

is reminiscent to process languages as CCS or CSP. For a process with some parameter, $\texttt{Proc}(X)$ say, occurring at the left-hand side of a consistency rule, a so-called change clause is added to the right-hand side of a consistency rule of the format $\texttt{Proc}(X) \Longrightarrow \texttt{Proc}(X')$. The idea is that the rule can only be fired if the data of Proc has value X. As an immediate consequence of firing the rule, the data X of Proc will be changed into the data X' on behalf of the relevant manager.

Consider, e.g., in the setting of Section 3, the case where the broker gives a client the same server as before. If the client has not been brokered yet, the broker simply selects one non-deterministically. We introduce the variable H (for history) containing a pair (i, j) if client i was served by server j before. The consistency rules are then augmented with the parameters and change clauses. See Table 5. Now there are two consistency rules in place corresponding to the local transition $\texttt{mediate}(i) \to \texttt{check}(i+1)$ of the broker: If there exists a

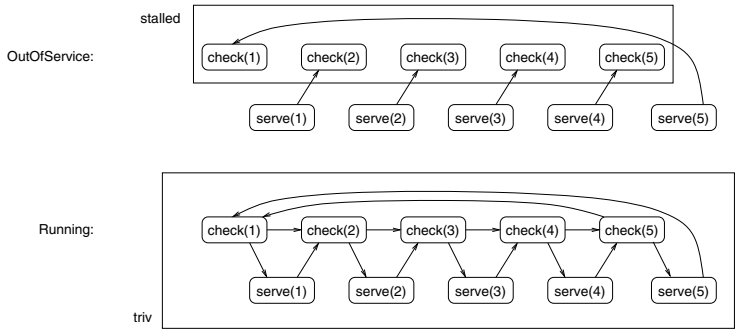


Fig. 13. Subprocesses of the Server process in `MAINTENANCE` partition

pair (i, j) in H than the server j is allocated for client i by rule $(B2a)$; if no such pair (i, k) exists in H, any of the servers can be appointed to deal with client i by rule $(B2b)$. The other consistency rules remain the same.

6 Concluding Remarks

In this paper we showed how delegation can be modeled with Paradigm. For two basic cases and variations we indicated what the Paradigm model looks like and how the consistency rules capture the coordination of the processes involved. The main point is that local or detailed behaviour of a process that is manager of part of the system, is consistent with the global behaviour of its employee processes, thus assuring horizontal consistency in that part of the system. Manager role and employee role can change dynamically. Paradigm does not only allow for multiple employees of one manager, but also for multiple managers of one employee, thus allowing delegation and even self-management. The advantage of being able to relate local and global behaviour is that of abstraction. Modeling or reasoning about the behaviour of one process does not require to have knowledge in full detail of the other processes that are involved. Here it is vertical consistency between the local behaviour and the global behaviour that matters, as illustrated above for delegation.

In the master's thesis of Van Kampenhout [15], related to work of [1], some initial work has been performed on verification of Paradigm models. In a case-study concerning an insurance company typical properties such as allocation and fairness have been checked. This was done using SMV. It is plausible, that the software architecture arising from a Paradigm model by 'cutting along partitions' is amendable to architecture slicing as proposed in [4] in the context of the Charmy framework. It would be interesting to see how Paradigm and Spin can be exploited, e.g., for the case study reported in [13], where also the issue of coordination and UML is addressed. More generally, with the increased expressiveness and flexibility of Paradigm, the pattern trail is a promising line of research. Currently, in joint work with Andries Stam, we are adapting Paradigm models for the ToolBus machinery [2, 14] for prototyping purposes.

References

1. J.C. Augusto and R.S. Gómez. A temporal logic view of Paradigm models. In *Proc. SEKE 2002, Ischia*, pages 497–503. ACM, 2002.
2. J.A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Proc. Coordination '96*, pages 75–88. LNCS 1061, 1996.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide*. Addison Wesley, 1999.
4. M. Caporuscio, P. Inverardi, and P. Pelliccione. Formal analysis of architectural patterns. In *Proc. ESWA*, pages 10–24. LNCS 3047, 2004.
5. J.O. Coplien and N.B. Harrison. *Organizational Patterns of Agile Software Development*. Prentice Hall, 2004.
6. G. Engels and L.P.J. Groenewegen. *Software Process Modelling and Technology*, chapter SOCCA: Specifications of Coordinated and Cooperative Activities, pages 71–102. Research Studies Press, 1994.
7. G. Engels, R. Heckel, and J.M. Küster. The consistency workbench: A tool for consistency management in UML-based development. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003*, pages 356–359. LNCS 2863, 2003.
8. P. Clements et al. *Documenting Software Architectures: Views and Beyond*. SEI Series in Software Engineering. Pearson Education, 2002.
9. M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd edition)*. Addison Wesley, 2003.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
11. L.P.J. Groenewegen, N. van Kampenhout, and E.P. de Vink. Coordination in networked organizations: the Paradigm approach. Technical Report CSR 03/13, Technische Universiteit Eindhoven, 2003.
12. L.P.J. Groenewegen and E.P. de Vink. Operational semantics for coordination in paradigm. In F. Arbab and C. Talcott, editors, *Proceedings Coordination 2002*, pages 191–206. LNCS 2315, 2002.
13. P. Inverardi and H. Muccini. A coordination process based on UML and a software architectural description. In H.R. Arabnia, editor, *Proc. PDPTA*, 2000. 7pp.
14. H. de Jong and P. Klint. Toolbus: The next generation. In F.S. de Boer et al., editor, *FMCO 2002, Revised Lectures*, pages 220–241. LNCS 2852, 2003.
15. N. van Kampenhout. Systematic specification and verification of coordination: towards patterns for Paradigm models. Master's thesis, Leiden University, 2003.
16. J.M. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, 2004.
17. B. Nuseibeh, S.M. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *IEEE Computer*, 33:24–29, 2000.
18. B. Nuseibeh, J. Kramer, and A. Finkelstein. Viewpoints: meaningful relationships are difficult! In *Proc. ICSE 2003, Portland, Oregon*, pages 676–683. IEEE, 2003.
19. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
20. P.J. Toussaint. *Integration of information systems: a study in requirements engineering*. PhD thesis, Leiden University, 1998.

Dynamically Adapting Tuple Replication for Managing Availability in a Shared Data Space

Giovanni Russello[1], Michel Chaudron[1], and Maarten van Steen[2]

[1] Eindhoven University of Technology
[2] Vrije Universiteit Amsterdam

Abstract. With its decoupling of processes in space and time, the shared data space model has proven to be a well-suited solution for developing distributed component-based systems. However, as in many distributed applications, functional and extra-functional aspects are still interwoven in components. In this paper, we address how shared data spaces can support separation of concerns. In particular, we present a solution that allows developers to merely specify performance and availability requirements for data tuples, while the underlying middleware evaluates various distribution and replication policies in order to select the one that meets these requirements best. Moreover, in our approach, the middleware continuously monitors the behavior of application and system components, and switches to different policies if this would lead to better results. We describe our approach, along with the design of a prototype implementation and its quantitative evaluation.

1 Introduction

The shared data space model has proven to be a useful abstraction for the development of distributed applications. Notably its support for decoupling processes in space and time makes it attractive for distributed systems that require dynamic configuration of applications by the insertion and removal of components at runtime. This dynamic configuration is possible when components encapsulate functionality that has been coded independent of any runtime environment. When extra-functional requirements have been addressed (such as those for performance), widespread component deployment becomes more difficult. In essence, we are facing the problem of separating various concerns when developing and deploying components in distributed systems.

One solution to address this separation is exploiting the underlying middleware. In particular, we believe that the middleware should provide the mechanisms for specifying and enforcing extra-functional concerns. For example, if replication is required, the middleware should ideally offer mechanisms that would allow the application developer to select from different replication policies that can be subsequently enforced at runtime. If necessary, new policies can be developed and deployed as well, independent of the basic functionality implemented by legacy components.

Somewhat surprisingly, research on shared data spaces has been largely ignoring the support for this separation of concerns. A plethora of solutions have been proposed to

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 109–124, 2005.

distribute data items, without giving the application developer a choice on *how*, *where*, and *when* data should be distributed or replicated. To solve this problem, we have proposed an extension of the shared data space model with a mechanism for separating the distribution (and replication) of data items from their strict functional usage by application components. Moreover, by monitoring the behavior of application components, we have been able to dynamically adapt data distribution to the needs of an application. We have thus effectively created a closed feedback-control system, now often popularly coined as a self-managing or autonomic system.

So far, we have considered adaptation for performance, focusing on metrics such as application-perceived latency and consumed network bandwidth. For this paper, we concentrate on data availability. Assuming that components may unpredictably fail, particular care has to be taken for shared data items to remain available to other components. Similar issues arise when an application is deployed on mobile nodes. In such an environment, a node's connectivity may be highly unpredictable and a set of data items may unexpectedly disappear when a node disconnects.

A well-know solution to this problem is data replication. By replicating data on several nodes, the system can statistically guarantee that a data item is available even if the node where the item was inserted is no longer connected (or has failed). However, replicating for availability may conflict with replicating for performance. For example, high performance requirements may dictate that only weak data consistency can be supported, whereas high availability requires updating all replicas simultaneously.

Such tradeoffs generally require application-specific solutions. However, instead of imposing a single solution, we propose a framework that offers to the application developer a suite of replication policies. Each policy incurs costs with respect to performance, availability, consistency, etc. In our approach, a developer is offered a simple means to weigh these different costs such that the system can automatically choose the policy that meets the various (and often conflicting) objectives best. Moreover, through continuos monitoring of the environment the system can dynamically and automatically switch to another policy if it turns out that this would reduce overall costs.

We make the following contributions. First, we provide a simple mechanism that allows for separating concerns regarding performance and availability in shared data space systems. Second, we demonstrate how possibly conflicting objectives can be dealt with in these systems, such that the selection of a best policy can be done dynamically and in a fully automated fashion. Third, we show that the input needed from an application developer to support these optimal adaptations can be kept to a minimum, allowing the developer to concentrate on the design and implementation of functionality.

This paper is organized as follows. In Section 2 we present our proof-of-concept called GSpace, and mechanisms that drive GSpace decisions. To prove the soundness of our framework we conducted some experiments, of which the outcomes are discussed in Section 3. Section 4 focuses on related work. We conclude in Section 5 and give directions for future research.

2 GSpace

In this section, we first provide some background information on the shared data space model. Thereafter, we concentrate on our implementation of a shared data space, called *GSpace*. We describe the internal modules that compose GSpace.

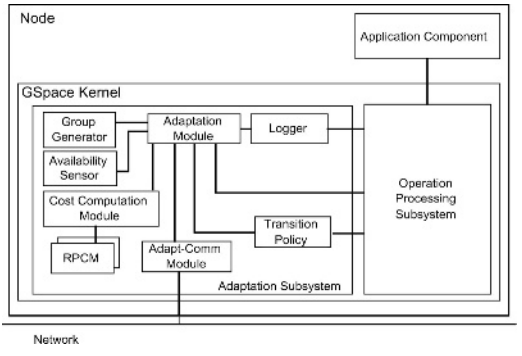


Fig. 1. Internal structure of a GSpace kernel deployed on a node

2.1 Architectural Design

The data space concept was introduced in the coordination language Linda [3]. In Linda, applications communicate by inserting and retrieving data through a data space. The unit of data in the data space is called *tuple*. Tuples are retrieved from the data space by means of *templates*, using an associative method. Multiple instances of the same tuple item can co-exist An application interacts with the data space using three simple operations: put, read and take.

GSpace is an implementation of a distributed shared data space. A typical setup of GSpace consists of several *GSpace kernels* instantiated on several networked nodes. Each kernel provides facilities for storing tuples locally, and for discovering and communicating with other kernels. GSpace kernels collaborate with each other to provide to the application components a unified view of the shared data space. Thus the physical distribution of the shared data space across several nodes is transparent to the application components, preserving its simple coordination model. In GSpace tuples are typed. This allows the system to associate different replication policies with different tuple types.

Figure 1 shows a GSpace kernel deployed on a networked node. A GSpace kernel consists of two subsystems: the *Operation Processing Subsystem (OPS)* and the *Adaptation Subsystem (AS)*.

The OPS provides the core functionality necessary for a node to participate in a distributed GSpace: handling application component operations; providing mechanisms for communication with kernels on other nodes; and monitoring connectivity of other GSpace nodes that join and leave the system; and maintaining the information about other kernels. Finally, the OPS provides the infrastructure to differentiate distribution strategies per tuple type. The internal structure of the OPS is described in [10].

The adaptation subsystem is an optional addition to GSpace that provides the functionality needed for dynamic adaptation of policies. The AS communicates with the co-deployed OPS for obtaining information about the status and actual usage of the system. In particular the *Logger* is responsible for logging all the space operations executed on the local kernel. When the number of operations for a particular type reaches a threshold, the logger notifies its local *Adaptation Module (AM)*. The AM is the core of each AS. The AM coordinates the different phases of the *adaptation mechanism*. The code of the AMs on all nodes is identical. However, for each tuple type in the system one AM operates as a *master* and all the others as *slaves*. The master AM takes decisions concerned which replication policy should be applied to a tuple type. The slaves AM follow the master's decisions. The *Cost Computation Module (CCM)* and *Replication Policy Cost Models (RPCM)* are responsible for computing the costs incurred by the replication policies for a given set of operation logs. The *Transition Policy* prescribes how to handle legacy tuples in order for them to be placed at locations where the new replication policy expects to find them. The *Adapt-Comm Module (ACM)* provides communication channels between the ASes on different nodes in the system.

The new modules that we added for dealing with availability are the following:

Availability Sensor: This module is responsible for measuring the availability of the node in which it is deployed. This is done by periodically writing timestamps in a file. When a failure occurs, this time-stamp is used to compute the duration that the node was not available.

Group Generator: Generating groups of nodes is the task of this module. Once the availability values for all the nodes have been collected the master AM passes this information to its local Group Generator. The Group Generator will aggregate nodes following some given strategy. For instance, in the experiments that we discuss in section 3 the Group Generator selects the best 3 nodes in term of availability. The generated group is then passed to the replication policies.

In the following section we describe in more detail how the different modules in the AS contribute to the mechanism that allows GSpace to select the replication policy that best suits the application behavior.

2.2 Autonomic Behavior in GSpace

This section describes the mechanism that allows GSpace to dynamically evaluate and select the replication policy that fits best the needs of the application.

In a distributed system such as GSpace, tuples are often stored and accessed remotely. Since nodes may fail or get disconnected, part of the shared data space could not be reachable. A common solution to this problem is the use of replication. By replicating tuples across several nodes we increase the probability of accessing a tuple even if some nodes are down. However, replication requires consumption of extra resources, such as extra memory for storing tuple replicas and bandwidth for exchanging information needed for keeping the replicas in a consistent state. Also, keeping replicas consistent comes at the price of global synchronization when updates occur.

Instead of proposing a one-size-fits-all solution, our approach sets flexibility as its primary goal. We included in GSpace a suite of replication policies each with its own

tradeoff between provided availability, resource consumption, and performance. In this paper, we ignore performance issues, allowing the application developer to specify only the availability requirements for the tuple types used by the application. The problem is now shifted to finding the replication policy that (a) minimizes resource consumption while (b) fulfilling the availability requirements. These conditions are generally in conflict with each other. As we will show, our simple mechanism is able to deal with such conflicting situations in a fully automated fashion.

As the environment's conditions change over time, a static assignment of replication policy to tuple type could eventually fail to provide the required performance of the system. As a solution to this issue, we monitor the environment. Application patterns are detected by logging each data space operation. Moreover, to guarantee that availability requirements are fulfilled, sensors are placed in each node to measure node availablity in real-time. By combining these data, our mechanism can automatically detect when to switch to another replication policy if it turns out that availability is at risk, or when resource consumption can be improved.

We identify three phases in our mechanism, that we explain in turn.

- monitoring phase
- evaluation phase
- adaptation phase

Monitoring Phase. During the first phase GSpace collects statistical data regarding its environment. This data consists of information about the availability of nodes and the usage profile of application components.

For collecting information on node availability, the GSpace kernel is instrumented with a sensor that monitors the availability of the node where it is running. Before diving into implementation details, we introduce the basic math behind the measurements that our system performs. The formula for calculating the availability of a single node is:

$$Availability = \frac{\textit{Mean Time To Failure}}{\textit{Mean Time To Failure + Mean Time To Recover}} \quad (1)$$

It is important to understand what exactly *Mean Time To Failure* (MTTF) and *Mean Time To Recover* (MTTR) mean. With MTTF we indicate the average time that the node is continuously operating, i.e. the average time between the end of one failure and the beginning of the next. With MTTR we address the average time necessary for the node to recover from an experienced failure.

Figure 2 sketches the time line of a node that experiences some failures. When a failure i occurs we indicate with sf_i and ef_i respectively the time when the failure i starts and

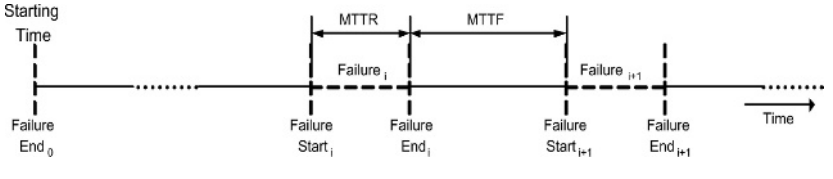


Fig. 2. The time line of a node that experiences some failures

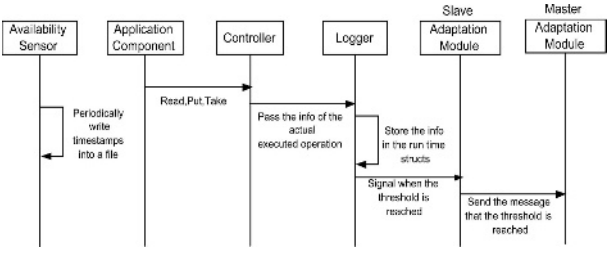


Fig. 3. The MSC of operation logging

ends. We assume that the starting time of the node (the very first time that the node is activated) is equivalent to ef_0 (end of failure 0). Figure 2 also provides a graphical representation of MTTF and MTTR to understand how to compute those values. For instance, the availability value after the n-th failure is obtained by the following formula:

$$Availability = \frac{\sum_{i=1}^{n}(sf_i - ef_{i-1})}{\sum_{i=1}^{n}(sf_i - ef_{i-1}) + \sum_{i=1}^{n}(ef_i - sf_i)} \quad (2)$$

For computing (2) we need to collect the starting and ending times of a failure. When the system is started for the first time, the sensor writes into a file the starting time of the system. Periodically, the sensor is actived and writes timestamps into the same file. Actually, a timestamp is just the time at which the sensor is active. After a node experiences a failure, at re-booting time the sensor detects that the system was down (since the timestamp file is stored persistently). The starting time of a failure then is considered as the time at which the last timestamp was written whereas the time at which the system is up again is considered as the end-of-failure time. GSpace simply calculates the down time as the difference between the new starting time and the time of the last executed timestamp.

For collecting information about the application behavior, we employ the same method as described in our previous work [12]. Each data-space operation that application components execute is logged and stored per tuple type. Figure 3 shows the message sequence chart during the operation logging. The data that is logged contains:

- Operation type: the space operation executed (either a read, take or put)
- Tuple type: the type of the tuple or template passed as argument with the operation
- Location: the address of the GSpace kernel (i.e., node) where the operation is executed
- Tuple ID: a unique id provided to each tuple that enters the shared data space
- Tuple size: the size of the tuple inserted through a put operation or returned by a read or take operation
- Template size: the size of the template passed as argument of a read or a take operation
- Timestamp: the time when the operation is executed

When the number of executed operations on a node reaches a given threshold the system starts the next phase.

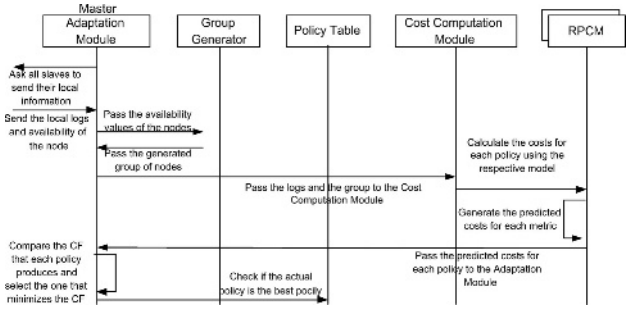


Fig. 4. The MSC of the evaluation phase

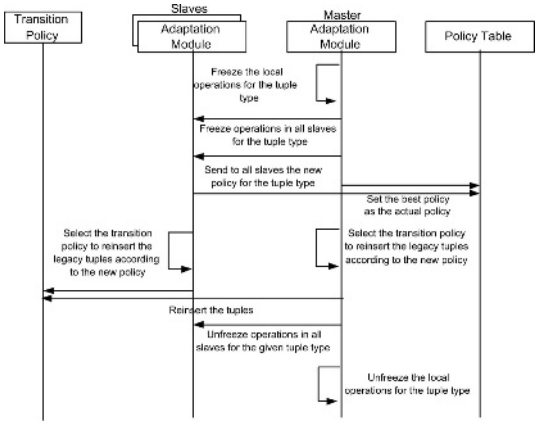


Fig. 5. The MSC of the policy adaptation phase

Evaluation Phase. The evaluation phase consists of collecting data from all nodes and comparing the cost of different replication policies.

Figure 4 shows the message sequence chart of the evaluation phase. The master AM requests all slave AMs to send their local data (logs and node availability). This data is combined and the costs for each policy are calculated by means of simulation.

For capturing the performance of the different distribution policies we use a *cost function*. Our cost function is a linear combination of various parameters. The values of these parameters are combined in an abstract value that quantifies the tradeoff between performance versus resource usage for a given replication policy. The parameters are defined in such a way that a lower value indicates lower costs (and thus better behavior). The replication policy that leads to the lowest costs is the best policy for the application.

In this work, we apply the same method as described in [12] but with the focus on data availability. Therefore, we use a different cost function: bu represents the bandwidth usage; mu represents the accumulative memory usage; and da represents the *derived availability*. The latter is calculated as follows:

$$da(p) = \begin{cases} 100 - availability(p) & \text{if } availability(p) \geq required\_availability, \\ MaxValue & \text{if } availability(p) < required\_availability \end{cases}$$

In this way, if the availability provided by a replication policy p does not satisfy the user's requirements then the value for da is set to $MaxValue$ so that the calculated costs will become very high and the system will automatically reject this policy. The cost function is defined as follows:

$$CF_p \ = \ w_1 * bu(p) + w_2 * mu(p) + w_3 * da(p) \tag{3}$$

The weights w_i tune the relative contribution of each parameter to the overall cost.

Once the costs are calculated for each replication policy, they are passed to the AM that selects the best replication policy. The AM checks whether the current policy is still the best one. If this is the case, no further actions are undertaken. Otherwise, the AM starts the phase described next.

Adaptation Phase. In this phase the system switches replication policy and adapts the data space content. In Figure 5 the actions executed during this phase are presented in a message sequence chart. The master AM freezes application operations for the given tuple type in all nodes. Afterwards, each kernel updates its own data structure and redistributes the tuples still in the space according to the new replication policy. When this transition period ends the master AM resumes the operations in all nodes.

3 Implementation and Experiments

This section describes the experiments that we performed using a simulator of GSpace. The experiments model a distributed system with 10 nodes connected via a LAN.

Our previous experiments focused on distributed systems in which application components dynamically join and leave a system during execution (but in which the nodes were always available). In [11] we showed that there is no single distribution policy that is best for this dynamic type of application behavior. Furthermore, in [12] we showed that dynamically adapting the distribution policy outperforms any static policy.

In this paper we do not only consider changes in the application behavior, but also in the underlying hardware infrastructure. In particular, we consider that the availability characteristic of nodes in the network may change. This occurs, for instance, in ad-hoc networks where devices join and leave a network.

We show the impact of changing infrastructure on sustaining a level of availability: without adaptation, no single static policy is able to sustain a given level of availability. Moreover, we show that the dynamic adaptation of the policy provides a better level of availability in the case of changing infrastructure. Furthermore, we show that the adaptation mechanism can handle situations where both the infrastructure as well as the application behavior change dynamically.

The goal of the experiments is to show that our system can adapt the policy it uses to changes in the availability characteristics of the nodes in the network. As a result, it can maintain a level of availability of tuples while the availability of nodes varies.

The results of this simulation are now being incorporated in an our distributed implementation of GSpace. Previous experience with the simulation [12] shows that the accuracy of the simulation is in the order of 5 percent. Hence the simulation provides fairly accurate predications about actual system behavior.

Next, we first describe the set-up of the experiments. Thereafter, we describe the used replication policies. We conclude discussing two interesting cases.

3.1 Set-Up of the Experiments

The experiments are based on the simulation of the deployment of GSpace in a network of 10 nodes connected via a LAN. We control the simulation experiment through the following parameters:

- *Application behavior*: the operations that the application components execute using GSpace. The simulation contains a library of different application usage patterns. A pattern consists of a series of read, put and take operations. A *run* of an experiment consists of the concatenation of a number of patterns. The patterns in a run may be of the same type, or they may be of different types. The approach we follow for the synthesis of application behavior is described in [12].
- *Node availability behavior*: the availability characteristics of the physical nodes where GSpace is deployed; including its change over time. The availability behavior of nodes during execution can be set to one of the following:
  - constant
  - increasing from a given value to a max value by increments of a given δ
  - decreasing from a given value to a min value by increments of a given δ
  - alternating between a min and max value by increments of a given δ

During the simulation, data about performance parameters is collected and passed to the Adaptation Manager. Using this data the Adaptation Manager evaluates the cost function, and determines which replication policy to use in the next phase.

3.2 Replication Policies

The set of replication policies for GSpace is extensible. For the experiments in this paper, we use the following set of replication policies:

- **Full Replication.** This policy puts a copy of every tuple on every node in the system (as soon as a tuple is inserted)
- **Fixed Replication.** This policy replicates tuples to a fixed number of nodes (as soon as the tuple is inserted). When awareness of node availability is enabled, the Group Generator provides the nodes where tuples should be replicated.
- **Dynamic Consumer Replication.** This policy replicates tuples to all nodes that host an application component that is a consumer of this type of tuple. In case the availability of the consumer group can not provide the required availability the policy includes in the group nodes provided by the Group Generator.
- **Dynamic Producer Replication.** This policy replicates tuples to all nodes that host an application component that is a produced of this type of tuple. Nodes provided by the Group Generator might be included in the group of producer nodes whenever this group can not sustain the required availability.

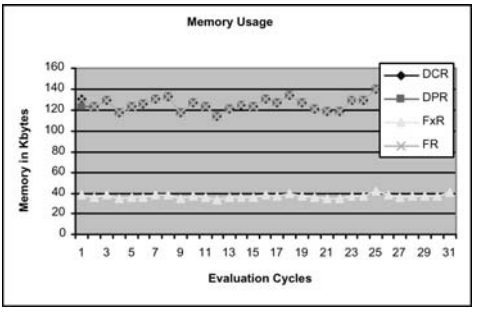


Fig. 6. Memory Usage measured for the different replication policies

For maintaining consistency among the nodes where replicas are stored, the replication policies collaborate using a Group Communication Protocol [5]. The nodes on which tuples are replicated are joined in a group where the operations are executed atomically. Moreover, the Group Communication Protocol takes care of consistency issues that could arise from the failure of some of the nodes in the group.

The availability of a given replication policy is determined by the availability of the group of nodes that is used for replicating tuples to. In particular, a group of nodes is considered available if at least one node of the group is available. Then, the group availability, GA, equals 1 minus the probability that all nodes within the group fail:

$$GA = 1 - P_{all\_nodes\_down} \tag{4}$$

We assume that failures of nodes are independent. Then the probability that all nodes fail is equal to the product of the probabilities of failure f_i of the individual nodes:

$$P_{all\_nodes\_down} = \prod_{i=1}^{n} f_i \tag{5}$$

3.3 Adding Awareness of Node Availability to Policies

In this section we introduce replication policies that base their decisions on the availability of nodes. The experiments in this section show that by constantly monitoring the underlaying infrastructure, the GSpace system improves sustainability of the required availability requirements despite the unpredictable behavior of the nodes.

In these experiments, we assume the application behavior is fixed. All the application components act both as consumers and producers.

For this application behavior, both Dynamic-Consumer and Dynamic-Producer policies replicate the tuples in all nodes. This means that the memory usage is the same as that for the Full Replication policy, as Figure 6 shows. Instead, the memory footprint of the Fixed Replication policy is smaller than that of the other policies since this policy replicates tuples on a smaller number of nodes.

First we consider the case when the availability monitoring is disabled. The required availability for the tuple type used in the experiments is 70%. The Fixed Replication

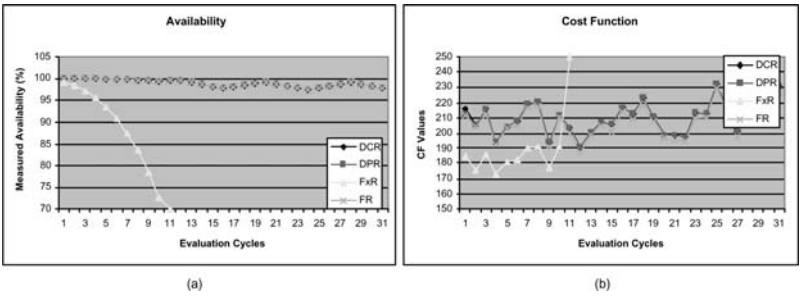


Fig. 7. Availability and Cost Function values for the replication policies when availability awareness is disabled

policy is defined to use the three nodes that provide the highest availability at the moment the system is started. However, the availability behavior of these nodes is programmed to decrease from 90 to 10 in steps of 5 (percent).

Using these three nodes the Fixed Replication policy initially satisfies the availability threshold. However, during execution, the nodes that are used by the Fixed Replication policy experience an increasing number of failures. Hence, the availability of the nodes decreases and as a result, the availability that the Fixed Replication policy provides decreases. Figure 7(a) clearly shows this decreasing behavior. The other replication policies provide a fairly stable availability with minor fluctuations. This becasue the changing availability of 3 nodes out of 10 impacts less the overall availability.

The previous graphs were concerned with availability. Next, we look at the effect of the replication policies on the cost function.

From Figure 8(b), we can conclude that as long as the availability requirements are met, Fixed Replication is the best policy since it uses the least memory. However, around the 10th evaluation cycle this policy can no longer sustain the required level of availability. As a result, the cost function value increases dramatically.

Next, we re-execute the same sequence of operations enabling the availability monitoring. The Fixed Replication policy still makes only a fixed number of copies, but now it selects the three nodes with the highest availability at the time of evaluation[1].

The memory usage graph is the same as the one shown in Figure 6 since the application behavior is the same. However, now the system is able to select nodes based on the measured availability of the nodes. At each evaluation, the system selects the three nodes that have highest availability. Now, Figure 8(a) shows that Fixed Replication is able to provide the required availability. Moreover, since the memory footprint is lower than that of the other policies, Fixed Replication is always the best policy. This is shown in the cost function graph on Figure 8(b).

3.4 Combining Dynamic Application Behavior and Dynamic Node Behavior

In this section we analyze when both the application components change their behavior and the availability of nodes changes during execution. The results will show that our

[1] These nodes are provided by the Group Generator module.

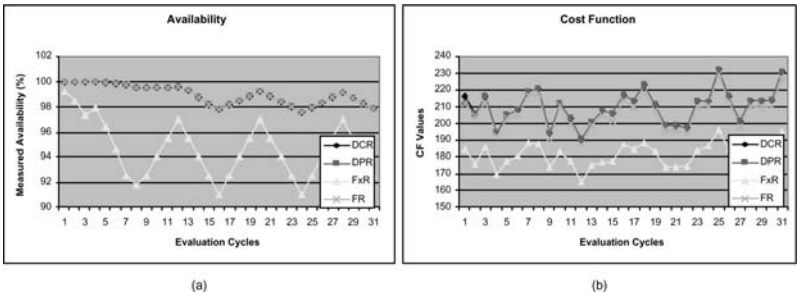


Fig. 8. Availability and Cost Function values for the replication policies when availability awareness is enabled

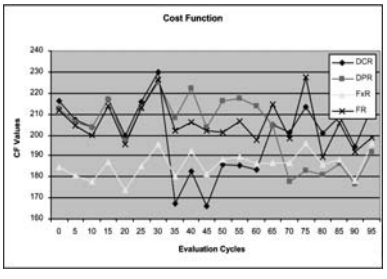


Fig. 9. Cost Function values when the application behavior changes

mechanism not only is able to select the replication policy that satisfies the availability requirements but also it selects the policy that best suits the components' behavior.

During these experiments the availability characteristics of nodes are measured from the system and made available to GSpace. The application component behavior is programmed to change during execution according to the following phases:

- Phase 1 (cycles 0–32): all application components are consumers and producers;
- Phase 2 (cycle 32–64): only the application components deployed on nodes n_9 and n_{10} act as consumers, all the other components act as producers;
- Phase 3 (cycle 64–95): only application components on nodes n_9 and n_{10} act as producers, the other components act as consumers.

Moreover, the availability of nodes n_9 and n_{10} is programmed to oscillate between 10% and 90%. Therefore, the group formed by these two nodes is not always able to sustain the required level of availability, which is fixed to 70%.

Let us begin analyzing the cost function values on Figure 9. During the first phase of the execution, the best policy that can guarantee the availability requirements with minimal memory usage is Fixed Replication.

During the second phase of execution, Dynamic Consumer Replication is the best policy. This is due to two factors. Firstly, only two nodes host application components that act as consumers. Therefore, Dynamic-Consumer Replication uses a group of nodes that is at most as large as the group used by Fixed Replication. This has a major im-

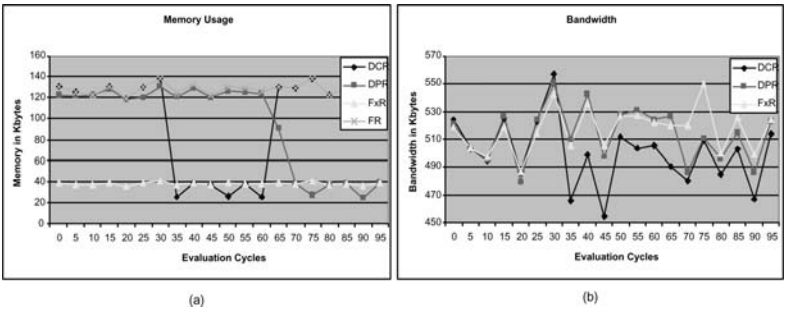


Fig. 10. Measured Memory and Bandwidth Usage when the application behavior changes

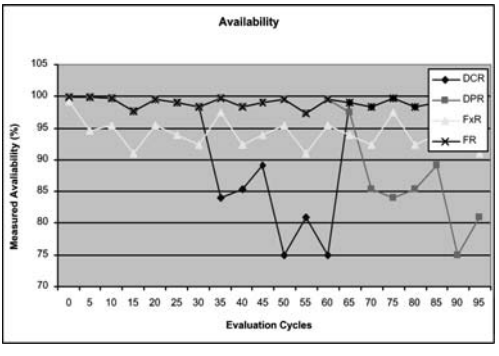


Fig. 11. Availability values when the application behavior changes

pact on the memory usage, as Figure 10(a) shows between evaluation cycles 32 and 64. In fact, when the combined availability of node n_9 and n_{10} is above the required availability, Dynamic Consumer Replication has a smaller memory usage footprint than Fixed Replication. However, sometimes those two nodes are not enough to guarantee the required availability. Thus, Dynamic Consumer Replication has to include other nodes to sustain the required availability. This is done by adding a node that is selected by the Group Generator module. The second factor is the reduced bandwidth usage that Dynamic Consumer Replication incurs. This is shown in Figure 10(b).

The last phase of execution witnesses another change. Application components switch behavior. In particular, after evaluation cycle 64, the application components on nodes n_9 and n_{10} start acting as producers. All the other components start to act as consumers. After a transition phase between cycles 64 and 70, where the components' behavior stabilizes, the Dynamic Producer Replication becomes the best policy, as Figure 9 shows. This is mainly due to the same factors that we discussed for Dynamic Consumer Replication. This is confirmed also by the graphs in Figure 10.

To conclude, we want to show that in all cases the availability sustained by the policies used in the different phases is always greater than the required value (Figure 11). This is an improvement over the behavior that is oblivious to changes in availability of nodes, yet the adaptation happens transparently to the application.

4 Related Work

This section describes other approaches for shared data space resilient to failures.

PLinda [4] is a variant of Linda that addresses fault-tolerant applications. In PLinda both data and processes are resilient to failures. In particular, by using a transaction mechanism extended with a process checkpoint scheme, PLinda ensures that a computation is carried out despite node failures. Compared to our approach, PLinda offers more functionality since it is resilient against process failures. On the other hand, in PLinda application developers have to explicitly declare which part of their application code should be executed in a fault-tolerant fashion. Threfoer, application code is interwoven with extra-funtional concerns not relevant to the application functionality.

Another fault tolerance implementation of Linda is FT-Linda [1]. As for PLinda, FT-Linda supports a transaction mechanism that allows the recovery of data and processes after a failure. However, FT-Linda requires the application developers to put extra effort in making their application resilient to failures. For instance, the application developer has to program the application to take care of removing intermediate results after a failure. Again, this is clearly against separating different concerns in the application design.

Although it was designed for taking advantage of idle time of workstations for running parallel applications, Piranha [6] could be used for addressing fault-tolerant applications as well. In Piranha, worker processes execute tasks on idle workstations. As soon as a workstation becomes busy, a worker process has to stop its current computation. The task has to be carried out by another Piranha worker on another idle workstation. Therefore, a retreat as the same effect as a failure. The Piranha model assumes that the execution of the task is carried out atomically despite the retreat. As for the FT-Linda, the Piranha system requires the application developer to program the application to clean-up intermediate results when a task has to retreat. Again, we see that application code is interwoven with fault-tolerant concerns.

An alternative approach to transaction mechanism for building shared tuple space resilient to fault tolerance is proposed in [9]. In this work, the author proposes exploiting code mobility as a mechanism for fault tolerance. By using code mobility, the system can guarantee an operational semantics in which either all operations are executed or none. The approach uses a run-time system that contains a checkpointing mechanism. In this way, the application developer does not need to interweave fault-tolerance code in her/his application since the run-time system will deal with this. To address the removal of legacy data left by mobile agent that is no longer alive, the author introduces the notion of *agent wills*. The agent will is a small piece of code embedded with the run-time system that describes what to do with data after the agent ceases activity. This will-code is executed by the run-time system whenever it detects that the respective agent crashed.

An evaluation of fault-tolerance methods for large scale distributed shared data spaces is described in [14].

Worthwhile to mention for the significance of their contributions, although not for fault tolerance, are the following implementations of shared data space. JavaSpaces [2] and TSpace [15] are commercial systems that have shown how the shared data space paradigm can be successfully used for building distributed applications. WCL [8] extends the basic primitives of the shared data space with some new ones. These new primitives

allow the execution of operations that are impossible to achieve by the standard ones. For instance, the multiple read primitive returns copies of all tuples that match with a given template. Finally, Lime [7] addresses the issues of coordination in a distributed environment.

5 Conclusions and Future Work

In this paper we made the following contributions. First, we provide a simple mechanism that allows for addressing availability concerns in shared data space systems separately from the functionality of applications. As a result, different policies can be employed for achieving different availability characteristics without affecting the functionality of the application.

Second, we demonstrate how possibly conflicting objectives (such as high availability and low resource use) can be dealt with in a fully automated fashion through the use of a cost-function.

Third, we show that the input needed from an application developer to support these optimal adaptations can be kept to a minimum, allowing the developer to concentrate on the design and implementation of functionality.

Finally, we showed the superior performance of dynamically adapting the replication policy that is used. The experiments showed that our mechanism is able to dynamically adapt the replication policy to the availability characteristics of the infrastructure. Moreover, the mechanism takes in consideration the application behavior and selects the policy that suits best the application needs.

This work is an extension of earlier work where we studies separation of extra-functional concerns in shared dataspaces. In [12] we showed how resource use could be treated as a separate policy and in [13] we studied the separation of real-time and exception handling concerns. The next challenge is combining multiple concerns in one architecture. Some of these concerns are inherently coupled, yet the challenge is to find a way of combining these concerns in a single architecture that enables ease of engineering and adaptability to changes in the usage profile.

References

1. D. E. Bakken and R. D. Schlichting. "Supporting Fault Tolerant Parallel Programming in Linda." *IEEE Trans. on Parallel and Distributed System*,1994.
2. E. Freeman, S. Hupfer, and K. Arnold. JavaSpaces principles, patterns, and practice. Addison-Wesley, Reading, MA, USA, 1999.
3. D. Gelernter. "Generative Communication in Linda." *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, 1985.
4. K. Jeong, D. Shasha. "PLinda 2.0: A Transactional/Checkpointing Approach to Fault Tolerant Linda." *Proc. 13th Symp. on Reliable Distributed Systems*, 96–105, Dana Point, CA, 1994.
5. M. F. Kaashoek and A. S. Tanenbaum. "Efficient reliable group communication for distributed systems." Internal Report IR-295 IR-295, Department of Computer Science, Vrije Universiteit of Amsterdam, 1992.
6. D. Kaminski. "Adaptive Parallelism in Piranha." PhD Thesis, Yale University, Department of Computer Science, 1994.

7. G. P. Picco, A. L. Murphy, and G.-C. Roman. "Lime: Linda Meets Mobility." In *Proc.* 21st International Conference on Software Engineering (ICSE'99), ACM Press, ISBN 1-58113-074-0, pp. 368-377, Los Angeles (USA), D. Garlan and J. Kramer, eds., May 1999.
8. A. Rowstron. "WCL: a Co-ordination Language for Geographically Distributed Agent." In *Worl Wide Web Journal*, Vol. 1, Issue 3, pp. 167–179, 1998.
9. A. Rowstron. "Using mobile code to provide fault tolerance in tuple space based coordination languages." In *Science of Computer Programming*, Vol. 46, Number 1-2, 137-162, Jan. 2003.
10. G. Russello, M. Chaudron, and M. van Steen. "Customizable Data Distribution for Shared Data Spaces." In *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2003)*, June 2003.
11. G. Russello, M. Chaudron, M. van Steen. "Exploiting Differentiated Tuple Distribution in Shared Data Spaces." *Proc. Int'l Conference on Parallel and Distributed Computing (Euro-Par)*, 3149:579–586, Springer-Verlag, Berlin, 2004.
12. G. Russello, M. Chaudron, M. van Steen. "Dynamic Adaptation of Data Distribution Policies in a Shared Data Space System." *Proc. Int'l Symp. On Distributed Objects and Applications (DOA)*, 3291:1225–1242, Springer-Verlag, Berlin, 2004.
13. R. Spoor. "Design and Implementation of a Real-Time Distributed Shared Data Space." Master's Thesis, Eindhoven University of Thechnology, Department of Computing Science, 2004.
14. R. Tolksdorf, A. Rowstron. "Evaluating Fault Tolerance Methods for Large-scale Linda-like systems." In *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, Vol. 2, pages 793-800, June 2000.
15. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. "T Spaces." *IBM System Journal*, 37(3):454-474, 1998.

Enforcing Distributed Information Flow Policies Architecturally: The SAID Approach

Arnab Ray

Department of Computer Science SUNY at Stony Brook, Stony Brook, NY 11794-4400, USA
arnabray@cs.sunysb.edu

Abstract. Architectural security of a distributed system is best considered at design time rather than further down the software life cycle where it may become very expensive to make even minor modifications to the software architecture. In this paper we take Architectural Interaction Diagrams (AID) [9, 8], an architecture description framework with an unique ability to encode communication efficiently and augment actions of AID components with security levels to produce SAID. This new architecture description language enables the designer to impose information flow restriction policies on system communications at design time which in turn allows a reduction of the information flow analysis problem for distributed systems to the simpler problem of information flow analysis of individual components of the distributed system.

1 Introduction

Model-driven architecture (MDA) [14] is an increasingly-popular paradigm of software development which looks upon *models* as first class entities in the development life-cycle.

There are two parts to any model-driven distributed system development process: specifying intra-model behavior (how a model does computation) and inter-model behavior (how different models communicate and coordinate). Architecture Description Languages (ADLs) try to make the development of the coordination infrastructure and the component models (ie users of the coordination infrastructure) as orthogonal to each other as possible in order to facilitate independent development and reuse.

Architectural Interaction Diagrams (AID) [9, 8] constitute an architecture description language with the following desirable properties:

- It supports abstract definitions of coordination that are separate from component models.
- It provides a parameterized notion of coordination (a coordination system generated for n component models does not need to be recoded for $n+1$ component models—unlike in CSP [3], CCS [12] based notations where it would need to be rewritten).
- It provides a coordination framework into which heterogenous models (models written in different modeling notation) can be plugged in and made to interoperate seamlessly.
- It allows for a variety of analysis routines (execution simulation, pre-order and equivalence checking, model-checking/counterexample generation) by virtue of the

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 125–139, 2005.

base formalism for AID being Labelled Transition Systems (LTS)s for which such algorithms have been extensively developed.

AID as described above are primarily used for certifying that the models satisfy properties related to the proper operation of the system (nothing bad can happen, something good will eventually happen etc). However the way coordination is encoded in AID makes it *easily extensible* for writing down *security policies for information-flow* between components plugged into the framework. *Distributed* information-flow policies are used to enforce security in a distributed system and adapting an unified framework for encoding coordination policies and information flow policies helps the designer deal with these two related issues at the same time. In this paper we propose an extension of AID called SAID (Secure Architectural Interaction Diagrams) which does exactly this.

An important aspect of distributed information flow security [4] is to ensure two kinds of security: security of computations of the components/processes (intra-component) and security in the flow of information between components (inter-component). SAID concerns itself with the latter problem. By providing the user the concept of *buses* (analogous to connectors in WRIGHT [7]) to encode interprocess communication, SAID allows for the specification of information flow policies on the distribution of information across components. The utility of this is twofold: firstly it confines the distributed information flow problem to a non-distributed context where there are several techniques [11] for dealing with it. Secondly it allows the designer to play with different information flow security policies (by encoding different buses) so as to consider different aspects of the functionality-security tradeoff before a design decision is committed to.

The contributions of the paper are as follows:

- Extending an existing coordination framework to encompass a wider domain of applicability: from purely *safety certification* to *safety-and-security* co-design.
- Providing *correct-by-construction* coordination rules that guarantee information-flow-safety during component composition.

While traditional approaches [4] first construct the entire system model (component models + coordination infrastructure) and then perform information flow analysis on it, SAID approaches the problem by providing *correct-by-construction* coordination rules that guarantee information-flow-safety during component composition (ie the composition operation does not introduce spurious information flow) and thus obviates the need for analyzing the entire composed system. In other words, it reduces the distributed information flow security problem to the more tractable *single component* information flow security problem where we can apply several well-studied methodologies for checking information flow of single components and then use the coordination framework assembly rules to guarantee global security properties.

1.1 Related Work

SAID extends the Architecture Interaction Diagrams paradigm [9, 8] by providing an enhanced methodology of writing down *security-aware* buses (Buses being the interprocess communication (IPC) entity used in AID). There are other architecture description languages s like WRIGHT [7], and coordination languages like Linda [5] that support some of the specification features used by SAID. However to our knowledge none of

the standard formal ADLs available have been used as a framework for distributed information flow analysis.

Security wrappers [13] are expressed as terms in a *boxed π calculus* and are used to compose untrusted components to form a secure system. Like the communication encodings in SAID, they too impose information flow policies on communication by filtering communications through the wrapper and in turn provide a correct-by-construction composition formalism. The aim of SAID is different from security wrappers: over here we are interested in extending a fully developed architecture description language so that it may support the expression of information-flow policies with the broad aim of having an unified method of dealing with coordination and information flow. Since SAID builds on AID, it inherits its beneficial features: state-space-efficient communication semantics and a more expressive communication vocabulary: as an example, in our formalism, a sophisticated event-coordination system used in an ubiquitious computing environment [10] can be efficiently encoded as a *bus* wheras it would be quite cumbersome to write a security wrapper that would encapsulate such a complicated communication discipline.

With respect to information flow analysis, our work is motivated by the distributed information flow analysis framework detailed by Mantel and Sabelfeld in [4]. In this work they motivate the need for distributed information flow which is defined to be the ability to check for spurious information flows not only when the process is performing its computation steps but also when messages are in transit via the communication infrastructure of the system. However our approach is different from the approach of Mantel and Sabelfield in the sense that while they analyze security properties globally, our approach is to localize the analysis to the components/processes by enforcing information flow safety across processes in a correct-by-construction fashion. The work in [4] proceeds by *requiring* a translation from their input formalism to an event-based system where an inherent limitation is introduced by the fact that the event-based system, by virtue of being a single-IPC-system, is merely able to simulate the richer modes of IPC that their input formalism uses (rather than support it natively). SAID on the other hand by virtue of its ability to provide different forms of IPC as a language-supported facility (rather than artificial simulation) provides a richer event-based mechanism for analysis with the result that systems may be directly defined and analyzed in SAID without the need of an explicit translation step.

The rest of the paper is arranged as follows. Section 2 provides a brief background to the theory of AID and distributed information flow which in turn lays the foundation for Section 3 which deals with SAID and examples that illustrate our approach. Section 4 contains discussions while Section 5 talks about future work and conclusions.

2 Background

2.1 Architectural Interaction Diagrams

Architectural Interaction Diagrams (AID) [8] is an Architecture Description Language for specifying systems, especially communication-intensive ones. Since SAID is derived from AID by augmenting transitions of its components, an understanding of the theory behind AID becomes imperative for understanding SAID.

The base formalism for AID is IOLTS (Input-Output Labeled Transition System), which are FSMs, consisting of states, transitions, a transition relation, a start state and a set of ports (the set being called an *interface*). A AID component has *output* transitions (writing data to a port), *input* transitions (reading data from a port) and another composite transition called *remote procedure-call* which consists of a single transition that denotes an output and an input action in sequence. This is analogous to a traditional remote procedure call in a programming language, with the output part signifying the supplying of actual parameters by the AID agent and the input part denoting the return value supplied back to the AID agent.

An AID component (agent) can take one of two forms: either it can be an IOLTS or it maybe a *network* containing other AID components embedded in interfaces and connected together in a communication topology as shown in Figure 1. The entities that actually perform the mechanism of communication and synchronization are called buses which like AIDs are also provided ports. The ports of a AID component and a bus are connected by *links*. It is also possible to export ports on an interface to an embedee interface through *gates*

The AID theory imposes no restrictions on how an IOLTS AID is described concretely: it could be a Statechart, or a term in process algebra, or a program. The only basic requirement is that the modeling formalism can be converted to IOLTSs ie for each input language there has to be translation to an IOLTS. We do not intend to provide a full semantic description of AID but the interested reader is requested to refer to [8] for the entire description including the Structural Operation Semantics (SOS) [6] rules that enable us to provide AID its uniqueness. What we do provide is the intuition behind buses ie the communication abstraction mechanism of AID.

In AID buses handle interactions between subsystems. As such, they have two responsibilities: the transfer of data between senders and receivers, and the synchronization of sender/receiver transitions, depending on the semantics of the interaction mechanism. For example, consider a synchronous binary handshaking interaction mechanism. Not only must a bus implementing this mechanism deliver a data value from a sender to a receiver, but it must also ensure that senders and receivers block until a communication partner is ready to execute. In the case of bounded-buffer non-lossy communication, on

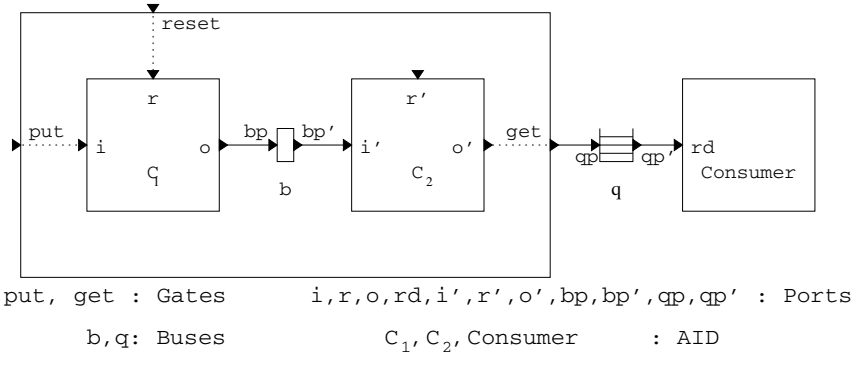


Fig. 1. A nested AID

the other hand, senders should be blocked when the buffer is full, while receivers should be blocked when the buffer is empty. In shared memory neither senders ("writers") nor receivers ("readers") ever block. Providing a common framework for explicating these subtleties is a central goal of the AID theory.

In particular, we wish to view buses as "devices" that combine transitions of subsystems connected to the bus into system-level transitions, according to the synchronization discipline the bus is intended to capture. This is where AID differ from conventional approaches . Normally the combining of subsystem transitions to form system-level transitions is done through the || (parallel) operator and the native handshaking discipline that is "hard-coded" into the semantics of the language. What we want however is to have a more general mechanism by which it would be possible for the user to define her own systems of communication and this newly created communication discipline can then be plugged seamlessly into the native semantics of the language. Buses are the means by which this goal is achieved.

Definition 1. *A* bus *is a tuple of form* $\langle I, B, T, b_0 \rangle$*, where* I *is an interface ie a pair of set of ports (the first set representing write and the second set the read ports),* B *is a set of bus states,* T *is a transition relation and and* $b_0 \in B$ *is the start state*

Intuitively, a bus contains a read and write interface, a set of states reflecting the internal status of the bus, a transition relation, and an initial state. Buses are similar to IOLTSs, but the transition relation is significantly different and requires more comment. A bus can be looked upon as an action transducer that takes as its input a bus state, a set of enabled write transitions (WV)represented by a set of tuples of the form (writeport,value), a set of enabled read transitions (R) represented by a set of read ports and provides as its outputs another bus state, a set of fired write transitions (W) represented by a set of write ports and a set of tuples (RV) of form (readport,value) that represents the set of fired read transitions.

A bus transition is of the form

$$b \xrightarrow[WV\ R]{W\ RV}_M b'$$

is intended to be read as: "if the bus is in state b, and subsystems connected to the bus enable write transitions as indicated in WV and read transitions as enabled in R, then the bus fires read transitions as indicated in RV and write transitions as indicated in W and goes to state b'." This firing of selected read and write transitions in systems connected to the bus is also done atomically: thus one bus transition may "consume" several transitions from the components connected to it. Also, "writing" to a bus is interpreted with respect to components connected to a bus: so write ports on a subsystem are connected to write ports on a bus, and similarly for read ports.

A bus transition may be thought of as consisting of an "enabling condition" and a "firing condition". The former requires that certain transitions be enabled on component ports that are connected to different bus ports. The latter then indicates which of the enabled transitions actually fire when the bus transition fires, thus causing state changes in the components as well as the bus.

In order to provide bus transitions, we have two obligations. The first is to define a transition predicate T_P involving free variables WV, R, RV and W with the property that

$$b \xrightarrow[WV\ R]{W\ RV}_M b'$$

holds exactly when $T_P(WV, R, W, RV)$ is true. The second is to show how the target of the transition ie b' is related to b

A Simple Example. The Calculus of Communicating Systems (CCS) [12] supports synchronous message passing as its only form of communication. This form of communication is common in other process algebras like CCS and CSP as well, and we show how it may be encoded as a bus. Buses in GCCS place no limit on how many subsystems are allowed to use them. They require all senders and receivers to block until at least one sender and receiver are enabled; then an exchange of data occurs, with the selected sender and receiver free to continue executing.

T contains all transitions for which T_P is true:

$$\exists \langle w, v \rangle \in WV.\ r \in R.W = \{w\} \wedge RV = \{\langle r, v \rangle\}.$$

Since the bus does not need to store the data and merely needs to pass it on, there is only a single state in the bus. Hence the target of the transition b' is always equal to b

A bus $M_S = \langle I, B, T, b_0 \rangle$ encapsulating synchronous binary handshaking may be defined as follows.

-I = a tuple consisting of two finite set of ports (read and write).
-$B = \{b\}$ consists of a single state.
- T is defined above.
- $b_0 = b$.

In other words, a bus transition is enabled any time there is at least one reader and writer, and the result of firing the transition is to cause exactly one writer and one reader to execute, with the value output by the writer being shifted to the reader. Note that the bus never changes state; the only role of M_S's transitions is to synchronize the transitions of users of the bus.

2.2 Information Flow Analysis

Information flow analysis is concerned with studying and characterizing flow of information within a system with the aim of enforcing confidentiality of system data. Access-control mechanisms restrict resources to certain class of users or allow only certain operations on a controlled resource for a particular class of users. Information flow is traditionally considered to be more fine-grained than access control: over here the untrusted users are allowed access to the resource (unlike in access control where the access itself is restricted) but the information that the untrusted observers can obtain by using the resource would be regulated by the design of the communication/coordination framework so that the untrusted observer may not deduce anything that it is *not supposed* to know.

Let us consider the following pseudo-code fragment:

$$\text{If } (salary \geq 10,000) \text{ then } public = 0 \text{ else } public = 1;$$

Here if $salary$ is a protected variable (with a HI security level) and $public$ be a public variable (with a LO security level) then an untrusted user by observing the value of $public$ can deduce some infromation about the value of $salary$ (even if he does not get the exact value).

Information flow policies are *rules* that are checked to see that such leaks are not present. For instance, a possible information flow policy to prevent the above information flow might be that if the guard of a command (the *if* statement) contains a reference to a HI variable and if there is any LO variable in its body (the *then* part) then that code fragment is *insecure*.

Distributed information flow analysis studies the information flow analysis in a distributed domain taking into account the flow of information between components. Let us consider another simple example as before:

$$\text{If } (salary \geq 10,000) \text{ then } put(private = 0);$$

Here $private$ is a HI security level variable and put is a network operation which puts the variable $private$ on the network (after setting it to 0). Here the guard and body of the *if* statement are both HI-level and so it *passes* the policy of the previous case. Now if there are untrusted agents listening on the network, then they are able to observe the fact that a variable was put onto the bus (they are still not able to read the value of $private$) and from that deduce some information about $salary$. (the problem here is that while $private$ is HI-level the coordination operation put is not).While in the previous example, we could impose information safety by simply looking at the isolated component, here the policy will have to be more subtle.

The insecurity in the above example arises from the fact that the put operation happens for one branch of a conditional statement (the *if-then* part and not the *if-else* part). A non-distributed policy might be to stipulate that regardless of the outcome of the guarding *if* condition, the network operation put will take place (with different values of course) so that an untrusted observer cannot deduce which part of the conditional statement got executed. However this would neccesiate modifying the design of the component which we want to avoid (for the sake of re-use). So instead of the above component-specific policy, we attempt to restrict the coordination inherent in the put operation so it becomes *invisible* to any untrusted observer. Consequent examples in this paper show how this *coordination invisibilty* is encoded in terms of a specific distributed policy and how our framework handles it.

One way to go about enforcing this policy at the model level would be to assign a HI security level to any put operation, and following the CCS [12] and CSP [3] paradigm of synchronous handshaking, stipulate that any reader that wishes to coordinate with put would have to do so on a label with a HI security level. While this would work on simple handshake type of communication, it is rare that real-world systems use such a simple method of inter-process communication/coordination—-multicast, asynchronous communication and other sophisticated forms of communication are more realistic IPC mechanisms. These are not supported natively by CCS/CSP-based ADLs and so in such frameworks, any such communication would have to be simulated by a sequence of biparty handshakes. Now there is no simple way to write down a policy that works on the labels of the transitions which are simulating a single co-ordination operation: what instead is needed is a way of encoding *coordination* abstraction entities which

encapsulate the semantics of the specific IPC faithfully but are actually single transitions. This is where SAID steps in by enabling us to obtain single-transition coordination abstractors on which we may impose information flow policies in an *atomic* fashion.

3 SAID

In SAID, there are three kinds of component transitions that use ports ($T_{\text{write}}, T_{\text{read}}$ and T_{rpc}):

- An *output transition* $\langle q, w\,!\,v, q' \rangle \in T_{\text{write}}$ indicates a state change from q to q' when value v is written out to the environment on write port w.
- In *input transition* $\langle q, r\,?, f \rangle \in T_{\text{read}}$, f is a function mapping values (of variables) to states. This transition indicates a state change from q to $f(v)$ if the system's environment supplies value v on read port r.
- A *remote procedure call transition (rpc)* $\langle q, w\,!\,v, r\,?, f \rangle \in T_{\text{rpc}}$, where the input parameters are supplied by v onto port w and the output result is obtained in r.

In SAID each transition is associated with a security level $=\{H, L\}$ with the function $sec\_level(t)$ returning the security level of a particular transition t. (Multiple levels of security are also possible in the SAID setting but we do not consider them in this paper) These security levels are user-supplied while defining the base Input Output Labeled Transition Systems. Another place where security levels need to be supplied by the user are in the bus data structures (whose snapshots form the bus states). As an example for a shared variable or a message queue bus the data structure that implements the communication must be provided a security level. A subsequent example will make clear this intuition.

The role of the bus is now to look at the security levels of all the transitions that want to write to it or read from it and then based on the security policy the bus enforces, decide which of these transitions will be given the chance to write and which of these transitions will be given a chance to read.

3.1 Examples

Synchronous Broadcast. Let us first consider synchronous broadcast without any security labels. In this communication discipline, there is one writer and multiple readers. It is analogous to the example in the previous section which had multiple writers and readers blocking till the communication fired. The only difference between synchronous broadcast and the synchronous biparty handshake in the previous example was that while in the previous example we chose *one* writer and *one* reader non-deterministically and made them "handshake" by passing a value between them, over here *all* the readers who want to read will be supplied the data value (in contrast to just one). Another difference is that we do not require the presence of at least one reader. The choice of a single writer however is non-deterministic as before. The definition of the transition predicate encapsulates this intuition.

T contains all transitions for which T_P is true:

$$\exists \langle w, v \rangle \in WV.W = \{w\} \wedge RV = \{\langle r, v \rangle \mid r \in R\}.$$

Since the bus does not need to store the data and merely needs to pass it on, there is only a single state in the bus. Hence the target of the transition b' is always equal to b.

A bus $M_{broadcast} = \langle I, B, T, b_0 \rangle$ encapsulating synchronous binary handshaking may be defined as follows.

-I = a tuple consisting of two finite set of ports (read and write).
-$B = \{b\}$ consists of a single state.
- T is defined above.
- $b_0 = b$.

Now by the communication discipline imposed by the above bus any of the writers who want to write (denoted by their membership in WV) can be allowed to write. This means if there is a maximum of n writers who may want to use the bus at a single point of execution there will be n different non-deterministic choices. Among the n non-deterministic choices, some of them may be considered insecure due to particular information flow policies and may be omitted from the global state space. SAID allows us to impose these policies in a natural fashion.

Let us assume the following information flow policy:

> *"Only those readers that have security level greater than or equal to the writer will be allowed to read; all the rest of the readers shall be blocked."* (P_1).

T contains all transitions for which T_P is true:

$$\exists \langle w, v \rangle \in WV. W = \{w\} \wedge RV = \{\langle r, v \rangle \mid r \in R \wedge sec\_level(r) \geq sec\_level(w)\}.$$

This rule encapsulates the policy that if a writer has a security level of L (in other words the transition contributed by the writer to the bus has level L) then anyone will be allowed to read what it has written. If the writer's security level is H then only those readers that have a level of H will be allowed to read the value and the readers who have security level L will be blocked. Let us consider another security policy:

> *"Only those writers with a security level lesser than or equal to all the readers will be given a chance to write"* (P_2)

T contains all transitions for which T_P is true:

$$\exists \langle w, v \rangle \in WV. (\forall r \in R. sec\_level(w) \leq sec\_level(r)) \wedge W = \{w\} \wedge RV = \{\langle r, v \rangle \mid r \in R\}.$$

Over here we apply the restriction at the writer's end (whereas in P_1 the restriction was at the reader's end) in that rather than having an unrestricted non-determinstic choice among all writers as in the previous policy we restrict our choice of writers from among those writers whose transitions have security level lesser than or equal to all readers.

It should be clear now how subtle changes in the way T_P is constructed can lead to the definition of different kind of security policies: some constraining the choice of writer and some the readers. The convenience afforded by this methodology of defining communication discipline and access constraints at the same time is quite signficant

to the designer; she may now at very insignificant incremental effort construct these different *buses* and play around with them in the software architecture.

For instance it may be interesting to consider the security-functionality tradeoff in a particular coordination architecture between the application of P_1 and P_2:in one sense P_2 may be deemed to be more secure than P_1 because in P_2 a writer with a transition of level H cannot write if there is even one reader with a security level of L listening in on the bus interaction. However in P_1 the write takes place even if there are insecure readers; its just that they do not receive the data value.

In terms of information flow, P_1 can still be considered secure because the low-privileged readers will not know if the broadcast took place at all because they will always remain blocked. In other words, a low-privileged reader will not be able to distinguish whether it is waiting because there has been no write by the high privileged process or whether the write operation has already been completed. (the communication here is *silent* with respect to the untrusted/low-privilege reader. Contrast this with the following policy P_3 which is closely similar to P_1.

> *"Only those readers that have security level greater than or equal to the writer will be allowed to read; all the rest of the readers shall be denied access."* (P_3)

T contains all transitions for which T_P is true:

$$\exists \langle w, v\rangle \in WV.W = \{w\} \wedge RV = \{\langle r, v\rangle \mid r \in R \wedge sec\_level(r) \geq sec\_level(w)\} \cup \{\langle r, DENIED\rangle \mid r \in R \wedge sec\_level(r) < sec\_level(w)\}.$$

Over here the low-privileged readers are no longer blocked and get an explicit $DENIED$ message. Despite the fact that the low-privileged readers do not know the value of the secure data passed through the bus, they could still obtain the information that a secure communication took place by checking for the $DENIED$ message. Thus for P_3 there is information leak which is not present in the closely related policy P_1

In terms of functionality however the reverse is true: P_3 is more functional than P_1 because P_1 works by blocking lower privileged components while P_3 allows lower privileged components to continue with their operation even if they are not allowed to participate in an interaction.

Another point to consider: P_2 leaves the door open for a denial of service attack on the communication whereas a low privileged reader can keep on entering into a secure group communication and prevent any kind of high privileged data from being transmitted. This attack would fail on P_1 and P_3 because transmission of any kind of data can go in even in the presence of low privileged readers.

Summarizing the lessons from the above discussion, it is very important for the designer to play around with different policies at design time and study their effects on her design decisions. SAID provides her with an efficient framework for doing so.

Asynchronous Communication. So far we have been considering synchronous communication where the bus state does not change. This simplifies the definition of security policies as we only need to be considered with the security labels of the transitions participating in an interaction. But once we enter the domain of asynchronous communication,

buses are no longer *stateless* and we need to supply security labels on bus data structures also and incorporate them into the information flow policies.

One may find many different forms of asynchronous communication used in modeling distributed systems: bounded / unbounded buffer, shared variables, etc. All involve a shared data structure into which writers deposit data and from which readers extract data. In what follows we give a general scheme for defining non-lossy asynchronous communication primitives in AID and show how it may be specialized to implement asynchronous mechanisms.

We begin by defining a generalized "storage structure".

Definition 2. *A* storage structure *is a tuple* $\langle B, put, get\rangle$*, where* B *is the set of states,* $put \in \mathbb{V} \times B \to B$ *is a partial function, and* $get \in B \to (\mathbb{V} \times B)$ *is a partial function.*

Intuitively, the states of a storage structure indicate "what's stored", while put and get insert and extract, respectively, data stored in a state. As an example, consider how a storage structure corresponding to a five-place FIFO buffer might be defined, where $\mathbb{V}^*$ is the set of sequences of values, $|\ell|$ is the length of sequence ℓ, and $\cdot$ is the sequence concatenation operator.

- $B_{FIFO} = \{\ell \in \mathbb{V}^* \mid |\ell| \leq 5\}$.
- $put_{FIFO}(v, \ell) = \ell \cdot v$ if $|\ell| < 5$, and is undefined otherwise.
- $get_{FIFO}(v \cdot \ell) = (v, \ell)$; $get_{FIFO}(\ell)$ is undefined if ℓ is empty.

A storage structure may also be given for a shared variable. In this case, the states of the variable correspond to the values that the variable can hold.

- $B_{SV} = \mathbb{V}$.
- $put_{SV}(v', v) = v'$.
- $get_{SV}(v) = (v, v)$.

Both put_{SV} and get_{SV} are total functions. Note that get_{SV} does not change the state of a variable, reflecting the fact that read operations on a shared variable do not change the state of the variable.

Given a storage structure $D = \langle B_D, put_D, get_D\rangle$ and distinguished storage state $b_D \in B_D$, we may define an asynchronous D-bus $\langle I, B, T, b_0\rangle$ as follows.

- I is a tuple consiting of two finite sets of ports (read and write).
- $B = B_D$.
- T is defined below.
- $b_0 = b_D$.

T contains all bus transitions of the form

$$b \xrightarrow[WV\ R]{W\ RV} b'$$

such that $WV \neq \emptyset$ or $R \neq \emptyset$, and: either $RV = \emptyset$ and

$$\exists \langle w, v\rangle \in WV.\, W = \{w\} \wedge b' = put_D(v, b),$$

or $W = \emptyset$ and

$$\exists r \in R, v \in \mathbb{V}.\, get_D(b) = \langle v, b'\rangle \wedge RV = \{\langle r, v\rangle\}.$$

In other words, M_D does not limit the connections coming into it, and its transitions are candidates for firing if at least one writer or reader wants access *and the relevant* put_D *or* get_D *operations are defined in the current storage state*. If e.g. $get_D(b)$ is undefined, then no reads can be performed because the condition "$get_D(b) = ...$" is untrue.

Adapting this into the SAID setting we can now modify the $get_D(b)$ and the $put_D(b)$ to be security aware. For instance we could have the policy that:

> *If the security level of the storage structure is* l *then it can be written to/read from by a write/read transition with a security level higher than* l.

Let us look at how we would implement the sub-policy that

> *If the security level of the storage structure is* l *then it can be written to by a write transition with a security level higher than* l.

In order to do that we need to first assign a security level to the storage structure under consideration. Then we would need to modify the definition of $put_D(b)$ such that a put operation is completed if and only if the security label of the transition contributing the data value to be *added* to the storage structure is greater than or equal to the security level of the storage structure.

The way asynchronous communication is defined above, the writing transition will be blocked in case the shared structure is *full* (ie it has reached its capacity) or the write transition does not have the privilege to write to the shared structure. If the writing transition is sure that the shared structure is not full or that it can never be full (for instance a shared variable can always be written upon) then it can deduce some information from the write operation (the information being that it does not have access privilege on the shared structure). Even this information leak can be plugged by redefining the put operation so that it is always defined even if the write operation failed. In that case the writer will not know if his write went "through" and thus will not be able to deduce any information. However this comes at the cost of usability as even a component executing a write transition that has the proper security level will not know if the data it sent got written onto the shared data-space or not.

The other subpolicy (relating to read) can be implemented in an analogous manner.

Transitive Security Policies. Some security policies cannot be expressed as a policy on a single interaction but instead needs to be defined on relationships between multiple interactions. As an example, let us consider a transitive security policy (a policy actually used in X-Windows [1])

> *Information displayed by an Xclient* X *can be copied by Xclient* Y *but not by Xclient* Z

In order to enforce this policy globally, we need to impose information flow restrictions on interactions between X and Y as well as between Y and Z. This is to prevent Z from indirectly obtaining X's information through the indirection of reading from Y.

We define a storage structure B_{TB} which is a table. A table is represented as a set of values with operations for insertion into table and a boolean $match(v, TB)$ operation which takes a value and returns true if it is present inside the table and false otherwise.

- B_{TB} = a set of values
- $put_{TB}(v) = B_{TB} \bigcup \{v\}$
- $match(v, TB) = true$ if $v \in B_{TB}$ else $false$.

T contains all bus transitions of the form

$$b \xrightarrow[WV\ R]{W\ RV} b'$$

$\exists \langle w, v\rangle \in WV, r \in R.w \in I(X) \wedge r \in I(Y) \wedge W = \{w\} \wedge b' = put_{TB}(v) \wedge RV = \{\langle r, v\rangle\}$
or $\exists \langle w, v\rangle \in WV, r \in R.w \in I(Y) \wedge r \in I(Z) \wedge match(v, TB) = false \wedge W = \{w\} \wedge RV = \{\langle r, v\rangle\}$
This policy states that 1) if Xclient X wants to write and if a Xclient Y wants to read then transfer the value between X and Y and record the data in the table 2) if Xclient Y wants to write to Z then allow it to write only if the data is not in the table i.e. was not part of a privileged communication between X and Y.

Here the entire security is being enforced at the communication layer without altering the components in any way. Y may be a insecure component which may want to transmit its value to Z but the communication policy will prevent it from doing so (thus we obtain a secure composition of untrusted components).

4 Discussion

In distributed information flow analysis, designers are interested in checking to see if the global pattern of information flow through the system satisfies certain policies. The way this is accomplished is by looking at the system in its totality and then checking for policy violations.

In our approach we follow the general AID paradigm of separating communication definition from component definition and apply it to information flow analysis. Buses in AID are abstractions of the communications being used in a distributed system which by virtue of SOS rules stich together component transitions to form global system-level transitions . Looking at it another way, rather than telling us exactly how the communication is being accomplished (its implementation) buses define the abstract behavior of the particular communication it encodes . In the final analysis this is all that we need in order to study the behavior of the components.

Similarly in SAID we abstract away details of how exactly the information flow policy is enforced. Instead we represent the behavior of communication under the imposed security policy by SAID buses with the aim of obtaining a precise description of component behavior in the particular coordination framework of the distributed system without considering the details of how the policy is enforced. Once this is achieved, we can then do information flow analysis on the components themselves independent of the communication.

A relevant question that may be asked is how can we guarantee that our assumptions on the information policies imposed on inter-component communication are actually satisfied by the actual implementation of the communication. Revisiting our synchronous

broadcast example from the previous section: how can we be sure that the actual communication infrastructure that performs the message broadcast satisfies the policies P_1, P_2 or P_3. After all, it can be argued that all we are doing is that we are asserting that a particular policy holds for the communication and based on that assertion we are then looking inside each component and analyzing information flow inside each of them.

The answer to this question lies in understanding the hierarchical way we build up complex safety and security critical systems. In general the synchronous communication "bus" used in the example may in turn be built up from multiple interacting components with their own communication disciplines and policies. In that case we need to break down the global policy of the bus into sub-policies on the simpler communications used by the distributed system implementing the broadcast. Then we need to encode the broadcast system as a SAID system and iteratively go down the hierarchy to simpler systems where the policies may be shown to be trivially true. For now this decomposition of complex policies into simpler policies is manually done but future work lies in automatically generating these sub-policies from the policies as we go down the analysis hierarchy.

Continuing the synchronous broadcast example let us assume that it is implemented as a reliable broadcast protocol on the lines of the protocol in [2] . Then in order to validate our assumptions on the information flow we need to construct a SAID description of the broadcast protocol and validate the sub-policies on the system in the same way as we did for the higher level system that uses the broadcast bus.

Our correct-by-construction approach (where correctness is defined as *adherence to an information flow policy*) for composing components makes it unnecessary for us to apply post-construction information flow based analysis routines for the entire system. Our policy-enforcement method is very similar to the way we enforce the semantics of coordination–which is what enables us to reuse the entire AID framework with minimum modifications (addition of security levels on transitions). As a result, the state-space benefits (due to the one-transition-per-communication principle of AID) and the ability to plug in heterogenous components (as long as they can be translated to a LTS) are features SAID inherits from AID making it a robust environment for safety-and-security codesign. (The advantages of AID alluded to here are not discussed in this paper; the interested reader is asked to refer to [8, 9] for details.)

5 Future Work and Conclusions

Future work consists of equipping SAID with an explicit notion of time so as to enable the expression of policies which depend on the temporal ordering of messages. Other future work lies in finding automated ways of taking a security policy imposed on the abstraction and breaking it down to sub-policies that can be checked on the implementation.

In conclusion the utility of SAID lies in its ability to reduce the global information flow problem to local information flow by a form of assume-guarantee reasoning (where the assumptions are restrictions on the flow of information between components and the guarantee part is the information flow analysis on the local components) and its reuse of the coordination infrastructure provided by an ADL to perform *unified communication and information flow* specifications on a software architecture.

Acknowledgments. I wish to thank Rance Cleaveland for his detailed comments and suggestions on this paper. I would also wish to acknowledge the comments of the anonymous reviewers.

References

1. Sun solaris documentation. *Solaris X Windows Developers Guide: SUN Microsystems*, 1999.
2. Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, 1984.
3. C.A.R. Hoare. Communicating sequential processes. 1985.
4. H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *J. Computer Security*, 11(4):615–676, 2003.
5. N.Carriero and D.Gelertner. Linda in context. *Communications of the ACM*, 32(4):445–458, 1989.
6. G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.
7. R.Allen and D.Garlan. Formalizing architectural connection. *16th International Conference on Software Engineering*, 1994.
8. Arnab Ray. Compositional modeling of interaction centric concurrent systems. *Ph.D thesis, State University of New York at Stonybrook*, 2004.
9. Arnab Ray and Rance Cleaveland. Architectural interaction diagrams: Aids for system modeling. *Proceedings of the International Conference on Software Engineeri ng,(ICSE)*, pages 396–406, 2003.
10. Arnab Ray and Rance Cleaveland. Formal modeling of middleware-based distributed systems. *Workshop on Formal Foundations of Embedded Software and Component-Based Architecture, Barcelona, Spain, April 2004. Satellite workshop of the European Joint Symposia on Theory and Practice of Software*, To appear in Electronic Notes in Theoretical Computer Science,2004.
11. R.Focardi, R.Gorrieri, and F.Martinelli. Information flow analysis in a discrete-time process algebra. *IEEE Computer Security Foundations Workshop*, pages 170–184, 2000.
12. R.Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 1980.
13. Sewell and Vitek. Secure composition of insecure components. In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.
14. Richard Soley and the OMG Staff Strategy Group. Model driven architecture.

Experience Using a Coordination-Based Architecture for Adaptive Web Content Provision*

Lindsay Bradford, Stephen Milliner, and Marlon Dumas

Centre for Information Technology Innovation,
Queensland University of Technology, Australia
{l.bradford, s.milliner, m.dumas}@qut.edu.au

Abstract. There are many ways of achieving scalable dynamic web content. In previous work we have focused on dynamic content degradation using a standard architecture and a design-time "Just In Case" methodology. In this paper, we address certain shortcomings witnessed in our previous work by using an alternate coordination-based architecture, which has interesting applicability to run-time web server adaptation. We first establish the viability of using this architecture for high-volume dynamic web content generation. In doing so, we establish its ability to maintain high throughput in overload conditions. We go on to show how we used the architecture to achieve a "Just In Time" adaptation to achieve dynamic web content degradation in a running web application server.

1 Introduction

Researchers have recently discussed the need for adaptable web-provision technologies, particularly in terms of architectures that cater to varying degree of adaptability [1]. This focus on architectures is perhaps due to a growing realisation that the architecture chosen is one of the key factors to successful system deployment [2]. If we want an adaptable system, its architecture must first support adaptation.

Architectures that offer a coordinated model of interaction (for example, JavaSpaces [3]) provide certain characteristics that are attractive to achieving adaptation. These architectures separate components from how the components interact via some form of coordination, and should thus make run-time component replacement and changes to component interaction easier to accomplish. Such architectures introduce decoupling across space (allowing distributed behaviour), time (allowing asynchronous communications) and interface (allowing easier replacement and interaction of components), which in turn, allows a wide range of choice in the types of adaptation that can be implemented.

* This research is funded in part by SAP Research Centre, Brisbane and Queensland Government.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 140–156, 2005.

Many modern web-centric architectures take a somewhat coarse-grained/ static approach where components and their interactions are fixed at design time. As a result, developers can be locked into limited types of adaptability, and, at worst, could be forced to construct adaptation techniques at design time along with the core deliverables and in ways that may not be appropriate. Adaptive systems that take this "Just In Case" (JIC) approach run the risk of not being able to adapt key components to environmental changes. Even if the components to adapt are correctly identified and alterable, the system designer needs to also successfully guess the right type and amount of adaptation to apply when designing their systems. That is, they must have complete a priori knowledge of all possible situations.

JIC adaptation techniques are highly predictive, whereas "Just In Time" (JIT) adaptation techniques are highly reactive, even to the point of being manually constructed for specific short-lived circumstances. More formally, we describe JIT adaptation as the introduction of behavioural change into a system once some event has occurred that requires such change, and that the adaptation made is targeted specifically to this event. By breaking the HTTP content delivery of a web application server into a number of components interacting via architectural coordination, the degree of predictiveness required for adaptation can be minimised, or even removed, by altering just those components and interactions that need changing at run-time.

In this paper, we aim to establish whether our coordination architecture, with its optimisations for localised coordination of network deliverable components, is capable of web content delivery with sufficient latency and throughput make it viable for use as a web application server. We also aim to establish whether the architecture can significantly and rapidly adapt via a JIT delivery of new behaviour, even under extreme load conditions.

Our focus in web application adaptation is on achieving scalability at a single web application server via behavioural change. In contrast, other popular techniques for supplying adequate web scalability involve the over-provision of a service-provider's computing resources, typically by supplying several duplicate machines that share requests through some load balancer. This solution is not only costly, but also requires increased configuration and administration effort.

In previous work [4], we constructed a dynamic web content degradation system based on elapsed-time measured at the server. Elapsed-time is a critical factor of *user-perceived quality of service* on the Web [5] [6] [7], and in human–computer interactions more generally [8]. Aspects such as layout, graphics and such are far less likely to effect user-perceived quality of service, suggesting that degrading such aspects for better elapsed-time responses is an area of web adaptability deserving further investigation.

We used a mainstream web application server, namely Tomcat[1], to supply several approaches to generating web content for a given URI. An elapsed-time-based algorithm was used to decide when to degrade the web content delivered

[1] http://jakarta.apache.org/tomcat/

by choosing between these approaches. For example, a baseline approach might be a complete web-page portal collating results from several other web-pages. Under load, the base approach generating this complete portal might be replaced by a lightweight approach that returns only those portlet images that the server has cached. The algorithm and alternate approaches required are an example of JIC web adaptation, and is typical of the extra pre-emptive overhead inherent in such schemes.

There are uncertainties in using our coordination architectures for web content delivery that we wish to explore. Firstly, how does it behave under load conditions, and secondly, how should it be used to achieve high-performance web content delivery. To that end, we offer two contributions in this paper, being i) we establish that our coordination architecture can be used to serve high-demand dynamic web content fast enough to be considered viable and show that it exhibits good throughput characteristics under load, and ii) we present a method for exploiting our architecture to seamlessly adapt to very different behavioural patterns in a JIT fashion. To illustrate our second contribution, we introduce automated content degradation into an overloaded web application server at run-time.

Section 2 discusses the design of the system by first describing the base architecture (Sect. 2.1), then the adaptation of the base architecture into a web application server at run-time (SecT. 2.2), and finally the adaptation this web application server into one capable of automated content degradation (Sec. 2.3). Section 2.4 discusses some of the lessons learned with early attempts. Section 3 establishes the viability of our design via experiments. Section 4 discusses related work and Section 5 concludes the paper.

2 Design

In Section 2.1, we discuss our coordination architecture and the localised optimisations that make this architecture a viable candidate for delivering both a) high load web service provision and b) flexible service adaptation. In Section 2.2, we discuss how we used this architecture to deliver our JIT adaptable web application server.

2.1 Service Provision with ActiveObjectSpaces

ActiveObjectSpaces (AOS) is a distributed coordination middleware drawing on three main predecessors: Blackboard Architectures (for example, Hearsay [9]), Linda [10] and Sun's JavaSpaces [3]. The AOS supports four main primitives `read`, `take`, `write` and `notify`. Respectively, these primitives copy objects from the AOS to clients, move objects from the AOS to clients, copy objects from clients to the AOS, and instruct the AOS to notifty clients of template matching objects the AOS holds. The AOS has an extended notification API to JavaSpaces (notification can trigger on arrival, on existence and on deletion). Like JavaS-

paces, coordination is achieved via notification templates describing essential aspects of objects on the AOS that clients wish to interact with.

Perhaps the most novel aspect of the AOS is the notion of *Active Objects*. Active Objects are AOS-aware objects that execute in their own thread of control within the AOS itself. The AOS runs with a thread pool that it allocates to execute active object behaviour upon template matching events. Thus, a number of threads (to the AOS server limit) could be executing in a single active object instance concurrently. By sending appropriate active objects[2] to a running AOS, and incorporating a coordination protocol for their interaction, we can dynamically deploy the components of a stand-alone server. Having constructed a service via a collection of loosely coupled coordinated components we can achieve fine-grained intra-server adaptability.

An AOS server initially knows nothing about receiving HTTP requests or delivering web content. Our HTTP module (Sect. 2.2) is delivered to the AOS which the AOS then executes. Once the module is running, the AOS has dynamically been converted into a web application server. Later, when the server exhibits poor response times, a content degradation module (Sect. 2.3) is delivered to the AOS, converting the AOS into a server capable of selecting degraded content based on elapsed server response-times.

2.2 Dynamic Web Content Delivery via ActiveObjectSpaces

A number of active objects and object class definitions are delivered to the AOS via the HTTP module (see Fig. 1). The class definitions and active objects are removed from the object space and installed for use by the AOS. For most of the active objects, this simply means telling the server to execute a callback method on them when data objects matching the notification templates specified

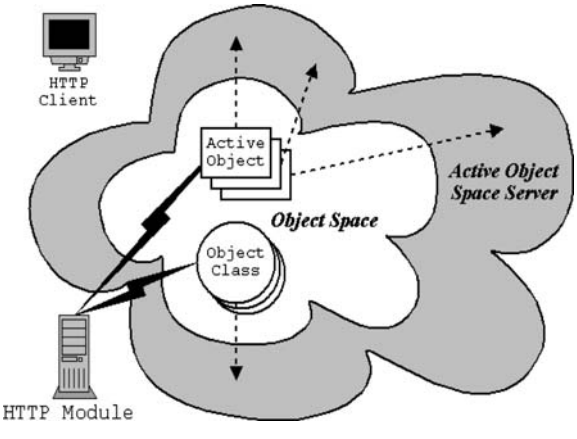


Fig. 1. Turning the AOS into a HTTP Server

[2] In this paper, we send one instance of each type of active object to the AOS.

are delivered to the object space. Later, when the AOS starts accepting client requests, the class definitions delivered via the module will allow the active objects to create and manipulate objects needed for HTTP content delivery.

Most of the content delivered in the HTTP module is generic HTTP handling code, and much like 'out of the box' application servers, is expected to remain unchanged over time. However, we do have the option of making quite radical changes even at the HTTP layer if required. The content degradation module (discussed later), alters behaviour, and is roughly analogous to delivering a new WAR file to a running Tomcat application server. What makes our approach novel with respect to more contemporary application servers is that both the HTTP generic behaviour and the content delivery behaviour can be adapted, and at a fine level of granularity. By using the space as our core state repository, we can also draw on looser, more interactive ways of generating web content that are not possible with call and return style architectures.

Figure 2 gives an overview of the design principles employed when building our server, supporting a subset of the HTTP/1.1 standard[3]. The active object that communicates with HTTP clients first asks the object space to deliver it any completed responses that may be deposited there. It then establishes a server socket and falls into a continuous loop, accepting client HTTP requests that it converts into request objects and delivers to the object space. Whenever a completed response is placed on the object space, the response is delivered back to this active object and the object transmits the response back to the client. Response delivery is managed via notification template processing.

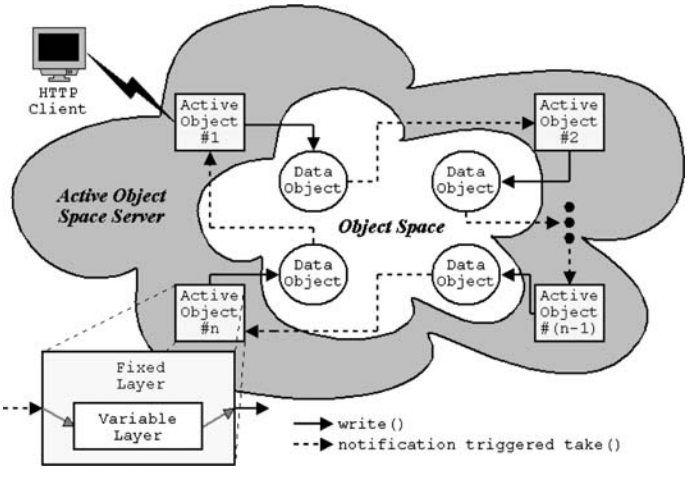


Fig. 2. HTTP Server Design

In early experiments, we discovered that the active object responsible for communicating with the HTTP clients needed to contain a mapping of open

[3] http://www.ietf.org/rfc/rfc2616.txt

client sockets to requests as part of its internal state. Java socket objects represent underlying operating system resources, a class of object that cannot be sensibly copied into the object space. It was therefore necessary to have this active object retain socket objects to support the HTTP/1.1 requirement for using the same socket for request receipt and response delivery.

With careful usage of notification templates, the active objects cooperate to produce HTTP responses in a way reminiscent of a "Pipes and Filters" architecture [11], where each active object acts as a filter and enacts some self-contained transformation of the HTTP content. However, there are some important adaptations our architecture allows that are not available to more rigidy coordinated architectures.

Firstly, the object space allows decoupling of response generation across time. A response in a partially complete state could be left that way for an indefinite period, and service other requests instead in a manner reminiscent of Floyd and Jacobson's link-sharing [12]. We see this desirable for certain classes of HTTP client, such as search bots that adversely effect response times and, in turn, displease human clients who are more judgmental of poor web response times than computers [6].

Secondly, active objects are loosely decoupled across their interfaces, using template matching to move data through its lifecycle. Though we have not taken full advantage of this loose coupling, we could (as an example) have certain active objects with the same template effectively "compete" for processing a HTTP request at some point in its lifecycle. As active objects are just a special type of client for the object board, certain behaviour and its partially complete state could be relocated to a remote location if it made sense to do so.

To make behaviour replacement easier, we deliberately minimise the state each active object holds by making as much use of the object space as is appropriate. Data objects represent the core state of a HTTP response at various points in its lifecycle from request receipt through to response delivery.

We separate active objects loosely into a "fixed" layer and a "variable" layer. The fixed layer typically focuses on communication with the object space. It delegates behaviour that manipulates the content retrieved from the object space to its variable layer component(s). The notification template that this layer establishes with the object space describes the nature of objects it desires from the object space. The fixed layer is also responsible for spotting and refreshing variable layer components when they are delivered to the object space, again via notification templates. The variable layer consists of one or more objects that work together to process data handed to it by the fixed layer. There is nothing stopping a variable layer object from talking directly to the object space or even containing a nested fixed/variable layering itself.

We considered an alternative to behaviour update, where active objects direct the object space to inform them of the delivery of a replacement for them. A notification signaling that a replacement had arrived would trigger their termination. A drawback we saw to this is that a great deal of code (especially for marshalling data to and from the object space) would be shared the same across

new and old active objects. We opted for an active object design that expects more frequent changes in component behaviour than changes to coordination.

2.3 Dynamic Web Content Degradation via ActiveObjectSpaces

Figure 3 shows what happens generally when we vary our HTTP Server behaviour. A number of new active objects, class definitions and variable-layer objects are delivered to the object space. The server then installs the active objects and class definitions. Finally, active objects that are informed of matching variable-layer objects will retrieve and install these new variable-layer objects.

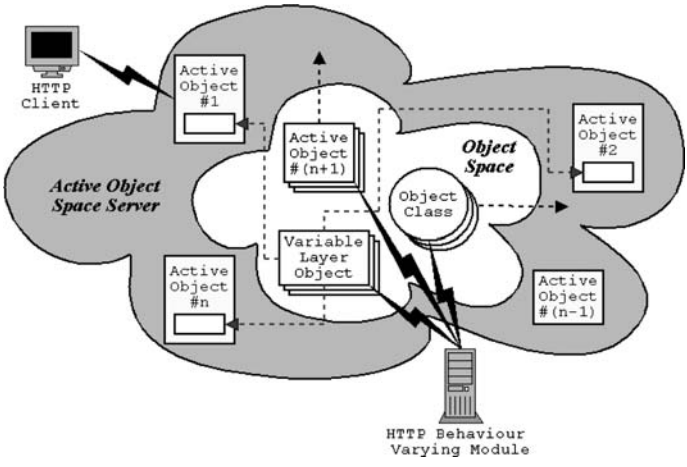


Fig. 3. Content-Degrading HTTP Server via AOS

In our specific case for web content degradation, a second client module connects to the server and delivers a single active object and a number of support objects and their class definitions. As this active object is being installed in the server, it asks the server to deliver `ResponseTime` objects (a new type of data object the server was initially unaware of) to the active object. The active object then delivers a variable-layer object to the server space capable of generating these new data objects.

The active object responsible for HTTP request/response communications objects is informed of the existence of the new variable-layer object, and installs it as a new piece of functionality to be run each time a response is successfully delivered back to a client.

This newly delivered active object implements the algorithm for elapsed-time web content degradation that we described in [4]. We leave a more detailed discussion of the algorithm to Section 3.4. Here, we will simply state that if this active object decides a new approach to generating web content is needed to ensure adequate response times, it delivers a new variable-layer object to the object space which will be picked up and installed by the appropriate active

object, for use in content generation from then on. To revert to a HTTP server that does not degrade content, we would replace the variable-layer object creating `ResponseTime` objects with a "do nothing" equivalent, and the variable-layer object of our new decision-making active object to one that always delivers a baseline variable-layer object.

2.4 Pitfalls Discovered

In building the base server module, we learned several important lessons in ensuring high throughput and low latency using the AOS to serve web content.

One lesson learned was to minimise the number of objects on the space growing over time. An early attempt at building the server had tried to place nearly all state on the object space. Some information, however, needed mutual exclusion (mutex) sections to ensure the correct history was being generated. As the state requiring mutex was in the object space, the mutex behaviour was coded using AOS primitive methods. The application-level implementation of mutual exclusion was enough to ensure that `ResponseTime` objects started backing up on the object space, which slowed down template matching on `take()` operations, establishing a systemic collapse of response times as more and more new request messages mixed in with the steadily growing number of `ResponseTime` objects.

In a related theme, the object space requires a great deal more object creation and destruction than a traditional call and return architecture. We learned through profiling that garbage collection on our single CPU server machine tended to contribute to poor response times when large amounts of memory were being reclaimed. Through careful configuration of the garbage collector, we were able to minimise its impact, though a more attractive proposition for future work is having a machine where garbage collection can run in parallel with service provision.

Finally, without centralised control, it became difficult to determine whether we would achieve intended behaviour until run-time. Run-time diagnostic tools such as server logs and profilers, giving real-time views into the running system became our primary method for debugging and system validation. Unit testing tended to be trivial, involving testing highly specialised, mostly stateless active objects. We couldn't verify intended overall behaviour had been achieved until actually seeing the AOS successfully marshalling request state between the loosely coupled active objects in a running environment.

3 Experiments

3.1 Experimental Setup

Experiments were carried out on a set of eight dedicated Sun boxes running Debian Linux, and a single target web server machine. The target web server machine was an 804MHz Pentium 3, with 256MB of memory, running Fedora

Core 1[4]. The AOS was run on a Sun 1.4.2-b28 JVM. The machines were connected on an isolated 100 megabit per second LAN.

One of the Sun boxes acted as a traffic synchroniser by broadcasting UDP messages to synchronise the activity of the other client machines. The remaining seven Sun boxes were used as test clients and would listen on heartbeat signals from the traffic synchroniser. For the tests we describe here, the traffic synchroniser was used only to ensure that all client machines started requesting content from the server at the same moment.

One of the test client machines was used to generate a sufficient request rate to ensure response times for the baseline approach were above one second. This client machine sent a request and, once the request was received, would immediately disconnect without informing the server, generating the equivalent of a denial-of-service attack. Both architectural styles (Tomcat and AOS) continue to process requests and fail only when attempting to transmit responses back to the non-sampling client on a defunct client socket. The remaining six machines sent requests and waited for responses before sending new requests. The figures reported in our third experiment have been sourced from the data collected off these six machines.

We simulated memory-intensive web content provision by generating HTML responses that ran a number of loops, accessing a random element of a 2K block of memory in each loop as they processed output. We considered four approaches for supplying a reply to a given URI that we wanted to automatically vary content generation on. Given the capacity of the target machine running the web server, we settled on four approaches doing 1,000,000, 500,000, 250,000 and 125,000 loops respectively. The 1,000,000 loop approach we name our "baseline" approach, which represents the original content being delivered before content degradation becomes necessary.

There is compelling human/computer interface research suggesting that we should try returning responses to web requests within one second for human clients if we want them to remain unaware of having waited for those responses [6]. When we present our results, we label responses adequate if they took under one second to return to the sampling clients. The aim we set out to achieve with our content degradation module is to maximise the number of adequate responses returned by trying to get response times back under one second once a server fails to do so for some type of web content.

3.2 Experiment 1: Sustainable Throughput per Approach

We used the tool httperf [13] to establish the sustainable throughput each approach can deliver, and used that figure to establish the number of client machines required to significantly tax the server past this rate for our baseline approach. We did this by sequentially requesting 10,000 results per approach, and recording the output from httperf. The average response time we get from

[4] http://fedora.redhat.com/

this is around where a server starts degrading performance if requests arrive at rates faster than this. The key results are displayed in Figure 4(a).

Loops	Resp. Time (ms)				Std. Dev.
	Avg.	Med.	Min.	Max.	
125,000	51	50.5	50	162.5	2.4
250,000	84.1	83.5	83	105.8	2.1
500,000	149.7	148.5	148.6	477.8	4.4
1,000,000	280.5	279.5	278.3	696.3	6.8

(a) AOS HTTP Module

Loops	Resp. Time (ms)				Std. Dev.
	Avg.	Med.	Min.	Max.	
125,000	47.4	47.5	47.1	58.4	0.5
250,000	91.6	91.5	91.1	104.0	0.6
500,000	179.9	179.5	179.3	192.2	0.9
1,000,000	356.5	356.5	355.5	413.3	1.6

(b) Tomcat Application Server

Fig. 4. Sequential approach response times across architectures

Using the same hardware, we ran httperf against a Tomcat 5.0.28 implementation [14] configured as closely as possible to our HTTP module with the approaches embedded in servlets. Our aim was simply to establish whether AOS could handle a similar amount of throughput for the same workloads as a contemporary web application server. The results are displayed in Figure 4(b).

Minimum, average and median response times were similar in both the AOS and Tomcat. As the AOS implements a subset of the HTTP protocol, we argue only that the architectural overhead of the AOS makes it a viable candidate for delivering HTTP content. We note the large difference in maximum response times and thus standard deviation between AOS and Tomcat. A small minority of AOS responses reported much longer response times. The remainder, like Tomcat, returned much closer to their minimum response times.

Via profiling, we saw two contributors to this variability in AOS response times. Firstly, AOS requires significantly more short-lived objects than Tomcat in response processing, and is thus more prone to the *excessive dynamic allocation* anti-pattern [15]. Secondly, how the AOS uses threading has introduced extra non-determinism. There is a single thread per request for Tomcat, meaning no extra threading overhead whilst requests were being fed to it sequentially. In the AOS, each active object notification is executed in a separate thread. Even when sending requests sequentially, the AOS still invokes threading overhead as requests moves through its lifecycle.

We conclude that in non-overload conditions, the overheads of the AOS architecture do not exclude it from being used as a viable HTTP application server. We can expect occasionally longer response times than a single-thread per request architecture, but for the most part, responses will be returned with little extra evident latency.

3.3 Experiment 2: Understanding Overload Behaviour

As the architectures handle request processing very differently, we were interested in what to expect in worsening overload conditions on each server. In this

Requests per Sec.	Response Time (ms)				Std. Dev.	Resp. per Sec.	
	Avg.	Median	Min.	Max.		Avg.	Std. Dev.
2	279.1	278.5	277.8	338.6	3.2	2	0
3	287.4	286.5	277.9	301.5	2.5	3	0.1
5	9358.9	12635.5	298.9	13341.3	4986.1	2.8	0.4
10	10854.5	12722.5	319.7	13382.6	4009.8	3.3	0.3

(a) AOS HTTP Module

Requests per Sec.	Response Time (ms)				Std. Dev.	Resp. per Sec.	
	Avg.	Median	Min.	Max.		Avg.	Std. Dev
2	357.4	357.5	356.0	467.4	2.7	2	0
3	2382.9	1048.5	381.9	9804.4	2602.3	0.4	0.9
5	3597.3	1782.5	536.1	9680.8	3288.7	0.1	0.3
10	1475.3	1606.5	1475.3	1746.9	135.8	0	0.1

(b) Tomcat Application Server

Fig. 5. Request Rates on Baseline across architectures

experiment, we used httperf to request the baseline approach at varied request rates: from below, at around, and above the sustainable request rates derived from Experiment 1.

From Figure 5 we see that as we increase our request rates past the sustainable point, Tomcat rapidly degraded in terms of responses per second, but the AOS settled on an average response rate at around its sustainable request rate. We also see worsening AOS response times as request rates increases beyond sustainable rates. Either way, past the sustainable request rate, both architectures rapidly reached the point of poor adequacy.

The reason behind this difference in throughput between architectures is described by Welsh et al. [16]. Architectures that combine threading (primarily for IO and network blocking) with event-driven task scheduling offer excellent throughput characteristics for overload situations, though latency increases as request rates exceed a sustainable limit. Because of our globally shared object space amongst active objects with no inherent queuing per thread, we suspect that the AOS lies somewhere between Tomcat and SEDA [17] (the architecture Welsh et al. built from the principles described in [16]) in its ability to maintaining high throughput in overload conditions.

We conclude that the AOS will maintain high throughput rates that should slowly degrade (as a function of number of objects in the object space) in overloaded server conditions, but that latency will continue to increase until such time as we make content cheaper to generate in terms of CPU consumption or we offload requests elsewhere.

3.4 Experiment 3: Validating JIT Content Degradation

Our web content degradation module uses an "elapsed-time of response generation" based algorithm to choose faster or slower approaches to generating content for a given URI [4] [5]. Other content degradation algorithms could have been attempted, but are outside the scope of this research. Instead, our aim is to use the algorithm to verify that the AOS is capable of handling quite radical adaptation of both its coordination and bottleneck components in an already overloaded web application server.

We chose not to compare AOS content degradation against Tomcat. The difference between the two architectures seen in Experiments 1 and 2 lead us to suspect that the behaviour of our algorithm was not readily comparable across architectures given their very different behaviours in overload conditions. Here we concentrate on the AOS architecture only.

We used the results from Figure 4(a) to guide us in configuring the request-rate of our denial of service client. Our aim was to find a request rate that, combined with requests from the sampling clients, would ensure that response-times for the AOS baseline approach would be above one second. We settled on a request rate of one request every 10 milliseconds from this client.

Experiment 3 was run for 20 minutes. The first 10 minutes used the baseline approach delivered with the first module, and the second 10 used content degradation, delivered via the second module.

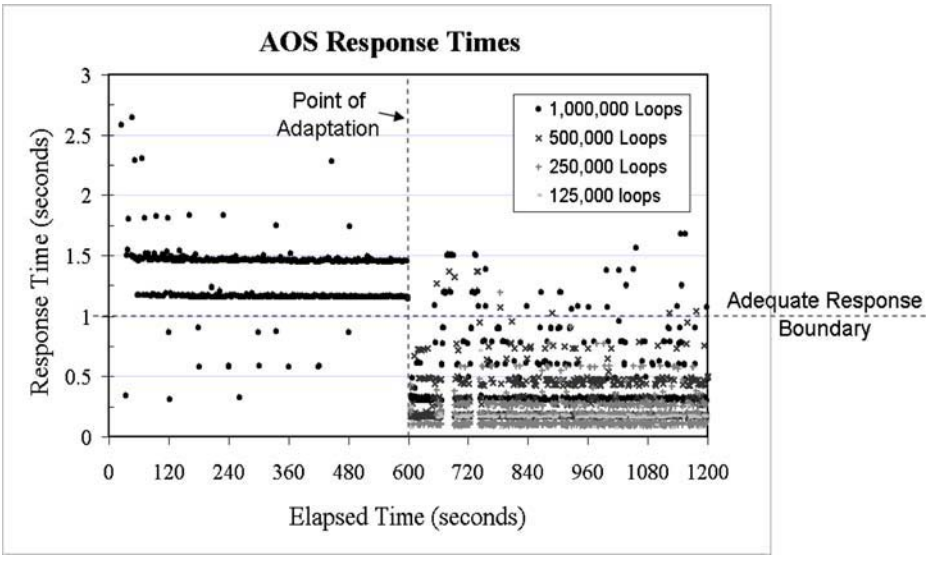


Fig. 6. AOS Responses Times

[5] We pessimistically configure the algorithm's parameters to 300ms for our *upper-time limit*, and 200ms for our *lower-time limit* in these experiments.

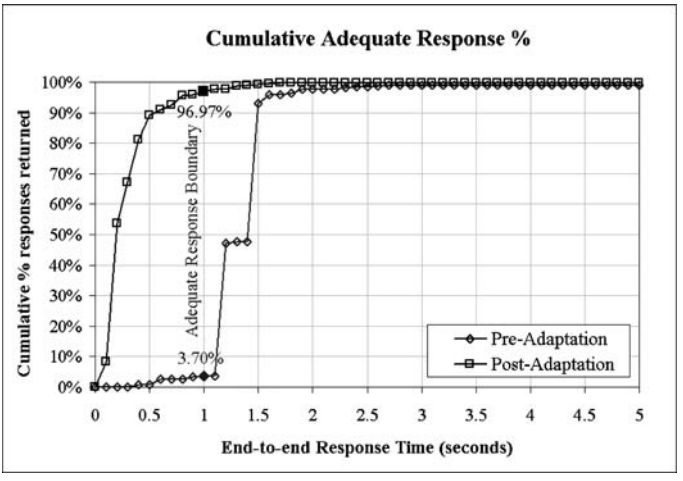


Fig. 7. Measured AOS Adequacy pre- and post-adaptation

Figure 6 shows a scatter-plot diagram of response times recorded from our sampling clients. At the 600 second (10 minute) mark, the content degradation module's delivery radically altered the response-times being reported by our target server. Where most responses were definitely taking longer than one second to deliver pre-adaptation, most responses delivered post-adaptation fell below our one-second target.

Figure 7 shows the number of adequate responses returned pre- and post-adaptation. We achieved very high adequacy. What we draw from this result is that the AOS allows relatively complex adaptation to occur under load conditions, not only in terms of a one-off change in active-object coordination, but also in terms of automated, rapid, fine-grained changes to content-generation behaviour. We have establish that the AOS is a viable candidate for further exploration in JIT adaptation of web-service architectures, even under taxing conditions.

Of secondary importance, no special effort was made here to better match the content degradation algorithm to the AOS. Because of the very different way in which the AOS behaves in overload conditions, there is some argument for making the algorithm less reactive to severe, short-lived changes in response times, and for it to take more consideration of how object space numbers might influence overall response latency.

4 Related Work

There are many ways to scale dynamic web content at a single server, such as caching ([18] lists several strategies), resource management (see [19] as an example), and content degradation. Caching dynamic web content is limited both in terms of when it can be applied, and how much benefit it can deliver [20]. Re-

source management policies often also have the undesirable "denial of service" characteristic that breaks a web user's expectations of "service on demand" from web service offerings [6].

As users seem less judgemental of content makeup (and several other quality of service measures [21]) than response times [5] [6], we see web content degradation as an area deserving further exploration. Web content degradation techniques (also called transcoding, see [22], [23] and [24]) rely on the idea that web content can be degraded to a user-tolerable degree when servers are overloaded. The degraded content should require less resources to deliver and, in turn, allow more clients to consume the desired content.

Until recently, very little has been discussed in terms of degrading dynamic web content. New adaptive architectures are being introduced that support dynamic web content degradation, but they do not explore specific degradation techniques [1]. Chen and Iyengar described a dynamic web content degradation system involving a number of tiered servers, offering content at decreasing degrees of fidelity the further from the core server a support server is in the tier [25]. We are currently unaware of any other dynamic web content degradation techniques targeting a single server besides our own [4].

Our early work has led us to the conclusion that the architecture is the key to the degree of adaptability we can achieve. Our longer-term goal is being able to apply adaptations to suit changing situations. Maximising the range of run-time adaptability we can achieve has been a driving force in this more recent work.

To that end, space-based architectures look particularly promising to us. Early blackboard architectures such as Hearsay [9] introduced the concept of expert software components watching a blackboard, and working on parts of a problem they understood. These experts were basically self-contained components with minimal state and were loosely coupled via the blackboard. As a consequence they were easier to replace with equivalent components. Later, Gelertner's Linda project [10] on generative communication discussed the greater flexibility available across both time and space by storing and manipulating generic tuples via a tuple space.

JavaSpaces replaces tuples with Java objects, which supports the movement of behaviour in addition to state [3]. JavaSpaces, however, focuses on distributed behaviour over a network. In contrast, our own interest in these architectures is in how we might take advantage of such loose coupling to introduce localised, fine-grained behaviour alterations whilst the server we are altering continues to operate. This is where the notion of active objects plays a crucial role. To the best of our knowledge, there has been no previous attempt at applying coordination architectures to the problem of highly adaptable web application servers.

5 Concluding Remarks

We show in this paper that our coordination architecture *ActiveObjectSpaces* (AOS), can serve as a viable base for adaptive web application servers. The active object primitives offered by our architecture allow us to easily deliver and

execute new components to a running AOS, and have them interact via localised coordination.

We described a way of using the strengths of this architecture to construct a simple HTTP server on top of the architecture, and then, to deliver significant alterations to the behaviour of the running HTTP server under load. Along the way, we learned that achieving good latency from the architecture required us to minimise the number of objects stored in the object space at any one time. We also discovered that mutual exclusion behaviour is best encapsulated inside active objects; building mutual exclusion via AOS primitives proves too costly in a busy server.

We have established that we can achieve our desired adaptability at the cost of some extra variability in server response times when compared against a more traditional web application architecture in unloaded conditions. The AOS matches the throughput and latency results of this traditional architecture close enough to make deployment viable. Overloading the architectures showed the AOS capable of maintaining high throughput, whereas the throughput of the traditional architecture rapidly degraded as request frequency was increased. Previous research has shown that a mix of threading and event-driven task scheduling (as implemented in the AOS) is the reason for the continued throughput under load witnessed in the AOS.

We used an automated web content degrading adaptation based on elapsed-time as a non-trivial example of the type of run-time adaptations we desire from the AOS. The adaptation involved the delivery of behaviour and new class definitions from a remote location as the server was suffering overloaded conditions. We have shown the AOS is capable of automated, rapid, and fine-grained changes to content-generation behaviour. Given the radically different behaviours of the architectures used under load, we've also seen that automated content degradation via elapsed-time measures should be handled differently to better suit each architecture.

From here we aim to better understand the AOS and how to best use it for adaptive, high volume HTTP service provision. Firstly, we wish to look at the granularity of work each active object performs and its impact on overall throughput and latency. We have seen in other research [16] that thread queuing mechanisms (including the joining of two tasks into a single queue) can be of benefit and we are interested in how these concepts might carry across into our own work.

Secondly, there is still much to understand in elapsed-time based automated content degradation. With a better understanding of what elements matter in such adaptations, we might be able to supply at least partial automation. For example, we could deliver the framework for content degradation from a library of standard adaptation modules to a running application server, and allow developers to insert new versions of degraded content as they become available.

Thirdly, our content degrading adaptation adds an extra step to the end of a pipeline of coordinated tasks. This extra step supplies replacement behaviour at run-time to the bottleneck in the process. We are interested in other types of

adaptation that benefit response times. Some examples might include i) altering the flow of objects by changing notification templates at run-time, ii) introducing active objects to compete with others for certain steps, or iii) automated migration of active objects and partially complete state to less loaded AOS environments.

References

1. Colajanni, M., Lancellotti, R.: System Architectures for Web Content Adaptation Services. Distributed Systems Online, Web Systems Topic (2004) http://dsonline.computer.org/was/adaptation.htm.
2. Garlan, D.: Software Architecture: a Roadmap. In Finkelstein, A., ed.: The Future of Software Engineering. ACM Press (2000)
3. Doberkat, E.E.: E. Doberkat, E. Freeman, S. Hüpfer, K. Arnold: JavaSpaces Principles, Patterns and Practice. Softwaretechnik-Trends **20** (2000)
4. Bradford, L., Milliner, S., Dumas, M.: Scaling Dynamic Web Content Provision Using Elapsed-time-based Content Degradation. In: Proceedings of the 5th International Conference on Web Information Systems Engineering (WISE 2004), Brisbane, Australia, Springer Verlag (2004) 559–571
5. Ramsay, J., Barbesi, A., Peerce, J.: A psychological investigation of long retrieval times on the World Wide Web. In: Interacting with Computers. Volume 10., Elsevier (1998) 77–86
6. Bhatti, N., Bouch, A., Kuchinsky, A.: Integrating user-perceived quality into Web server design. Computer Networks (Amsterdam, Netherlands: 1999) **33** (2000) 1–16
7. Bouch, A., Kuchinsky, A., Bhatti, N.: Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. In: Proceedings of the CHI 2000 Conference on Human Factors in Computing Systems, ACM (2000) 297–304
8. Miller, R.: Response Time in Man–Computer Conversational Transactions. In: Proc. AFIPS Fall Joint Computer Conference. Volume 33. (1968) 267–277
9. Reddy, D.R., Erman, L., Neely, R.: A model and a system for machine recognition of speech. In: IEEE Transactions on Audio and Electroacoustics. Volume 21. (1973) 229–238
10. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7** (1985) 80–112
11. Shaw, M., Garlan, D.: Software Architecture. Prentice Hall, Upper Saddle River, New Jersey (1996)
12. Floyd, S., Jacobson, V.: Link-sharing and resource management models for packet networks. IEEE/ACM Transactions on Networking **3** (1995) 365–386
13. Mosberger, D., Jin, T.: httperf-A Tool for Measuring Web Server Performance. SIGMETRICS Perform. Eval. Rev. **26** (1998) 31–37
14. The Apache Group: Tomcat web application server (2004) http://jakarta.apache.org/tomcat/.
15. Williams, L.G., Smith, C.U.: PASA^{SM}: A Method for the Performance Assessment of Software Architectures. In: WOSP 2002: Third International Workshop on Software and Performance, Rome, Italy, ACM Press New York, NY, USA (2002)

16. Welsh, M., Gribble, S.D., Brewer, E.A., Culler, D.: A Design Framework for Highly Concurrent Systems. Technical Report UCB/CSD-00-1108, UC Berkeley (2000)
17. Welsh, M., Culler, D.: Adaptive Overload Control for Busy Internet Servers. In: USENIX Symposium on Internet Technologies and Systems. (2003)
18. Shi, W., Collins, E., Karamcheti, V.: Modeling Object Characteristics of Dynamic Web Content. Technical Report TR2001-822, New York University (2001)
19. Chen, X., Heidemann, J.: Flash Crowd Mitigation via Adaptive Admission Control based on Application-level Observations. Technical Report ISI-TR-557, USC/Information Science Institute (2002)
20. Thomas M. Kroeger, Darrell D. E. Long, J.C.M.: Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In: 1st USENIX Symposium on Internet Technologies and Systems, Monterey, California, USA (1997)
21. Sahai, A., Durante, A., Machiraju, V.: Towards Automated SLA Management for Web Services. Technical Report HPL-2001-310R1, HP Labs (2001)
22. Amir, E., McCanne, S., Katz, R.H.: An Active Service Framework and Its Application to Real-Time Multimedia Transcoding. In: SIGCOMM. (1998) 178–189
23. Chandra, S., Ellis, C.S., Vahdat, A.: Differentiated Multimedia Web Services using Quality Aware Transcoding. In: INFOCOM 2000. Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Volume 2., IEEE (2000) 961–969
24. Tarek F. Abdelzaher and Nina Bhatti: Web Server QoS Management by Adaptive Content Delivery. In: Proceedings of the 8th International World Wide Web Conference, Toronto, Canada (1999)
25. Chen, H., Iyengar, A.: A Tiered System for Serving Differentiated Content. In: World Wide Web: Internet and Web Information Systems. Volume 6., Netherlands, Kluwer Academic Publishers (2003) 331–352

Global Computing in a Dynamic Network of Tuple Spaces*

Rocco De Nicola[1], Daniele Gorla[2,**], and Rosario Pugliese[1]

[1] Dipartimento di Sistemi e Informatica, Università di Firenze
[2] Dipartimento di Informatica, Università di Roma "La Sapienza"
{denicola, pugliese}@dsi.unifi.it
gorla@di.uniroma1.it

Abstract. We present a calculus inspired by KLAIM whose main features are: explicit process distribution and node interconnections, remote operations, process mobility and asynchronous communication through distributed tuple spaces. We first introduce a basic language where connections are reliable and immutable; then, we enrich it with two more advanced features for global computing, i.e. *failures* and *dynamically evolving connections*. In each setting, we use our formalisms to specify some non-trivial global computing applications and exploit the semantic theory based on an observational equivalence to equationally establish properties of the considered case-studies.

1 Introduction

Programming computational infrastructures available globally for offering uniform services has become one of the main issues in Computer Science. The challenges come from the necessity of dealing at once with issues like communication, co-operation, mobility, resource usage, security, privacy, failures, etc. in a setting where demands and guarantees can be very different for the many different components. A key issue is the definition of innovative theories, computational paradigms, linguistic mechanisms and implementation techniques for the design, realization, deployment and management of global computational environments and their application.

On the linguistic side, we believe that a language for global computing should be equipped with primitives that support *network awareness* (i.e. locations can be explicitly referenced and operations can be remotely invoked), *disconnected operations* (i.e. code can be moved from one location to the other and remotely executed), *flexible communication mechanisms* (like distributed repositories [11, 8, 15] storing content addressable data), and *remote operations* (like asynchronous remote communications). On the foundational side, the demand is on the development of tools and techniques to

* This work has been partially supported by EU FET - Global Computing initiative project MIKADO IST-2001-32222. The funding bodies are not responsible for any use that might be made of the results presented here.

** Most of this work was carried on while the second author was a PhD student at the University of Florence.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 157–172, 2005.

build safer and trustworthy global systems, to analyse their behaviour, and to demonstrate their conformance to given specifications. Clearly, such semantic theories should reflect all the above listed distinctive features of global systems.

In this paper, we introduce a foundational language that retains the main features of KLAIM [12] (explicit distribution, remote operations, process mobility and asynchronous communication through distributed data spaces), but extends it with new constructs (somehow inspired by [4]) to flexibly model the interconnection structure underlying a network. The resulting formalism, called TKLAIM (*topological* KLAIM), permits to explicitly model inter-node connections and to establish and remove them dynamically. Connections are then essential to enable TKLAIM remote operations: such an operation can be performed only if the node where it is executed and that on which it acts upon are directly connected. Routing algorithms are then needed to enable remote operations between nodes that are not directly connected.

TKLAIM takes its origin from two formalisms with opposite objectives. On one hand, we have the programming language X-KLAIM [3], a full fledged programming language for global computers based on KLAIM; on the other hand, we have the π-calculus [22], the generally recognised minimal common denominator of calculi for mobility. By following well-established techniques for the π-calculus, in a companion paper [13] we formally develop the semantic theory of TKLAIM. Here, we use such a theory to state and prove properties of some meaningful global computing applications. Given the direct correspondence of TKLAIM with X-KLAIM, we believe that the behavioural study carried on at the level of the calculus can be faithfully transposed at the level of the programming language to let programs run in a controlled way on an actual global computer, like the Internet.

To softly introduce the reader to our language, we start in Section 2 by presenting a very basic model where inter-node connections are explicitly programmable but fixed at the outset. This scenario is very close to LANs, where physical connections are reliable and immutable (or change very rarely). We then present two variations of this basic formalism. In Section 3, we enrich the language with different forms of failures, another key feature of global computers. We start with a scenario where only nodes and node components (i.e., data or processes) can fail and use it to establish soundness of a distributed fault-tolerant protocol, the *'k-set agreement'* [10]; then, we briefly present a way to also encompass link failures. The second variation of the basic framework is in Section 4, where links can be dynamically changed by processes. The use of the language with both link failures and dynamic connections is exemplified by programming two routing scenarios and by establishing their soundness.

Section 5 concludes the paper with a discussion on related work. In all the examples, properties of the proposed case-studies are formulated and proved by exploiting *may testing* [14], an intuitive observational equivalence. Our proofs rely on a tractable (bisimulation-based) proof technique whose definition has been omitted from this paper for the sake of space and can be found in [13].

2 The Language

Syntax. The syntax of TKLAIM, given in Table 1, is parameterised with respect to the following syntactic sets, which we assume to be countable and pairwise disjoint: $\mathcal{L}$, of

Table 1. Syntax of TKLAIM

NETS:	COMPONENTS:
$N ::= \mathbf{0} \mid l :: C \mid \{l_1 \leftrightarrow l_2\} \mid (\nu l)N \mid N_1 \| N_2$	$C ::= P \mid \langle t \rangle \mid C_1 \mid C_2$
PROCESSES:	TUPLES:
$P ::= \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid X \mid \mathbf{rec}\, X.P$	$t ::= e \mid \ell \mid t_1, t_2$
ACTIONS:	
$a ::= \mathbf{in}(T)@\ell \mid \mathbf{read}(T)@\ell \mid \mathbf{out}(t)@\ell \mid \mathbf{eval}(P)@\ell \mid \mathbf{new}(l)$	
TEMPLATES:	EXPRESSIONS:
$T ::= e \mid !\, x \mid \ell \mid !\, u \mid T_1, T_2$	$e ::= V \mid x \mid \ldots$

localities, ranged over by l; $\mathcal{U}$, of *locality variables*, ranged over by u; $\mathcal{V}$, of *basic values*, ranged over by V; $\mathcal{Z}$, of *basic variables*, ranged over by x; $\mathcal{X}$, of *process variables*, ranged over by X. Finally, ℓ is used to denote elements of $\mathcal{L} \cup \mathcal{U}$.

The exact syntax of *expressions*, e, is deliberately omitted; we just assume that expressions contain, at least, basic values and variables. *Localities*, l, are the addresses (i.e. network references) of nodes. *Tuples*, t, are sequences of expressions, localities or locality variables. *Templates*, T, are used to select tuples: in particular, $!\, x$ and $!\, u$, that we call *formal fields*, are used to bind variables to values.

Processes, ranged over by $P, Q, R, \ldots$, are the TKLAIM active computational units and may be executed concurrently either at the same locality or at different localities. They are built up from the terminated process **nil** and from the basic actions by using prefixing, parallel composition and recursion. *Actions* permit removing/accessing/adding tuples from/to tuple spaces (actions **in**/**read**/**out**, resp.), activating new threads of execution (action **eval**) and creating new nodes (action **new**). Action **new** is not indexed with an address because it always acts locally; all the other actions explicitly indicate the (possibly remote) locality where they will take effect.

Nets, ranged over by $N, M, \ldots$, are finite collections of nodes and inter-node connections. A *node* is a pair $l :: C$, where locality l is the address of the node and C is the (parallel) component located at l. *Components*, ranged over by $C, D, \ldots$, can be either processes or data, denoted by $\langle t \rangle$. *Connections*, or *links*, are pairs of node addresses $\{l_1 \leftrightarrow l_2\}$ stating that the nodes with address l_1 and l_2 are directly (and bidirectionally[1]) linked. In $(\nu l)N$, name l is private to N; the intended effect is that, if one considers the term $M \| (\nu l)N$, then locality l of N cannot be referred from within M.

Names (i.e. localities and variables) occurring in TKLAIM processes and nets can be *bound*. More precisely, prefixes $\mathbf{in}(T)@\ell.P$ and $\mathbf{read}(T)@\ell.P$ bind T's formal fields in P; prefix $\mathbf{new}(l).P$ binds l in P, and, similarly, net restriction $(\nu l)N$ binds l in N; finally, $\mathbf{rec}\, X.P$ binds X in P. A name that is not bound is called *free*. The sets $fn(\cdot)$ and $bn(\cdot)$ (respectively, of free and bound names of a term) are defined accordingly. The set $n(\cdot)$ of names of a term is the union of its sets of free and bound names. We say that two

[1] For the sake of simplicity, we assumed bidirectional links; nevertheless, all the theory and the examples we develop here could be tailored to the framework where links are directed.

terms are *alpha-equivalent* if one can be obtained from the other by renaming bound names. In the sequel, we shall work with terms whose bound names are all distinct and different from the free ones. Moreover, as usual, we shall only consider *closed* terms, i.e. processes and nets without free variables.

Notation 1. We write $A \triangleq W$ to mean that A is of the form W; this notation is used to assign a symbolic name A to the term W. We shall use notation $\widetilde{\cdot}$ to denote sets of objects (e.g. $\widetilde{l}$ is a set of names). We shall sometimes write $\mathbf{in}()@l$, $\mathbf{out}()@l$ and $\langle\rangle$ to mean that the argument of the actions or the datum are irrelevant. Finally, we shall omit trailing occurrences of process **nil** and write $\Pi_{j\in J}\ W_j$ for the parallel composition (both '|' and '||') of terms (components or nets, resp.) W_j.

Operational Semantics. tKlaim operational semantics is given in terms of a structural congruence and a reduction relation. The *structural congruence*, $\equiv$, identifies nets which intuitively represent the same net. It is inspired to π-calculus's structural congruence (see, e.g., [25]) and includes laws stating that '||' is commutative, associative and has $\mathbf{0}$ as identity element, laws equating alpha-equivalent nets, laws regulating commutativity of restrictions, and laws allowing to freely fold/unfold recursive processes. Moreover, the following laws are crucial to our setting:

(Clone)	(Self)	(BiDir)
$l :: C_1 \vert C_2 \ \equiv\ l :: C_1 \parallel l :: C_2$	$l :: \mathbf{nil} \ \equiv\ \{l \leftrightarrow l\}$	$\{l_1 \leftrightarrow l_2\} \ \equiv\ \{l_2 \leftrightarrow l_1\}$

(RNode)	(Ext)
$(\nu l)N \equiv (\nu l)(N \parallel l :: \mathbf{nil})$	$N \parallel (\nu l)M \equiv (\nu l)(N \parallel M)$ if $l \notin fn(N)$

Law (Clone) turns a parallel between co-located components into a parallel between nodes; law (Self) states that nodes are self-connected; law (BiDir) states that links are bidirectional; law (Ext) is the standard π-calculus' rule for scope extension. Finally, law (RNode) states that any restricted name can be used as the address of a node; indeed, we consider restricted names as private network addresses, whose corresponding nodes can be activated and deactivated when needed. By relying on rule (RNode), we shall only consider nets where each bound name is associated to a node.

The reduction relation is given in Table 2 and relies on two auxiliary functions: $\mathcal{E}[\![\ \_\ ]\!]$ and $match(\_\,;\_)$. The *tuple/template evaluation* function, $\mathcal{E}[\![\ \_\ ]\!]$, evaluates componentwise the expressions occurring within the tuple/template $\_$; its definition is simple and, thus, omitted. The *pattern matching* function, $match(\_\,;\_)$, verifies the compliance of a tuple w.r.t. a template and associates values to variables bound in the template. Intuitively, a tuple matches a template if they have the same number of fields, and corresponding fields match. Formally, function *match* is defined as

$$match(l; l) = \epsilon \qquad match(!u; l) = [^{l}/_{u}] \qquad \frac{match(T_1; t_1) = \sigma_1 \quad match(T_2; t_2) = \sigma_2}{match(T_1, T_2; t_1, t_2) = \sigma_1 \circ \sigma_2}$$

$$match(V; V) = \epsilon \qquad match(!x; V) = [^{V}/_{x}]$$

where we let 'ϵ' be the empty substitution and '$\circ$' denote substitutions composition. Here, a substitution σ is a mapping of localities and basic values for variables; $P\sigma$ denotes the (capture avoiding) application of σ to P.

Table 2. tKlaim Reduction Relation

$$\text{(R-Out)}\quad \frac{\mathcal{E}[\![\, t \,]\!] = t'}{l :: \mathbf{out}(t)@l'.P \parallel \{l \leftrightarrow l'\} \;\longmapsto\; l :: P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t' \rangle}$$

$$\text{(R-Eval)}\quad l :: \mathbf{eval}(P_2)@l'.P_1 \parallel \{l \leftrightarrow l'\} \;\longmapsto\; l :: P_1 \parallel \{l \leftrightarrow l'\} \parallel l' :: P_2$$

$$\text{(R-In)}\quad \frac{match(\mathcal{E}[\![\, T \,]\!]; t) = \sigma}{l :: \mathbf{in}(T)@l'.P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle \;\longmapsto\; l :: P\sigma \parallel \{l \leftrightarrow l'\}}$$

$$\text{(R-Read)}\quad \frac{match(\mathcal{E}[\![\, T \,]\!]; t) = \sigma}{l :: \mathbf{read}(T)@l'.P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle \;\longmapsto\; l :: P\sigma \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle}$$

$$\text{(R-New)}\quad l :: \mathbf{new}(l').P \;\longmapsto\; (\nu l')(l :: P \parallel \{l \leftrightarrow l'\})$$

$$\text{(R-Par)}\quad \frac{N_1 \;\longmapsto\; N_1'}{N_1 \parallel N_2 \;\longmapsto\; N_1' \parallel N_2} \qquad \text{(R-Res)}\quad \frac{N \;\longmapsto\; N'}{(\nu l)N \;\longmapsto\; (\nu l)N'}$$

$$\text{(R-Struct)}\quad \frac{N \equiv M \quad M \longmapsto M' \quad M' \equiv N'}{N \longmapsto N'}$$

The operational rules of tKlaim can be briefly motivated as follows. Rule (R-Out) evaluates the expressions within the argument tuple and sends the resulting tuple to the target node. However, this is possible only if the source and the target nodes are directly connected. Rule (R-Eval) is similar: a process can be spawned at l' by a process running at l only if l and l' are directly connected. Rules (R-In) and (R-Read) require existence of a matching datum in the target node and of a connection between the source and the target node. The tuple is then used to replace the free occurrences of the variables bound by the template in the continuation of the process performing the actions. With action **in** the matched datum is consumed while with action **read** it is not. Rule (R-New) says that execution of action **new**(l') adds a restriction over l' to the net and a link between the creating node l and the created node l'; from then on, a new node with locality l' can be activated/deactivated by using law (RNode). Rules (R-Par), (R-Res) and (R-Struct) are standard.

tKlaim adopts a Linda-like [18] communication mechanism: data are anonymous and associatively accessed via pattern matching, and communication is asynchronous. Indeed, even if there exist prefixes for placing data to (possibly remote) nodes, no synchronization takes place between (sending and receiving) processes, because their interactions are mediated by nodes, that act as data repositories.

To conclude the presentation of tKlaim operational semantics, we want to stress that interactions between directly linked nodes can be used to permit interactions between nodes that are not directly linked. However, as it happens in practice, this feature must be explicitly programmed. If a process running at l wants to send a tuple t to l' and there

exists a path of links from l to l' in the underlying connection graph, the one needs a mobile process be spawned by l 'towards' l' that delivers tuple t. The main challenges of such a process is to discover the shortest (or, at least, a possible) path connecting l and l'. This functionality can be accomplished by relying on *routing tables*, i.e. distributed data structures that record information on routing paths. We leave the formal specification of this process, together with a proof of its soundness, for a forthcoming full paper.

Observational Semantics. We now present a preorder on tKLAIM nets yielding sensible semantic theories. We follow the approach put forward in [14] and use *may testing* preorder and the associated equivalence. Intuitively, two nets are may testing equivalent if they cannot be distinguished by any external observer taking note of the data offered by the observed net. More precisely, an *observer* O is a net containing a node whose address is a reserved locality name `test`. A computation reports *success* if, along its execution, a datum at node `test` appears; this is written $\stackrel{OK}{\Longrightarrow}$.

Definition 2 (May Testing Preorder and Equivalence). May testing preorder, $\sqsubseteq$, *is the least preorder on* tKLAIM *nets such that, for every* $N \sqsubseteq M$, *it holds that* $N \parallel O \stackrel{OK}{\Longrightarrow}$ *implies* $M \parallel O \stackrel{OK}{\Longrightarrow}$, *for any observer* O.

May testing equivalence, $\simeq$, *is defined as the intersection of* $\sqsubseteq$ *and* $\sqsupseteq$.

To conclude, we give a simple Proposition collecting a few equational laws that will simplify the proofs of the case-studies considered in this paper. Soundness of such laws can be easily established by exploiting the co-inductive (bisimulation-based) proof technique provided in [13]. Indeed, directly establishing may-testing may be very hard because of the universal quantification over contexts.

Proposition 1.

1. $l :: \mathbf{out}(t)@l'.P \parallel \{l \leftrightarrow l'\} \simeq l :: P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle$
2. $l :: \mathbf{eval}(Q)@l'.P \parallel \{l \leftrightarrow l'\} \simeq l :: P \parallel \{l \leftrightarrow l'\} \parallel l' :: Q$
3. $(\nu l')(l :: \mathbf{in}(T)@l'.P \parallel l' :: \langle t \rangle) \simeq (\nu l')(l :: P\sigma)$ if $match(\mathcal{E}[\![T]\!]; t) = \sigma$
4. $(\nu l)(l :: C) \simeq \mathbf{0}$ whenever C is a datum $\langle t \rangle$, a stuck process **nil** or the parallel composition of such components
5. $l :: \mathbf{new}(l').P \simeq (\nu l')(l :: P \parallel \{l \leftrightarrow l'\})$
6. $(\nu l')\{l \leftrightarrow l'\} \simeq l :: \mathbf{nil}$

3 Modelling Failures

We now enrich the basic framework with a mechanism for modelling various forms of failures, a key feature of global computers. We start by modelling failure of nodes and of node components; then, we use the new setting to prove some properties of a distributed fault-tolerant protocol. Finally, we sketch a minor modification of our framework to take into account link failures.

3.1 Failure of Nodes and Node Components

We start by modelling a framework where only nodes and node components fail. This can be achieved by adding the operational rule

$$\text{(R-FailN)} \qquad l :: C \longmapsto \mathbf{0}$$

that models corruption of data (*message omission*) if $C \triangleq \langle t_1 \rangle | \ldots | \langle t_n \rangle$, node (*fail-silent*) failure if $l :: C$ collects all the clones of l, and abnormal termination of some processes running at l otherwise. Modelling failures as disappearance of a resource (a datum, a process or a whole node) is a simple, but realistic, way of representing failures in a global computing scenario [6]. Indeed, while the presence of data/nodes can be ascertained, their absence cannot because there is no practical upper bound to communication delays. Thus, failures cannot be distinguished from long delays and should be modelled as totally asynchronous and undetectable events.

For the sake of clarity, we shall denote with $\sqsubseteq_f$ and $\simeq_f$ the may testing preorder and equivalence obtained when adding rule (R-FailN) to the rules in Table 2.

A Distributed Fault-tolerant Protocol: *k*–set Agreement. We now use may testing to verify the correctness of *k–set agreement* [10], a simple distributed fault-tolerant protocol. Let us consider a totally-connected distributed system with n principals relying on an asynchronous message-passing communication paradigm. Moreover, we also assume that principals can fail according to a fail-silent model of failures; however, the communication medium is reliable, i.e. messages sent will surely be received although the order and the moment in which messages will arrive are unpredictable because of asynchrony.

Each principal has an input value (taken from a totally ordered set) and must produce an output value. The *agreement* problem requires to find a protocol that satisfies three properties: *termination* (i.e. the non-faulty principals eventually produce an output), *agreement* (i.e. the non-faulty principals produce the *same* output value) and *validity* (i.e. the output value must be one of the input values). It is well-known (see, e.g. [2]) that a solution for this problem does not exists even if a single failure occurs.

The *k–set agreement* problem relaxes the agreement property to enable the existence of a solution. Indeed, for each $1 \leq k \leq n$, it requires that, assuming at most $k-1$ faulty principals, the non-failed principals successfully complete their execution by producing outputs taken from a set whose size is at most k. Notice that for $k = 1$ we get the agreement problem without failures.

A possible solution for the k-set agreement problem is given by the following protocol, taken from [2], executed by each principal:

(i) send your input value to all principals (including yourself)
(ii) wait to receive $n - k + 1$ values
(iii) output the minimum value received

In this way, if we call $\mathcal{I}$ the set of the input values, the set of output values $\mathcal{O}$ is formed by the k smallest values in $\mathcal{I}$. For the sake of simplicity, we assume that the elements in $\mathcal{I}$ are pairwise distinct; however, the protocol works also if input values are duplicated (in this case $\mathcal{I}$ and $\mathcal{O}$ are multisets).

We use integers as input/output values, while principals are represented as distinct nodes, whose addresses are taken from the set $L = \widetilde{l} \triangleq \{l_1, \ldots, l_n\}$; moreover, we use $d_i \in \mathcal{I}$ to denote the input value of the principal associated to the node whose address is l_i. Once we fix the value for k, node l_i hosts the process

$$P_i^k \triangleq \mathbf{out}(d_i)@l_1. \ldots .\mathbf{out}(d_i)@l_n.\mathbf{in}(!z_1^i)@l_i. \ldots .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}(m_i)@l$$

with $m_i \triangleq min\{z^i_j : j = 1, \ldots, n-k+1\}$ and l be a distinct locality used to collect output values. The net implementing the whole protocol is

$$N^k_n \;\triangleq\; (\nu \widetilde{l})\Big(\prod_{i=1}^{n} \; l_i :: P^k_i \parallel \Gamma \Big)$$

where

$$\Gamma \;\triangleq \prod_{i \neq j} \{l_i \leftrightarrow l_j\}$$

We restricted the localities associated to the principals because no external context is allowed to interfere with the execution of the protocol. Notice that, having restricted the $\widetilde{l}$, all the principals are connected and no **out** prefix will ever block P^k_i (because of law (RNode)). However, this does not prevent failures: the failure of (a reduct of) P^k_i is indeed the failure of principal i.

A formulation of the three properties for the k-set agreement problem is given by Equations (1) and (2) below. The formalisation of k-set agreement and validity properties is given by the Equation

$$N^k_n \;\simeq_f\; M^k_n \tag{1}$$

There, we exploit the auxiliary net

$$M^k_n \;\triangleq\; (\nu \widetilde{l}, \widetilde{l'})\Big(\prod_{i=1}^{n} \; l_i :: \mathbf{new}(l'_i).(Q^k_i \mid \prod_{w \in O} \mathbf{out}(w)@l'_i) \;\parallel\; \Gamma \Big)$$

where

$$Q^k_i \triangleq \mathbf{out}(d_i)@l_1.\cdots.\mathbf{out}(d_i)@l_n.\mathbf{in}(!z^i_1)@l_i.\cdots.\mathbf{in}(!z^i_{n-k+1})@l_i.\mathbf{in}(m_i)@l'_i.\mathbf{out}(m_i)@l$$

We assume that nodes whose addresses are in $\widetilde{l'}$ cannot fail; this is reasonable because they are only auxiliary nodes and hence their failure is irrelevant for the original formulation of the problem. Intuitively, node l'_i acts as a repository for l_i and contains the possible output values (i.e. the elements of O), while the last **in** action of Q^k_i is a test for checking that the output value produced by the principal i is in O. The net M^k_n obviously satisfies the wanted properties since its principals output only values present in O. The fact that $|O| = k$ then implies the k-set agreement property, while the fact that $O \subseteq \mathcal{I}$ implies validity.

In order to prove the termination property, it suffices to prove that

$$l :: \prod_{j=1}^{n-k+1} \langle\rangle \quad \sqsubseteq_f \quad \hat{N}^k_n \tag{2}$$

where $\hat{N}^k_n \;\triangleq\; (\nu \widetilde{l})\Big(\prod_{i=1}^{n} \; (l_i :: \hat{P}^k_i \parallel \{l_i \leftrightarrow l\}) \parallel \Gamma \Big)$ and processes $\hat{P}^k_i$ is defined like P^k_i with action $\mathbf{out}()@l$ in place of $\mathbf{out}(m_i)@l$. Clearly, if we only consider termination, N^k_n and $\hat{N}^k_n$ are equivalent, in the sense that a non-faulty principal produces an output value in the first net if and only if its counterpart produces an output in the second net. Equation (2) implies termination of the protocol, since it requires that at least $n-k+1$ tuples are produced at l; by definition of the protocol, this is possible only if $n-k+1$ principals terminate successfully.

Before proving the equations stating the soundness of the protocol, we want to remark that other solutions to the agreement problem in presence of failures have been given in literature. Some of these solutions use *failure detectors* [9, 2]. Recently, one such solution has been formalised and proved sound by using a process algebraic approach [17]. The solution in *loc.cit.* is, however, heavier than ours and exploits properties of the operational semantics, instead of working in a (simpler) equational setting. Moreover, it exploits failure detectors which are hardly implementable in a global computing scenario.

Proof of Equations (1) and (2). To prove the properties formulated above, we first need a new equality

$$l :: I_1 | \ldots | I_n \ \sqsubseteq_f \ l :: I_1 | \ldots | I_m \quad \text{if } n \leq m \qquad (\dagger)$$

Second, we need to smoothly adapt some of the equalities put forward by Proposition 1: the first equality holds only under the hypothesis that l' is restricted, while the third equality holds only under the further hypothesis that $\langle t \rangle$ is not corruptible at l' (with "$\langle t \rangle$ is not corruptible at l' ", we mean that $l' :: \langle t \rangle$ does never fail). Then, we prove Equation (1) as follows:

$$\begin{aligned}
N_n^k &\simeq_f (\nu \widetilde{l})\Big(\prod_{i=1}^{n} l_i :: \mathbf{in}(!z_1^i)@l_i. \ \ldots \ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}(m_i)@l \,|\, \langle d_1 \rangle \,|\, \ldots \,|\, \langle d_n \rangle \parallel \Gamma \Big) \\
&\simeq_f (\nu \widetilde{l})\Big(\prod_{i=1}^{n} l_i :: \mathbf{out}(m_i')@l \,|\, \langle d_{i_1} \rangle \,|\, \ldots \,|\, \langle d_{i_{k-1}} \rangle \parallel \Gamma \Big) \\
&\simeq_f (\nu \widetilde{l}, \widetilde{l'})\Big(\prod_{i=1}^{n} (l_i :: \mathbf{in}(m_i')@l_i'.\mathbf{out}(m_i')@l \,|\, \langle d_{i_1} \rangle \,|\, \ldots \,|\, \langle d_{i_{k-1}} \rangle \\
&\qquad\qquad \parallel \ l_i' :: \prod_{w \in O} \langle w \rangle \parallel \{l_i \leftrightarrow l_i'\}) \ \parallel \Gamma \Big) \\
&\simeq_f (\nu \widetilde{l}, \widetilde{l'})\Big(\prod_{i=1}^{n} (l_i :: \mathbf{in}(!z_1^i)@l_i. \ \ldots \ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{in}(m_i)@l_i'.\mathbf{out}(m_i)@l \\
&\qquad\qquad \,|\, \langle d_1 \rangle \,|\, \ldots \,|\, \langle d_n \rangle \ \parallel \ l_i' :: \prod_{w \in O} \langle w \rangle \parallel \{l_i \leftrightarrow l_i'\}) \ \parallel \Gamma \Big) \\
&\simeq_f M_n^k
\end{aligned}$$

where m_i' denotes $m_i[\widetilde{d}/\widetilde{z}]$, with $\widetilde{d} \triangleq \{d_1, \ldots, d_n\} - \{d_{i_1}, \ldots, d_{i_{k-1}}\}$ and $\widetilde{z} \triangleq \{z_1, \ldots, z_{n-k+1}\}$. The first and the last steps have been inferred by applying several times (the revised formulation of) Proposition 1.1 . The second and the fourth steps have been inferred by applying several times (the revised formulation of) Proposition 1.3; notice that, since the number of failures is at most $k-1$, the number of non-corruptible data present in each l_i is at least $n-k+1$. The third step relies on Proposition 1.3, .4 and .6 . It is worth to notice that $m_i' \in O$ because, since $|O| = k$, at least one principal whose input value, say d', is in O has not failed; hence d' has been received by all the (non-failed) principals. Moreover, we assumed that the $\widetilde{l'}$ cannot fail and hence the data they host are uncorruptable.

To conclude, we are left with proving Equation (2). This can be done very similarly as follows:

$$\begin{aligned}
\hat{N}_n^k &\simeq_f (\nu \widetilde{l})\Big(\prod_{i=1}^{n} l_i :: \mathbf{in}(!z_1^i)@l_i. \ \ldots \ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}()@l \,|\, \langle d_1 \rangle \,|\, \ldots \,|\, \langle d_n \rangle \parallel \{l_i \leftrightarrow l\}\Big) \\
&\simeq_f (\nu \widetilde{l})\Big(\prod_{i=1}^{n} l_i :: \mathbf{out}()@l \,|\, \langle d_{i_1} \rangle \,|\, \ldots \,|\, \langle d_{i_{k-1}} \rangle \parallel \{l_i \leftrightarrow l\}\Big)
\end{aligned}$$

$$\simeq_f l :: \prod_{j=1}^{n} \langle\rangle$$
$$\sqsupseteq_f l :: \prod_{j=1}^{n-k+1} \langle\rangle$$

The first two steps are derived in the same way. The third step is derived using (the revised version of) Proposition 1.1/.4 and Proposition 1.6 . The fourth step derives from law (†).

3.2 Failure of Inter-node Connections

The philosophy underlying our failure model can be easily adapted to deal with link failures too. To this aim, we only need to add the operational rule

$$\text{(R-FAILC)} \quad \{l_1 \leftrightarrow l_2\} \longmapsto \mathbf{0}$$

that models the (asynchronous and undetectable) failure of the link between nodes l_1 and l_2.

Discovering Neighbours. When the (multi)set of links in a net can change during computations, routing tables must be dynamic, because the original topology can change at runtime. This task is usually carried on by the so-called *adaptive* (or *dynamic*) routing algorithms. Several proposals have been presented in literature and different standards use different solutions. However, in general, routing algorithms are repeated at regular time intervals and consist in two main phases: first, each node discovers its neighbours; then, it calculates its routing table by usually sharing local information with its neighbours. We present here a simple way to implement in TKLAIM the first phase; the (more challenging) study of the second phase is left for future work.

Neighbours can be discovered in a simple way. Each node l can try to send a "hello" message to another node l'; if this action succeeds, then a connection between l and l' does exist; otherwise, nothing can be said (e.g., the message could get lost or the link could be congested and this caused a delay to the message). In our framework, no explicit message is needed: a simple action $\mathbf{eval}(\mathbf{nil})@l'$ performed at l can be used as test for existence of link $\{l \leftrightarrow l'\}$ in the net.

By letting $\sqsubseteq_f$ still denote the may testing preorder in this refined framework, soundness of our solution follows by proving that

$$l :: \mathbf{eval}(\mathbf{nil})@l'.\mathbf{out}(\text{“}CONN\text{”}, l, l')@l \quad \sqsubseteq_f \quad \{l \leftrightarrow l'\} \parallel l :: \langle \text{“}CONN\text{”}, l, l' \rangle$$

The equation above states that if the left hand side successfully passes the test of an observer looking for a tuple $\langle$“$CONN$”$, l, l'\rangle$ at l, then the link $\{l \leftrightarrow l'\}$ must exist. Its soundness can be easily proved by exploiting the co-inductive proof technique in [13].

4 Modelling Dynamic Connections

Finally, we present another variation of the basic language that let connections dynamically evolve. To this aim, we add two actions to create and destroy a link, respectively. Formally, we add the productions

$$a \ ::= \ \ldots \ \mid \ \mathbf{conn}(\ell) \ \mid \ \mathbf{disc}(\ell)$$

to the syntax of Table 1. Intuitively, the first action, when executed at node l, creates a new link between l and ℓ, if the latter name is associated to a network node. Conversely, the second action, when executed at node l, removes a link between l and ℓ, if such a link exists. These intuitions are formalised by the following operational rules, that must be added to those in Table 2:

$$\text{(R-Conn)} \quad l :: \mathbf{conn}(l').P \parallel l' :: \mathbf{nil} \longmapsto l :: P \parallel \{l \leftrightarrow l'\}$$

$$\text{(R-Disc)} \quad l :: \mathbf{disc}(l').P \parallel \{l \leftrightarrow l'\} \longmapsto l :: P \parallel l' :: \mathbf{nil}$$

Again, for the sake of clarity, we denote with $\simeq_d$ the may testing equivalence in the calculus with dynamic connections.

Message Delivering in a Dynamic Net. To conclude, we now give an application of our theory in a setting where node links change dynamically. To this aim, we use a simplified scenario inspired by the *handover protocol*, proposed by the European Telecommunication Standards Institute (ETSI) for the GSM Public Land Mobile Network (PLMN). The formal specification of the protocol and its service specification are in [24]; we use here an adaption of their approach.

The PLMN is a cellular system which consists of Mobile Stations (MSs), Base Stations (BSs) and Mobile Switching Centres (MSCs). MSs are mobile devices that provide services to end users. BSs manage the interface between the MSs and a stationary net; they control the communications within a geographical area (a cell). Any MSC handles a set of BSs; it communicates with them and with other MSCs using a stationary net.

A new user can enter the system by connecting its MS with a MSC that, in turn, will decide the proper BS responsible for such a MS. Then, messages sent from the user are routed to their destinations by the BS, passing through the MSC handling the BS. However, it may happen that the BS responsible for a MS should be changed during the computation (e.g., because the MS left the area associated to the BS and entered in the area associated to a different BS). In this case, the MSC should carry on the rearrangements needed to cope with the new situation, without affecting the end-to-end communication.

We now model the key features of a PLMN in TKlaim; however, for the sake of simplicity, several aspects will be omitted, like, e.g., the criterion to choose a proper BS for a given MS, or the event originating an handover. Both MSs, BSs and MSCs are modelled as nodes. For the sake of simplicity, we consider a very simple PLMN, with one MSC (whose address is $\mathtt{M}$) and two BSs (whose addresses are $\mathtt{B}_1$ and $\mathtt{B}_2$, resp.).

Let us start with the process that performs the connecting formalities in $\mathtt{M}$.

$$\begin{aligned} ENTER \triangleq \ & < \textit{gather a new connection from } l > .\mathbf{read}(!B)@\mathtt{BSlist}. \\ & \mathbf{eval}(\mathbf{conn}(l))@B.\mathbf{disc}(l).\mathbf{out}(l, B)@\mathtt{Table} \end{aligned}$$

When a new user want to enter the PLMN, it has to perform a $\mathbf{conn}(\mathtt{M})$ from his MS, whose address is l; this generates an interrupt in $\mathtt{M}$ (that we do not model here) by which the MSC can gather the address of the MS. This address, together with other information (like the geographical area of the user or its credentials), are used by the

MSC to choose a proper BS; in our simplified framework, we let $\mathtt{M}$ take a BS's address from a private repository $\mathtt{BSlist}$. Then, the MSC creates a new link from the chosen BS to the MS and destroys the link from itself to the MS. Finally, it records in a private repository $\mathtt{Table}$ the fact that the new MS is under the control of the chosen BS.

Once entered the PLMN, the new user can send some data d to (the MS of) a remote user (whose address is l'); this is achieved by letting his MS (whose address is l) perform an action of the form $\mathbf{out}(\text{`send'}, l', d)@l$. Then, the BSs associated to l and l' come into the picture to properly deliver the message. In particular, let $\mathtt{B}_i$ be the BS associated to l and $\mathtt{B}_j$ be the BS associated to l' (for $i, j \in \{1, 2\}$). Then, the message is forwarded from $\mathtt{B}_i$ to $\mathtt{B}_j$ by the process

$$FWD_i \triangleq \mathbf{read}(!x, \mathtt{B}_i)@\mathtt{Table}.\mathbf{in}(\text{`send'}, !y, !z)@x.\mathbf{in}(y, !B)@\mathtt{Table}.\mathbf{out}(y, z)@B$$

This process first retrieves the address of a MS associated to $\mathtt{B}_i$ (in particular, l); then, it collects the message and forwards it to the BS associated to the destination MS. Notice that, in doing this, it 'locks' the link between l' and $\mathtt{B}_j$ until the message will be delivered to l' (see below); this is necessary to avoid that a handover may interfere with the message delivering. Then, the message is collected by $\mathtt{B}_j$ and passed to l' by the process

$$CLT_j \triangleq \mathbf{in}(!dest, !mess)@\mathtt{B}_j.\mathbf{out}(mess)@dest.\mathbf{out}(dest, \mathtt{B}_j)@\mathtt{Table}$$

This process retrieves the message sent by $\mathtt{B}_i$ and passes it to the final MS; then, it releases the 'lock' on the link $\{\mathtt{B}_j \leftrightarrow l'\}$ acquired by $\mathtt{B}_i$ by putting back in $\mathtt{Table}$ the tuple $\langle l', \mathtt{B}_j \rangle$. Clearly, there are also processes FWD_j and CLT_i running in $\mathtt{B}_j$ and $\mathtt{B}_i$ respectively, but they do not play any role here.

Finally, the handover is handled by the MSC via the following process:

$$\begin{aligned} HNDVR \triangleq\ & \mathbf{in}(!x, !B)@\mathtt{Table}.\mathbf{read}(!B')@\mathtt{BSlist}. \\ & \mathbf{eval}(\mathbf{disc}(x))@B.\mathbf{eval}(\mathbf{conn}(x))@B'.\mathbf{out}(x, B')@\mathtt{Table} \end{aligned}$$

This process first selects a MS-to-BS association to be changed (the reason why this is needed is not modelled here); then, it chooses a new BS, properly changes the links between the MS and the BSs, and updates the repository $\mathtt{Table}$.

The overall resulting system is

$$\begin{aligned} SYS \triangleq\ & (\nu \mathtt{Table}, \mathtt{BSlist}, \mathtt{B}_1, \mathtt{B}_2)(\mathtt{M} :: *ENTER \mid *HNDVR \\ & \parallel \mathtt{BSlist} :: \langle \mathtt{B}_1 \rangle \mid \langle \mathtt{B}_2 \rangle \parallel \mathtt{Table} :: \mathbf{nil} \\ & \parallel \mathtt{B}_1 :: *FWD_1 \mid *CLT_1 \parallel \mathtt{B}_2 :: *FWD_2 \mid *CLT_2) \end{aligned}$$

where $*P$ denotes the replication of P and stands for an unbounded number of copies of P running in parallel. Replication can be easily encoded through recursion by letting $*P$ be a shortcut for $\mathbf{rec}\, X.(P|X)$. Soundness of the system can be formulated as:

$$\begin{aligned} & (\nu l)(l :: \mathbf{conn}(\mathtt{M}).\mathbf{out}(\text{`send'}, l', \text{`HI'})@l \parallel l' :: \mathbf{conn}(\mathtt{M}) \parallel SYS) \\ & \quad \simeq_d \ (\nu l)(l' :: \langle \text{`HI'} \rangle \parallel SYS) \end{aligned} \tag{3}$$

Notice that l is restricted only to simplify proofs: soundness of the protocol is not affected by the fact that the MSs are public or not.

Proof of Equation (3). To prove the equation above, we first need two laws for the primitives **conn** and **disc**, that are quite expectable.

$$l :: \mathbf{conn}(l').P \parallel l' :: \mathbf{nil} \;\simeq_d\; l :: P \parallel \{l \leftrightarrow l'\} \qquad (\star)$$

$$(\nu l')(l :: \mathbf{disc}(l').P \parallel \{l \leftrightarrow l'\}) \;\simeq_d\; (\nu l')(l :: P \parallel l' :: \mathbf{nil}) \qquad (\star\star)$$

Moreover, we also need an adapted version of Proposition 1.3 to deal with action **read**. It is defined as follows:

$$(\nu l')(\, l :: \mathbf{read}(T)@l'.P \parallel l' :: \langle t \rangle\,) \;\simeq_d\; (\nu l')(l :: P\sigma \parallel l' :: \langle t \rangle) \quad \text{if } \mathit{match}(\mathcal{E}[\![\, T \,]\!], t) = \sigma. \qquad (\ddagger)$$

We are ready to prove Equation (3), yielding the soundness of the protocol for the PLMN. It is easy to prove that

$$(\nu l)(l :: \mathbf{conn}(\mathtt{M}).\mathbf{out}(\text{`send'}, l', \text{`HI'})@l \parallel l' :: \mathbf{conn}(\mathtt{M}) \parallel SYS\,)$$

$$\begin{aligned}
\simeq_d\;\; & (\nu l, \mathtt{Table}, \mathtt{BSlist}, \mathtt{B}_1, \mathtt{B}_2)(\; l :: \langle \text{`send'}, l', \text{`HI'} \rangle \;\parallel\; l' :: \mathbf{nil} \\
& \quad \parallel\; \mathtt{M} :: *\mathit{ENTER} \mid *\mathit{HNDVR} \;\parallel\; \mathtt{BSlist} :: \langle \mathtt{B}_1 \rangle \mid \langle \mathtt{B}_2 \rangle \\
& \quad \parallel\; \mathtt{Table} :: \langle l, \mathtt{B}_i \rangle \mid \langle l', \mathtt{B}_j \rangle \;\parallel\; \{l \leftrightarrow \mathtt{B}_i\} \;\parallel\; \{l' \leftrightarrow \mathtt{B}_j\} \\
& \quad \parallel\; \mathtt{B}_1 :: *\mathit{FWD}_1 \mid *\mathit{CLT}_1 \;\parallel\; \mathtt{B}_2 :: *\mathit{FWD}_2 \mid *\mathit{CLT}_2\;) \\
\simeq_d\;\; & (\mathit{nul}, \mathtt{Table}, \mathtt{BSlist}, \mathtt{B}_1, \mathtt{B}_2)(\; l :: \mathbf{nil} \;\parallel\; l' :: \mathbf{nil} \\
& \quad \parallel\; \mathtt{M} :: *\mathit{ENTER} \mid *\mathit{HNDVR} \;\parallel\; \mathtt{BSlist} :: \langle \mathtt{B}_1 \rangle \mid \langle \mathtt{B}_2 \rangle \\
& \quad \parallel\; \mathtt{Table} :: \langle l, \mathtt{B}_i \rangle \mid \langle l', \mathtt{B}_j \rangle \;\parallel\; \{l \leftrightarrow \mathtt{B}_i\} \;\parallel\; \{l' \leftrightarrow \mathtt{B}_j\} \\
& \quad \parallel\; \mathtt{B}_1 :: *\mathit{FWD}_1 \mid *\mathit{CLT}_1 \;\parallel\; \mathtt{B}_2 :: *\mathit{FWD}_2 \mid *\mathit{CLT}_2 \\
& \quad \parallel\; \mathtt{B}_i :: \mathbf{in}(l', !B)@\mathtt{Table}.\mathbf{out}(l', \text{`HI'})@B\;) \\
\triangleq\;\; & K
\end{aligned}$$

The first equality can be inferred using laws ($\star$) and ($\ddagger$), Proposition 1.2, laws ($\star$) and ($\star\star$), and Proposition 1.1; the second equality can be inferred using law ($\ddagger$) and Proposition 1.3. Now we cannot proceed equationally: indeed, there are two parallel components that may want to retrieve the tuple $\langle l', \mathtt{B}_j \rangle$ at `Table`, i.e. the process $\mathbf{in}(l', !B)@\mathtt{Table}.\mathbf{out}(l', \text{`HI'})@B$ running at $\mathtt{B}_i$ and the process $\mathit{HNDVR}$ running at `M`. This fact makes Proposition 1.3 not applicable here.

To overcome this problem, we observe that there are only three possible evolutions for K: make a handover for l, make a handover for l', or complete the delivering of the message that l sent to l'. The first evolution is compatible with the latter two ones that, in turn, are mutually exclusive. Thus, let $\mathcal{H}$ be the set of pairs $(N, (\nu l)(l' :: \langle \text{`HI'} \rangle \parallel SYS))$, where $N$ is any reduct of $K$ obtained by giving the precedence to the handover of $l'$ w.r.t. the message delivering. Symmetrically, let $\mathcal{D}$ be the set of pairs $(N, (\nu l)(l' :: \langle \text{`HI'} \rangle \parallel SYS))$, where N is any reduct of K obtained by giving the precedence to the message delivering w.r.t. the handover of l'. Now, it can be easily proved that

$$\{\, (K, (\nu l)(l' :: \langle \text{`HI'} \rangle \parallel SYS))\,\} \;\cup\; \mathcal{H} \;\cup\; \mathcal{D}$$

is a bisimulation. By the fact that $\approx_d \subset \simeq_d$ and by transitivity of $\simeq_d$, this suffices to prove Equation (3).

5 Conclusions and Related Work

We have presented a calculus inspired by KLAIM whose main feature is the handling of inter-node connections. We have first presented a basic setting where connections are reliable and immutable; then, we have enriched the basic framework with failures and dynamically evolving connections, two more advanced features for global computing. In each setting, we have used our formalisms to specify and verify some non-trivial global computing applications, by exploiting a may-testing equivalence.

Related work. In the last decade, several languages for modelling and programming distributed and global computing systems have been proposed in literature; we mention here only the most strictly related ones.

In DJoin [16], located mobile processes are hierarchically structured and form a tree-like structure evolving during the computation. Entire subtrees, not just single processes, can move and fail. Communication takes place in two steps: first, the sending process sends a message on a channel; then, the ether (i.e. the environment containing all the nodes) delivers the message to the (unique) process that can receive on that channel. Failures are programmed (i.e., they result from the execution of some process actions) and can be detected by processes. We believe that the setting presented in this paper is more realistic than DJoin because the considered interconnection topology is more general than trees and also because we do not assume any implicit engine for distant communications. Finally, we model failures in a way that is closer to actual global computers.

The Ambient calculus [7] is an elegant notation to model hierarchically structured distributed applications. Like in our work, the calculus is centered around the notion of connections between ambients, that are containers of processes and data. Each language primitive can be executed only if the ambient hierarchy is structured in a precise way; e.g., an ambient n can enter an ambient m only if n and m are sibling, i.e. they are both contained in the same ambient. However, like DJoin, Ambient strongly relies on a tree-like structure for the ambient hierarchy. Moreover, to the best of our knowledge, no explicit notion of failures, close to actual global computing requirements, has been ever given for Ambient.

[27] presents NOMADIC PICT, a distributed and agent-based language based on the π-calculus. It relies on a flat net where named agents can roam. Communication between two agents can take place only if they are located at the same node (thus no low-level remote communication is allowed). However, the language also provides a (high-level) primitive for remote communication, that transparently delivers a message to an agent even if the latter is not co-located with the sender. This primitive is then encoded in the low-level calculus by a central forwarding server, implemented by only using the low-level primitives. The assumption that only co-located agents can communicate is, in our opinion, too demanding. Moreover, it is not clear to us how the theory can be adapted to consider failures.

Another distributed version of the π-calculus is presented in [21]; the resulting calculus contains primitives for code movement and creation of new localities/channels in a net with a flat architecture. The main feature of the language is the possibility of controlling process activities via (sophisticated and non-standard) type systems. No notion of explicit connections and of failures have been integrated in the framework yet.

We now touch upon some formalisms for distributed computing relying on the powerful paradigm put forward by LINDA [18]. In *TuCSoN* [23], tuple spaces are enhanced with the capability of programming their behaviour in response to communication events; moreover, the computational model relies on a hierarchical collection of (possibly) distributed tuple spaces. *MARS* [5] is a coordination tool for Java-based mobile agents that defines LINDA-like tuple spaces programmable to react when accessed by agents. Such mechanisms can be used to control accesses to specific tuples. In TKLAIM, this can be obtained either by using dynamically created (private) nodes or by tailoring the capability-based type systems presented in [19, 20]. *Lime* [26] exploits multiple tuple spaces to coordinate mobile agents and adds mobility to tuple spaces: it allows processes to have private tuple spaces and to transparently and transiently share them. In TKLAIM, sharing of resources can be somehow achieved via dynamic handling of links; however, tuple spaces are bound to nodes and nodes cannot move.

Finally, we want to remark that the use of observational equivalences to state and proof soundness of protocols is a well-established technique in the field of process calculi; some notable examples are [1, 22, 24, 28]. In particular, in the last paper, an automatic verification tool to prove equivalences in the π-calculus is described. As an application, the authors automatically verify an equality, similar to ours, stating the soundness of the PLMN example.

Acknowledgements. We would like to thank the anonymous referees for some suggestions that helped in improving the paper.

References

1. M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the Spi calculus. In *Proc. of CONCUR'97*, volume 1243 of *LNCS*, pages 59–73. Springer, 1997.
2. H. Attiya and J. Welch. *Distributed Computing*. McGraw Hill, 1998.
3. L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. of the 7th WETICE*, pages 110–115. IEEE, 1998.
4. L. Bettini, M. Loreti, R. Pugliese. An Infrastructure Language for Open Nets. In *Proc. of the 2000 ACM Symposium on Applied Computing*, pages 373–377, ACM Press, 2002.
5. G. Cabri, L. Leonardi and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In *Proc. of the 2nd Int. Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 237–248. Springer, 1998.
6. L. Cardelli. Abstractions for mobile computation. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS, pages 51–94. Springer, 1999.
7. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
8. S. Castellani, P. Ciancarini, and D. Rossi. The ShaPE of ShaDE: a coordination system. Tech. Rep. UBLCS 96-5, Dip. di Scienze dell'Informazione, Univ. di Bologna, Italy, 1996.
9. T. Chandra and S.Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.
10. S. Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105(1):132–158, 1993.

11. N. Davies, S. Wade, A. Friday, and G. Blair. L^{2}imbo: a tuple space based platform for adaptive mobile applications. In *Int. Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP'97)*, 1997.
12. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
13. R. De Nicola, D. Gorla, and R. Pugliese. Basic observables for a calculus for global computing. Technical Report 07/2004, Dip. di Informatica, Univ. di Roma "La Sapienza". Available at `http://www.dsi.uniroma1.it/~gorla/papers/bo4k-full.pdf`.
14. R. De Nicola and M. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.
15. D. Deugo. Choosing a Mobile Agent Messaging Model. In *Proc. of ISADS 2001*, pages 278–286. IEEE, 2001.
16. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. of CONCUR '96*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.
17. R. Fuzzati, M. Merro, and U. Nestmann. Modelling Consensus in a Process Calculus. In *Proc. of CONCUR'03*, volume 2761 of *LNCS*. Springer-Verlag, 2003.
18. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
19. D. Gorla and R. Pugliese. Resource Access and Mobility Control with Dynamic Privileges Acquisition. In *Proc. of ICALP'03*, volume 2719 of *LNCS*, pages 119-132. Springer, 2003.
20. D. Gorla and R. Pugliese. Enforcing Security Policies via Types. In *Proc. of Security in Pervasive Computing (SPC'03)*, volume 2802 of *LNCS*, pages 88-103. Springer, 2003.
21. M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.
22. R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993.
23. A. Omicini and F. Zambonelli. Coordination of Mobile Information Agents in Tucson. *Journal of Internet Research*, 8(5):400-413, 1998.
24. F. Orava and J. Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, 4:497–543, 1992.
25. J. Parrow. An introduction to the pi-calculus. In *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.
26. G.P. Picco, A.L. Murphy and G.-C. Roman. LIME: Linda Meets Mobility. In *Proc. of the 21st Int. Conference on Software Engineering (ICSE'99)*, pages 368–377. IEEE, 1999.
27. A. Unyapoth and P. Sewell. Nomadic Pict: Correct Communication Infrastructures for Mobile Computation. In *Proc. of POPL'01*, pages 116–127. ACM Press, 2001.
28. B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In *Proc. of CAV '94*, volume 818 of *LNCS*, pages 428–440. Springer, 1994.

Mobile Agent Based Fault-Tolerance Support for the Reliable Mobile Computing Systems

Taesoon Park

Department of Computer Engineering, Sejong University,
Seoul 143-747, Korea
tspark@sejong.ac.kr

Abstract. To support the fault tolerance of mobile computing systems, many checkpointing coordination and message logging schemes have been proposed. However, due to the mobility of mobile hosts, coordination and control of these schemes are quite complicate and have the possibility of inefficiency. In this paper, a fault-tolerance scheme based on mobile agents is proposed. In the proposed scheme, a set of mobile and stationary agents are used to trace and maintain the recovery information for the mobile host. The mobile agents properly trace the mobile host and manage the suitable distance between the recovery information and the mobile host. Also, the migration of the recovery information of the mobile host can be performed asynchronously with the hand-off of the mobile host and hence the fault-tolerance service by mobile agents dose not incur any unnecessary hand-off delay.

1 Introduction

Fault-tolerance is one of most important design issues to build a reliable computing system. Especially for the mobile computing systems in which the mobility of mobile hosts complicates the design of many system components, the design of fault-tolerant services may also be very complex and costly, unless it is carefully designed. Many fault-tolerant schemes have been proposed to deal with the mobility of mobile hosts and to reduce the recovery cost of the mobile computing system. The proposed schemes can be categorized into the checkpointing coordination schemes and the message logging schemes.

For the checkpointing coordination of mobile hosts, some coordination algorithms used for the distributed computing system consisting of static hosts have been extended [1, 3, 7, 8, 12]. The extension of these algorithms mainly focused on the reduction of the network bandwidth usage, since the cost of wireless networks is very high. As a result, algorithms with the less frequent synchronization, the less number of coordination messages and the fewer number of processes participating in the coordination have been suggested. However, when the recovery is concerned, checkpointing-coordination schemes in the literature have a common problem for rollback. Because of the live-lock problem[6], which causes the recursive rollbacks during the recovery, either the rollback of the related mobile

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 173–187, 2005.

hosts has to be synchronized or the centralized rollback coordination is required, which is undesirable for mobile hosts using the wireless network.

One way to avoid the rollback coordination and to guarantee the asynchronous recovery is to use pessimistic or optimistic message logging with the independent checkpointing [5, 9, 14]. These logging schemes do not require intensive message communication for checkpointing coordination. Instead, the application messages between mobile hosts should be logged in the stable storages. Asynchronous recovery without checkpointing coordination can be a great advantage for the mobile computing systems. However, the checkpoint and the message log may cause another problem in the mobile computing environment, since a mobile host may be far away from the stable storage carrying its checkpoint or message log in the event of recovery. Hence, for the mobile computing systems, distributed recovery information management has been another important design issue and the migration of recovery information should be considered during the fail-free operation, balancing the migration cost and the recovery cost.

The distributed recovery information management schemes [10, 11, 15], however, have common problems as follows: First, the information migration decision is made during the hand-off procedure of the mobile host and hence the hand-off should be delayed until the migration of the checkpoint and the message log is completed. Another problem is that the migration frequency of the checkpoint and the message log is decided without considering their weights. The checkpoint is usually required in the early stage of the recovery of a mobile host while the logged messages are consumed gradually. Therefore, it is desirable to use separate migration frequencies for the checkpoint and the message log. The control information regarding the checkpoint and the message log can also be a big burdon on the system, since one mobile support station supports a large number of mobile hosts. The garbage collection of the distributed recovery information and their control information can be another problem.

To solve the problems of existing solutions, we propose a mobile agent based scheme to build a fault-tolerant mobile computing system. In the proposed scheme, a stationary agent residing in each mobile support station site takes care of the recovery information of mobile hosts so that the mobile support station can concentrate on its own tasks, such as the location management and the call query handling. Also, a set of two mobile agents for each mobile host make a suitable decision on the checkpoint migration and take care of the distributed log information, separately. Therefore, the migration of the recovery information of a mobile host can be performed asynchronously with the hand-off procedure and the fault-tolerance service does not cause any unnecessary delay on the hand-off procedure. Moreover, two different weights can be put on the checkpoint migration and the log migration since two mobile agents separately manages the checkpoint and the message log.

The rest of this paper is organized as follows: Section 2 briefly describes the mobile computing system model and the fault-tolerant system model. Section 3 presents the proposed system architecture for mobile agent based fault-tolerant services and also the coordination model for the stationary agents and the mobile agents. In Section 4, the performance of the proposed scheme is briefly discussed and Section 5 concludes the paper.

2 Background

2.1 Mobile Computing System Model

The mobile computing system [1, 2] considered in this paper consists of a number of mobile hosts (MHs) and mobile support stations (MSSs). The MSSs are static and between any two MSSs, a high speed wired communication link is assumed. The MSS provides various services to support the mobility of the MHs, such as the location management and the call query handling. The physical service region of an MSS is restricted and the service region covered by an MSS is called a *cell*. The communication between an MSS and the MHs in its service region is done by the wireless link supporting FIFO communication in both directions.

An MH may traverse a number of cells while communicating with another MH or performing some computation. To provide continuous services throughout the system, a *hand-off* process is performed between two MSSs servicing the corresponding cells as the MH crosses the boundary between two cells. For the hand-off, the MH first ends its previous connection by sending a *leave(r)* message to the local MSS when the MH leaves a cell, where r is the sequence number of the last message received from the MSS. The MH then establishes a new connection by sending a *join(MH-id, previous MSS-id)* message to the new MSS, when it enters another cell. Then the MSS servicing the new cell begins the hand-off process with the previous MSS to update the location information of the MH and retrieve the undelivered messages from the previous MSS.

A list of identifiers of MHs, called an *Active_MH_List*, is maintained by each MSS and in this list, the identifiers of the MHs that are currently supported by the MSS are included. For the location management of the MHs, the scheme based on the two-level data hierarchy, consisting of the *home location register(HLR)* and *visitor location register(VLR)*, is assumed.

2.2 Distributed Computation Model

A distributed computation performed by a set of N processes running concurrently on MHs in the system is assumed. Each process experiences a sequence of state transitions during its execution and an atomic action which causes the state transition is called an *event*. The event is said to be *internal* if it causes no interaction with another process. Otherwise, the event is called an *external* event. Message-sending and message-receiving events are the external events. A sequence of events within a process is called a *computation*. The computation in this paper is assumed to follow a *piece-wise deterministic model*, in which a process can always produce the same sequence of states for the same sequence of message-receiving events.

Let $R_{(i,\alpha)}$ denote the α-th message-receiving event of a process P_i and the *state interval*, $I_{(i,\alpha)}$, denote the sequence of states generated between $R_{(i,\alpha-1)}$ and $R_{(i,\alpha)}$, where $\alpha > 0$ and $R_{(i,0)}$ denotes the initial event. Then, the inter-process dependency [9] caused by the message communication can be defined as follows:

Definition 1: A state interval $I_{(i,\alpha)}$ is said to be *dependent on* another state interval $I_{(j,\beta)}$ if one of the following conditions is satisfied; and the dependency relation is denoted $I_{(j,\beta)} \rightarrow I_{(i,\alpha)}$:

(1) $i = j$ and $\alpha = \beta + 1$, or
(2) For an event $R_{(i,\alpha-1)}$, the corresponding message-sending event has happened in $I_{(j,\beta)}$, or
(3) For any $I_{(k,\gamma)}$, $I_{(j,\beta)} \rightarrow I_{(k,\gamma)}$ and $I_{(k,\gamma)} \rightarrow I_{(i,\alpha)}$.

2.3 Checkpointing, Message Logging and Consistent Recovery

The failure model considered in the paper follows the fail-stop [13] model, in which a process stops the execution and does not perform any malicious action in case of a failure. When an MH fails and loses the contents of the volatile memory, the processes running on the MH should restart and checkpoints are used to prevent the restart from the initial states. Checkpointing is an operation to save the intermediate state of a process in a stable storage so that the process can resume the execution from the restored state, called a *checkpoint*, after a failure. The resumption of the computation from the checkpoint is called the *rollback*.

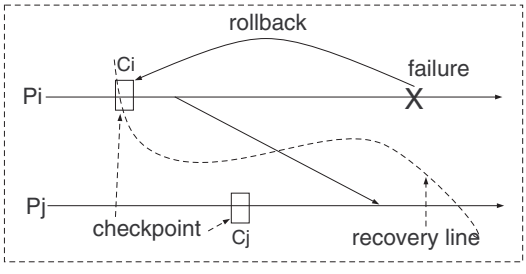


Fig. 1. An Example of Inconsistent Recovery

When a process fails and rolls back, the set of processes participating in the same distributed computation may result in the inconsistent state caused by the dependency relation. Figure 1 shows a typical example of the inconsistent system recovery. After the rollback of the process P_i followed by a failure, P_i resumes the computation from its previous checkpoint, C_i. Then, the states of two processes, P_i and P_j, denoted by the *recovery line* in the figure, are inconsistent because the current state of P_j is dependent on the states of P_i which have been discarded by the rollback of P_i. A state interval dependent on any discarded interval is called an *orphan* and any orphan interval also has to be discarded by another rollback. The recovery from a failure is said to be *consistent*, if no orphan state interval exists after the rollback-recovery.

Let D be a set of processes participating in a distributed computation, F_i^f be the f-th failure of P_i in D and L_i^f be the set of state intervals discarded by the

rollbacks of the processes in D, after the failure, F_i^f. Then, the consistent recovery can be defined as follows:

Definition 2: The recovery from a failure, F_i^f, is said to be consistent, if for any $I_{(i,\alpha)} \in L_i^f$, the relation, $I_{(i,\alpha)} \rightarrow I_{(j,\beta)}$, holds, then $I_{(j,\beta)}$ is also in L_i^f.

One way to avoid the inconsistent recovery is to use the *pessimistic message logging*, in which every message delivered to a process is logged into the stable storage so that the process, P_i, can reconstruct the exactly same interval, $I_{(i,\alpha)}$, with the recomputation of $R_{(i,\alpha-1)}$ using the logged message. Since every interval after the latest checkpoint can fully be recovered even after a failure, the recovery of a process with the pessimistic message logging can be consistent.

2.4 Distributed Recovery Information Management

For the efficient recovery of an MH, each MH has to manage its checkpoint and message log. However, because of the storage limitation of an MH, checkpoints and message logs of MHs are usually managed by the MSSs. Hence, the storage for checkpoints and message logs of an MH becomes dispersed over a number of MSSs as an MH moves around the cells. When the MH fails, it has to retrieve the latest checkpoint and the sequence of logged messages, which may cause the delay in recovery and increase the recovery cost.

Considering relatively high failure rates of MHs, instant recovery from a failure must be very important. However, checkpoints and logs distributed over the network may cause the severe message traffic and the delay during the recovery. For the efficient implementation, the mobility of MHs must be carefully traced and a proper mechanism for gathering distributed recovery information must be prepared. Also, since the size of a stable storage managed by each MSS is limited, checkpoints and logs no longer required for any recovery must be discarded for the reuse of the space. Such garbage collection may have to put extra communication overhead if the logs are dispersed over a large number of MSSs.

Some works related to the distributed storage management have been proposed in [11, 15]. For the fast recovery, it is desirable for checkpoints and message logs to be near the MH on recovery, and hence, checkpoints and logs of a MH in [11] keep moving as the MH performs the hand-off between two cells. As a result, instant recovery can be possible, however, the failure-free communication overhead cannot be negligible, considering the size of a checkpoint and logged messages. One suggestion made in [15] utilizes the home of each MH to maintain the recovery information. As an MH moves, it transfers checkpoints or logs to the home, and in case of a failure, it can find the recovery information at home. However, if the MH is far from home, the transfer cost can also be a problem.

3 Mobile Agent Based Fault-Tolerance Service

3.1 Overview

Mobile agent based fault-tolerance service has two main goals: One is to minimize the burden of fault-tolerance services on the MSS so that the MSS can

concentrate on its original tasks, such as the location management and the communication support for MHs. The other is to provide the efficient and also the most appropriate fault-tolerance service for each MH.

To achieve these two goals, the proposed system employs three types of agents: One is the stationary agent, called a *Recovery Agent (RA)*, which is designed to relieve the burden of the MSS and hence one RA is employed for each MSS. The RA takes care of the recovery information, such as the checkpoint, the logged messages and the related information, for the MHs covered by the MSS. In case that one of those MHs fails, the RA takes the responsibility of the recovery of the MH. To make efficient and proper recovery information migration decision for each MH, two mobile agents are employed for each MH. One is the *checkpointing agent (CPA)* and the other is the *log agent (LGA)*. The CPA takes care of the checkpoint of the MH and makes a decision on the checkpoint migration. The LGA manages the location information of the distributed message log and takes care of the garbage collection for the unnecessary logs.

One thing notable in the proposed model is that the checkpoint of an MH and its logged messages are separately managed, unlike the existing schemes. For an MH to recover from a failure, it should restore its previous state from a checkpoint and then redo the computation using the logged messages. Hence, the checkpoint near the MH can help the early start of the recovery. However, the logged messages would be used gradually as the recomputation proceeds. In other words, the slight delay in the collection of logged messages may not cause the severe delay in the recovery. Therefore, in the proposed scheme, two different weights are put on the checkpoint migration and the log migration. This is the reason why two mobile agents are used for the checkpoint and the message log management.

3.2 Recovery Agent

The recovery agent is a stationary agent residing in an MSS site and its main task is to manage recovery information of the MHs which have been in the cell covered by the corresponding MSS. Three types of recovery information are maintained by the recovery agent: One is the MH's location information. To trace the location of an MH, the recovery agent records the previous location and the next location of every MH which has visited and these information can be obtained from the MSS during the hand-off procedure. The recovery agent also temporarily manages checkpoints and message logs of those MHs. Since checkpoints and message logs would be migrated on the decision of the checkpointing agent or the log agent, the recovery agent holds them until their migration is decided or their deletion is decided for the garbage collection.

The recovery agent, RA_p, working for MSS_p, maintains the following information:

- $INF(i,k)$: The k-th recovery information entry of MH_i, where k denotes the k-th visit of MH_i. $INF(i,k)$ contains the following six fields.

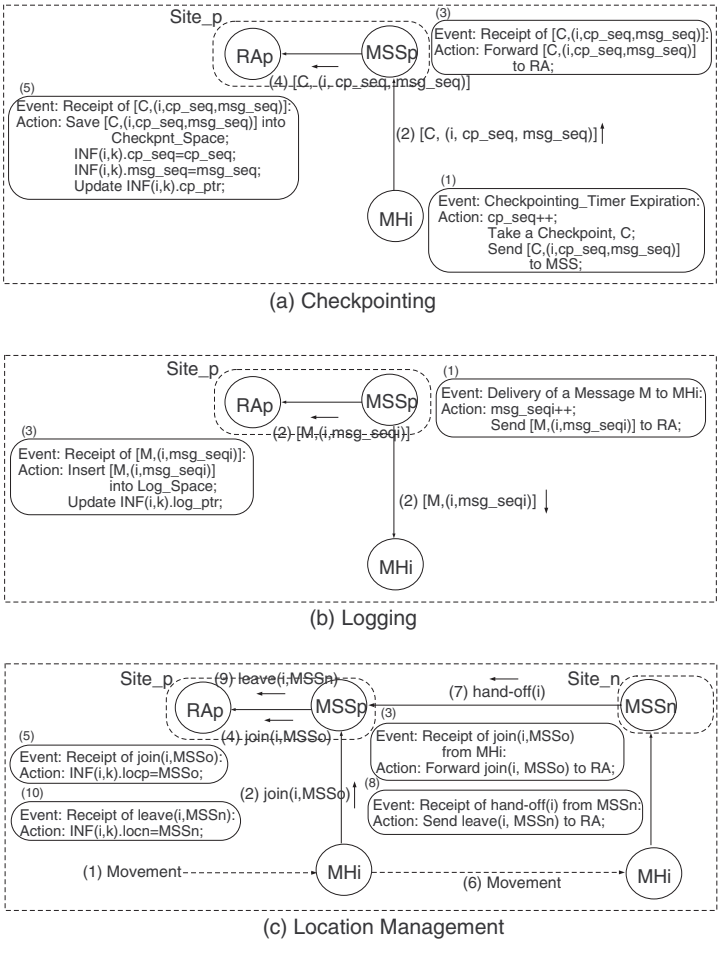


Fig. 2. Task Description of the Recovery Agent

- loc_P, loc_N : Identifiers of the previous and the next MSSs visited by MH_i. loc_P is filled when MH_i sends the *join* message to MSS_p and loc_N is filled when MSS_p performs the hand-off of MH_i with the next MSS.
- *cp_seq*, *msg_seq*, *cp_ptr* : When MH_i takes a new checkpoint, RA_p temporarily saves the checkpoint in the secure storage and holds the pointer to the new checkpoint until the checkpointing agent takes care of it. The checkpoint sequence number, *cp_seq*, and the sequence number of the last message which has been received before the checkpointing, *msg_seq*, are also recorded. *msg_seq* is recorded to indicate the recomputation point.
- *log_ptr* : When MSS_p delivers a message for MH_i, the message is also sent to RA_p and logged by RA_p. RA_p maintains the list of message log and *log_ptr* is the pointer to the header of the message log list.

- $status_{CPA}$, $status_{LGA}$: Status fields to indicate whether the checkpointing agent and the log agent of MH_i are currently residing in RA_p. Since the hand-off of MH_i and the migration of the two mobile agents are asynchronously processed, the existence of MH_i does not necessarily indicate the presence of CPA_i and LGA_i.

Figure 2 describes the task of the recovery agent, RA_p, and its cooperation model with the corresponding MSS_p. As shown in the figure, the recovery agent performs three main tasks as follows:

- **Checkpointing:** When MH_i takes a new checkpoint, it sends the checkpoint with its sequence number to the current MSS, which then forwards the checkpoint to the recovery agent, RA_p. RA_p saves the checkpoint in the stable storage and maintains the checkpoint related information.
- **Message Logging:** Whenever MSS_p delivers a message to MH_i, it sends a copy of the message to RA_p. RA_p adds the message into the list of logged messages for MH_i, which is stored in $INF(i,k)$, and updates log_ptr.
- **Location Management:** When MH_i enters a new cell or leaves the cell, a hand-off procedure is performed with the adjacent MSS. After a hand-off for MH_i, the MSS sends the $join(i, MSS_o)$ or $leave(i, MSS_n)$ message to the recovery agent, where the $join(i, MSS_o)$ message includes the identifier of the previous MSS, say MSS_o in the example, and the $leave(i, MSS_n)$ message includes the identifier of the next MSS, say MSS_n in the example. The recovery agent, RA_p, then properly updates the location information of MH_i in $INF(i,k).loc_P$ and $INF(i,k).loc_N$.

3.3 Checkpointing Agent

The checkpointing agent is a mobile agent and one agent is created for an MH. One main task of the checkpointing agent is to manage the latest checkpoint of the MH. The checkpoint taken by an MH is temporarily managed by the recovery agent and on the request of the checkpointing agent, it is transferred to the checkpointing agent. Another important task of the checkpointing agent is to make a migration decision of the checkpoint. For the early start of the recovery, the checkpoint is necessary to be near the MH. However, considering the migration cost of the checkpoint, frequent checkpoint migration should be a burden on the system. Hence, the checkpointing agent considers both the migration cost and the recovery cost to make an efficient decision.

One notable point of the proposed migration model is that the migration of the checkpointing agent is asynchronously processed with the migration of the MH. As a result, the checkpoint migration causes no delay for the hand-off process. The checkpointing agent, CPA_i, working for MH_i, maintains the following information:

- CP_INF : The information of the latest checkpoint for MH_i, which includes the following three fields.

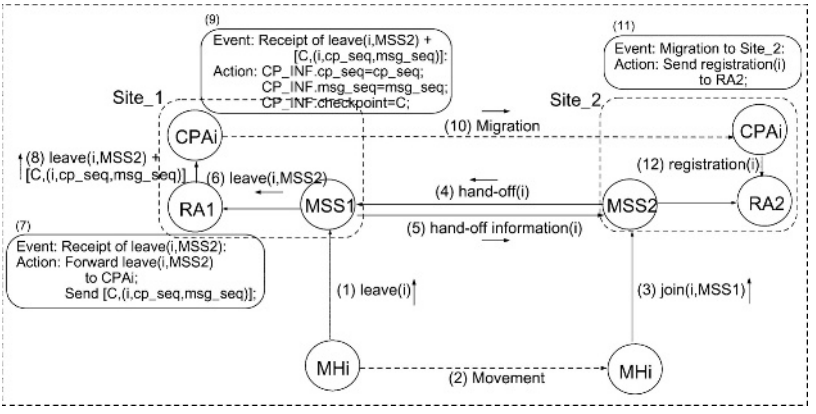


Fig. 3. Task Description of the Checkpointing Agent

- *cp_seq*, *msg_seq* : The sequence number of the latest checkpoint and the sequence number of the last message which has been received before the checkpointing.
- *checkpoint* : The latest checkpoint of MH_i.

Figure 3 describes the task of the checkpointing agent, CPA_i, and its cooperation model with the recovery agents. As shown in the figure, the checkpointing agent performs two main tasks as follows:

• **Checkpoint Tracking:** When an MH takes a new checkpoint, the recovery agent of the current cell temporarily takes care of the checkpoint. The checkpointing agent, CPA_i, retrieves the latest checkpoint of MH_i, when the current recovery agent, RA_1, forwards the $leave(i, MSS_2)$ message with the latest checkpoint information. On the receipt of this message, CPA_i replaces the old checkpoint with the new one, knowing that MH_i has taken a new checkpoint in MSS_1 and then left for the next cell covered by MSS_2. Since MH_i periodically takes a new checkpoint, CPA_i does not necessarily retrieve the new checkpoint from every recovery agent. When the MH does not take any new checkpoint before leaving the cell, the recovery agent just sends the *leave* message with the null checkpoint information.

Also, the hand-off of MH_i and the migration of CPA_i are not synchronized. Hence, when CPA_i arrives in a new MSS site, MH_i may already have left the corresponding cell. In this case, the recovery agent sends the *leave* message with any checkpoint information as the response of the *registration* message from CPA_i. Otherwise, if MH_i is still active in the current cell, the response from the recovery agent is delayed until MH_i leaves the cell.

• **Migration:** On the receipt of the *leave* message from the recovery agent, the checkpointing agent should decide its migration with the latest checkpoint. For the checkpoint to be near the MH_i, CPA_i should migrate to the next site. However, considering the migration cost, CPA_i should stay in the current site if

the checkpoint is not that far away from MH_i. Hence, the checkpointing agent usually decides its migration balancing the migration cost and the recovery cost. Two migration strategies can be used as follows:

- **Pessimistic Migration:** The checkpointing agent migrates every time when the migration of the MH is notified. Hence, there is a high probability that the checkpointing agent and the MH are in the same site when a failure occurs. As a result, the recovery cost can be minimized.
- **Time-based Migration:** This scheme tries to reduce the migration cost by the checkpointing agent staying in one site for a pre-determined time period. After CPA_i receives a $leave(i, MSS_{n1})$ message from RA_p, it stays in the current site for the time period, T, where T should be the time long enough for MH_i to perform a number of migration. CPA_i then directly migrates to the site currently supporting MH_i, which reduces the frequency of the migration and the migration cost.
  To search the site currently supporting MH_i, CPA_i first sends the *query* message to RA_{n1} and RA_{n1} responses with $leave(i, MSS_{n2})$ message and any new checkpoint information if MH_i has already left the site. Otherwise, if MH_i is still active in the current site, RA_{n1} sends the $in\_active(i)$ message to CPA_i. If CPA_i receives a $leave(i, MSS_{n2})$ message, it sends another query to RA_{n2} and so on, until CPA_i finally receives the $in\_active(i)$ message. After finding out the site currently supporting MH_i, CPA_i migrates to the target site with the latest checkpoint. If CPA_i receives a new checkpoint information as the response of the query, it replaces the checkpoint information before migration.

3.4 Log Agent

The log agent is also a mobile agent and one agent is created for an MH. The main task of the log agent is to manage the distributed log information. One principle of our fault-tolerance service model is to put less weight on the log migration, since collection of logged messages for the failure-recovery is less important to the early start of the recovery. Also, considering the high migration cost of the logged messages, the log migration may not be desirable in many cases. Hence, the log agent manages only the distributed log information, such as the identifier of the recovery agent which manages the message log and the sequence number to sort the logged messages for the recomputation. Another task of the log agent is the garbage collection of the unnecessary log entries. When the log agent is informed of new checkpointing from the recovery agent, it should send the *garbage collection* messages to the recovery agent so that the unnecessary log entries can be deleted.

For these tasks, the log agent, LGA_i, working for MH_i, maintains the following information:

- LOG_INF : The list of message log entries for MH_i. Each entry includes the following two fields.

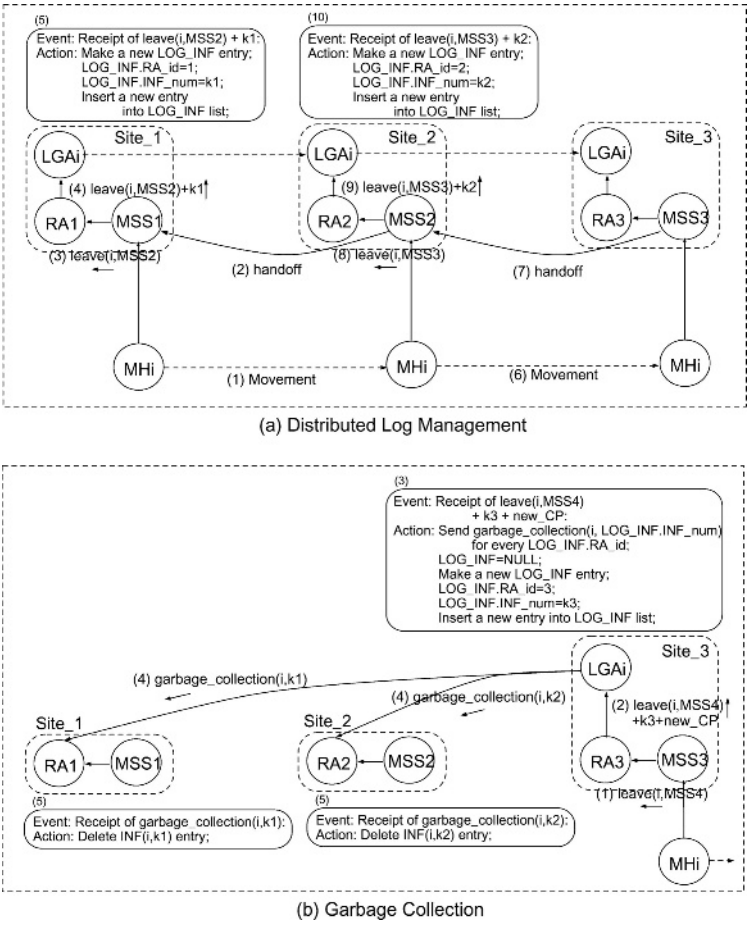


Fig. 4. Task Description of the Log Agent

- RA_id : The identifier of the recovery agent which holds the message logs.
- INF_num : The entry number k of the $INF(i,k)$ structure in which the recovery agent, RA_id, records the corresponding message log.

Figure 4 describes the task of the log agent, LGA_i, and its cooperation model with the recovery agents. The migration of LGA_i basically follows the pessimistic migration scheme. As shown in the figure, the log agent performs two main tasks as follows:

• Distributed Log Management: When LGA_i migrates to a site in the traveling path of MH_i, it first sends the $registration(i)$ message to the recovery agent, RA_p. When MH_i leaves the corresponding cell, RA_p sends the $leave$ message with the k value for the latest entry of $INF(i,k)$ which records the message log of MH_i. LGA_i then inserts a new entry into LOG_INF, where RA_id is set to

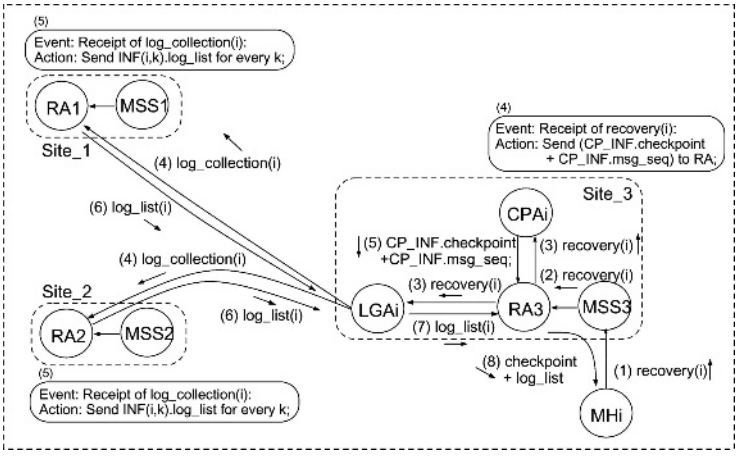


Fig. 5. Failure-Recovery Process

p and INF_num is set to k. As a result, LOG_INF of LGA_i contains a list of the recovery agents which hold the message logs for MH_i.

• Garbage Collection: RA_p also sends any new checkpointing information with the *leave* message. In the pessimistic message logging approach, the message logs taken before any checkpointing are not necessary any more. Hence, LGA_i performs the garbage collection of the message logs in its LOG_INF by sending the *garbage collection* message to each recovery agent indicated by $LOG\_INF.RA\_id$.

3.5 Failure-Recovery

Figure 5 describes the failure-recovery process for MH_i. The recovery agent, RA, takes the prime responsibility of the recovery of an MH. Hence, on the receipt of the $recovery(i)$ message from MH_i, the current recovery agent, RA_p, contacts with CPA_i and LGA_i to collect the checkpoint and the logged messages for MH_i. On the request from RA_p, CPA_i replies with the latest checkpoint and LGA_i sends the $log\_collection(i)$ messages to the recovery agent in its LOG_INF list. After collecting the logged messages from other recovery agents, LGA_i, forwards the log list pointer to RA_p. For the early start of the recovery process, RA_p starts the recovery of MH_i as soon as it receives the latest checkpoint from CP_i, without waiting for the completion of log collection. Then, RA_p sorts the logged messages in the message sequence order and sends them to MH_i.

4 Performance

Table 1 briefly compares the performance of the proposed scheme with the existing schemes suggested in [10, 11, 15]. To denote the migration frequency, the notations found in [4] are used: FM denotes the full migration in which the

Table 1. Performance Comparison

	Migration Frequency		
	Checkpoint Migration	Message Log Migration	Synchronization
Pessimistic Scheme in [11]	FM	FM	YES
Lazy Scheme in [11]	NM	NM	YES
Home-based Scheme in [15]	FM	FM	YES
Mobility-based Scheme in [10]	JM	JM	YES
Pessimistic CPM + Lazy LGM	FM	NM	NO
Pessimistic CPM + Partial LGM	FM	FM+NM	NO
Time-based CPM + Lazy LGM	JM	NM	NO
Time-based CPM + Partial LGM	JM	FM+NM	NO

recovery information is migrated for each hand-off of the mobile host. NM indicates no migration of any information. JM denotes the jump migration in which the migration frequency can be controlled so that the recovery information is migrated for a number of hand-offs of the mobile host.

One notable point of our proposed scheme is that the checkpointing agent and the log agent separately decide the migration frequencies of the checkpoint and the message log based on their importance. Compared to this, the existing schemes make one migration decision for both of the checkpoint and the message log. The pessimistic scheme suggested in [11] and the home-based scheme suggested in [15] require the full migration for both of the checkpoint and the message log while the lazy scheme in [11] requires no migration of the checkpoint and the message log. The mobility-based scheme proposed in [10] allows the jump migration however the same migration frequency is used for both of the checkpoint and the message log.

Compared to this, the mobile-agent based scheme can use different frequency for the checkpoint migration and the log migration. For the checkpoint migration, the pessimistic or time-based checkpoint migration(CPM) scheme can be chosen. For the log migration, the basic scheme is the lazy log migration(LGM) scheme. However, a slightly modified scheme, called a partial log migration scheme, can be used. In this scheme, early parts of the message log are migrated with the log agent so that a part of the message log can follow the full migration while the rest follows no migration. Another notable point of the proposed scheme is that the migration of the recovery information is not necessarily synchronized with the hand-off process of the mobile host as indicated in the *Synchronization* column of the table.

5 Conclusions

In this paper, we have proposed an agent based fault-tolerance service for the mobile computing systems. In the proposed scheme, one stationary agent is used to relieve the burden of the mobile support station and takes the prime responsibility of the recovery. Two mobile agents are employed to make an efficient migration decision of the checkpoint and to manage the distributed message log information, respectively. This paper have described the tasks of three main components of the fault-tolerance service and also presents the interaction model of three cooperating agents. In the proposed scheme, the migration of the recovery information of a mobile host can asynchronously be performed with the hand-off procedure and the fault-tolerance service does not cause any unnecessary delay on the hand-off procedure. Another notable point of the proposed scheme is that mobile agents which have the mobility take care of the recovery information and garbage collection of those information; and make a decision suitable for each mobile host.

Acknowledgments

This work was supported by grant No. 04-Kicho-051 from the University Fundamental Research Program of Ministry of Information & Communication in Republic of Korea.

References

1. Acharya, A., Badrinath, B.R.: Checkpointing Distributed Applications on Mobile Computers. Proc. of the 3rd Int'l Conf. on Parallel and Distributed Information Systems (1994) 73–80
2. Ayildiz, I.F., Ho, J.S.M.: On Location Management for Personal Communications Networks. IEEE Communications Magazine (1996) 138–145
3. Cao, G., Singhal, M.: Low-Cost Checkpointing with Mutable Checkpoints in Mobile Computing Systems. Proc. of the 18th Int'l Conf. on Distributed Computing Systems (1998) 464–471
4. Cao, J., Feng, X., Das, S.K.: Mailbox-Based Scheme for Mobile Agent Communications. IEEE Computer (2002) 54–60
5. Damani, O.P., Garg, V.K.: How to Recover Efficiently and Asynchronously When Optimism Fails. Proc. of the 16th Int'l Conf. on Distributed Computing Systems (1996) 108–115
6. Koo, R., Toueg, S.: Checkpointing and Rollback-Recovery for Distributed Systems. IEEE Transactions on Software Engineering, Vol. SE-13, No. 1 (1987) 23–31
7. Manivannan, D., Singhal, M.: Failure Recovery Based on Quasi-Synchronous Checkpointing in Mobile Computing Systems. OSU-CISRC-796-TR36, Dept. of Computer and Information Science, The Ohio State University (1996)
8. Neves, N., Fuchs, W.K.: Adaptive Recovery for Mobile Environments. Communications of the ACM, Vol. 40, No. 1 (1997) 68–74
9. Park, T., Woo, N., Yeom, H.Y.: An Efficient Optimistic Message Logging Scheme for Recoverable Mobile Computing Systems. IEEE Transactions on Mobile Computing, Vol. 1, No. 4 (2002) 265–277

10. Park, T., Woo, N., Yeom, H.Y.: An Efficient Recovery Scheme for Fault-Tolerant Mobile Computing Systems. Future Generation Computer Systems, Vol. 19, No. 1 (2003) 37–53
11. Pradhan, D.K., Krishna, P., Vaiday, N.H.: Recoverable Mobile Environment : Design and Trade-Off Analysis. Proc. of the 26th Int'l Symp. on Fault Tolerant Computing Systems (1996) 16–25
12. Prakash, R., Singhal, M.: Low-Cost Checkpointing and Failure Recovery in Mobile Computing. IEEE Transactions on Parallel and Distributed Computing Systems, Vol. 7, No. 2 (1996) 1035–1048
13. Schlichting, R.D., Schneider, F.B.: Fail-Stop Processors: An Approach to Designing Fault tolerant Computing Systems. ACM Transactions on Computer Systems, Vol. 1, No. 3 (1983) 222–238
14. Smith, S.W., Tygar, J.D., Johnson, D.B.: Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. Proc. of the 25th Int'l Symp. on Fault Tolerant Computing Systems (1995) 361–370
15. Yao, B., Ssu, K., Fuchs, W.K.: Message Logging in Mobile Computing. Proc. of the 29th Symp. on Fault Tolerant Computing Systems (1999) 294–301

Preserving Architectural Properties in Multithreaded Code Generation

Marco Bernardo and Edoardo Bontà

Università di Urbino "Carlo Bo"
Istituto di Scienze e Tecnologie dell'Informazione
Piazza della Repubblica 13, 61029 Urbino, Italy
{bernardo, bonta}@sti.uniurb.it

Abstract. Architectural descriptions can provide support for a formal representation of the structure and the overall behavior of software systems, which is suitable for an early assessment of the system properties as well as for the automated generation of code. The problem addressed in this paper is to what extent the properties verified at the architectural level can be preserved during the code generation process for multithreaded programs. In order to limit the human intervention, we propose to separate the thread synchronization management from the thread behavior translation. While a completely automated and architecture-driven approach can guarantee the correct thread coordination, we show that only a partial translation based on stubs is possible for the behavior of the threads, with the preservation of the architectural properties depending on the way in which the stubs are filled in.

1 Introduction

One of the major objectives of the software architecture level of design [14, 15] is that of producing a reference document – shared by all the people involved in the development process – which describes the structure of the software system as well as the main functional and non-functional aspects of its overall behavior. Whenever such a document is made formal through the use of a suitable architectural description language (ADL), an early assessment of the gross system properties can be carried out. This is the case with process algebraic ADLs, for which several techniques based on equivalence checking have been developed for the component-oriented verification and diagnosis of architectural mismatch freedom [3, 11, 10, 9, 6, 5, 1].

As observed in [7], one of the big issues in the software engineering field is guaranteeing that the implementation of a software system conforms to its architectural description. In other words, a way has to be found to check whether the properties verified at the architectural level are preserved at the code level. In this respect, it may be helpful to generate code directly from the architectural description, as the latter represents an abstract model of the final system. Indeed, the purpose of automatic code generation should be not only to speed up the system implementation, but also to ensure conformance by construction.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 188–203, 2005.

In order to reconcile the (architecture-based) early property assessment and the (architecture-driven) automated code generation, in this paper we investigate to what extent the architectural properties can be preserved during the translation of architectural descriptions into code. On the upstream side we shall concentrate on process algebraic architectural descriptions, while on the downstream side we shall focus on multithreaded Java programs.

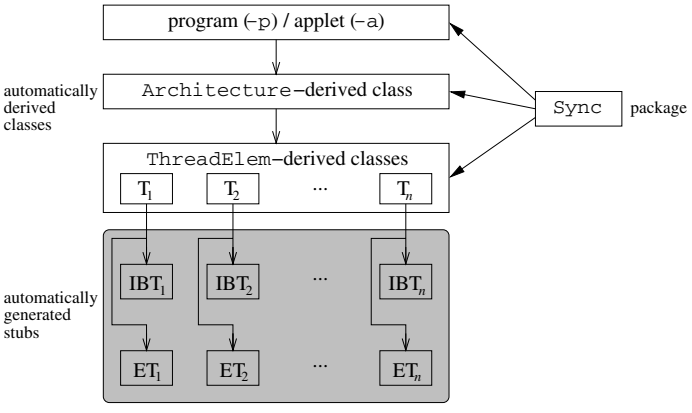


Fig. 1. File structure of the generated code

As discussed in [4] and recalled in Sect. 2, a completely automated and architecture-driven approach can guarantee the correct thread coordination. Given an architectural description in PADL [1], the approach of [4] is based on the use of a translator, called PADL2Java, and a Java package, called `Sync`. The package is structured around four conceptual layers: `Architecture`, `ThreadElem`, `Port`, and `Connector`. As shown in the upper part of Fig. 1, the translator synthesizes a class derived from `Architecture` plus as many classes derived from `ThreadElem` as there are architectural element types in the PADL description. The instances generated for the `ThreadElem`-derived classes, i.e. the threads, are guaranteed to interact as expected thanks to the generation of the appropriate instances of `Port` and `Connector`. On top of the previous classes, there may be another class with the `main()` method or an `Applet`-derived class, depending on the option with which PADL2Java is invoked.

In order to limit the human intervention, in this paper we propose to separate the thread synchronization management from the thread behavior handling, then we complete the approach of [4] by taking care of the thread behavior translation. In the process algebraic framework of PADL, the behavior of an architectural element – to be implemented as a thread – is given through a sequence of equations based on a restricted number of operators: action prefix, choice, and behavior invocation. In Sect. 3 we show how such process algebraic operators can be translated into the Java framework. In particular, for the action prefix we shall see that a different treatment is needed depending on whether the action – which corresponds to a sequence of thread statements – is an internal action

or an interaction. In fact, while the latter is involved in the communications, hence it is managed as explained in [4], the former can only be rendered as a stub during the translation of the thread behavior. More precisely, we shall have stubs for the actions internal to the behavior of the threads (IBT), together with stubs for the handling of the exceptions raised by the interactions of the threads (ET), as shown in the lower part of Fig. 1. This means that the translation of the thread behavior cannot be completely automatic, as should be expected due to the different levels of abstraction of an ADL and a programming language.

On the theoretical side, we show in Sect. 4 that the preservation of the architectural properties during the translation process critically depends on the way in which the IBT and ET stubs are filled in. We shall provide to this purpose a set of guidelines that guarantee the property preservation along the translation.

In Sect. 5 some remarks are finally reported about related and future work.

2 Thread Coordination

In this section we recall the approach proposed in [4] to ensure a correct thread coordination for Java programs generated from PADL descriptions.

2.1 The Language PADL

Every PADL description comprises two sections. In the first section, the types of architectural elements of the software system are defined by specifying their behavior and their interfaces. The behavior of an architectural element type is described through a sequence of behavioral equations of the form $B(\underline{x}) = P$. The process algebraic term P has the following syntax:

$$\begin{aligned} P &::= \texttt{stop} \mid \texttt{a}.P' \mid \texttt{choice}\{P, \ldots, P\} \\ P' &::= B(\underline{e}) \mid P \end{aligned}$$

where `stop` is the terminated process, $\texttt{a}.P$ is an application of the action prefix operator, $\texttt{choice}\{P_1, ..., P_n\}$ is an application of the choice operator, and $B(\underline{e})$ is a behavioral equation invocation.

The interfaces are actions occurring in the behavioral equations, which are used to interact with other architectural elements in the system. Each interaction has three qualifiers associated with it. The first qualifier establishes whether the interaction is an input (receiver side) or an output (sender side). The second qualifier determines the multiplicity of the communications in which the interaction can be involved: one-to-one (uni-interaction), conjunctive one-to-many (and-interaction – e.g. broadcast), and disjunctive one-to-many (or-interaction – e.g. client-server). The third qualifier establishes whether the interaction participates in a communication in a synchronous or asynchronous way.

In the second section of a PADL description, the topology of the system is defined by declaring the instances of the previously defined types of architectural elements, the interactions that act as interfaces for the whole system (useful for hierarchical modeling), and the attachments among the other interactions of the architectural element instances.

We conclude by exemplifying PADL through the description of a simple architectural element type, which will be reused in Sect. 3.6 to illustrate the thread generation. The architectural element `Checker_Type` that we consider gets values from another architectural element until a value is received that coincides with an expected one, then forwards the value to another architectural element and terminates. The values that are different from the expected one are printed out as long as they are received. Here is the PADL description of `Checker_Type`:

```
ELEM_TYPE Checker_Type(const integer expected_value)
  BEHAVIOR
    Getting_Value(void; local integer received_value) =
      <get_value?(received_value), _> . Checking_Value(received_value);
    Checking_Value(integer received_value; void) =
      choice {
        cond(received_value = expected_value) ->
          <prepare_to_forward, _> .
            <forward_value!(received_value), _> . stop,
        cond(received_value != expected_value) ->
          <prepare_to_print, _> . Printing_Value(received_value)
      };
    Printing_Value(integer received_value; void) =
      <print_value!(received_value), _> . Getting_Value()
  INPUT_INTERACTIONS UNI SYNC get_value
  OUTPUT_INTERACTIONS UNI SYNC forward_value
```

`Checker_Type` has three behavioral equations: `Getting_Value`, `Checking_Value`, and `Printing_Value`. Within the equations, every action is represented through its name – possibly followed by a parameter that is separated from the action name by "`?`" (resp. "`!`") in the case of an input (resp. output) action – and the information about its priority level and its weight – "`_`" is used to denote the default values. Unlike the other actions, the two actions `prepare_to_forward` and `prepare_to_print` are preceded by a boolean guard, which establishes the condition under which these actions can be executed based on the comparison of the received value with the expected one. These two actions together with `print_value` are internal, whereas `get_value` is an input synchronous uni-interaction and `forward_value` is an output synchronous uni-interaction.

2.2 The Java Package `Sync`

The package `Sync` offers a set of facilities to support the development of multithreaded Java programs by handling the details of the thread synchronization in a way that is transparent to the software developer. This package is organized into four conceptual layers: `Architecture`, `ThreadElem`, `Port`, and `Connector`. Each layer corresponds to a different architectural abstraction and comprises a set of components realized through Java classes and interfaces.

The first layer, `Architecture`, is an abstract class containing components belonging to lower layers. On the implementation side, `Architecture` is derived

from the class `ThreadElem`. This means that an arbitrarily complex architecture is a thread, which is useful for systems modeled in PADL in a hierarchical way.

The second layer, `ThreadElem`, is a class that inherits from the Java class `Thread`. The reason why a derived class has been defined instead of directly using the class `Thread` is related to the translation of the behavior of the architectural elements of a PADL description into the corresponding Java threads. As we shall see in Sect. 3, this is accomplished through additional methods made available in the class `ThreadElem`.

The third layer, `Port`, realizes the abstraction corresponding to a set of statements through which a thread interacts with other threads. Based on the synchronization model adopted in PADL, there are twelve types of `Port`: six are synchronous and six are asynchronous. If a `Port` is synchronous, it waits – and the thread that contains it passivates – until the communication has been established. If a `Port` is asynchronous, it communicates if the other connected `Port` is willing to communicate with it, otherwise an exception is raised and no communication takes place. Both in the synchronous and in the asynchronous mode, three types of `Port` are on the sender side and the other three are on the receiver side. A sending `Port` can transfer to one or more receiving `Port`s an array of generic `Object`s using the method `send()`. Similarly, a receiving `Port` can receive from one or more sending `Port`s an array of generic `Object`s using the method `receive()`. A null array represents a pure synchronization signal. Both on the sender and on the receiver side, the three types are termed "uni", "and", and "or". A uni-`Port` interacts with only one `Port` of another thread. An and-`Port`, instead, interacts with several `Port`s of other threads in a broadcast fashion. Finally, an or-`Port` interacts with only one `Port` of a thread selected out of a set of other threads.

The fourth layer, `Connector`, sets up a communication link between a sending `Port` and a receiving `Port` of two different threads. There are four types of `Connector` in order to cover all the possible combinations of the types (synchronous vs. asynchronous) of the two connected `Port`s. The package contains an internal mechanism – through the method `attach()` defined in the layer `Architecture` – that, given two `Port`s, creates a `Connector` of the right type.

2.3 The Translator PADL2Java

Once the PADL description of a multithreaded Java program has been provided, the software developer can use the translator PADL2Java to generate a skeleton of the program itself. As illustrated in the upper part of Fig. 1, the automatically generated Java code imports the package `Sync` and is composed of several classes derived from `ThreadElem`, each representing the corresponding architectural element type defined in the PADL description, as well as a class derived from `Architecture`, which corresponds to the overall PADL description.

The `Architecture`-derived class has five completely specified sections: *declaring threads*, *declaring architectural interactions*, *defining constructor*, *building architecture*, and *running architecture*.

Each `ThreadElem`-derived class has instead the following four sections: *defining constructor*, *defining behavior*, *instantiating input interactions*, and *instan-*

tiating output interactions. All of these sections are completely specified, except for *defining behavior*, whose generation is the focus of this paper and will be discussed in Sect. 3.

Finally, in the automatically generated code there is in addition a class with the `main()` method or an `Applet`-derived class, whenever PADL2Java is invoked with option `-p` (for program) or `-a` (for applet), respectively. If option `-c` (for class) is used, then no wrapper class is added to the generated code.

3 Thread Behavior

The approach of [4] only deals with thread coordination. The contribution of this section is to show how to translate the PADL description of an architectural element type into a thread. The resulting code will fill in the *defining behavior* section of the corresponding `ThreadElem`-derived class, as anticipated in Sect. 2.3.

The basic idea is that, besides the method `run()` already defined as abstract in the Java base class `Thread`, within the *defining behavior* section we need to generate some additional methods that are the Java translation of the PADL behavioral equations. The generation of these behavioral methods must then be complemented by the generation of some stubs, as depicted in the lower part of Fig. 1. Each such stub is a class containing either a method for every internal action occurring in the PADL description of the behavior of the thread (IBT), or a method for every interaction occurring in the PADL description of the behavior of the thread that can result in an exception to be handled (ET).

Thus, for every architectural element type contained in a PADL description, PADL2Java will have to generate not only a `ThreadElem`-derived class, but also two classes for the internal action translation and the interaction-related exception handling, respectively. The `ThreadElem`-derived class will declare as protected members two objects of class IBT and ET, respectively, and will instantiate these objects within the method `run()`.

Unlike the method `run()` and the behavioral methods, which are completely generated in an automatic way, the methods contained in the IBT and ET stubs have to be manually filled in by the developer. The reason is that, in the case of the IBT stubs, the methods are associated with the execution of the internal actions. An internal action describes at a high level of abstraction a set of operations to be carried out by the thread, so in general it will correspond to a sequence of Java statements to be defined by the developer. Likewise, the way in which an exception raised by an interaction has to be handled must be established by the developer.

In this section, we first analyze the execution flow of a thread and show how to generate the method `run()` avoiding recursion (Sect. 3.1). Then we show how to generate the code for the behavioral methods by proceeding by induction on the syntactical structure of the process algebraic terms occurring in the right-hand side of the corresponding behavioral equations (Sect. 3.2 to 3.5). Finally, we show an example of Java thread code generated from the PADL description of a specific architectural element type (Sect. 3.6).

3.1 The Execution Flow and the Method `run()`

The execution flow of a Java thread generated with PADL2Java is determined by the behavioral equations – translated into as many behavioral methods – of the corresponding architectural element type of the PADL description. In order to generate efficient Java code, in the translation process we have to get rid of the frequently occurring recursive behavioral invocations. In the process algebraic syntax only tail recursion comes into play, which is easy to transform into iteration.

Following [8], one possibility is to generate a `while` statement containing an `if-elseif` statement – or equivalently a `switch-case` statement – through which the next behavioral method to be executed is selected based on a label variable. Instead of invoking the next method, before returning each method properly sets the label variable.

Unfortunately this approach is not efficient when several method calls are contained within the conditional statement, because several checks must be done for different cases in every cycle. We therefore propose a variant of this approach, in which the address of the method to be executed is used, instead of a label variable. Since pointers are not available in Java, we exploit Java reflection in a transparent way to accomplish this task.

In order to implement our proposal, three protected methods are added to the class `ThreadElem`. The first method, `behavEqList()`, accepts as input an array of elements of class `BehavEqId`, each of which is an object containing information – name and formal parameters – about a behavioral method belonging to a `ThreadElem`-derived class. The method `behavEqList()` builds a map of the behavioral methods that translate the behavioral equations in the architectural element type for the `ThreadElem`-derived class. This static map is then used to retrieve a behavioral method, given its index (indices start from `0`).

The second method, `behavEqNext()`, requires to specify the index and the actual parameters of the next behavioral method to be executed. The method `behavEqNext()` returns `false` if the index is equal to or greater than the number of behavioral methods, or if the actual parameters do not match the formal parameters previously specified with `behavEqList()`. Otherwise, the method `behavEqNext()` returns `true` and the method retrieved from the map is placed into a private `ThreadElem` member variable. The value of this variable is set to `null` if the next behavior of the thread is described by process term `stop`.

The third method, `behavEqCall()`, invokes the behavioral method previously placed in the private `ThreadElem` member variable by `behavEqNext()`. The method `behavEqCall()` returns `true` if the last invocation of `behavEqNext()` has been successful and the value of the private `ThreadElem` member variable is not `null`. The returned value determines whether the body of the `while` statement has to be repeated or not.

In the automatically generated code, `behavEqList()` with the appropriate parameters is invoked by the constructor of the `ThreadElem`-derived class. The

methods `behavEqNext()` and `behavEqCall()`, instead, are invoked by `run()`. The latter is redefined in the *defining behavior* section of the `ThreadElem`-derived class, just before the definition of the behavioral methods:

```
public void run() {
  <IBT instantiation>
  <ET  instantiation>
  behavEqNext(0, <actual parameters of the first equation>);
  while (behavEqCall());
}
```

The call to the method `behavEqNext()` specifies the first behavioral method to be executed, which has index `0` in the map, together with its actual parameters.

3.2 Behavioral Invocations

The behavioral invocation $B(\underline{e})$ represents a process term that behaves as the behavioral equation whose identifier is B, when passing the possibly empty sequence of actual parameters $\underline{e}$. A behavioral invocation, which can occur only within the scope of an action prefix operator, is not translated into a behavioral method call, as this may result in the generation of inefficient code in case of recursion. Instead, a behavioral invocation is translated into an invocation of `behavEqNext()`, to which the index and the actual parameters of the behavioral method associated with the invoked behavioral equation are passed.

3.3 Stop

Process term `stop` represents the situation in which no further action can be executed. The process term `stop` is translated into an invocation of the method `behavStop()` defined in the class `ThreadElem`. The method `behavStop()` is similar to the method `behavEqNext()`, but it does not return any value, it has no input parameters, and causes the method `run()` to terminate because the next invocation of `behavEqCall()` returns `false`.

3.4 Action Prefix

The action prefix operator is used to represent a process term that can execute an action and then behaves as described by another process term. Every action has three pieces of information associated with it: (*i*) a boolean guard, expressing the possible constraint under which the action can be enabled (default value `true`), (*ii*) a positive integer number representing a priority level, which is used when resolving choices among several enabled actions (default value `1`), and (*iii*) a positive real number representing a weight, which is used when resolving choices among several enabled actions with the same priority (default value `1.0`).

In PADL an action can be an interaction or an internal action. In the `Sync`-based thread coordination, the output/input interactions are translated into invocations of the methods `send()`/`receive()` within the instances of the classes of layer `Port`, as described in Sect. 2.2. However, since `send()` and `receive()`

can be subject to the following two exceptions, the translation of the interactions must be completed by filling in the corresponding ET stubs.

The first exception, `UnattachedPortException`, is raised when an architectural interaction is executed, which is not attached to any other interaction. In this case, the architectural interaction is like an internal action, hence the sequence of Java statements translating it has to be manually provided.

The second one, `AsyncPortNotReadyException`, is raised when an asynchronous interaction is executed, but the other party is not ready to communicate with it. In this case, the thread containing the asynchronous interaction goes on, but the developer may want to add some Java statements to deal with this event.

Unlike the interactions, which are completely translated in an automatic way up to the handling of the exceptions that they may rise, the internal actions cannot be treated automatically at all. A method for each of them is placed in an IBT stub, which has to be filled in by the developer with the corresponding Java statements. As a consequence, every occurrence of an internal action is translated into an invocation of the related method in an IBT stub.

3.5 Choice

The choice operator expresses a selection among a certain number of alternative behaviors described through process terms. A choice-based process term is translated into a `switch-case` statement, whose condition is given by an invocation of the method `choice()` defined in the class `ThreadElem`.

There are two cases that must be addressed in order to translate the choice operator. The first one is the case where every process term involved in the choice starts with an action prefix operator. In this case the method `choice()` is directly employed, which accepts as input an array of objects of class `ChAct`, each of which contains the three pieces of information mentioned in Sect. 3.4 about one of the starting actions. Should one of the starting actions be an interaction, an additional piece of information is contained in the corresponding object, which is a reference to the object `Port` associated with the interaction. The method `choice()` returns the index (within the array) of the starting action selected for execution.

A starting action can be selected if: (*i*) its guard evaluates to `true`, (*ii*) the corresponding `Port` object is ready to send or receive, whenever the starting action is an interaction, and (*iii*) its priority is not less than the priority of the other enabled starting actions, with its weight being used to probabilistically solve the choice among the enabled starting actions with the highest priority. If all the enabled starting actions are interactions, the method `choice()` waits – and the thread that contains it passivates – until one of the associated `Port`s is ready to communicate. If the array that contains the objects of class `ChAct` is empty or all the guards of the starting actions evaluate to `false`, the method `choice()` returns a negative value.

Based on the index returned by `choice()`, the `switch-case` statement invokes the method associated with the execution of the selected starting action.

This method is `send()` or `receive()` in the case of an interaction, whereas for an internal action it is the corresponding method in an IBT stub. The invocation of this method is followed in turn within the `switch-case` statement by the translation of the process term prefixed by the selected action. In the default clause, which comes into play when a negative value is returned by `choice()`, the method `behavStop()` is invoked.

The second case is the one in which some of the process terms involved in the choice do not start with an action prefix operator. If one of these process terms is `stop`, then nothing has to be added for it in the `ChAct` array and the `switch-case` statement, because the method `behavStop()` is selected by default whenever the other involved process terms cannot be selected. If instead one of these process terms is a nested choice, then a flattening of the nested choice takes place during the translation. This means that the `ChAct` array and the `switch-case` statement for the outer choice are extended in order to include all the alternative starting actions that are contained in the inner choice. The event in which one of the process terms involved in the choice is a behavioral invocation cannot happen, because a behavioral invocation can only occur within an action prefix operator.

3.6 Example of Thread Behavior Generation

In order to illustrate all the features of the proposed approach, we conclude by showing the translation into a Java thread of the architectural element type described with PADL in Sect. 2.1. A more complex and realistic example can be found at `http://www.sti.uniurb.it/bonta/java_audio_proc/`.

Below we exhibit the *defining behavior* section automatically generated for the `ThreadElem`-derived class `Checker_Type`, which comprises the method `run()` along with the three behavioral methods associated with the three behavioral equations of the considered architectural element type:

```
public void run() {
  act_Checker_Type = new IBT_Checker_Type();
  // no ET instantiation as there are
  // no architectural interactions and no asynchronous interactions
  behavEqNext(0, null);
  while (behavEqCall());
}
public void Getting_Value() {
  Integer received_value;
  try {
    Object obj[] = get_value.receive();
    received_value = (Integer)obj[0];
  } catch (SyncException e) {}
  behavEqNext(1, new Object[] {received_value});
}
```

```
public void Checking_Value(Integer received_value) {
  switch(
    choice(new ChAct[] {
      new ChAct(received_value.intValue() == expected_value, 1, 1.0),
      new ChAct(received_value.intValue() != expected_value, 1, 1.0)
    })
  )
  {
    case 0:
      act_Checker_Type.prepare_to_forward();
      try {
        forward_value.send(new Object[] {received_value})
      } catch (SyncException e) {}
      behavStop();
      break;
    case 1:
      act_Checker_Type.prepare_to_print();
      behavEqNext(2, new Object[] {received_value});
      break;
    default:
      behavStop();
      break;
  }
}
public void Printing_Value(Integer received_value) {
  act_Checker_Type.print_value(received_value);
  behavEqNext(0, null);
}
```

The three behavioral methods are numbered `0`, `1`, and `2` in the array built by the method `behavEqList()` invoked from within the constructor of the Java class `Checker_Type`. Therefore, these are the indices that occur above in the invocations of method `behavEqNext()`.

The interactions `get_value` and `forward_value` are translated into invocations of the methods `receive()` and `send()`, respectively, which are defined in the corresponding `Ports`. Since such methods can throw exceptions of class `SyncException` defined in the package `Sync`, their invocation must be controlled using a `try-catch` statement. The class `SyncException` is the superclass of `UnattachedPortException` and `AsyncPortNotReadyException` mentioned in Sect. 3.4, which need to be handled with suitable methods to be manually defined in the ET stub. Since none of `get_value` and `forward_value` is architectural or asynchronous, no exception handler is needed for them, which explains why no ET stub is instantiated by the method `run()`.

The three methods `prepare_to_forward()`, `prepare_to_print()`, and `print_value()` related to the internal actions are invoked on the object `act_Checker_Type`, which is of class `IBT_Checker_Type`. This object is declared as a protected member within the class `Checker_Type` and is then instantiated within the method `run()` above. The class `IBT_Checker_Type` is defined as follows:

```
class IBT_Checker_Type {
  // ADD CLASS MEMBER DECLARATIONS IF NEEDED
  IBT_Checker_Type() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }
  void prepare_to_forward() {
    // FILL IN THE METHOD BODY
  }
  void prepare_to_print() {
    // FILL IN THE METHOD BODY
  }
  void print_value(Integer received_value) {
    // FILL IN THE METHOD BODY
  }
}
```

The methods associated with the three internal actions will have to be manually filled in by the developer based on the semantics of the internal actions themselves. The developer is also allowed to fill in the body of the constructor of the class `IBT_Checker_Type` and to add member declarations whenever needed.

4 Preservation of Architectural Properties

PADL is equipped with a component-oriented technique based on equivalence checking for verifying the freedom from architectural mismatches [1]. These are the malfunctionings that arise when assembling together several components that are correct if considered in isolation. More precisely, the class of properties dealt with by the technique – which includes for instance deadlock freedom – is characterized by three constraints. First, the properties can only be concerned with the interactions, as they are the actions through which the components communicate. Second, for each property $\mathcal{P}$ in the class, there must exist a weak equivalence $\approx_{\mathcal{P}}$ coarser than $\approx_{\mathrm{B}}$ (weak bisimulation [13]) that preserves $\mathcal{P}$ – it never equates two process terms such that one of them satisfies $\mathcal{P}$ while the other does not – and is a congruence with respect to the static process algebraic operators. Third, the (action-based) temporal logic in which the properties of the class are expressed cannot allow the negation to be freely used.

An important issue is to guarantee that the properties proved at the architectural level are then preserved at the code level. Since we have taken an approach based on automatic code generation, property preservation should be achieved by construction. In other words, the translation from PADL to Java illustrated before should have been defined in a way that ensures the property preservation. This is what we are going to investigate in this section.

4.1 Code Generated for the Thread Management

The code for handling the threads is completely generated in an automatic way by means of the package `Sync`. As far as the system topology is concerned, this

is built in the `Architecture`-derived class in the same way as prescribed by the second section of the PADL specification.

As far as the thread coordination is concerned, both PADL and `Sync` adhere to the same synchronization model. On the PADL side, each interaction is given three qualifiers: output vs. input, uni vs. and vs. or, synchronous vs. asynchronous. Each interaction is then translated into an invocation of the method `send()` or `receive()` defined in the corresponding `Port`, depending on whether it is an output or an input interaction, respectively. Additionally, the kind of this `Port` (uni vs. and vs. or, synchronous vs. asynchronous) is the same as that of the interaction.

As a consequence, the code generated for managing the threads cannot infringe the preservation of the architectural properties, up to the methods for handling the exceptions raised by architectural and asynchronous interactions.

4.2 Code Generated for the Behavioral Equations

Each behavioral equation occurring in the PADL description of an architectural element type is translated into a behavioral method of the corresponding `ThreadElem`-derived class. The translation proceeds by induction on the syntactical structure of the process term on the right-hand side of the behavioral equation, based on the operators that occur in such a process term. The way in which the translation is carried out, together with the way in which the thread execution flow proceeds according to the order established by the invocations of the behavioral equations, ensures the preservation of the process algebraic semantics, up to the methods related to the internal actions.

4.3 Code Provided for Filling in the Stubs

In conclusion, the preservation of the architectural properties critically depends on the way in which the developer manually fills in the IBT and ET stubs. Here we shall consider only the IBT stubs, as the ET stubs can be treated similarly.

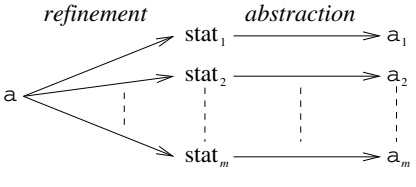


Fig. 2. Internal action refinement and related statement abstraction

In order to be able to reason about the architectural property preservation, we have to compare the internal actions and the corresponding sequences of Java statements on the same process algebraic ground. As shown in Fig. 2, the Java statements into which an internal action is refined during the translation process can be abstractly viewed as fresh actions. The following theorem provides

a sufficient condition for ensuring the preservation of an architectural property of the considered class. Below, we denote by _/_ the hiding operator and by $\simeq_B$ the observational congruence of [13].

Theorem 1. *Let* T *be the process algebraic description of the behavior of a thread and let* a *be an internal action occurring in* T*. Let* a_1, a_2, ..., a_m *be the fresh actions abstracting the statements into which* a *is translated and let* T' *be the process algebraic description of the behavior of the thread obtained from* T *by replacing every occurrence of* a._ *with* $a_1.a_2. \ldots .a_m$._*. Let* H *be the set of internal actions occurring in* T *or* T'*. Whenever* T *satisfies* $\mathcal{P}$ *and* a.stop / H $\simeq_B$ $a_1.a_2. \ldots .a_m$.stop / H*, then* T' *satisfies* $\mathcal{P}$ *as well.*

Proof. Since $\simeq_B$ *is a congruence with respect to all the process algebraic operators, from* a.stop / H $\simeq_B$ $a_1.a_2. \ldots .a_m$.stop / H *it follows that* T / H $\simeq_B$ T' / H*, hence* T / H $\approx_B$ T' / H*. Since* $\mathcal{P}$ *must be equipped with a weak equivalence* $\approx_{\mathcal{P}}$ *coarser than* $\approx_B$*, it follows that* T / H $\approx_{\mathcal{P}}$ T' / H*. Since* T *satisfies* $\mathcal{P}$*,* $\approx_{\mathcal{P}}$ *preserves* $\mathcal{P}$*, and* $\mathcal{P}$ *can only make assertions about the interactions (which do not belong to* H*), it follows that* T' *satisfies* $\mathcal{P}$ *as well.*

Note that in the theorem above it is not necessarily the case that all of the actions a_1, a_2, ..., a_m associated with the Java statements provided by the software developer belong to H. As an example, one of such actions may correspond to an invocation of `send()`/`receive()` or of a behavioral method. Fortunately, both cases are prevented from occurring by the fact that the `Port` instances – which contain methods `send()` and `receive()` – and the `ThreadElem`-derived class instances – which contain the behavioral methods – are not visible within the stubs.

We conclude by providing some guidelines that the developer should follow when filling in the stubs in order to preserve the architectural properties:

- No `synchronized` methods should be defined within the stubs, so that methods like `wait()` and `notify()` – which could not be abstracted through internal actions – cannot occur within the stubs.
- No further thread should be created within the stubs, as this would have an observable impact on the system topology and the thread coordination.
- There should be no variables/objects that are visible from several stub classes. This means that all the data shared by several threads should be exchanged only through suitable components of the package `Sync`.
- In the stub method associated with the first internal action following an invocation of the method `send()` (resp. `receive()`), every object that has been passed in that invocation should be copied, with all the stub methods associated with the subsequent internal actions working on that copy of the object. This avoids interferences among threads stemming from the fact that the method `send()` always keeps a reference to the passed objects – so that it can be defined in the package `Sync` in a way that supports arbitrarily many parameters of arbitrary types – and such objects may be modified by the stub method associated with some internal action.

– All the exceptions that can be raised when executing a stub method should be caught or prevented from being raised inside the stub method itself.
– Non-terminating statements should be avoided within the stub methods.

5 Conclusion

In this paper we have addressed the problem of automatically generating multithreaded programs from formal architectural descriptions, in a way that builds on [4] and preserves the properties proved at the architectural design level. Since the preservation of the architectural properties critically depends on the way in which certain methods are manually filled in by the developer, we have provided some guidelines that should be followed when completing such methods.

Concerning related work that addresses both architecture-driven code generation and architectural property preservation, we have ArchJava and C2SADEL. ArchJava [2] is an extension of Java aiming at the unification of software architecture with implementation, in order to ensure that the implementation conforms to the architectural description with respect to communication integrity. According to this property, each component in the implementation may only communicate directly with the components to which it is connected in the architecture. Our approach differs from ArchJava in several ways. First, it does not extend Java, but generates Java code from process algebraic architectural descriptions. In our approach the developer is then required to fill in some stubs to complete the code for the behavior of the threads, thus giving a certain degree of flexibility. The price to be paid is that the guidelines may be violated, whereas a similar situation is not possible in ArchJava. Second, our approach focuses on the issue of correct thread coordination with respect to a rich synchronization model implemented in `Sync`. This guarantees a property that is even stronger than communication integrity: Implementation threads directly communicate only with the threads they are connected to in the architectural description, in the way prescribed by the architectural description itself with respect to the communication mode (synchronous, asynchronous, asymmetric) and the communication multiplicity (uni-uni, and-uni, or-uni). Third, our approach is more general in the sense that it considers the preservation of a class of architectural properties related to the system behavior, rather than a specific static property. This is formalized through a theorem and a set of guidelines that the developer should follow when filling in the stubs.

C2SADEL [12] is an ADL tied to the C2 style, which combines the usual architectural concepts with type theory. Type checking is used to analyze the architectural descriptions for consistency by unifying corresponding operations required and provided by different components. Moreover, Java code can be automatically generated from C2SADEL descriptions. Since type checking is a static analysis technique, while the architectural properties on which we focus are dynamic and concerned with a rich synchronization model, the differences between our approach and C2SADEL are similar to those above between our approach and ArchJava.

For the future we plan to make some experiments to assess on the field the effectiveness of the proposed framework as well as the performance of the generated code. We also would like to investigate the applicability of our approach to C2SADEL, in order to take advantage of both type checking and behavioral analysis from the architectural level to the code level. Moreover, it would be interesting to combine our approach with ArchJava – by generating ArchJava code instead of Java code – in order to exploit their complementary strengths.

References

1. A. Aldini and M. Bernardo, *"On the Usability of Process Algebra: An Architectural View"*, to appear in Theoretical Computer Science.
2. J. Aldrich, C. Chambers, and D. Notkin, *"ArchJava: Connecting Software Architecture to Implementation"*, in Proc. of the *24th Int. Conf. on Software Engineering (ICSE 2002)*, IEEE-CS Press, pp. 187-197, Orlando (FL), 2002.
3. R. Allen and D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:213-249, 1997.
4. M. Bernardo and E. Bontà, *"Generating Well-Synchronized Multithreaded Programs from Software Architecture Descriptions"*, in Proc. of the *4th Working IEEE/IFIP Conf. on Software Architecture (WICSA 2004)*, IEEE-CS Press, pp. 167-176, Oslo (Norway), 2004.
5. M. Bernardo, P. Ciancarini, and L. Donatiello, *"Architecting Families of Software Systems with Process Algebras"*, in ACM Trans. on Software Engineering and Methodology 11:386-426, 2002.
6. C. Canal, E. Pimentel, and J.M. Troya, *"Compatibility and Inheritance in Software Architectures"*, in Science of Computer Programming 41:105-138, 2001.
7. D. Garlan, *"Formal Modeling and Analysis of Software Architectures: Components, Connectors, and Events"*, in *Formal Methods for Software Architectures*, LNCS 2804:1-24, 2003.
8. R. Guimarães and W. Borelli, *"An Automatic Java Code Generation Tool for Telecom Distributed Systems"*, in Proc. of the *Int. Conf. on Software, Telecommunications and Computer Networks (SOFTCOM 2002)*, Split (Croatia), 2002.
9. P. Inverardi and S. Uchitel, *"Proving Deadlock Freedom in Component-Based Programming"*, in Proc. of the *4th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2001)*, LNCS 2029:60-75, Genova (Italy), 2001.
10. P. Inverardi, A.L. Wolf, and D. Yankelevich, *"Static Checking of System Behaviors Using Derived Component Assumptions"*, in ACM Trans. on Software Engineering and Methodology 9:239-272, 2000.
11. J. Magee and J. Kramer, *"Concurrency: State Models & Java Programs"*, Wiley, 1999.
12. N. Medvidovic, D.S. Rosenblum, and R.N. Taylor, *"A Language and Environment for Architecture-Based Software Development and Evolution"*, in Proc. of the *21st Int. Conf. on Software Engineering (ICSE 1999)*, IEEE-CS Press, pp. 44-53, Los Angeles (CA), 1999.
13. R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989.
14. D.E. Perry and A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992.
15. M. Shaw and D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, 1996.

Prioritized and Parallel Reactions in Shared Data Space Coordination Languages

Nadia Busi and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Piazza di Porta S. Donato 5, I-40127 Bologna, Italy
{ busi, zavattar}@cs.unibo.it

Abstract. Reactive programming has been added to the traditional Linda programming style in order to deal with the dynamic aspects of those applications in which it is important to observe modifications of the environment which occur quickly. Reactive programming is embedded in shared data spaces by defining the *observable* entities and the corresponding *reactions* that usually are processes possibly executing coordination primitives. Typical observable entities are the presence in the space of a datum (state-based) or the execution of a particular primitive (event-based). In this paper we consider different models of reaction execution adopted in the coordination literature so far, namely the *prioritized* model (used e.g. in MARS [3], TuCSoN [8] and LIME [9]) and the *parallel* execution model (considered e.g. in JavaSpaces [13], TSpaces [14] and WCL [11]). Prioritized reactions are usually implemented at server-side as during their execution no other coordination primitives can take place; *parallel* reactions, more suited when reactions are executed at client-side, are executed in parallel with the other processes in the system (thus other coordination primitives can interleave with the reactions). Using a process algebraic setting we perform a rigorous investigation of the relative advantages and disadvantages of these two models for reaction execution.

1 Introduction

The development of Linda-like coordination languages [4] for use over wide-area or mobile networks has driven the consideration of new primitives being added to allow new styles of coordination. One set of such primitives are those that allow reactive-programming, essentially allowing a program to be notified on the availability of a tuple or the execution of a particular primitive. Examples of Linda-like coordination languages including such reactive mechanisms are MARS [3], TuCSoN [8], LIME [9], JavaSpaces [13], TSpaces [14] and WCL [11].

Similarly, in all these languages reactions are processes that possibly execute coordination primitives; the difference is in *what* can activate the reactions and *how* the reactions are executed. We call *observable entities* those entities that can activate a reaction; we call reaction *execution model* the rules governing the execution of the activated reactions.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 204–219, 2005.

In this paper we focus on the two different reaction execution models considered in the coordination literature so far. In the *prioritized* model the reactions are executed (activated and completed) immediately after the observation occurs. While reactions are under execution, the other processes in the system cannot access the data space. This model is usually adopted when reactions are executed on server-side, that is within the implementation of the shared data space. On the contrary, in the *parallel* execution model, the reactions are only activated immediately after the observation, and they are executed in parallel with the other processes in the system; in this way, reactions do not block the execution of the other processes in the system. This model is particularly suited when the reactions are executed on client-side, i.e. when the shared data space implementation simply invokes the reactions that are actually executed by the devices hosting the processes accessing the data space. The prioritized model is adopted in MARS [3] and TuCSoN [8]. LIME [9] supports both the reaction execution models. JavaSpaces [13], TSpaces [14] and WCL [11] adhere to the parallel model.

In order to compare these two approaches, we consider two Linda-like process calculi extended with reactive mechanisms that differ for the reaction execution model. As far as the parallel execution model is concerned, we consider the calculus and the results proved in a previous paper [1] where a calculus modeling the $notify$ primitive of JavaSpaces has been introduced. More precisely, the primitive $notify(a, R)$ is analysed that installs a listener responsible for activating the execution of the reaction R everytime a new instance of a datum a is introduced in the shared data space. As far as the prioritized execution model is concerned, we introduce a new calculus named `ReactLinda` which is essentially the adaptation of the calculus in [1] to the prioritized model. The new calculus has only two differences with respect to the previous one; the first difference is the name of the reactive coordination primitive ($reactsTo(a, R)$ instead of $notify(a, R)$); the second difference is in the operational semantics: in `ReactLinda` the presence of uncompleted reactions disables the other processes in the system from executing coordination primitives.

We use the two process calculi in order to investigate the possibility to encode one mechanism into the other one. Intuitively, we have that the parallel execution model can be encoded in the prioritized one. The idea is to delegate the execution of the reactions to parallel processes that are blocked willing to consume some activation data that are produced by the prioritized reactions. More formally, we can consider an encoding function $[\![\ ]\!]$ such that

$$[\![notify(a, R)]\!] \quad = \quad reactsTo(a, out(a_R)) \mid !in(a_R).R$$

where a_R is the name used for the activation datum, and $!in(a_R).R$ is the parallel composition of an unbonded amount of processes willing to consume the activation datum and subsequently execute the reaction R.

On the contrary, the modeling of $reactsTo$ in terms of $notify$ is clearly more complex because it is necessary to block the execution of the processes while reactions are in execution. This can be realized exploiting locks that forbid the

processes from accessing the data space. Following this approach, the problem is how to detect the termination of the reactions in order to remove the lock; indeed, the number of listeners that observe a specific event is unpredictable (it depends on the run time behaviour of the system) thus also the number of reactions that are triggered by that event.

This informal discussion shows (some of) the difficulties that are encoutered while trying to encode the prioritized into the parallel reaction execution model. Using the process calculi we are able to formally prove an interesting impossibility result. This result is based on the decidability of properties such as process *termination* (i.e. existence of a finite completed computation) and process *divergence* (i.e. existence of an infinite computation). In [1] we proved that in the calculus with $notify$ termination is undecidable while divergence is decidable. In `ReactLinda`, on the contrary, both termination and divergence are undecidable.

As a consequence we have the following impossibility result: there exists no computable encoding of the prioritized reaction execution model into the parallel model that preserves divergence. From a theoretical point of view, this proves that the prioritized model is strictly more expressive than the parallel model. On the other hand, from a practical point of view, the undecidability of divergence (and termination) is a rather negative result. In fact, in the prioritized reaction execution model, reactions are expected to complete with neither deadlock nor divergence otherwise the rest of the system is blocked indefinitely.

In light of this observation about the practical usage of prioritized reactions, we continue our investigation trying to point out a significant decidable fragment for `ReactLinda`. In this perspective, we eliminate the unique explicit infinite operator, i.e. the replication operator, from `ReactLinda` and we investigate the decidability of termination and divergence in the obtained fragment, that we call `ReactLinda-!`.

The elimination of the replication operator does not necessarily brings to a finite calculus as reactive programming permits to simulate replication. For instance, the process $reactsTo(a, out(a).P).out(a)$ activates an unbounded amount of copies of process P, simply by permitting to a reaction to perform an *out* operation that triggers another instance of the reaction itself.

Even if the replication operator can be simulated following this approach, `ReactLinda-!` is more decidable than `ReactLinda`. Indeed, in `ReactLinda-!` both termination and divergence turn out to be decidable (while they are both undecidable in `ReactLinda`). This result is proved resorting to a Petri net semantics.

This theoretical result is of interest also from a practical viewpoint. Indeed, `ReactLinda-!` can be used as a reference model for prioritized reactions where both termination and divergence can be decided. In languages such as MARS and LIME restrictions are imposed to the reactive model in order to guarantee the completion of reactions. For instance, in MARS reactions cannot trigger other reactions, while in LIME blocking operations cannot be used inside reactions. Our decidability results open an alternative approach which can be based on the verification of the reactive programs before their actual execution. In the case the reactions can deadlock or diverge the run time system can decide to avoid their execution.

The paper is structured as follows. In Section 2 we present the syntax and the operational semantics of the considered calculi. In Section 3 we prove the undecidability results while in Section 4 the decidability results are discussed. Finally, Section 5 reports some conclusive remarks. Due to space limitation the formal proofs of theorems are not reported, but the proof techniques are carefully detailed.

2 The Process Calculi

In this Section, we introduce the process calculus `ReactLinda` based on the prioritized reaction execution model discussed in the Introduction. It is obtained as the adaptation to the prioritized model of the calculus introduced in [1] based on the parallel model. Syntactically, the main difference is that in the new calculus we separate processes, data, listeners and reactions in four different syntactic cathegories while in [1] they are all treated as parallel processes.

By borrowing typical techniques from the tradition of process calculi for concurrency (e.g., Milner's CCS [6]), a process (as well as a reaction) is described as a term of an algebra where the basic actions are the Linda coordination primitives *in*, *rd* and *out*, or the reactive operation *reactsTo*. The data space is represented by the multiset of the data actually available. The listeners are modeled with pairs (a, R) where a is the datum whose production is observed, and R is the corresponding reaction.

Formally, we consider a denumerable set of names for data, called *Data*, ranged over by a, b, ... A data space is a multiset of *Data*; formally the possible data spaces, ranged over by DS, DS', $\oplus$ are taken from $DataSpace = \mathcal{M}(Data)$. In the following we use $\oplus$ to denote multiset union, $\bigoplus_i M_i$ to denote the multiset union of the indexed multisets M_i, and we use a to denote also the singleton $\{a\}$.

Reactions are finite programs that can only execute input, read and output operations. We follow the LIME assumption according to which reactions cannot contain reactive statements. The set *Reac* of reactions, ranged over by R, R', ..., is the set of terms generated by the following grammar:

$$R \quad ::= \quad \mathbf{0} \;\mid\; \mu.R \;\mid\; R|R \qquad\qquad \mu \quad ::= \quad out(a) \;\mid\; in(a) \;\mid\; rd(a)$$

where μ denotes an instance of one of the (non-reactive) coordination primitives. In the following we use $\prod_i R_i$ to denote the parallel composition of the indexed reactions R_i or $\mathbf{0}$ if the set of index is empty.

The reaction $\mathbf{0}$ represents the empty program; $\mu.R$ is a reaction that starts executing the primitive μ then becomes R; $R|R'$ is the parallel composition of the two reactions R and R'.

Listeners are formally modeled as follows. Let *Lst*, ranged over by L, L', ..., be the set of multisets on the cartesian product between *Data* and *Reac*, namely $Lst = \mathcal{M}(Data \times Reac)$. The listener (a, R) is responsible for activating the reaction R on production of datum a.

Table 1. The operational semantics of `ReactLinda`

$$(\texttt{inP}) \quad [in(a).P|Q, a \oplus DS, L, \mathbf{0}] \rightarrow [P|Q, DS, L, \mathbf{0}]$$

$$(\texttt{rdP}) \quad [rd(a).P|Q, a \oplus DS, L, \mathbf{0}] \rightarrow [P|Q, a \oplus DS, L, \mathbf{0}]$$

$$(\texttt{inR}) \quad [P, a \oplus DS, L, in(a).R|R'] \rightarrow [P, DS, L, R|R']$$

$$(\texttt{rdR}) \quad [P, a \oplus DS, L, rd(a).R|R'] \rightarrow [P, a \oplus DS, L, R|R']$$

$$(\texttt{rct}) \quad [reactsTo(a, R').P|Q, DS, L, \mathbf{0}] \rightarrow [P|Q, DS, L \oplus (a, R'), \mathbf{0}]$$

$$(\texttt{outP}) \quad \frac{(b, R) \in L \text{ implies } b \neq a}{\begin{array}{c}[out(a).P|Q, DS, \bigoplus_i (a, R_i) \oplus L, \mathbf{0}] \rightarrow \\ [P|Q, a \oplus DS, \bigoplus_i (a, R_i) \oplus L, \prod_i R_i]\end{array}}$$

$$(\texttt{outR}) \quad \frac{(b, R) \in L \text{ implies } b \neq a}{\begin{array}{c}[P, DS, \bigoplus_i (a, R_i) \oplus L, out(a).R|R'] \rightarrow \\ [P, a \oplus DS, \bigoplus_i (a, R_i) \oplus L, \prod_i R_i|R|R']\end{array}}$$

$$(\texttt{cong}) \quad \frac{P \equiv P'' \quad P' \equiv P''' \quad R \equiv R'' \quad R' \equiv R''' \quad [P, DS, L, R] \rightarrow [P', DS', L', R']}{[P'', DS, L, R''] \rightarrow [P''', DS', L', R''']}$$

The set $Proc$ of processes, ranged over by P, P', ... is defined by the following grammar:

$$P \quad ::= \quad \mathbf{0} \;\mid\; \mu.P \;\mid\; reactsTo(a, R).P \;\mid\; P|P \;\mid\; !P$$

where μ is defined as above for reactions. Processes extend reactions with the $reactsTo(a, R)$ prefix and the replication operator $!P$. In the following we use $\prod_i P_i$ to denote the parallel composition of the indexed processes P_i or the process $\mathbf{0}$ if the set of index is empty.

We will reason upto a structural congruence relation used for rearranging parallel composed processes (as well as reactions), for abstracting away from the empty program and for (un)folding the replicated processes. Formally, let $\equiv$ be the least congruence on processes and reactions such that

$$P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R \qquad P|\mathbf{0} \equiv \mathbf{0} \qquad !P \equiv P|!P$$

The set of configurations $Conf = Proc \times DataSpace \times Lst \times Reac$ is ranged over by $[P, DS, L, R]$ where P denotes the active processes, DS the current

data space, L the set of actually installed listeners, and R represents the active reactions. We assume that initially the configurations start with an empty data space, without listeners installed, and without active reactions.

The semantics is defined in terms of a reduction relation $[P, DS, L, R] \rightarrow [P', DS', L', R']$ used to denote that the configuration on the left hand side can evolve, performing one step, to the configuration on the right hand side. The reduction semantics is defined as the least relation on configurations satisfying the axioms and rules in the Table 1.

The axioms `(inP)` and `(rdP)` model the execution of the *in* and *rd* operations performed by processes: the former removes a datum from the space, while the latter simply checks the presence of a datum leaving the data space unchanged. It is important to note that in both the axioms the reactions are assumed to be $\mathbf{0}$; in this way processes can perform *in* and *rd* operations only if there are no active reactions. The same assumption is made in the rules `rct` and `outP` modeling the other two operations that processes can perform. The axioms `(inR)` and `(rdR)` are the corresponding ones for reactions. The axiom `(rct)` models the execution of a $reactsTo(a, R)$ primitive; its effect is the addition of a listener.

The rule `(outP)` define the semantics of the $out(a)$ operation. The premise of the rule guarantees that all the interested listeners are taken into account. The rule `(outR)` is the corresponding one for output operations performed by reactions. Finally, rule `(cong)` is used to ensure that structurally congruent terms have the same operational semantics.

We denote with `ReactLinda-!` the fragment of `ReactLinda` obtained by removing the replication operator.

As described in the Introduction, we will focus on the decidability of two typical properties, *termination* (i.e. the existence of a finite completed computation) and *divergence* (i.e. the existence of an infinite computation). Formally, let $[P, DS, L, R] \not\rightarrow$ denote that there exists no $[P', DS', L', R']$ such that $[P, DS, L, R] \rightarrow [P', DS', L', R']$. We say that P terminates, denoted with $P\downarrow$, if there exist $[P_1, DS_1, L_1, R_1], \ldots, [P_n, DS_n, L_n, R_n]$ such that $[P, \emptyset, \emptyset, \mathbf{0}] \rightarrow [P_1, DS_1, L_1, R_1] \rightarrow \ldots \rightarrow [P_n, DS_n, L_n, R_n] \not\rightarrow$. On the other hand, we say that P diverges, denoted with $P\uparrow$, if for any natural number i there exists $[P_i, DS_i, L_i, R_i]$ such that $[P, \emptyset, \emptyset, \mathbf{0}] = [P_0, DS_0, L_0, R_0]$ and $[P_i, DS_i, L_i, R_i] \rightarrow [P_{i+1}, DS_{i+1}, L_{i+1}, R_{i+1}]$.

3 The Undecidability Results

In this Section we discuss the undecidability of termination and divergence for `ReactLinda`.

We adopt the following technique: we reduce the decidability of termination (resp. divergence) to the decidability of termination (resp. divergence) on Random Access Machines, a well known Turing powerful formalism.

A Random Access Machines (RAM) [12] is composed of a finite set of registers, that can hold arbitrary large natural numbers, and by a program, that is a

sequence of simple numbered instructions, like arithmetical operations (on the contents of registers) or conditional jumps.

It is not restrictive to assume that the registers $r_1, \ldots, r_n$ are initially empty. The execution of the program begins with the first instruction and continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached. If the program terminates, the result of the computation is the contents of the registers.

In [7] it is shown that the following two instructions are sufficient to model every recursive function:

- $Succ(r_j)$: adds 1 to the content of register r_j;
- $DecJump(r_j, s)$: if the content of register r_j is not zero, then decreases it by 1 and go to the next instruction, otherwise jumps to instruction s.

We start presenting an encoding of RAMs in `ReactLinda`. This encoding is inspired by a previous encoding described in [1] based on the $notify(a, R)$ primitive; the main difference is that the reactions do not perform blocking operations. Indeed, under the prioritized model a blocked reaction blocks also the rest of the system.

The encoding is *nondeterministic* as it introduces some extra infinite computations; nevertheless, it is ensured that a RAM terminates if and only if the corresponding encoding has a terminating computation. As termination cannot be decided in Turing equivalent formalisms, the same holds also for `ReactLinda`.

The encoding implements nondeterministically *DecJump* operations: two possible behaviours can be chosen, the first is valid if the tested register is not zero, the second otherwise. If the wrong choice is made, the computation is ensured to be infinite; in this case, we cannot say anything about the corresponding RAM. Nevertheless, if the computation terminates, it is ensured that it corresponds to the computation of the corresponding RAM. Conversely, any computation of the RAM is simulated by the computation of the corresponding encoding in which no wrong choice is performed.

Given the RAM with program PRG (composed by the instructions $I_1 \ldots I_k$) and registers $r_1 \ldots r_n$, the corresponding encoding $[\![PRG]\!]$ is defined in Table 2. The basic idea underlying this encoding is to represent instructions with replicated processes, the program counter with a datum that activate the corresponding instruction, and the content of each register r_j with a corresponding number of r_j data. Namely, the instruction I_i is modeled with the process $!in(p_i).[\![I_i]\!]$ where p_i is the program counter tuple and $[\![I_i]\!]$ is a process responsible for updating the register and the program counter. Before starting the computation, a listener is installed executing $reactsTo(div, out(div))$; this listener has the ability to activate an infinite computation. More precisely, the infinite computation is started in case a *loop* datum is consumed by the initially spawn process $in(loop).out(div)$. Finally, the actual computation is started emitting the program counter tuple p_1.

Table 2. Termination preserving encoding of RAMs in `ReactLinda`

$$
\begin{array}{ll}
[\![PRG]\!] & = \prod_{i\in 1\ldots k} !in(p_i).[\![I_i]\!] \mid reactsTo(div, out(div)).out(p_1) \\
 & \mid\ !in(inc).INC \mid !in(dec).DEC \mid in(loop).out(div) \\
[\![i : Succ(r_j)]\!] & = out(r_j).reactsTo(zero_j, out(inc)).out(p_{i+1}) \\
[\![i : DecJump(r_j, s)]\!] & = out(c_i) \\
 & \mid\ in(c_i).out(loop).in(r_j).in(loop). \\
 & \quad reactsTo(zero_j, out(dec)).out(p_{i+1}) \\
 & \mid\ in(c_i).out(zero_j).in(zero_j).out(p_s) \\
\text{where:} & \\
INC & = out(loop).in(match).in(loop) \\
DEC & = out(match)
\end{array}
$$

Every time an increment (resp. a decrement) on the register r_j is performed, a new listener $(zero_j, out(inc))$ (resp. $(zero_j, out(dec))$) is spawn. The presence of these listeners permits to check if the actual content of a register r_j is zero by verifying if the occurrences of $(zero_j, out(inc))$ corresponds to those of $(zero_j, out(dec))$.

An increment instruction on r_j simply increments the number of the occurrences of the datum r_j and install a new listener $(zero_j, out(inc))$. When a *DecJump* instruction is executed, a nondeterministic choice between the two possible branches of the instruction occurs. The choice is modeled putting in parallel the two branches that compete for consuming the datum c_i. We now analyse the computation in case the wrong choice is done. There are two cases to analyse: (i) a decrement on a register containing zero, (ii) a jump for zero on a non-empty register.

In the case (i), $out(loop).in(r_j).in(loop).reactsTo(zero_j, out(dec)).out(p_{i+1})$ is activated with no r_j data available. Thus, the program produces the datum *loop* and blocks trying to execute $in(r_j)$. The produced datum *loop* will be consumed by the process $in(loop).out(div)$ thus an infinite computation is started.

In the case (ii), the process $out(zero_j).in(zero_j).out(p_s)$ is activated when there are more occurrences of the listener $(zero_j, out(inc))$ than those of the listener $(zero_j, out(dec))$. When the datum $zero_j$ is emitted, its production is notified to the listeners; then the corresponding data *inc* and *dec* are produced. These data activates corresponding processes INC and DEC. Each DEC emits a datum *match* while each INC produces a datum *loop*, and requires a *match* datum to be consumed before removing the emitted *loop*. As there are more INC processes than DEC, one of the processes INC will block waiting for an unavailable *match* datum; thus it will not consume its corresponding *loop*. As before, an infinite computation will be activated.

The correspondence between the RAMs and their modeling in ReactLinda is stated by the following Theorem. The undecidability of termination in the calculus ReactLinda is a trivial corollary of this Theorem.

Theorem 1. *Let us consider a RAM with program PRG (composed by the instructions $I_1 \dots I_k$) and registers $r_1 \dots r_n$, and the corresponding encoding $[\![PRG]\!]$ in* ReactLinda *defined in Table 2. We have that the RAM terminates its computation if and only if $[\![PRG]\!] \downarrow$.*

Now, we prove that also divergence is undecidable in ReactLinda. In order to prove this result we present how to encode RAMs preserving divergence, i.e. a RAM does not terminate if and only if the corresponding encoding has an infinite computation.

The new encoding is similar to the encoding above because it is nondeterministic: the branch of *DecJump* is selected nondeterministically independently of the actual contents of the registers. The difference is that in case the wrong branch is selected the computation is guaranteed to block. Thus, the encoding has an infinite computation if and only if the RAM does not terminate.

Table 3. Divergence preserving encoding of RAMs in ReactLinda

$[\![PRG]\!]$	$= \prod_{i \in 1 \dots k} !in(p_i).[\![I_i]\!] \mid out(p_1)$
$[\![i : Succ(r_j)]\!]$	$= out(r_j).reactsTo(zero_j, INC).out(p_{i+1})$
$[\![i : DecJump(r_j, s)]\!]$	$= out(c_i)$
	$\mid in(c_i).in(r_j).reactsTo(zero_j, DEC).out(p_{i+1})$
	$\mid in(c_i).out(zero_j).in(zero_j).out(p_s)$
where:	
INC	$= in(match)$
DEC	$= out(match)$

We overload the notation $[\![PRG]\!]$ used also for the new encoding reported in Table 3. If the decrement branch is wrongly selected all the computation blocks because the input operation $in(r_j)$ cannot be executed; in case a wrong jump is selected, the computation blocks because one of the INC reactions will not terminate due to the impossibility to execute the $in(match)$ operation.

The following theorem states that the new RAM encoding preserves divergence, thus divergence is undecidable in the calculus ReactLinda.

Theorem 2. *Let us consider a RAM with program PRG (composed by the instructions $I_1 \dots I_k$) and registers $r_1 \dots r_n$, and the corresponding encoding $[\![PRG]\!]$ in* ReactLinda *defined in Table 3. We have that the RAM does not terminate its computation if and only if $[\![PRG]\!] \uparrow$.*

4 The Decidability Results

In this Section we show that both termination and divergence are decidable in `ReactLinda-!`. We reduce termination on `ReactLinda-!` to termination on Place/Transition Petri nets. As termination is decidable on such class of Petri nets [2], we get the decidability result for termination on `ReactLinda-!`. Divergence on `ReactLinda-!` is reduced to a property similar to divergence on the class of nets corresponding to `ReactLinda-!` processes.

4.1 P/T Nets

We recall Place/Transition nets with unweigthed flow arcs (see, e.g., [10]). Here we provide a characterization of this model which is convenient for our aims.

Definition 1. *Given a set S, a* finite multiset *over S is a function $m : S \to \mathbb{N}$ such that the set $dom(m) = \{s \in S \mid m(s) \neq 0\}$ is finite. The* multiplicity *of an element s in m is given by the natural number $m(s)$. The set of all finite multisets over S, denoted by $\mathcal{M}_{fin}(S)$, is ranged over by m. A multiset m such that $dom(m) = \emptyset$ is called* empty*. The set of all finite sets over S is denoted by $\wp_{fin}(S)$.*

Given the multiset m and m', we write $m \subseteq m'$ if $m(s) \leq m'(s)$ for all $s \in S$ while $\oplus$ denotes their multiset union*: $m \oplus m'(s) = m(s) + m'(s)$. The operator $\setminus$ denotes* multiset difference*: $(m \setminus m')(s) =$ if $m(s) \geq m'(s)$ then $m(s) - m'(s)$ else 0. The* scalar product*, $j \cdot m$, of a number j with m is $(j \cdot m)(s) = j \cdot (m(s))$.*

To lighten the notation, we sometimes use the following abbreviation. If m is a multiset containing only one occurrence of an element s (i.e., $dom(m) = \{s\}$ and $m(s) = 1$) we denote m by only s. Multiset union is represented also by comma, i.e., $m, m' = m \oplus m'$. Let m be a multiset over S and m' a multiset over $S' \supseteq S$, such that $(m'(s') = 0)$ for each $s' \in S' \setminus S$; with abuse of notation, we sometimes use m in place of m', and vice versa.

Definition 2. *A P/T net is a pair (S, T) where S is the set of* places *and $T \subseteq \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S)$ is the set of* transitions*.*

Finite multisets over the set S of places are called markings*. Given a marking m and a place s, we say that the place s contains $m(s)$* tokens*.*

A P/T net is finite if both S and T are finite.

A P/T system is a triple $N = (S, T, m_0)$ where (S, T) is a P/T net and m_0 is the initial marking*.*

A transition $t = (c, p)$ is usually written in the form $c \longrightarrow p$. The marking c, usually denoted by ${}^\bullet t$, is called the preset *of t and represents the tokens to be* consumed*; the marking p, usually denoted by $t^\bullet$, is called the* postset *of t and represents the tokens to be* produced*.*

A transition t is enabled *at m if ${}^\bullet t \subseteq m$. The execution of a transition t enabled at m produces the marking $m' = (m \setminus {}^\bullet t) \oplus t^\bullet$. This is written as $m \xrightarrow{t} m'$ or simply $m \longrightarrow m'$ when the transition t is not relevant. We use σ, τ to range over sequences of transitions; the empty sequence is denoted by ε;*

let $\sigma = t_1, \ldots, t_n$, *we write* $m \xrightarrow{\sigma} m'$ *to mean the* firing sequence $m \xrightarrow{t_1} \cdots \xrightarrow{t_n} m'$.

A marking m *is* dead *if* $m \not\longrightarrow$. *We say that the P/T system* N terminates *if there exists a dead marking reachable from the intial marking* m_0, *i.e.,* $m_0 \longrightarrow {}^*m \not\longrightarrow$. *We say that the P/T system* N *is* divergent *if there exists an infinite sequence of markings* $m_1, \ldots, m_i, \ldots$ *such that* $m_i \longrightarrow m_{i+1}$ *for* $i \leq 0$.

4.2 Reducing Termination on `ReactLinda-!` to Termination on Nets

The basic idea underlying the definition of a net semantics is to decompose a process P in the (finite) multiset of its sequential subprocesses that appear at top-level (i.e., occur unguarded in P); this multiset is then considered as the marking of a P/T Petri net.

Besides processes, a `ReactLinda-!` configuration also contains data and listeners, that will be represented as places in the net as well. The execution of a computational step of the process will correspond to the firing of a transition in the corresponding net.

The main problems to face with in the costruction of a P/T net with the same behaviour of a process w.r.t. termination are the following:

- when a new datum is produced, *all* the reactions of the currently installed listeners are spawn,
- the execution of the process is blocked until the execution of all the reactions is terminated.

A faithful representation of the above described features in a Petri net requires to consider nets extended with transfer and inhibitor arcs; unfortunately, termination becomes undecidable on such an extended class of nets.

However, a closer look to the `ReactLinda-!` calculus permits to note that the sets of *reactsTo* and *out* operations that an initial process can perform are finite.[1] Hence, for a given process P,

- the number of occurrences of the *reactsTo* primitive is an upper bound to the number of occurrences of a given listener,
- the number of occurrences of the *out* primitive occurring in the initial process is an upper bound to the number of times that the reactions start executing.

The upper limit to the number of occurrences of a given listener permits to faithfully model the activation of a reaction for each active listener when an output of a datum is performed. Instead of representing n active occurrences of listener (a, R) as n tokens in place (a, R), we adopt the following representation: each pair (a, R) of potential listeners is represented by the following set of places in the net: $\{0\star(a, R), 1\star(a, R), \ldots, k\star(a, R)\}$, where k is the number of *reactsTo*

[1] Note that this finiteness property holds because the calculus does contain neither replication nor the possibility to install new listeners during the execution of reactions.

primitives occurring in the initial process; a token in place $h \star (a, R)$ models the situation where exactly h active occurrences of listener (a, R) appear in the net.

The prioritized execution of reactions cannot be faithfully modeled by a Petri net; however, the upper limit to the number of times that the reactions set can be activated permits to construct a net with the same behaviour of a `ReactLinda-!` process w.r.t. termination. The basic idea is to consider $k + 1$ copies of the subnets that model the behaviour of processes, where k is the number of *out* primitives occurring in the initial process. One of the subnets represents the "main" process, whereas the remaining subnets are used to model the execution of reactions: each time an output of datum a is performed by the main process, a yet unused subnet is filled with the marking corresponding to the reactions of the listeners waiting for the emission of a.

After activation, the computation is carried out by the subnet. According to the operational semantics of Section 2, if an $out(b)$ operation is performed the reactions of the installed listeners (b, R_i) are activated by producing tokens in those places of the subnet that correspond to the R_i.

In any moment, exactly one subnet is active. A distinct label is associated to the places of each subnet: label m is associated to the places of the subnet representing the main process, while labels $r_1, \ldots, r_k$ decorate places of the subnets used for reactions. Each subnet is equipped with three control places, decorated with the same label of the subnet. A token in place $l : running$ represents the fact that the subnet corresponding to label l is active. Places $l : unused$ and $l : finished$ are used only for subnets that represent reactions. A token in place $l : unused$ represents the fact that the subnet labelled by l has not been used yet, and can be used to execute a reaction; a token in place $l : finished$ denotes the fact that the subnet labelled by l is supposed to have successfully terminated its execution.

The behaviour of reactions is modeled in the following way: a subnet representing a reaction, and labelled with r, may choose to terminate its execution and to pass the control to the main process. If the execution was not terminated, the behaviour of the net does not correspond to the behaviour of a configuration. However, some tokens remain in some place of the subnet, corresponding to a sequential subprocess; the presence of a token in such places, together with the presence of a token in the $finished$ place, enables a transition that produces a token in a place *loop*; place *loop* has a self-loop on itself, hence a token in *loop* forbids termination of the net.

To define the net corresponding to a process, we introduce some auxiliary definitions:

Definition 3. *The set of sequential subprocesses of a process is defined inductively as follows:*

$$sub(\mathbf{0}) = \emptyset \qquad sub(\mu.P) = \{\mu.P\} \cup sub(P)$$
$$sub(reactsTo(a, R).P) = \{sub(R)\} \cup sub(P) \qquad sub(P|Q) = sub(P) \cup sub(Q)$$

The decomposition *of a process in the set of top-level subprocesses is defined as follows:*

Table 4. The transitions schemata

(in)	$l : running, l : in(a).P, a \longrightarrow l : running, l : dec(P)$
(rd)	$l : running, l : rd(a).P, a \longrightarrow l : running, l : dec(P), a$
(react)	$m : running, m : reactsTo(a, R).P, k \star (a, R) \longrightarrow m : running, m : dec(P), (k+1) \star (a, R)$
(out − M)	$m : running, m : out(a), P, r : unused, \bigoplus_{i:a_i=a} k_i \star (a_i, R_i) \longrightarrow m : dec(P), r : running, r : \bigoplus_{i:a_i=a} k_i \cdot R_i, \bigoplus_{i:a_i=a} k_i \star (a_i, R_i)$
(out − R)	$r : running, r : out(a).P, \bigoplus_{i:a_i=a} k_i \star (a_i, R_i) \longrightarrow m : dec(P), r : running, r : \bigoplus_{i:a_i=a} k_i \cdot R_i, \bigoplus_{i:a_i=a} k_i \star (a_i, R_i)$
(end − R)	$r : running \longrightarrow r : finished, m : running$
(startloop)	$r : finished, r : P \longrightarrow loop$
(loop)	$loop \longrightarrow loop$

$dec(\mathbf{0}) = \emptyset$ $\quad dec(\mu.P) = \{\mu.P\}$
$dec(reactsTo(a, R).P) = \{reactsTo(a, R).P\}$ $\quad dec(P|Q) = dec(P) \cup dec(Q)$

The set of names occurring in a process is defined as follows:

$names(\mathbf{0}) = \emptyset$ $\quad names(P|Q) = names(P) \cup names(Q)$
$names(in(a).P) = names(rd(a).P) = names(out(a).P) = \{a\} \cup names(P)$
$names(reactsTo(a, R).P) = \{a\} \cup names(R) \cup names(P)$

The set of potential listeners of a process is defined as follows:

$listeners(\mathbf{0}) = \emptyset$
$listeners(\mu.P) = listeners(P)$
$listeners(reactsTo(a, R).P) = \{(a, R)\} \cup listeners(P)$
$listeners(P|Q) = listeners(P) \cup listeners(Q)$

With $\#rct(P)$ we denote the number of occurrences of reactsTo primitives in P, with $\#out(P)$ the number of occurrences of out operations in P.

Given a process P, the set of labels for subnets is $L = \{m, r_1, \ldots, r_{\#out(P)}\}$, ranged over by l, l'. The set $Trans$ contains all the instances of the transitions schemata reported in Table 4. With $l : m$ we denote the multiset obtained by decorating each element in m with label l, i.e., for all $s \in S$, $l : m(s) = m(s)$. Axioms (in) and (rd) deal with the execution of an *in* and a *read* operation, respectively, in one of the subnets. Axiom (react), corresponds to the production of a new listener of kind (a, R): if there are k active occurrences of listener

(a, R), after the execution of the notify operation there are $k + 1$ active occurrences of listener (a, R); hence, a token is moved from place $k \star (a, R)$ to place $(k + 1) \star (a, R)$. Axiom `(out-M)` models the effect of an output operation of datum a in the main process: a not yet used subnet is chosen and marked with the tokens corresponding to the decomposition of all the reactions corresponding to listeners on datum a, and the control is passed to the chosen subnet. On the other hand, if the output operation is performed in a reaction, according to axiom `(out-R)` the tokens corresponding to the decomposition of reactions of active listeners on the datum are produced in the active subnet. Axiom `(end-R)` permits to end the execution of a (possibly not terminated) subnet corresponding to a reaction and to return the control to the main program. If the execution of a subnet corresponding to a reaction has been stopped before all reaction processes are terminated, by axiom `(startloop)` a token in place *loop* is produced; by axiom `(loop)`, this token prevents the whole net to terminate.

Definition 4. *Let P be a* `ReactLinda-!` *process. We define the system $Net(P) = (S, T, m_0)$, where*

$$\begin{aligned} S &= L \times sub(P) \cup \ names(P) \cup \\ &\quad \{k \star (a, R) \mid (a, R) \in listeners(P) \wedge 0 \leq k \leq \#rct(P)\} \cup \\ &\quad L \times \{running, unused, finished\} \cup \{loop\} \\ T &= \{(c, p) \in Trans \mid c, p \subseteq S\} \\ m_0 &= dec(P) \end{aligned}$$

Note that $Net(P)$ is a finite P/T net.

The following lemma permits to reduce the decidabiliy of termination for process P to the decidabiliy of termination for the P/T system $Net(P)$.

Lemma 1. *Let P be a* `ReactLinda-!` *process. P terminates if and only if $Net(P)$ terminates.*

Theorem 3. *Let P be a* `ReactLinda-!` *process. The termination problem $P \downarrow$ is decidable.*

4.3 Deciding Divergence on `ReactLinda-!`

The decidability of divergence is obtained by removing from the net constructed in the previous subsection the `(startloop)` and `(loop)` transition, and by using an adaptation of the procedure to check divergence of a P/T system. The basic technique consists in the construction of the so-called coverability tree [5]; the system diverges if a marking m is found that covers (i.e., is bigger than) another marking m' appearing in the path from the root of the tree (i.e., the initial marking m_0) to m. The net system used to check divergence may exhibit some wrong behaviour in the case the control is returned to the main program when some process in the reactions set is not terminated. For example, consider $reactsTo(a, in(b)).out(a).reactsTo(loop, out(loop)).out(loop)$. This process reaches a deadlock during the execution of the reaction $in(b)$, as no datum b will

be produced. However, in the net system corresponding to the process it may happen that the control is returned to the continuation of the main program, i.e., $reactsTo(loop, out(loop)).out(loop)$ even if the reaction $in(b)$ has not been executed. Hence, the net may exhibit a divergent computation. However, all the markings in such a divergent computation are "erroneous", and can be recognized by the following feature: there exist two places, $r : finished$ and $r : P'$ (for some process P'), such that both places contain at least one token. This means that the subnet r has stopped its execution when there is at least one reaction that has not finished its execution. Hence, the existence of a divergent computation of such a net system can be decided by a slight variation of the basic procedure: the system diverges if a marking m is found that covers another marking m' appearing in the path from the root of the tree to m, and moreover m is not an erroneous marking.

The set $Trans'$ contains all the instances of the transitions schemata reported in Table 4, but axioms `(startloop)` and `(loop)`. The P/T net used to decide divergence of P is constructed as follows:

Definition 5. *Let P be a* `ReactLinda-!` *process. We define the system $Net'(P) = (S', T', m'_0)$, where*

$$\begin{aligned} S' &= L \times sub(P) \cup \ names(P) \cup \\ &\quad \{k \star (a, R) \mid (a, R) \in listeners(P) \wedge 0 \leq k \leq \#rct(P)\} \cup \\ &\quad L \times \{running, unused, finished\} \\ T' &= \{(c, p) \in Trans' \mid c, p \subseteq S\} \\ m'_0 &= dec(P) \end{aligned}$$

Note that $Net'(P)$ is a finite P/T net.

Definition 6. *A marking m of $Net'(P)$ is* erroneous *is there exist i and P' s.t. the following holds: $m(r_i : finished) = 1$ and $m(r_i : P') > 0$.*

Note that if a marking m is erroneous then any marking reachable from m is erroneous. The following lemmata permit to reduce decidability of divergence for the calculus `ReactLinda-!` processes to decidability of a similar property on the class of net systems constructed in Definition 4.3.

Lemma 2. *Let P be a* `ReactLinda-!` *process. The existence of a divergent computation not containing erroneous markings is a decidable property of the net system $Net'(P)$.*

Lemma 3. *A* `ReactLinda-!` *process P diverges if and only if there exists a divergent computation in $Net'(P)$ that does not contain erroneous markings.*

Theorem 4. *Let P be a* `ReactLinda-!` *process. The divergence problem $P \uparrow$ is decidable.*

5 Conclusion

The contribution and the theoretical results proved in this paper have been already discussed in the Introduction. A relevant result, that has some relevance

also from a practival viewpoint, is the decidability of both termination and divergence in `ReactLinda-!`, the replication free calculus based on the prioritized reaction execution model. It is worth to notice that a similar result does not hold for the calculus in [1] based on the parallel execution model. Indeed, if we remove replication from that calculus termination is still undecidable. This result follows from the possibility to install new listeners during the execution of reactions. This is natural under the parallel model due to the interleaving between processes and reactions. For instance, the *notify* operation of JavaSpaces [13] can be freely executed within reactions. On the contrary, under the prioritized model, the set of the installed listeners is usually leaved unchanged during the reactions execution. This holds, e.g., in LIME [9] where the *reactsTo* operation cannot be performed within reactions.

References

1. N. Busi and G. Zavattaro. On the Expressivenes of Event Notification in Data-Driven Coordination Languages. In *Proc. of ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 41–55. Springer-Verlag, Berlin, 2000.
2. A. Cheng, J. Esparza, and J. Palsberg, Complexity results for 1-safe nets. *Theoretical Computer Science*, 147:117–136, 1995.
3. G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 237–248. Springer-Verlag, Berlin, 1998.
4. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
5. R. M. Karp and R. E. Miller, Parallel Program Schemata. *Journal of Computer and System Sciences* 3: 147-195, 1969.
6. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
7. M.L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, 1967.
8. A. Omicini and F. Zambonelli. Coordination of mobile information agents in TuCSoN. *Journal of Internet Research*, 8(5), 1998.
9. G.P. Picco, A.L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *Proc. of the 21st ICSE*, 1999.
10. W. Reisig. *Petri nets: An Introduction.* EATCS Monographs in Computer Science, Springer, 1985.
11. A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.
12. J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.
13. J. Waldo et al. Javaspace specification. Technical report, Sun Microsystems, 1998.
14. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

Synchronized Hyperedge Replacement for Heterogeneous Systems*

Ivan Lanese and Emilio Tuosto

Dipartimento di Informatica, Largo Bruno Pontecorvo 3,
56127 Pisa – Italy

Abstract. We present a framework for modelling heterogeneous distributed systems using graph transformations in the Synchronized Hyperedge Replacement approach, which describes complex evolutions by synchronizing local rules. In order to deal with heterogeneity, we consider different synchronization algebras for different communication channels. The main technical point is the interaction between synchronization algebras and name mobility in the π-calculus style. The power of our approach is shown through a few examples.

1 Introduction

Nowadays, Internet is becoming more and more integrated with *non* IP-based networks and the so-called *overlay networks*, which combine different kinds of networks (e.g., ATM, wireless etc.), are beginning to appear in the scene. Overlay networks are changing the nature of Internet, indeed new protocols and communication policies are required in order to allow applications (such as web services) to interact across different kinds of networks.

Doubtless, applications should consider these changes of perspective in order to take advantage of the new technological possibilities offered by overlay networks. Indeed, not only this evolution of Internet has impact on the communication infrastructure, but it also influences the level of the applications and the middlewares they rely on. For instance, in some cases it would help to have point-to-point communications while in others broadcast is preferable. Of course, this kind of situation may be present also in the usual practice of concurrent programming. A typical example is when a server first acquires data over which it computes and then must send the results to several waiting clients. Classical languages for concurrent/distributed programming offer a limited number of communication policies while models usually restrict on a single synchronization mechanism. For example, the π-calculus [13, 15] adopts point-to-point communication, hence, broadcast communications, if desired, must be encoded. Classical formalisms, however, cannot uniformly deal with dynamical/unpredictable variations of the synchronization policies.

* I. Lanese has been supported by EU-FET project **AGILE** IST-2001-32747. E. Tuosto has been supported by EU-FET project **PROFUNDIS** IST-2001-33100.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 220–235, 2005.

The *Synchronized Hyperedge Replacement* approach [1, 3] (SHR, for short) is a uniform graph transformation framework for dealing with many facets of wide area network applications [7, 16, 8, 2, 6, 11, 5]. Systems are hypergraphs, namely graphs where each hyperedge connects an arbitrary number of nodes, computations correspond to rewrite hypergraphs by applying *productions* which are rules of the form $p : L(\boldsymbol{x}) \rightarrow G$ where $L(\boldsymbol{x})$ is an hyperedge and G an hypergraph. Informally, applying production p to a graph means to replace an instance of L with G in the graph. The replacements are coordinated in SHR through the requirements that p imposes to the attachment nodes of L, namely, in order to replace L with G, it is necessary that the components connected to the attachment nodes of L in the graph synchronize (according to a given synchronization policy) with the requirements imposed by p. Intuitively, this implies that synchronizing corresponds to resolving a distributed constraint satisfaction problem as pointed out in [14]. The SHR approach has the advantage, w.r.t. other graphical frameworks, such as double push-out [4] and bigraphs [10], of allowing a distributed implementation since productions have a local effect and synchronization can be performed using a distributed algorithm.

A first abstraction w.r.t. synchronization mechanisms has been done in [12], where SHR has been equipped with a general notion of *synchronization algebra with mobility* (SAM, for short). Synchronization algebras [17] abstractly define the basic properties of synchronization policies by distilling the axioms capturing them. Therefore, the programmer can define his own synchronization mechanism and exploit SHR for representing systems and their computations. However, the proposal of [12] lacks the possibility of representing different SAMs in a single rewriting system. The main contribution of this work is to solve this problem.

Following [9], the SHR mechanism used here, allows mobility by means of node fusions. Our extension adds to fusion the peculiarity of changing the synchronization policy of a given node. In particular, SAMs form a commutative monoid which is programmer-defined. Moreover, whenever x is the node resulting from merging y and z, the SAM associated to x is obtained by composing the SAMs associated to y and z via the monoidal operation. Even though this might have no counterpart at the level of the communication infrastructure, this mechanism may result extremely useful at the level of the applications. For instance, the synchronization policy of a new channel can be dynamically determined as the composition of different constraints imposed by interacting components.

Structure of the Paper. §2 gives some background on graphs. §3 discusses synchronization algebras with mobility and labelled graphs. The following sections §4 and §5 present productions and transitions for SHR, respectively via an informal example and with rigorous mathematical definitions. Another example is presented in §6. Finally, §7 presents conclusions and traces for future work.

2 Background

Before describing hypergraphs, some notations are given.

Given a vector $\boldsymbol{v}$, $|\boldsymbol{v}|$ is its length and $\boldsymbol{v}[i]$ is its i-th element.

We denote with $\mathrm{mgu}(E)$ an idempotent substitution resulting from computing the most general unifier on the set of equations E. The transitive closure of E is E^+.

We use $\uplus$ for disjoint set-union and, in $A \uplus B$, $[1, n]$ (resp. $[2, n]$) is the element that corresponds to $n \in A$ (resp. $n \in B$). We denote with Int_n the set $\{1, \ldots, n\}$ (where $Int_0 \stackrel{\text{def}}{=} \emptyset$) while id_n is the identity function on it. Given two functions $f : A \to C$ and $g : B \to D$ we denote with $[f, g] : A \uplus B \to C \uplus D$ the function that applies f to the elements of A and g to the ones of B. The standard composition of functions is denoted by $\circ$. Given a function f, $f|_S$ (resp. $f|_{\setminus S}$) is the function obtained by restricting f to S (resp. to $\mathrm{dom}(f) \setminus S$) while $\mathrm{merge}(f)$ yields the set of equalities $\{x = y | f(x) = f(y)\}$. Finally, when set operations (e.g., $\cup$) are used on functions, it is implicitly assumed that these are represented as sets of pairs.

We want to model systems using hypergraphs, a generalization of graphs where hyperedges may connect any number of nodes. For simplicity, we use graph (resp. edge) instead of hypergraph (resp. hyperedge). We assume a set LE of labels and a function $\mathrm{rank} : LE \to \omega$ that assigns a rank to each $L \in LE$. An edge labelled by L is an atomic item with $\mathrm{rank}(L)$ ordered tentacles. A set of nodes, together with a set of edges, forms a graph if each edge is connected, by its tentacles, to its attachment nodes. A graph is connected to its environment by an interface which is a subset of its nodes. Nodes in the interface are called *free* nodes, while other nodes are said *bound*. We consider graphs up to isomorphisms that preserve free nodes, labels of edges, and connections between edges and nodes.

We use a textual representation for graphs as (syntactic) judgements which is more suitable for defining transformations [9]. In this representation nodes correspond to names, free nodes to free names and edges to basic terms of the form $L(x_1, \ldots, x_n)$, where x_i are arbitrary names and $L \in LE$ has rank n. The constant *nil* represents a graph without edges, the parallel composition operator $|$ builds large graphs from smaller ones and the ν operator binds nodes.

Definition 1 (Graphs as judgements). *Let $\mathcal{N}$ be a fixed infinite set of names. A* judgement *is a pair of the form $\Gamma \vdash G$ where:*

1. *$\Gamma \subseteq \mathcal{N}$ is a finite set of names (the free nodes of the graph);*
2. *G is a term generated by the grammar*
   $G ::= L(\boldsymbol{x}) \mid G|G \mid \nu\, y\; G \mid nil$
   where $\boldsymbol{x}$ is a vector of names, L is an edge label with $\mathrm{rank}(L) = |\boldsymbol{x}|$ and y is a name.

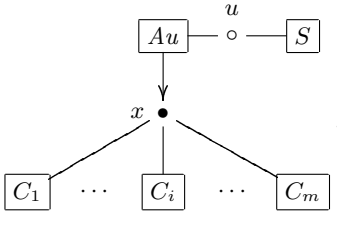
The restriction operator ν is a binder (similar to the binder of λ-calculus). We denote with $\mathrm{fn}(-)$ the function that yields the set $\mathrm{fn}(G)$ of free names in a term G. We demand that $\mathrm{fn}(G) \subseteq \Gamma$.

Graph terms are considered up to the axioms of structural congruence in Table 1: (ax1), (ax2) and (ax3) define respectively the associativity, commutativity and identity over *nil* for operation $|$. Axioms (ax4) and (ax5) state that nodes of a graph can be hidden only once and in any order. Thanks to axiom (ax4) we can write $\nu\, Z$ where $Z = \{x_1, \ldots, x_n\}$ instead of $\nu\, x_1 \ldots \nu\, x_n$. Axiom (ax6) defines α-conversion of bound names in a graph. Axiom (ax7) defines the interaction between restriction and parallel composition. Note that function $\mathrm{fn}(-)$ is well-defined on equivalence classes. As far as judgements are concerned, we define $\Gamma \vdash G \equiv \Gamma' \vdash G'$ iff $\Gamma = \Gamma'$ and $G \equiv G'$.

Theorem 1 (Soundness of the Representation [7]). *Judgements up to structural congruence are isomorphic to graphs up to isomorphisms.*

In order to explain the formal definitions, we describe here a sample scenario from the Internet realm: it will also be exploited later as a running example.

Some clients $C_1, \ldots, C_m$ can invoke a service offered by a remote server S provided that they are authorized. A possible solution might be to interpose an authority Au between S and the clients. Both S and the clients trust Au, and clients get access to S only after they have been authenticated by Au.



$$x \vdash (\nu\, u) \left(\begin{array}{c} Au(x,u) \mid S(u) \mid \\ C_1(x) \mid \cdots \mid C_i(x) \mid \cdots \mid C_m(x) \end{array} \right).$$

The picture and the judgement above represent clients connected to Au on a "public" node x. The server S is also connected to the authority Au, but this time on a "private" (i.e., restricted) node, graphically represented as an empty bullet. Notice that Au has an arrowed tentacle. In general, hypergraphs are not oriented, here this graphical convention is only used for representing the order of the tentacles in edges having rank $h > 1$. Namely, the arrowed tentacle is the first one while the others are numbered clock-wise.

Table 1. Structural congruence for graph terms

(ax1) $G_1 \mid (G_2 \mid G_3) \equiv (G_1 \mid G_2) \mid G_3$ (ax2) $G_1 \mid G_2 \equiv G_2 \mid G_1$ (ax3) $G \mid nil \equiv G$
(ax4) $\nu\, x\ \nu\, y\ G \equiv \nu\, y\ \nu\, x\ G$ (ax5) $\nu\, x\ G \equiv G$ if $x \notin \mathrm{fn}(G)$
(ax6) $\nu\, x\ G \equiv \nu\, y\ G\{y/x\}$ if $y \notin \mathrm{fn}(G)$
(ax7) $\nu\, x\ (G_1 \mid G_2) \equiv (\nu\, x\ G_1) \mid G_2$ if $x \notin \mathrm{fn}(G_2)$

3 Synchronization Algebras with Mobility

In this section we introduce *synchronization algebras with mobility* (SAM, for short), an extension of synchronization algebras [17] able to deal with name mobility in the style of name-passing calculi. We extend graphs by labelling nodes with SAMs that will be exploited to choose the policies for synchronizing rewriting rules.

Definition 2 (Synchronization Algebra with Mobility). *Let Act be a set of actions containing a distinguished element ϵ and at least an action $a \neq \epsilon$. A SAM on Act, $< N, Act, \mathrm{ar}, \epsilon, Fin, ActCmp >$, is identified by its name N and it includes a function* $\mathrm{ar} : Act \to \omega$ *such that* $\mathrm{ar}(\epsilon) = 0$*, a subset Fin of final actions containing ϵ and a relation $ActCmp$ for action composition. In particular, $ActCmp$ is a set of triples of the form $(a, b, (c, \mathrm{Mb}))$ where $a, b, c \in Act$ and* Mb *is a partial function from $Int_{\mathrm{ar}(a)} \uplus Int_{\mathrm{ar}(b)}$ to $\{1, 2, \dots\}$ such that:*

1. $c = \epsilon \Leftrightarrow a = b = \epsilon$*;*
2. Mb *is surjective on* $Int_{\mathrm{ar}(c)}$*;*
3. $(b, a, (c, \mathrm{Mb}')) \in ActCmp$*, with* $\mathrm{Mb}'([1, n]) = \mathrm{Mb}([2, n])$ *and* $\mathrm{Mb}'([2, n]) = \mathrm{Mb}([1, n])$ *for each n;*
4. $(c, d, (e, \mathrm{Mb}')) \in ActCmp \Rightarrow \exists (b, d, (f, \mathrm{Mb}'')) \in ActCmp, (a, f, (e, \mathrm{Mb}''')) \in ActCmp$ *and in that case the composition of* Mb *and* Mb' *gives the same mapping and the same fusions of the composition of* Mb'' *and* Mb'''*.*

The requirements for condition 4 can be written as:

$$\mathrm{Mb}_1 \mid_{Int_{\mathrm{ar}(e)}} = \mathrm{Mb}_2 \mid_{Int_{\mathrm{ar}(e)}}$$
$$(\mathrm{merge}(\mathrm{Mb}) \cup \mathrm{merge}(\mathrm{Mb}_1))^+ = (\mathrm{merge}(\mathrm{Mb}'') \cup \mathrm{merge}(\mathrm{Mb}_2))^+.$$

where $\mathrm{Mb}_1 = \mathrm{Mb}' \circ [\mathrm{Mb} \mid_{Int_{\mathrm{ar}(c)}}, id_{\mathrm{ar}(d)}]$ *and* $\mathrm{Mb}_2 = \mathrm{Mb}''' \circ [id_{\mathrm{ar}(a)}, \mathrm{Mb}'' \mid_{Int_{\mathrm{ar}(f)}}]$.

We define the functions factset, factfin *and* factcmp *that given a SAM A compute respectively its set of actions Act, its set of final actions Fin and its composition relation $ActCmp$.*

Intuitively, $\mathrm{ar}(a)$ is the number of nodes that are communicated with a, ϵ stands for "no-action", Fin contains the set of actions that represent complete synchronizations and thus can be executed also on restricted nodes. Notice that ϵ is always considered complete. Finally, $ActCmp$ defines action synchronization and name communication. More precisely, the triples of the form $(a, b, (c, \mathrm{Mb}))$ define the allowed synchronizations when actions a and b interact: each triple is an allowed behaviour, and if no such triple exists the actions are not compatible, i.e. they cannot synchronize. In particular c is the result of the synchronization and Mb describes how the parameters of a and b are mapped to the parameters of c. If many parameters are mapped to the same position, the corresponding nodes are merged and the resulting node is attached to action c. Only the parameters up to $\mathrm{ar}(c)$ are actually communicated, the others are used for performing additional fusions without showing the result in the final action.

Condition 1, present in normal synchronization algebras, amounts to say that the effect of a synchronization cannot be "no action". Condition 2 guarantees that each node attached to the composed action can be computed, that is it corresponds to a non empty set of nodes from component actions. Finally, conditions 3 and 4 state that action synchronization and mobility patterns are commutative and associative.

We inherit the characterization of mobility from [12] so that general mobility patterns can be modelled. Since this work aims at formalizing heterogeneous systems, we tailor the examples to show the use of multiple SAMs, while maintaining them as simple as possible. For an application of the full generality of the mobility patterns, we refer to [12]. The main difference w.r.t. the SAMs in [12] is that we allow nondeterminism, namely, instead of being fixed, the result of synchronizing two actions is nondeterministically chosen from a set of allowed behaviours specified by the synchronization relation *ActCmp*. This allows to model some more policies, for instance, in a SAM with priorities, when two messages with the same priority interact, one of them is nondeterministically discarded. Also, SAMs are named so that we can distinguish among those describing the same interactions. This can be useful when SAM composition does not depend only on their synchronization policy.

We present here two simple examples of SAMs, namely the one for *Milner synchronization* and the one for *broadcast*. Both of them rely on a mobility pattern that implements message passing, i.e. it merges the corresponding parameters. We define the function message passing $MP_{n,m}$ from $Int_n \uplus Int_m$ to $Int_{\max(n,m)}$ as follows: the element i of both the starting sets (when it exists) is mapped to the element i in the codomain.

When defining SAMs we suppose that *ActCmp* contains just the tuples given explicitly, plus the ones derivable from commutativity.

Example 1 (Milner SAM). Given the set of actions $Act = \{\tau, \epsilon\} \cup \bigcup_{i \in I} \{a_i, \overline{a_i}\}$ where $\mathrm{ar}(\overline{a_i}) = \mathrm{ar}(a_i)$ and $\mathrm{ar}(\tau) = 0$, the Milner SAM over Act is as follows:

- $(a, \epsilon, (a, MP_{\mathrm{ar}(a),0})) \in ActCmp$ for each $a \in Act$,
  $(a, \overline{a}, (\tau, MP_{\mathrm{ar}(a),\mathrm{ar}(\overline{a})})) \in ActCmp$ for each $a \in \bigcup_{i \in I} \{a_i\}$;
- $Fin = \{\tau, \epsilon\}$.

Milner synchronization represents message passing *à la* π-calculus, with one process executing input a and the other one executing output $\overline{a}$.

Example 2 (Broadcast SAM). Given the set of actions $Act = \{\epsilon\} \cup \bigcup_{i \in I} \{a_i, \overline{a_i}\}$ where $\mathrm{ar}(\overline{a_i}) = \mathrm{ar}(a_i)$, the broadcast SAM over Act is as follows:

- $(a, \overline{a}, (\overline{a}, MP_{\mathrm{ar}(a),\mathrm{ar}(\overline{a})})) \in ActCmp$ for each $a \in \bigcup_{i \in I} \{a_i\}$,
  $(a, a, (a, MP_{\mathrm{ar}(a),\mathrm{ar}(a)})) \in ActCmp$ for each $a \in \bigcup_{i \in I} \{a_i\}$,
  $(\epsilon, \epsilon, (\epsilon, MP_{0,0})) \in ActCmp$;
- $Fin = \{\epsilon\} \cup \bigcup_{i \in I} \{\overline{a_i}\}$.

This SAM implements broadcast, where one process performs an output $\overline{a}$ and *all* the others have to perform an input a. Notice that, if one wants to have

weak broadcast, where some process may not synchronize with the output, it is enough to add the triples $(a, \epsilon, (a, MP_{\mathrm{ar}(a),0}))$ for each $a \in Act$ to $ActCmp$.

We always consider a set $\mathcal{Alg}$ containing all the SAMs of interest. This set is assumed to be a commutative monoid w.r.t. an operator $\diamond$ of algebra composition. Names of SAMs are assumed to be unique in $\mathcal{Alg}$.

We can now extend graphs by labelling nodes with SAMs. We concentrate on the representation as syntactic judgements.

Definition 3 (Labelled Graphs). *Assuming $A \in \mathcal{Alg}$, we define a* labelled graph *as a pair of the form $\Gamma \vdash G$ where:*

1. *Γ is a finite function mapping nodes to SAMs, written as sequence of pairs $n : A$ where $n \in \mathcal{N}$;*
2. *G is a term generated by the grammar*
   $G ::= L(\boldsymbol{x}) \mid G|G \mid \nu\, y : A.G \mid nil$
   where $\boldsymbol{x}$ is a vector of names, L is an edge label with $\mathrm{rank}(L) = |\boldsymbol{x}|$ *and y is a name.*

In $\nu\, y : A.G$, ν binds y in G while recording the label A.

When defining the interfaces, $\Gamma, x : A$ denotes the set obtained by adding $x : A$ to Γ, assuming $x \notin \mathrm{dom}(\Gamma)$ and Γ_1, Γ_2 denotes the union of Γ_1 and Γ_2, assuming $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$.

The relation of structural congruence on graphs is the same that we defined in §2, with α-conversion preserving the labelling SAMs. Now graph isomorphisms also preserve SAMs that label nodes. Theorem 1 holds also with the new definitions.

4 SHR via an Example

Productions are the basic rules of SHR describing how edges can be rewritten. This section gives a flavour of productions and synchronization before the formal definitions which are in the following section. In particular, it highlights how multiple SAMs can model issues of distributed applications in a simpler way w.r.t. previous approaches.

A production takes the form $\boldsymbol{x} : \boldsymbol{A} \vdash L(\boldsymbol{x}) \xrightarrow{\Lambda} \Gamma \vdash G$ where

- $\boldsymbol{x} : \boldsymbol{A}$ abbreviates $x_1 : A_1, \ldots, x_{|\boldsymbol{x}|} : A_{|\boldsymbol{x}|}$ and $\boldsymbol{x} : \boldsymbol{A} \vdash L(\boldsymbol{x})$ is an edge such that all its attachment nodes are distinct,
- $\Gamma \vdash G$ is a labelled graph and
- Λ is a synchronization function mapping nodes in $\boldsymbol{x}$ to pairs $(a, \boldsymbol{y})$ where $a \in Act$ is the performed action and $\boldsymbol{y}$ contains the communicated nodes.

Roughly, such a production states that, in a given graph, an edge labelled L can be replaced by G provided that its attachment nodes $\boldsymbol{x}$ are labelled by $\boldsymbol{A}$ and it can synchronize through actions specified by Λ with the productions of the edges sharing the attachment nodes in $\boldsymbol{x}$.

We present the productions for our running example, i.e., the ones expressing the behaviours of Au, S and a generic client C_i. For the sake of simplicity, we consider the Milner SAM on actions $\{\text{req}\} \cup \bigcup_{i \in Int_m} \{\text{auth}_i\}$ (see Example 1) and call it *Mil*.

First consider a generic client C_i:

$$x : Mil \vdash C_i(x) \xrightarrow{(x,\overline{\text{auth}_i},\langle y \rangle)} x : Mil, y : Mil \vdash C_i'(y) \tag{1}$$

$$y : Mil \vdash C_i'(y) \xrightarrow{(y,\overline{\text{req}},\langle\rangle)} y : Mil \vdash C_i'(y). \tag{2}$$

Production 1 models the authentication phase where C_i is attached to a *Mil* node and asks Au for the access to the service S. If the authentication takes place, C_i connects to the server through y, the name which will be instantiated with the server "address" during the synchronization with Au. The right-hand-side of production 1 expresses that the client is connected to the server and can make its requests. Notice that the synchronization between Au and C_i takes place only if C_i is able to provide some information that allows Au to "recognize" C_i as a legal user, which is abstracted with auth_i. Production 2 simply states that a request is sent to the server. For simplicity, we assume that the server does not give back any answer.

Productions for Au simply have to accept authorized clients and give them the server address:

$$x : Mil, u : Mil \vdash Au(x, u) \xrightarrow{(x,\text{auth}_i,\langle u \rangle)} x : Mil, u : Mil \vdash Au(x, u), \tag{3}$$

where i ranges over the indexes of valid clients.

Finally, the server simply accepts requests from clients:

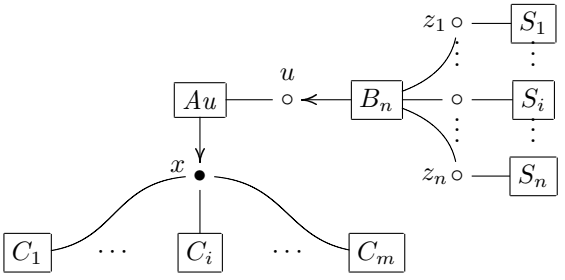
$$u : Mil \vdash S(u) \xrightarrow{(u,\text{req},\langle\rangle)} u : Mil \vdash S'(u),$$

where S' is used to model the server when it is busy. It becomes again available via the production

$$u : Mil \vdash S'(u) \xrightarrow{(u,\epsilon,\langle\rangle)} u : Mil \vdash S(u),$$

which does not require any synchronization since it corresponds to an internal transition of the server.

Consider that we want to extend our example by making each request to be served by many servers at once. For instance, assume that the request is a search query on the web that clients want to submit to different search engines. This extension can be easily obtained by interposing an edge B_n between Au and the n servers. This system can be represented by the following figure:



B_n will simply acquire the requests from clients and forward them to each server. This behaviour is formally stated by the following production:

$$u : Mil, \boldsymbol{z} : \mathbf{Mil} \vdash B_n(u, \boldsymbol{z}) \xrightarrow{\substack{(u, \mathrm{req}, \langle\rangle),\\ (\boldsymbol{z}, \overline{\mathrm{req}}, \langle\rangle)}} u : Mil, \boldsymbol{z} : \mathbf{Mil} \vdash B_n(u, \boldsymbol{z}).$$

where $\boldsymbol{z} : \mathbf{Mil}$ shortens $z_1 : Mil, \ldots, z_n : Mil$ while with $(\boldsymbol{z}, \overline{\mathrm{req}}, \langle\rangle)$ we denote $(z_1, \overline{\mathrm{req}}, \langle\rangle), \ldots, (z_n, \overline{\mathrm{req}}, \langle\rangle)$.

This solution has a number of advantages:

- the introduction of B_n is completely transparent to the other components, hence we do not have to change the productions for Au, C or S,
- B_n triggers all the servers in parallel.

Nevertheless, there also are some drawbacks:

- if one of the servers crashes or autonomously disconnects from B_n, then B_n is blocked,
- if B_n crashes then all the servers are isolated,
- adding (resp. removing) a server implies to replace B_n with B_{n+1} (resp. B_{n-1}),
- the request is dangling if one of the servers does not accept it.

All these drawbacks are resolved by using weak broadcast synchronizations between the clients and the servers. Instead of introducing edges B_n, we can simply modify productions (1) and (3) as follows:

$$x : Mil \vdash C_i(x) \xrightarrow{(x, \overline{\mathrm{auth}_i}, \langle y\rangle)} x : Mil, y : Bdc \vdash C_i'(y) \tag{4}$$

$$x : Mil, u : Bdc \vdash Au(x, u) \xrightarrow{(x, \mathrm{auth}_i, \langle u\rangle)} x : Mil, u : Bdc \vdash Au(x, u),$$

where Bdc is the weak broadcast SAM on the action $\{\mathrm{req}\}$ (see Example 2). Notice that Au and C_i must accord on the label of the second attachment node of Au. Notice also that now only request actions can be executed on node u.

5 The Mathematics of Heterogeneous SHR

We present now the formal definitions of productions and transitions. In addition to what shown in Section 4, now the label of a transition also contains an idempotent substitution π that allows to merge nodes in the interface by mapping each of them into a standard representative of its equivalence class. Even though not used in this work, we include π since we extend the SHR approach from [5] where it has been exploited.

Definition 4 (SHR Transition). *A* SHR transition *is a relation of the form:*

$$\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$$

where $\Gamma \vdash G$ and $\Phi \vdash G'$ are labelled graphs, $\Lambda : \mathrm{dom}(\Gamma) \to (Act \times \mathcal{N}^*)$ is a total function and $\pi : \mathrm{dom}(\Gamma) \to \mathrm{dom}(\Gamma)$ is an idempotent substitution. If $\Lambda(x) = (a, \boldsymbol{y})$ then $|\boldsymbol{y}| = \mathrm{ar}(a)$. We define $\mathrm{act}_\Lambda(x) = a$, $\mathrm{n}_\Lambda(x) = \boldsymbol{y}$ and

- $\mathrm{n}(\Lambda) = \{z | \exists x.z \in \mathrm{n}_\Lambda(x)\}$ *set of exposed names;*
- $\Gamma_\Lambda = \mathrm{n}(\Lambda) \setminus \mathrm{dom}(\Gamma)$ *set of exposed fresh names.*

We require $\mathrm{dom}(\Phi) = \pi(\mathrm{dom}(\Gamma)) \cup \Gamma_\Lambda$, namely free nodes are never erased ($\supseteq$) and new nodes are bound unless exposed ($\subseteq$). The SAMs associated to new nodes in Φ (nodes in Γ_Λ) can be freely chosen. Instead, for each $x \in \pi(\mathrm{dom}(\Gamma))$, the associated algebra A is $A_1 \diamond \ldots \diamond A_n$ where $\{A_1, \ldots, A_n\}$ are the labels of the nodes in Γ mapped to x by π. Notice that A is well-defined since SAMs form a commutative monoid.

We want to be able to specify productions that can be applied to nodes with different labels, while keeping some control on them. Hence, we introduce types:

Definition 5 (Types). *A* type *t is any non empty set of SAMs.*

Definition 6 (Productions). *A* production *is an SHR transition of the form $x_1 : A_1, \ldots, x_n : A_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} \Phi \vdash G$ where $x_1, \ldots, x_n$ are all distinct and $A_1, \ldots, A_n$ are SAMs.*

A production schema *is a generalized production that has types instead of SAMs as labels for nodes.*

Productions can be derived from production schemas by α-converting the nodes in $\{x_1, \ldots, x_n\} \cup \mathrm{dom}(\Phi)$ and/or by specializing each type t_i into a particular SAM $A_i \in t_i$, provided that for any x_i, $\mathrm{act}_\Lambda(x_i) \in \mathrm{factset}(A_i)$ and that the result is a correct transition (namely, nodes in Φ that are the result of a fusion have the label computed by composing the labels of the merged nodes). We suppose to have for each edge label L of arity n a special idle production schema $\boldsymbol{x} : \boldsymbol{\mathcal{A}lg} \vdash L(\boldsymbol{x}) \xrightarrow{\Lambda_\epsilon,id} \boldsymbol{x} : \boldsymbol{\mathcal{A}lg} \vdash L(\boldsymbol{x})$ where $\Lambda_\epsilon(x_i) = (\epsilon, \langle\rangle)$ for each $i \in Int_{|\boldsymbol{x}|}$.

Transitions for SHR are derived by composing productions using the inference rules in the following definition.

Definition 7 (Heterogeneous SHR). *A* heterogeneous SHR rewriting system *consists of a triple* $(\mathcal{Alg}, \mathcal{P}, \Gamma \vdash G)$*, where* $\mathcal{Alg}$ *is the set of SAMs,* $\mathcal{P}$ *is a set of productions and* $\Gamma \vdash G$ *is the initial labelled graph.*

The set of transitions of $(\mathcal{Alg}, \mathcal{P}, \Gamma \vdash G)$ *is the smallest set obtained by applying the inference rules below starting from the productions in* $\mathcal{P}$*. The* computations of $(\mathcal{Alg}, \mathcal{P}, \Gamma \vdash G)$ *are sequences of transitions* $\Gamma_i \vdash G_i \xrightarrow{\Lambda_i, \pi_i} \Phi_i \vdash G'_i$*, for* $i \in \omega$*, such that* $\Gamma_0 \vdash G_0 = \Gamma \vdash G$ *and, for each* $i > 0$*,* $\Gamma_i \vdash G_i$ *is* $\Phi_{i-1} \vdash G'_{i-1}$*.*

$$\textit{(res)}\quad \frac{\Gamma, x : A \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G' \quad \mathrm{act}_\Lambda(x) \in \mathrm{factfin}(A) \quad (x\pi = y\pi \wedge x \neq y) \Rightarrow x\pi \neq x}{\Gamma \vdash \nu\, x : A.G \xrightarrow{\Lambda|_{\setminus\{x\}}, \pi|_{\setminus\{x\}}} \Phi' \vdash \nu\, Z\ G'}$$

where $Z = \Phi \setminus \Phi'$*.*

$$\textit{(new)}\quad \frac{\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G' \quad x \notin \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Phi) \quad A \in \mathcal{Alg}}{\Gamma, x : A \vdash G \xrightarrow{\Lambda \cup \{(x,\epsilon,\langle\rangle)\},\pi} \Phi, x : A \vdash G'}$$

$$\textit{(par)}\quad \frac{\Gamma_1 \vdash G_1 \xrightarrow{\Lambda_1,\pi_1} \Phi_1 \vdash G'_1 \quad \Gamma_2 \vdash G_2 \xrightarrow{\Lambda_2,\pi_2} \Phi_2 \vdash G'_2 \quad (\mathrm{dom}(\Gamma_1) \cup \mathrm{n}(\Lambda_1)) \cap (\mathrm{dom}(\Gamma_2) \cup \mathrm{n}(\Lambda_2)) = \emptyset}{\Gamma_1, \Gamma_2 \vdash G_1 | G_2 \xrightarrow{\Lambda_1 \cup \Lambda_2, \pi_1 \cup \pi_2} \Phi_1, \Phi_2 \vdash G'_1 | G'_2}$$

$$\textit{(merge)}\quad \frac{\Gamma, x : A, y : A \vdash G \xrightarrow{\Lambda \cup \{(x,a_1,\boldsymbol{v}_1),(y,a_2,\boldsymbol{v}_2)\},\pi} \Phi \vdash G' \quad (a_1, a_2, (c, \mathrm{Mb})) \in \mathrm{factcmp}(A)}{\Gamma, x : A \vdash G\sigma \xrightarrow{\Lambda', \pi'} \Phi' \vdash \nu\, Z\ G'\sigma\rho}$$

where:

- $\sigma = \{x/y\}$*;*
- $E = \{\boldsymbol{v}_{i_1}[j_1] = \boldsymbol{v}_{i_2}[j_2] \mid \mathrm{Mb}([i_1, j_1]) = \mathrm{Mb}([i_2, j_2])\}$*;*
- $\rho = \mathrm{mgu}(\{(E \cup \mathrm{merge}(\pi))\sigma\})$ *where we choose node names in* $\mathrm{dom}(\Gamma) \cup \{x\}$ *as representatives whenever possible;*
- $\boldsymbol{w}[i] = (\boldsymbol{v}_j[k])\sigma\rho$ *if* $\mathrm{Mb}([j,k]) = i$*,* $i \in Int_{\mathrm{ar}(c)}$*;*
- $\Lambda'(z) = \begin{cases} (c, \boldsymbol{w}) \text{ if } z = x \\ (\mathrm{act}_\Lambda(z), (\mathrm{n}_\Lambda(z))\sigma\rho) \text{ for each } z \in \mathrm{dom}(\Gamma) \end{cases}$
- $\pi' = \rho|_{\mathrm{dom}(\Gamma) \cup \{x\}}$*;*
- $\mathrm{dom}(Z) = \mathrm{dom}(\Phi)\sigma\rho \setminus \mathrm{dom}(\Phi')$*;*
- *the label of each node* $x \in \mathrm{dom}(Z) \cup \mathrm{dom}(\Phi')$ *is computed as follows:* x *is the representative according to* ρ *of an equivalence class* $\{x_1, \ldots, x_n\}$ *of nodes which have in* Φ *labels* $A_1, \ldots, A_n$*. Then the label of* x *is* $A_1 \diamond \ldots \diamond A_n$*.*

Rule (res) binds nodes, allowing only complete actions (w.r.t. their SAMs) to take place on them. According to the last condition of (res), the bound node must not be the representative of the equivalence class induced by π when the class is not trivial. Nodes extruded just on the bound node must be bound after the transition, and thus they are in Z (the labels are preserved).

Rule (new) allows to add an isolated node x to the interface; initially, on x only the trivial action ϵ can be done.

Rule (par) performs the union of two transitions provided that they have disjoint sets of free names (accounting also for newly generated names).

Rule (merge) is the rule for synchronization. It allows to compute the effect of merging two nodes x and y with synchronizations $(a_1, \boldsymbol{v}_1)$ and $(a_2, \boldsymbol{v}_2)$ respectively on them, provided that they are labelled with the same SAM A, which will label also the resulting node. The synchronization is allowed iff there exists a triple $(a_1, a_2, (c, \mathrm{Mb})) \in \mathrm{factcmp}(A)$. In this case set E is computed, which accounts for merging names that are mapped to the same position by Mb (note that merges are performed even if the resulting representative is not attached to action c). We then compute ρ by means of an mgu on $E \cup \mathrm{merge}(\pi)$ after having fused nodes according to σ. Notice that $\mathrm{merge}(\pi)$ accounts for the node fusions due to π and that ρ also chooses a representative for each equivalence class. If at least one of the members of the class is in $\mathrm{dom}(\Gamma) \cup \{x\}$, then one of them must be chosen (otherwise undesired renamings of nodes may happen). After that, the new vector $\boldsymbol{w}$ is generated by choosing for each position the representative of the corresponding equivalence class. We can then compute the new synchronization function Λ', which takes into account the performed merges. Merges on nodes in the interface are traced by π'. Nodes that are no longer extruded (because the synchronization discards them) are bound. The SAMs associated to nodes are preserved from the premise for nodes that are not merged, and are computed using the operator of composition of SAMs otherwise. The result does not depend on the structure of the derivation, as witnessed by the following proposition:

Proposition 1. *Given a transition* $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$*, the labels in* Φ *of nodes in* $\mathrm{dom}(\Phi) \cap \mathrm{dom}(\Gamma)$ *depend only on* Γ *and* π*.*

Proof. By induction on the structure of the derivation.

Example 3 (Dynamically computing SAMs). Consider the example of § 4. Using weak broadcast communications between clients and servers is "statically" determined by the productions of A and C_i which also couple the behaviour of these components. Indeed, if $Mil \diamond Bdc = Bdc$, then rule (merge) ensures that the synchronization of productions (4) and (3) yields the expected result, namely, the attachment node of servers uses a weak broadcast SAM, since C_i requests it to do so.

6 A Further Example

We consider a scenario where multiple processes/hosts are connected through links with different features. Mainly, we consider two characteristics: the maximum packet size, which can be either 4kb or 16kb, and the presence/lack of some error-detecting mechanism e.g., the CRC algorithm. Before starting the real communication, two processes create a logical communication channel between them.

This channel supports 16kb packets and/or error-detecting capabilities only if all the underlying channels do.

We model this scenario using eight SAMs obtained by mixing three distinct features, namely:

- packet sizes (4kb/16kb),
- error-detecting mechanism (yes/no),
- control or communication link.

The first two items correspond to the aforementioned characteristics of links, while the last one specifies whether the link is used for exchanging control messages or for data. For simplicity we suppose that control messages are short and thus can be encoded using some error-correcting code. Hence, control messages are always correctly delivered. Since all the combinations of the three features are possible, we conventionally name the SAMs by means of expressions like CTR_4 or $\mathrm{COM}^{\surd}_{16}$ that respectively denote a control link without error-detecting mechanism and with 4kb maximum packet size and a communication link with error-detecting mechanism and 16kb maximum packet size.

The control SAMs provide essentially normal Milner communication (where we drop the distinction between action and coaction) for control messages. The SAMs for communication without error-detecting mechanism provide faulty Milner synchronization, i.e., the result of a synchronization can be either τ or err. In the former case, name passing is performed while no name passing is performed in the latter. Note that, in these SAMs, the processes cannot detect the result of synchronizations (unless they perform additional interactions). Communication actions can be of two kinds, in_4 for packets of 4kb and in_{16} for packets of 16kb. Corresponding output actions are provided. The same τ and err actions are used in both the cases. Both the kinds of synchronization are allowed by the SAM 16kb, while just the 4kb-size packets are allowed for SAMs 4kb. Finally, error-detecting SAMs allow two different kinds of input, $in^{\surd}_s$ which synchronizes giving τ and in_s which synchronizes giving a detected error, where $s \in \{4\mathrm{kb}, 16\mathrm{kb}\}$.

To clarify all that we will now formally define the SAM $\mathrm{COM}^{\surd}_{16}$:

- $N = \mathrm{COM}^{\surd}_{16}$;
- $Act = \{in^{\surd}_4, in_4, out_4, in^{\surd}_{16}, in_{16}, out_{16}, \tau, err, \epsilon\}$;
- for simplicity we suppose that all actions but τ, err and ϵ have arity 1 while these three have arity 0;
- $Fin = \{\tau, err, \epsilon\}$;
- $(in^{\surd}_4, out_4, (\tau, MP_{1,1})) \in ActCmp$,
  $(in_4, out_4, (err, MP_{0,0})) \in ActCmp$,
  $(in^{\surd}_{16}, out_{16}, (\tau, MP_{1,1})) \in ActCmp$,
  $(in_{16}, out_{16}, (err, MP_{0,0})) \in ActCmp$,
  $(\epsilon, \epsilon, (\epsilon, MP_{0,0})) \in ActCmp$.

We can now define the function of algebra composition. We will define a partial order on our SAMs, and the greatest lower bound will be our composition

operator. The order is defined component-wise on the three features of SAMs over which the following orders are considered: 16kb > 4kb, the presence of an error-detecting mechanism is greater than its absence and $CTR > COM$. Note that the set of SAMs with the resulting operation forms a commutative monoid.

We consider as types all the singletons, the universal type $\mathcal{Alg}$ and the type CT that contains the four control SAMs.

For simplicity we consider that our system contains two kinds of subsystems: end systems $x \vdash Idle(x)$ and routers $x, y, z \vdash R(x, y, z)$. End systems can start a communication with the production schema:

$$x : CT \vdash Idle(x) \xrightarrow{(x,connect,\langle y \rangle)} x : CT, y : \mathrm{COM}_{16}^{\surd} \vdash Active(x, y) \tag{5}$$

After connecting to another system, an *Idle* process becomes *Active*, i.e. able to send/receive data over the links built toward other systems. For lack of space, we do not formally model their behaviour.

We now show the productions for routers. We write just a meta-rule, with type meta-variables t_1 and t_2 to be instantiated with each singleton containing a control SAM. Furthermore the production must be written for each permutation of attachment nodes.

$$\begin{array}{c} x : t_1, y : t_2, z : CT \vdash R(x, y, z) \xrightarrow{(x,connect,\langle w \rangle),(y,connect,\langle w \rangle)} \\ x : t_1, y : t_2, z : CT, w : (\mathrm{COM}_{16}^{\surd} \diamond t_1 \diamond t_2) \vdash R(x, y, z) \end{array} \tag{6}$$

We can now show some examples. Let us consider as starting graph:

$$\vdash \begin{array}{c} (\nu\, x_1 : \mathrm{CTR}_{16}^{\surd}, x_2, x_3 : \mathrm{CTR}_{16}, x_4 : \mathrm{CTR}_4, x_5, x_6 : \mathrm{CTR}_4^{\surd}). \\ (Idle(x_1)|R(x_1, x_2, x_4)|R(x_2, x_3, x_5)|Idle(x_3)|R(x_4, x_5, x_6)|Idle(x_6)), \end{array}$$

where $x_1, \ldots, x_n : A$ shortens $x_1 : A, \ldots, x_n : A$.

For instance, we can build a communication channel between $Idle(x_1)$ and $Idle(x_3)$ using productions (5) and (6). The algebra for the resulting connection is COM_{16}, obtained via $\mathrm{COM}_{16}^{\surd} \diamond \mathrm{COM}_{16} \diamond \mathrm{COM}_{16}$. In that case we obtain the graph:

$$\vdash \begin{array}{c} (\nu\, x_1 : \mathrm{CTR}_{16}^{\surd}, x_2, x_3 : \mathrm{CTR}_{16}, x_4 : \mathrm{CTR}_4, x_5, x_6 : \mathrm{CTR}_4^{\surd}, y : \mathrm{COM}_{16}) \\ (Active(x_1, y)|R(x_1, x_2, x_4)|R(x_2, x_3, x_5)|Active(x_3, y)|R(x_4, x_5, x_6)|Idle(x_6)) \end{array}$$

Similarly we can build a connection between $Idle(x_1)$ and $Idle(x_6)$, but this time the resulting channel will use the algebra COM_4. When either of these connections has been established, communication can happen using that channel. Notice that the label of the channel is computed from the constraints imposed by interacting routers.

7 Conclusions

We have presented a framework for modelling heterogeneous distributed systems using SHR. In addition to standard SHR properties, namely the ability to build

a compositional description of system behaviour, our extension allows to deal with systems where different kinds of communication/synchronization policies coexist, as usually happens in WANs and overlay networks. From the technical point of view the main points are the labelling of nodes with SAMs and the management of these labels when nodes are communicated and merged, which is obtained by requiring a commutative monoid structure on the set of SAMs.

As future work we plan to analyze the behavioural properties of SHR systems by defining a suitable bisimulation which is a congruence w.r.t. composition of graphs. We also want to develop a framework that merges the advantages of SHR and bigraphs, thus allowing a compositional description of systems with both a communication and a location structure. Another important point is to build an implementation of SHR systems, both for modelling system evolution (and also performing behavioural analysis using model-checking) and for programming real systems. In this second case SHR must be integrated with a standard programming language such as Java or C++, thus enabling to write normal code for computation and using some flavour of SHR (probably in a more process-calculi like style) for coordination.

References

1. I. Castellani and U. Montanari. Graph Grammars for Distributed Systems. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Proc. 2nd Int. Workshop on Graph-Grammars and Their Application to Computer Science*, volume 153 of *LNCS*, pages 20–38. Springer, 1983.
2. R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A Formal Basis for Reasoning on Programmable QoS. In N. Dershowitz, editor, *International Symposium on Verification – Theory and Practice – Honoring Zohar Manna's 64th Birthday*, volume 2772 of *LNCS*, pages 436 – 479. Springer, 2003.
3. P. Degano and U. Montanari. A model of distributed systems based on graph rewriting. *JACM*, 34:411–449, 1987.
4. H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: an algebraic approach. In *Proceedings IEEE Conference on Automata and Switching Theory*, pages 167–180, 1973.
5. G. Ferrari, U. Montanari, and E. Tuosto. A LTS Semantics of Ambients via Graph Synchronization with Mobility. In *ICTCS*, volume 2202 of *LNCS*. Springer, 2001.
6. G. Ferrari, U. Montanari, and E. Tuosto. Graph-based Models of Internetworking Systems. In T. Aichernig, Bernhard K. Maibaum, editor, *Formal Methods at the Crossroads: from Panaces to Foundational Support*, volume 2757 of *LNCS*, pages 242 – 266. Springer, 2003.
7. D. Hirsch. *Graph Transformation Models for Software Architecture Styles*. PhD thesis, Departamento de Computación, UBA, 2003. http://www.di.unipi.it/~dhirsch.
8. D. Hirsch, P. Inverardi, and U. Montanari. Reconfiguration of Software Architecture Styles with Name Mobility. In A. Porto and G.-C. Roman, editors, *Coordination 2000*, volume 1906 of *LNCS*, pages 148–163. Springer, 2000.
9. D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility: A graphical calculus for name mobility. In *CONCUR*, volume 2154 of *LNCS*, pages 121–136, Aalborg, Denmark, 2001. Springer.

10. O. Jensen and R. Milner. Bigraphs and transitions. *SIGPLAN Not.*, 38(1):38–49, 2003.
11. I. Lanese and U. Montanari. Software architectures, global computing and graph transformation via logic programming. In L. Ribeiro, editor, *Proc SBES'2002 - 16th Brazilian Symposium on Software Engineering*, pages 11–35. Anais, 2002.
12. I. Lanese and U. Montanari. Synchronization algebras with mobility for graph transformations. In *Proc. FGUC'04 – Foundations of Global Ubiquitous Computing*, ENTCS, 2004. To appear.
13. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Inf. and Comp.*, 100(1):1–40,41–77, September 1992.
14. U. Montanari and F. Rossi. Graph rewriting and constraint solving for modelling distributed systems with synchronization. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference COORDINATION '96, Cesena, Italy*, volume 1061 of *LNCS*. Springer, April 1996.
15. D. Sangiorgi and D. Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2002.
16. E. Tuosto. *Non-Functional Aspects of Wide Area Network Programming*. PhD thesis, Dipartimento di Informatica, Università di Pisa, May 2003. TD-8/03.
17. G. Winskel. Synchronization trees. *TCS*, 34:33–82, May 1985.

Synthesis of Reo Circuits for Implementation of Component-Connector Automata Specifications

Farhad Arbab[a,c], Christel Baier[b], Frank de Boer[a,c], Jan Rutten[a,d], and Marjan Sirjani[e]

[a] CWI, Amsterdam, The Netherlands
{farhad, janr, frb}@cwi.nl
[b] Universität Bonn, Institut für Informatik I, Germany
baier@cs.uni-bonn.de
[c] Universiteit Leiden, The Netherlands
[d] Vrije Universiteit, Amsterdam, The Netherlands
[e] Sharif University of Technology, Tehran, Iran

Abstract. Composition of a concurrent system out of components involves coordination of their mutual interactions. In component-based construction, this coordination becomes the responsibility of the glue-code language and its underlying run-time middle-ware. Reo offers an expressive glue-language for construction of coordinating component connectors out of primitive channels. In this paper we consider the problem of synthesizing Reo coordination code from a specification of a behavior as a relation on scheduled-data streams. The specification is given as a constraint automaton that describes the desired input/output behavior at the ports of the components. The main contribution in this paper is an algorithm that generates Reo code from a given constraint automaton.

1 Introduction

Composing components into a concurrent system involves coordination of their mutual interactions. The internals of black-box components cannot be modified to implement such coordinated interactions. Coordination, therefore, becomes the responsibility of the "glue-code" that inter-connects the constituent components of a composite system, and of its underlying run-time middle-ware. Reo [2] offers a powerful glue language for implementation of coordinating component connectors that resemble electronic circuits and are based on a calculus of mobile channels. Reo is being used, for instance, in the context of the Cybernetic Incident Management project [9] for composition of web services, which constitute the black-box components of dynamically configured distributed applications [11]; to model business processes, such as electronic auctions [20]; and for modeling coordination in biological systems [10].

This paper addresses the *synthesis problem of component connectors* with Reo as our target implementation language. The input for this problem is a specification of a coordination protocol and its output is a Reo connector circuit that implements this protocol. Synthesis problems address the issue of the (algorithmic) generation of an implementation from a given specification and have a long tradition in computer science. In the context of switching circuits, the synthesis problem was first raised by Church [8]

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 236–251, 2005.

and is nowadays well-understood. For temporal logical specifications, several synthesis algorithms have been suggested that rely on the close relationship between the synthesis and satisfiability problem or on a game-theoretic view, see e.g. [12, 15, 6, 16, 19, 18, 13]. The output of these synthesis algorithms are some kind of automata or state-transition graphs. Our goal is a step further toward an implementation by generating Reo code from a given automaton specification. Thus, our contribution is more in the spirit of gate-level hardware synthesis from given automata specifications. Our starting point is a specification of a component connector as a relation over *timed data streams* [7, 5], represented by a *constraint automaton* [4]. Constraint automata are variants of labeled transition systems that operationally describe the maximally parallel data-flow activity through the nodes in a Reo circuit. In [4], constraint automata are used to provide an operational semantics for coordination mechanisms formalized by composition of Reo connector graphs. In a constraint automaton, the states of the automaton represent the possible configurations (e.g., the contents of the FIFO-channels of the Reo-connector); transitions going out of a state represent data-flow at that state and its effect on the configuration.

In this paper we are not primarily concerned with the derivation of (constraint) automata representations from higher-level behavior specifications, such as in temporal logic or relations on timed data streams. Similar derivations, for instance, in the field of digital circuit design, are well-known. The main contribution of this paper is an algorithm that takes as input a constraint automaton $\mathcal{A}$ and produces a Reo connector graph that implements the relation on timed data streams specified by $\mathcal{A}$. This is tantamount to compiling an automaton down to actual concurrent executable code for a distributed implementation of the coordination behavior specified by that automaton. Superficially, compiling constraint automata specifications to Reo circuits seems simple. By analogy, derivation of digital circuits from Mealy automata specifications are well understood. However, constraint automata (and Reo circuits) can exhibit far more complex behavior than digital circuits, including combinations of synchrony and asynchrony, and relational, as well as simple (input/output) functional, interdependencies. In the light of this fact, it is far from obvious if synthesis of Reo circuits from constraint automata is possible at all, and if so, whether it can be done efficiently.

The rough idea of our synthesis algorithm is as follows. We first transform the automaton $\mathcal{A}$ into an equivalent *scheduled-data expression* which is a slight variant of an ordinary ω-regular expression. We then construct circuits for the atomic expressions and composition operators on Reo circuits that capture the semantics of concatenation, union, and infinity-closures. The major difficulty is the treatment of the atomic expressions that describe a complex "one-step" coordination scenario with possibly data-dependent synchronous and asynchronous behavior.

The rest of this paper is organized as follows. Section 2 contains a summary of the main features of Reo. Section 3 recalls the definition of constraint automata and their accepted TDS-languages. In Section 4, we show the equivalence of scheduled-data expressions and constraint automata. The construction of a Reo circuit from a given expression is explained in Section 5. Section 6 concludes the paper.

2 A Reo Primer

Reo [2] is a channel-based exogenous coordination model wherein complex coordinators, called *connectors*, are compositionally built out of simpler ones. The simplest connectors in Reo are a set of *channels* with well-defined behavior supplied by users. Components can instantiate, compose, connect to, and perform I/O operations through connectors. Here, as in [5, 4], we do not consider the dynamic creation, composition, and reconfiguration of connectors by components. We restrict our attention to connectors that have a static graphical representation as a *Reo circuit* which coordinates the data-flow through the channels connecting the input/output ports of components.

Reo's notion of *channel* is far more general than its common interpretation and allows for any primitive communication medium with exactly two ends. The channel ends are classified as *source* ends through which data enters and *sink* ends through which data leaves a channel. Although Reo allows for an open-ended set of channel-types with user-defined semantics, for our purposes in this paper, we restrict ourselves to the channel-types shown in Fig. 1.

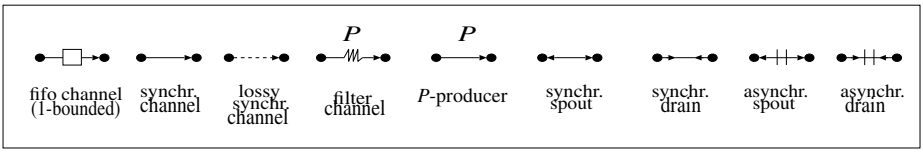


Fig. 1. Basic channel-types in Reo

The simplest form of an asynchronous channel is a *FIFO channel* with one buffer cell (called a 1-bounded FIFO channel or simply a FIFO1 channel). We graphically represent a FIFO1 channel by a small box in the middle of an arrow. In the example in Fig. 1, the left channel-end is a source, and the right end is a sink. The buffer is assumed to be initially empty if no data item is shown in the box (this is the case in Fig. 1). The graphical representation of a FIFO1-channel whose buffer initially contains a data element d shows d inside the box. FIFO channels with two or more buffer cells can be produced by composing several FIFO1 channels, as for instance, explained in [5, 4].

A *synchronous channel* (depicted as a simple solid arrow) has a source and a sink end, and no buffer. It accepts a data item through its source end iff it can simultaneously dispense it through its sink. A *lossy synchronous channel* (depicted as a dashed arrow) is similar to a synchronous channel, except that it always accepts all data items through its source end. If it is possible for it to simultaneously dispense the data item through its sink (e.g., there is a take operation pending on its sink) the channel transfers the data item; otherwise the data item is lost. For a *synchronous filter channel*, its "pattern" P (for our purposes here, formalized as a set $P \subseteq Data$) specifies the type of data items that can be transmitted through the channel. Any value $d \in P$ is accepted through its source end iff its sink end can simultaneously dispense d; all data items $d \notin P$ are always accepted through the source end but are immediately lost. The P-producer is a variant of a synchronous channel whose source end accepts any data item $d \in Data$, but the value dispensed through its sink end is always a data element $d \in P$.

More exotic channels permitted in Reo are (a)synchronous *drains* that have two source ends. Because a drain has no sink end, no data value can ever be obtained from these channels. Thus, a synchronous drain accepts a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well. All data accepted by this channel are lost. An asynchronous drain accepts and loses data items through its two source ends, but never simultaneously. Synchronous and asynchronous *spouts* are duals of their corresponding drain channel types, as they have two sink ends.

A complex connector has a graphical representation, called a *Reo circuit*, which can be produced by applying certain composition operators to channels. In our setting, where we do not consider dynamic aspects of the Reo language, a Reo-circuit is a finite graph where the *nodes* are labeled with pair-wise disjoint, non-empty sets of channel ends, and where the edges represent their connecting channels. The set of channel ends coincident on a node A is disjointly partitioned into the sets $\mathsf{Src}(A)$ and $\mathsf{Snk}(A)$, denoting the sets of source and sink channel ends that coincide on A, respectively. A node is called a *source node* if $\mathsf{Src}(A) \neq \emptyset \wedge \mathsf{Snk}(A) = \emptyset$. Analogously, A is called a *sink node* if $\mathsf{Src}(A) = \emptyset \wedge \mathsf{Snk}(A) \neq \emptyset$. Node A is called a *mixed node* if $\mathsf{Src}(A) \neq \emptyset \wedge \mathsf{Snk}(A) \neq \emptyset$. In this paper, it suffices to assume that all mixed nodes are hidden. In other words, we abstract away from their names and formalize the behavior of a Reo circuit by means of the data-flow at its sink and source nodes. Intuitively, source nodes of a circuit are analogous to the input ports, and sink nodes to the output ports of a component, while mixed nodes are its hidden internal details. Components cannot connect to, read from, or write to mixed nodes. Instead, data-flow through mixed nodes is totally specified by the circuits they belong to.

A component can write data items to a source node of a Reo circuit that it is connected to. A write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a *replicator*. A component can obtain data items from a sink node of a Reo circuit that it is connected to through input operations.[1] A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic *merger*. A mixed node is a self-contained "pumping station" that combines the behavior of a sink node (merger) and a source node (replicator) in an atomic iteration of an endless loop: in every iteration a mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends. A data item is suitable for selection in an iteration only if it can be accepted by all source channel ends that coincide on the mixed node.

Example 1 (Exclusive Router and Shift-Lossy FIFO1 Channel). Fig. 2 a. shows an implementation of an exclusive router built by composing five synchronous channels, two lossy synchronous channels and a synchronous drain. The intuitive behavior of this cir-

[1] We consider only the destructive take operation here which, e.g., on a FIFO channel, reads and removes the first data item in its buffer.

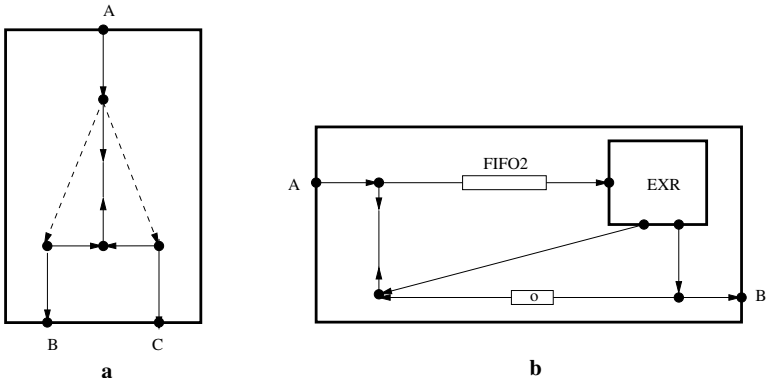


Fig. 2. Exclusive router and shift-lossy FIFO1 channel

cuit is that through its source node *A*, it obtains a data item *d* from its environment and delivers *d* to one of its sink nodes *B* or *C*. If both *B* and *C* are willing to accept *d* then (the merger in the mixed node in the middle of) the exclusive router nondeterministically decides to deliver *d* to either *B* or *C*. No data that passes through *A* can be lost because of the synchronous drain and the two synchronous channels in the middle of the circuit. The synchronous drain ensures that data flow at *A* is synchronized with data flow through the node at its opposite end. The merger inherent in this mixed node guarantees that at most one of its two coincident synchronous channels transfer data, synchronized with the data flow at either *B* or *C*.

The circuit in Fig. 2.b shows an implementation of a shift-lossy FIFO1 channel with source node *A* and sink node *B*. This implementation uses four synchronous channels, a synchronous drain, a FIFO1 channel whose buffer initially contains a token data item, *o*, an empty FIFO2 channel, and an instance of the exclusive router of Fig. 2.a shown as the box labeled EXR. A shift-lossy FIFO1 channel behaves the same as a FIFO1 channel, except that writing to its source end is never blocked. If at the time of a write operation its buffer is full, the stored data item in the buffer is lost and the new data item replaces it in the buffer. The observable behavior of each of these Reo circuits is represented by a constraint automaton in Fig. 3. Derivation of these constraint automata as compositions of the constraint automata representing the behavior of the individual primitives used in their respective Reo circuits appears in [4]. □

In spite of its simplicity, the semantics of Reo is indeed very rich, yielding a surprisingly expressive language [2]. For instance, the relational (as opposed to functional) dependencies that result in "propagation of synchrony" as well as the way in which the local behavior of, e.g., lossy synchronous channels imposes non-local constraints on a circuit, are already evident in the exclusive router of Fig. 2.a. (We use this exclusive router later in this paper in our synthesis of Reo circuits.) Examples of Reo circuits with more interesting behavior can be found elsewhere [1], and the reader is encouraged to see [17] and [5] for the simple, rich, and expressive formal semantics of Reo.

In the remainder of the paper, we discuss the synthesis problem of Reo circuits where the input specification of the desired coordination is given as a *constraint automaton*, as defined in the next section.

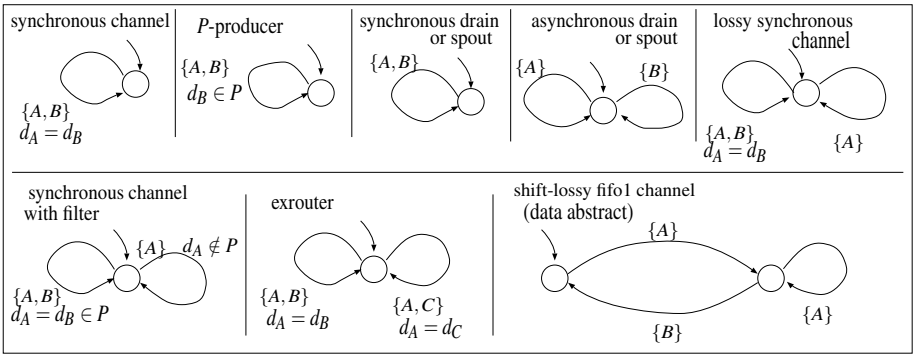
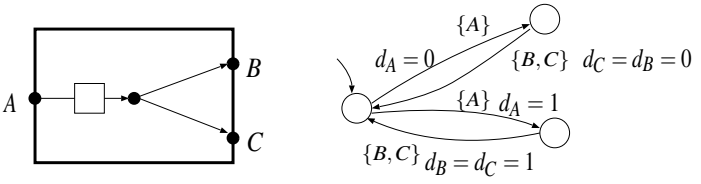


Fig. 3. Constraint automata for some basic channels in Reo

3 Constraint Automata

Constraint automata can serve as an operational model for Reo circuits [4]. The states of an automaton represent the configurations of its corresponding circuit (e.g., the contents of the FIFO channels), while the transitions encode its maximally-parallel stepwise behavior. The transitions are labeled with the maximal sets of nodes on which data-flow occurs simultaneously, and a data constraint (i.e., boolean condition for the observed data values). We start with a simple example for a constraint automaton that models a component with input port A and two output ports B and C which is modeled by a Reo circuit as shown in the left of the picture below.



The picture on the right shows the corresponding constraint automaton where we assume that only bits 0 and 1 can be transmitted through the channels. The initial state stands for the configuration where the buffer is empty, while the two other states represent the configurations where the buffer is filled with one of the data items. The outgoing transitions from the initial state are labeled with the singleton set $\{A\}$ which reflects the fact that in the initial configuration only data-flow at A is possible. If the buffer is filled then data-flow at A is impossible and only B and C can take the value from the buffer.

In the sequel, we specify constraint automata using a nonempty and finite set *Data* consisting of data items that can be sent (and received) via channels and a nonempty and finite set $\mathcal{N} = \{A_1, \ldots, A_n\}$ of names. Intuitively, we may think of the A_i's to be the source or sink nodes of a Reo circuit. We refer to the subsets of $\mathcal{N}$ as node-sets. A data assignment for $\emptyset \neq N \subseteq \mathcal{N}$ is a function $\delta : N \to \mathit{Data}$. $\mathit{DA}(N)$ denotes the set of all data assignments for N, and DA the set of all data assignments (on any N). Data *constraints*, which can be viewed as a symbolic representation of *sets* of data assignments, are formally defined as propositional formulas built from the atoms "$d_A \in P$"

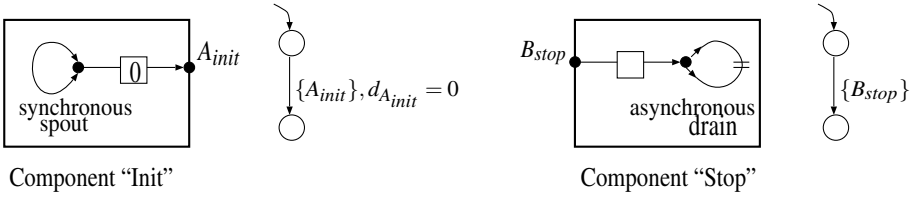


Fig. 4. Reo circuits and automata for an initializer and a terminator

and "$d_A = d_B$", where $A, B \in \mathcal{N}$, $d_A, d_B \in Data$, and $P \subseteq Data$. $DC(N)$ denotes the set of data constraints using only names from N, and DC is a shorthand for $DC(\mathcal{N})$. We simply write "$d_A = d$" rather than "$d_A \in \{d\}$". The symbol $\models$ stands for the obvious satisfaction relation which results from interpreting data constraints over data assignments. Satisfiability and logical equivalence $\equiv$ of data constraints are defined as usual.

Definition 1 (Constraint automata, [4]). A constraint automaton (over *Data*) is a tuple $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$ where Q is a finite set of states, $\mathcal{N}$ a finite set of nodes, $\longrightarrow$ is a finite subset of $Q \times (2^{\mathcal{N}} \times DC) \times Q$, called the transition relation, and $Q_0 \subseteq Q$ a nonempty set of initial states. We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \longrightarrow$ and require that (1) $N \neq \emptyset$ and (2) $g \in DC(N)$ is satisfiable. We call N the node-set and g the guard of the transition. States without any outgoing transition are called terminal. □

The intuitive meaning of a constraint automaton as an operational model for Reo connectors is similar to the interpretation of labeled transition systems as formal models for reactive systems. The sink and source nodes of a Reo connector circuit play the role of the nodes in its corresponding constraint automaton. The states represent the configurations of the connector. The meaning of a transition $q \xrightarrow{N,g} p$ is that in configuration q all the nodes $A_i \in N$ perform (synchronously) I/O-operations that meet the guard g, resulting in a new configuration p, while at the same moment there is no data-flow at the other nodes $A_i \in \mathcal{N} \setminus N$.

Example 2 (Constraint automata). Constraint automata for the various basic channels types, the exclusive router and shift-lossy FIFO1 channel are shown in Figure 3 (where valid guards have been omitted). The automaton for a FIFO1 channel with source A and sink B is the same as the one for the example in the beginning of the section, except that C has to be removed. These automata do not have terminal states as in any configuration data flow at some nodes is possible. The left part of Fig. 4 shows the Reo circuit for an initializer, i.e., a component without input ports (source nodes) and a single output port A_{init} where data-flow at A_{init} happens exactly once.[2] Thus, if we connect A_{init} with an input port A of another component $\mathcal{C}$ via a synchronous channel with source A_{init} and sink A then data-flow at A_{init} activates the data-flow at

[2] Data-flow at the node on the left, where the two sink ends of a synchronous spout coincide, is never possible because on the one hand, the sink ends of the spout are obligated to produce their respective data items simultaneously, while on the other hand the merge semantics of sink/mixed nodes does not allows for simultaneous data-flow at both sink ends.

C but prevents any "restart" of C. The situation is similar for the component "Stop" on the right of the picture where the source node B_{stop} can put a value into the buffer exactly once, because afterward the buffer is filled forever as no data-flow is possible for an asynchronous drain with both source ends coincident on the same node. Thus, if an output port B of a component C is connected via a synchronous channel with B_{stop} then output at B is possible exactly once. In this sense, component "Stop" can serve to terminate data-flow in other components. □

In [4], we formalized the semantics of a constraint automaton as a relation on timed data streams. For the purposes of this paper, an equivalent, but simpler concept suffices which abstracts away from time and describes the "traces" of a constraint automaton by *scheduled-data streams*: finite or infinite sequences of pairs $\langle N, \delta \rangle$, consisting of a set N of all the nodes that are scheduled to be synchronously (i.e., atomically) active in the next step, together with a data assignment $\delta \in DA(N)$ describing the data values that are input and output.

Definition 2 (Scheduled-data streams, generated language). A scheduled-data stream $\Theta = \Theta(0);\Theta(1);\ldots$ is a finite or infinite sequence of pairs $\Theta(i) \in 2^{\mathcal{N}} \times DC$, denoted by

$$\Theta(i) = \langle \underbrace{\Theta.N(i)}_{\text{node-set}}, \underbrace{\Theta.\delta(i)}_{\text{data assignment}} \rangle,$$

such that $\Theta.N(i)$ is a non-empty node-set and $\Theta.\delta(i)$ a data assignment for $\Theta.N(i)$. We write $|\Theta|$ to denote the length of Θ (which can be ω). The empty scheduled-data stream is denoted by ε. $SDS_{\mathcal{N}}$ or briefly SDS denotes the set of all scheduled-data streams.
Let $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$ be a constraint automaton, $\Theta \in SDS$ and q a state in $\mathcal{A}$. A q-run for Θ in $\mathcal{A}$ is a path in $\mathcal{A}$

$$\mathbf{q} = q_0 \xrightarrow{N_0,g_0} q_1 \xrightarrow{N_1,g_1} q_2 \xrightarrow{N_2,g_2} \ldots$$

such that (1) $q_0 = q$ and (2) either $\mathbf{q}$ and Θ are infinite or $\mathbf{q}$ consists of $|\Theta|$ transitions and ends in a terminal state and (3) $N_i = \Theta.N(i)$, $\Theta.\delta(i) \models g_i$ for all $0 \leq i < |\Theta|$. The generated language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of all scheduled-data streams $\Theta \in SDS$ which have a q_0-run in $\mathcal{A}$ for some initial state $q_0 \in Q_0$. □

For instance, the SDS-language generated by the automaton for a synchronous channel consists of all infinite scheduled-data streams Θ with $\Theta.N(i) = \{A, B\}$ and where data assignment $\Theta.\delta(i)$ assigns the same data item to A and B.

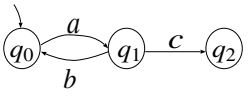
Although the formal definition of scheduled-data streams does not impose a relation between the data assignments $\Theta.\delta(i)$, for a given constraint automaton, there can be a link between the data constraints $\Theta.\delta(i)$ and $\Theta.\delta(i+1)$. For instance, the automaton for a FIFO1 channel with source node A and sink node B generates the SDS-language consisting of all infinite scheduled-data streams Θ with $\Theta.N(2i) = \{A\}$, $\Theta.N(2i+1) = \{B\}$, and with $\Theta.\delta(2i) = [A \mapsto d]$ and $\Theta.\delta(2i+1) = [B \mapsto d]$, for some $d \in Data$.

In [4], we explain how an automaton for a Reo circuit can be constructed in a compositional way. (For the purpose of this paper, the details of that construction do not matter. The only thing that we use later, in Section 5, is that by applying the above definition to the automaton for a Reo circuit R, we obtain an SDS-language $\mathcal{L}(R)$ for R.) In what follows we show, conversely, how to construct a Reo circuit from a constraint automaton.

4 Scheduled-Data Expressions

The first step of our construction of a Reo circuit from a given automaton is to transform the automaton into an equivalent ω-regular expression, a so-called *scheduled-data expression*. These are built by ε representing the singleton SDS-language $\{\varepsilon\}$ and the atoms $\langle N, g\rangle$ where $\emptyset \neq N \subseteq \mathcal{N}$ and g is a satisfiable data constraint for N. The SDS-language $\mathcal{L}(\langle N, g\rangle)$ consists of all scheduled-data streams Θ of length 1 such that $\Theta.N(0) = N$ and $\Theta.\delta(0) \models g$. Moreover, we use the standard composition operators; (concatenation), $\cup$ (union) and the closure operators α^ω (infinitely many repetitions) and α^∞ (finite or infinite repetitions). The formal definition of $\mathcal{L}(\alpha)$ for composite expressions is defined as for ordinary ω-regular expressions and is omitted here.

Similar to the construction of a finite automaton from ordinary regular expressions (see e.g. [14]), we can assign a constraint automaton to any scheduled-data expression that generates the same SDS-language and which is linear in the size of the expression. Since this construction does not play a role in the present paper, its description is omitted. Instead, we use the reverse construction, i.e., of a scheduled-data expression for a constraint automaton. Although to do so, we may apply the standard algorithms for generating (ω-)regular expressions from automata (see e.g. [14]), we suggest here an alternative algorithm. Rather than describing the construction in general, we treat a typical example. Consider the constraint automaton as shown on the left of the following picture where a, b, c are pairs of node-sets with corresponding data constraints.



$$\begin{aligned} \alpha_0 &= a;\alpha_1 \\ \alpha_1 &= (b;\alpha_0) \cup (c;\alpha_2) \\ \alpha_2 &= \varepsilon \end{aligned}$$

Let α_i denote the scheduled-data expression corresponding to (the SDS-language generated by) state q_i, for $i = 0, 1, 2$. The three transitions of this automaton give rise to three equations for the expressions as shown above. Together, they imply the following equation: $\alpha_0 = (a;b;\alpha_0) \cup (a;c)$. This equation can be solved, using the following general laws for scheduled-data expressions: "if $\alpha = (\beta;\alpha) \cup \gamma$ and $\varepsilon \notin \beta$ then $\alpha = \beta^\infty;\gamma$" and "if $\alpha = \beta;\alpha$ and $\varepsilon \notin \beta$ then $\alpha = \beta^\omega$". Applying the first law to the equation above yields the expression $\alpha_0 = (a;b)^\infty;a;c$ for the state q_0.

5 From Scheduled-Data Expressions to Reo

We now address the issue of constructing a Reo circuit for a scheduled-data expression α_0. Because the source and the sink nodes of a Reo circuit play different roles with respect to its environment, and this distinction is abstracted away in scheduled-data expressions (and constraint automata), we first need to identify the "input" and "output" of a circuit by partitioning its node set $\mathcal{N}$. That is, our starting point is a description of a component connector by its input ports $C_1, \ldots, C_n$ and its output ports $D_1, \ldots, D_m$ and by (the scheduled-data expression α_0 of) a given constraint automaton that specifies the observable data flow at the C_i's and D_j's.

In the sequel, let $\mathcal{N} = \{C_1, \ldots, C_n\} \cup \{D_1, \ldots, D_m\}$ contain all nodes occurring in the node-sets N of the atoms $\langle N, g\rangle$ in α_0, where we assume that the C_i's are source

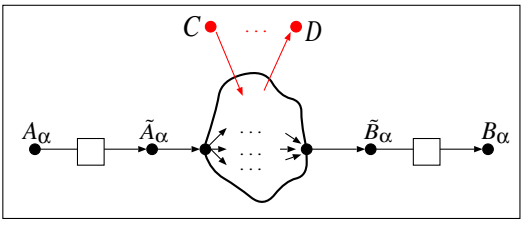


Fig. 5. Structure of the Reo-circuit R_α

nodes and the D_j's are sink nodes. Our goal is the construction of a Reo circuit R with source nodes $C_1,\ldots,C_n$ and sink nodes $D_1,\ldots,D_m$ such that $\mathcal{L}(\alpha_0) = \mathcal{L}(R)$.

For the construction of R, we use a compositional approach that builds a Reo circuit R_α for each subexpression α of α_0. Fig. 5 shows the general structure of R_α: if the source node A_α is fed from outside with some data element, then it is put into the buffer between A_α and $\tilde{A}_\alpha$. As soon as $\tilde{A}_\alpha$ takes the data element from the buffer, the sub-circuit in the middle is "activated". Similarly, data-flow inside this sub-circuit stops as soon as a data element arrives at $\tilde{B}_\alpha$, which puts it into the buffer between $\tilde{B}_\alpha$ and B_α. Thus, data-flow at the sink node $\tilde{B}_\alpha$ can be viewed as a signal that R_α has "terminated".

The nodes C, D in Fig. 5 are there to indicate that there will be some channels connecting the sub-circuit in the middle of R_α with (some of) the source nodes C and (some of) the sink nodes D in $\mathcal{N}$. The construction of a circuit R for an expression α_0 will be completed by a last step, in which "Init" and "Stop" components, defined in Example 2, are added to begin and end the data-flow of in the circuit R_{α_0}, as shown in Fig. 6. The construction of the circuit will be such that at any moment, *exactly one* of the leftmost and rightmost buffers or buffers inside R_{α_0} will be filled. Thus, we may consider data-flow through R as a *token game*, where the token is passed on from left to right. The reason why we put R_{α_0} in the context of an initializer and a terminator is that

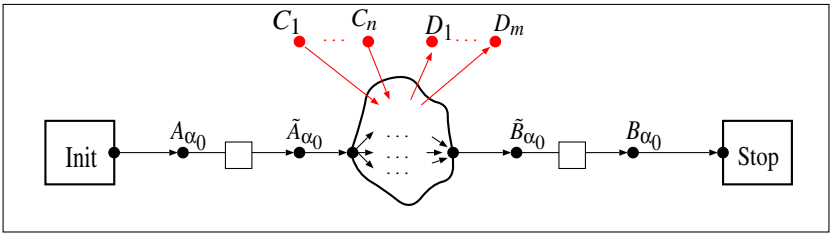
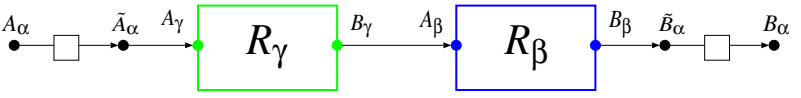


Fig. 6. The final Reo-circuit R

the circuit R_{α_0} allows a "restart" of data flow at node A_{α_0} whenever $\tilde{A}_{\alpha_0}$ has consumed the data item in the buffer between A_{α_0} and $\tilde{A}_{\alpha_0}$. In fact, the initializer ensures that data flow at A_{α_0} occurs exactly once. The reason for using the stop-component is similar.

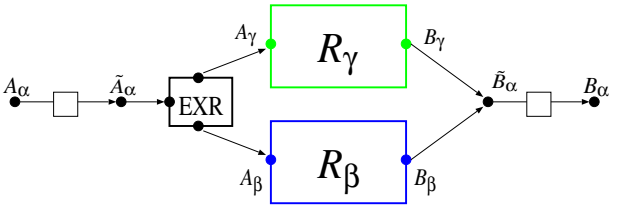
Concatenation, union and closure. We first explain how to construct a circuit R_α, assuming we have already constructed the circuits for α's subexpressions. (If a subexpression α occurs more than once in α_0, e.g. if $\alpha_0 = \alpha;\alpha$, then we need a copy of the

circuits R_α for every syntactic occurrence of α as a subexpression in α_0.) For $\alpha = \gamma;\beta$ the Reo circuit R_α results from combining R_γ and R_β as follows:

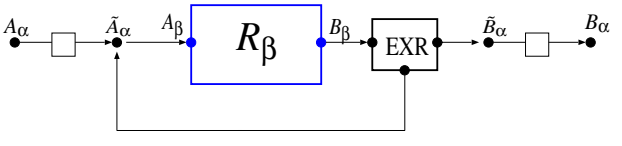


Note that the internal FIFO-channels "at the end" of R_γ and "at the beginning" of R_β (not drawn in the picture) ensure that in the concatenation $\gamma;\beta$ data-flow inside R_β cannot start before data-flow in R_γ has finished.

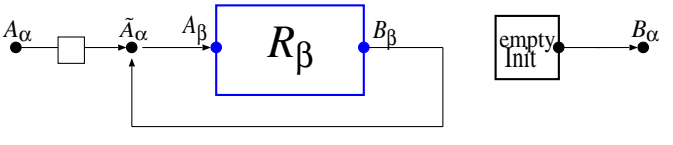
For $\alpha = \gamma \cup \beta$, the Reo circuit R_α is obtained by combining R_γ and R_β with an exclusive router that nondeterministically chooses to "activate" the data-flow in either R_γ or R_β:



The Reo circuit R_α where $\alpha = \beta^\infty$ is obtained from R_β as follows:[3]



For $\alpha = \beta^\omega$, the Reo circuit has the following structure.



Here, "empty Init" is a variant of the initializer in Ex. 2, where the buffer is initially empty. Thus, data-flow never occurs in "empty Init" or at node B_α. Being non-reachable, it may be omitted; we keep it here so that the circuit retains the general shape of Fig. 5.

The empty expression. For $\alpha = \varepsilon$, we simply use a FIFO1 channel with its source end on node A_ε and its sink end on node B_ε. (Using just a single channel departs from the general schema sketched in Fig. 5, but the nodes $\tilde{A}_\alpha$ and $\tilde{B}_\alpha$ are not needed in our compositional approach.)

Atomic expressions. So far the construction of Reo circuits for composite expressions has followed patterns that are familiar from automata theory. Next we come to the most

[3] The syntax of scheduled-data expressions does not include the Kleene closure $\alpha = \beta^*$. However, it could be treated by simply replacing the exclusive router with a fair exclusive router.

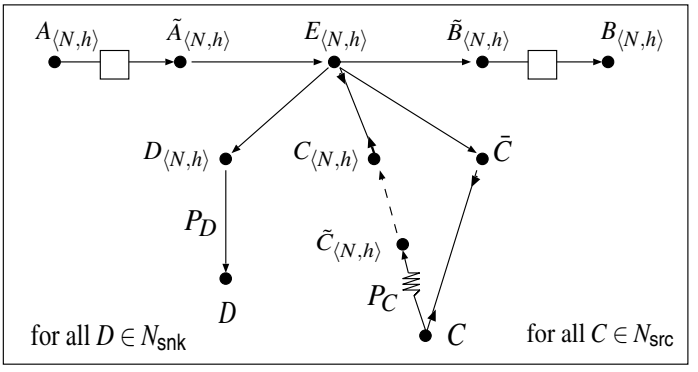


Fig. 7. Reo-circuit $R_{\langle N,h\rangle}$

complicated and most interesting step in our construction, namely the construction of a Reo circuit for atomic expressions $\langle N,g\rangle$. The difficulty lies in the fact that such expressions model a computation step of a corresponding Reo circuit, in which certain channel ends are active and others are not. Moreover, we must ensure that at every active channel end, the right data value is input or output.

Let Atoms denote the set of all atomic expressions $\langle N,g\rangle$ of α_0. Recall that N is a nonempty subset of $\mathcal{N} = \{C_1,\ldots,C_n\} \cup \{D_1,\ldots,D_m\}$ and g is a satisfiable data constraint for the nodes in N. We first describe a general technique to design a Reo circuit for the atoms $\langle N,g\rangle \in$ Atoms. (Later we explain how this technique can be made more efficient in various ways.) We first transform g into its canonical disjunctive normal form, which replaces it with an equivalent data constraint $h_1 \vee \ldots \vee h_r$ where each of the h's is a formula of the form

$$h \;=\; \bigwedge_{C \in N_{\mathsf{src}}} (d_C \in P_C) \wedge \bigwedge_{D \in N_{\mathsf{snk}}} (d_D \in P_D)$$

with $N_{\mathsf{src}} = N \cap \{C_1,\ldots,C_n\}$, $N_{\mathsf{snk}} = N \cap \{D_1,\ldots,D_m\}$ and P_C, $P_D \subseteq \mathit{Data}$. E.g., if g is "$d_C = d_D$" then we replace g with $\bigvee_{d \in \mathit{Data}} h_d$ where h_d is $(d_C = d) \wedge (d_D = d)$. Next, we replace $\langle N,g\rangle$ with the equivalent expression $\langle N,h_1\rangle \cup \ldots \cup \langle N,h_r\rangle$, construct the circuits $R_{\langle N,h_k\rangle}$ (see below) and combine them with the union-operator described above. With the formula h as above, a circuit $\mathcal{R}_{\langle N,h\rangle}$ for $\langle N,h\rangle$ is presented in Fig. 7, which we now explain. For the Reo circuit $R_{\langle N,h\rangle}$ of a given $\langle N,h\rangle$, we need a pair of nodes $C_{\langle N,h\rangle}$ and $\tilde{C}_{\langle N,h\rangle}$ for every source node $C \in N_{\mathsf{src}}$, and similarly, one node $D_{\langle N,h\rangle}$ for every sink node $D \in N_{\mathsf{snk}}$, plus one other node $E_{\langle N,h\rangle}$. The same node $\bar{C}$ must be used for all circuits $R_{\langle M,f\rangle}$ where $C \in M$ and $\langle M,f\rangle \in$ Atoms, while the nodes $C_{\langle N,h\rangle}$ and $\tilde{C}_{\langle N,h\rangle}$ are unique for every atomic data expression $\langle N,h\rangle \in$ Atoms where $C \in N$.

We can think of the node $E_{\langle N,h\rangle}$ as a switch that synchronizes the data-flow in the upper sub-circuit with the nodes $D_{\langle N,h\rangle}$, D, and $C_{\langle N,h\rangle}$, $\tilde{C}_{\langle N,h\rangle}$, $\bar{C}$, and C for all source nodes $C \in N_{\mathsf{src}}$ and all sink nodes $D \in N_{\mathsf{snk}}$. The synchronous channel from $E_{\langle N,h\rangle}$ to $D_{\langle N,h\rangle}$ and the P_D-producer connecting $D_{\langle N,h\rangle}$ with D ensure that any data-flow at $E_{\langle N,h\rangle}$ is synchronized with the receipt of a value $d \in P_D$ at sink node D.

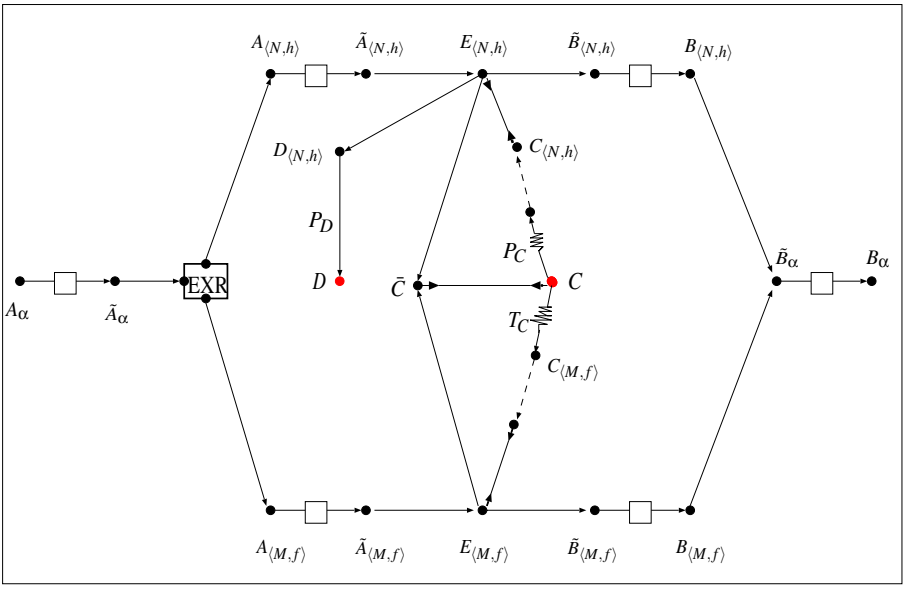


Fig. 8. Reo circuit R_α for $\alpha = \langle N,h\rangle \cup \langle M,f\rangle$ where $N = \{C,D\}$, $M = \{C\}$

For the source nodes C, the situation is a bit more complicated because we must ensure that C accepts an input value iff C synchronizes with exactly one of the nodes $E_{\langle M,f\rangle}$ where $\langle M,f\rangle$ is a subexpression of α_0 with $C \in M$. The use of perfect synchronous channels is not appropriate because of the replicator semantics of the source nodes. If C were connected with $E_{\langle N,h\rangle}$ via perfect synchronous channels only, then data-flow would block when C appears in two or more atomic subexpressions of α_0. (Note that simultaneous data-flow at different nodes $E_{\langle N,h\rangle}$, $E_{\langle M,f\rangle}$ is not possible.) For this reason, we connect C with $E_{\langle N,h\rangle}$ via a filter channel, a lossy synchronous channel and a synchronous drain through the nodes $C_{\langle N,h\rangle}$ and $\tilde{C}_{\langle N,h\rangle}$. These three channels (1) allow C to pass values even when $E_{\langle N,h\rangle}$ is not available to synchronize with C, and (2) force C to pass a value $d \in P_C$ when it synchronizes with $E_{\langle N,h\rangle}$. To prevent C from passing a value without synchronizing with one of the nodes $E_{\langle M,f\rangle}$ where $C \in M$, we use a synchronous channel connecting $E_{\langle N,h\rangle}$ with $\bar{C}$ and a synchronous drain between $\bar{C}$ and C. These channels ensure that for $C \in M$, C is active exactly when data-flow occurs at $\bar{C}$ and exactly one of the nodes $E_{\langle M,f\rangle}$.

A concrete example for the Reo-circuit which is constructed from the scheduled-data expression $\alpha = \langle N,h\rangle \cup \langle M,f\rangle$ is shown in Fig. 8. Here, we assume that $N = \{C,D\}$, $M = \{C\}$ and h is $(d_C \in P_C) \wedge (d_D \in P_D)$ while f is $d_C \in T_C$. The proof for the correctness of our synthesis algorithm is quite technical and omitted here.

Size of the constructed circuit. In the worst case, the treatment of the atoms $\langle N,g\rangle$ leads to an exponential blow-up (because every disjunctive normal form for g may be exponentially longer than g). However, when we assume that all data constraints in α_0 are given in canonical disjunctive normal form and when we measure the length of α_0

as the total length of all data constraints occurring in (one of the atoms in) α_0 then the total number of channels in the constructed circuit is *linear* in the length of α_0.

Preprocessing. We now explain how a preprocessing phase of the set Atoms can simplify the construction of the circuits for the atomic subexpressions of α_0. We first look for pairs $\langle C,D\rangle$ with $C \in \{C_1,\ldots,C_n\}$, $D \in \{D_1,\ldots,D_m\}$ such that for all $\langle N,g\rangle \in$ Atoms either $\{C,D\} \cap N = \emptyset$ or $\{C,D\} \subseteq N$ and $g \leq d_C = d_D$. ($\leq$ denotes logical implication.) Then, we establish a synchronous channel with its source end on node C, its sink end on node D and remove D in the sense that any $\langle N,g\rangle \in$ Atoms is replaced with $\langle N \setminus \{D\}, g[d_D/d_C]\rangle$ where $g[d_D/d_C]$ means the data constraint resulting from g by the syntactic replacement of any occurrence of d_D with d_C. Second, for any pair (C_i,C_j) of source nodes such that for all $\langle N,g\rangle \in$ Atoms either $\{C_i,C_j\} \cap N = \emptyset$ or $\{C_i,C_j\} \subseteq N$ and d_{C_j} does not occur in g, we establish a synchronous drain connecting C_i and C_j and remove C_j from Atoms. The same technique can be applied to sink nodes D_i, D_j such that for all $\langle N,g\rangle \in$ Atoms either $\{D_i,D_j\} \cap N = \emptyset$ or $\{D_i,D_j\} \subseteq N$ and d_{D_j} does not occur in g, where we generate a synchronous spout with its sink ends D_i and D_j and remove D_j. Finally, we look for sink nodes D_i,D_j such that $\langle N,g\rangle \in$ Atoms implies $\{D_i,D_j\} \cap N = \emptyset$ or $\{D_i,D_j\} \subseteq N$ and $g \leq d_{D_i} = d_{D_j}$ and insert a new sink node D_{ij} with synchronous channels from D_{ij} to D_i and D_j. We then remove D_i, D_j from Atoms and treat D_{ij} as a sink node. A similar transformation $\langle C_i,C_j\rangle \rightsquigarrow C_{ij}$ applies to source nodes such that for all $\langle N,g\rangle \in$ Atoms either $\{C_i,C_j\} \cap N = \emptyset$ or $\{C_i,C_j\} \subseteq N$ and $g \leq d_{C_i} = d_{C_j}$. However, here we need a Reo connector that checks the equality of two (synchronously) arriving input values.

Optimization. As in other algorithmic constructions, our resulting Reo circuits contain certain redundancies which can be optimized away. We can detect and remove them by applying circuit transformation rules that look for recognizable patterns of subcircuits and replace them with their simpler equivalents. For instance, every occurrence of a synchronous channel preceding or following any other channel X can be simplified to only X. In Fig. 8, there are multiple candidates that qualify for the application of various circuit transformation rule. For example, we know that in Fig. 8, data-flow can occur through only one of the top or bottom branches of the circuit (because there is only one token at a time that passes through the entire circuit; the exclusive router; and because the two branches are isolated from one another by drains). This makes the right-hand-side FIFO1 channels on both top and bottom branches redundant.

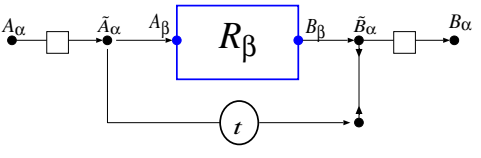
6 Conclusion

The main contribution of the present paper is a general construction of a Reo circuit from a constraint automaton. Although similar constructions exist in the classical area of automata and digital circuits, the situation here is far more complicated because of two major differences: (1) The behavior specified by constrained automata is generally not functional (from input to output) but relational. (2) In a digital circuit and the Mealy automaton describing it, behavior is always synchronous. In contrast, in Reo, behavior can be synchronous, asynchronous, or (at different steps) a combination of the two. Because of in particular point 2, the classical construction of a circuit from an automaton

breaks down, and at forehand, it was by no means obvious how to tackle the problem for Reo. We, therefore, see the algorithm described in the present paper as a major step forward in the automatic synthesis of Reo component connector circuits.

From the theoretical point of view, the results established here and in [4] yield that Reo connector circuits, constraint automata, and scheduled-data streams have the same expressiveness and can be transformed into each other via algorithmic transformations. This result can also be useful in practice as it allows to switch between these three formalisms. For instance, it enables one to use automata-models within the Reo framework to describe (and finally to synthesize) the interfaces of black-box components. On the other hand, our algorithm also illustrates the expressive power of the channel types presented in Fig. 1. (Note that our construction uses all of them, except for the asynchronous spout.)

To some extent, our construction can also be modified to treat real-time constraints, e.g., those formalized by timed scheduled-data expressions of the form $\alpha = \beta^{\leq t}$ stating that data flow described by β must be completed within t time units. (See [3] for a formal treatment of real-time within the Reo framework.). For this, we just connect the node $\tilde{A}_\alpha$ to the node $\tilde{B}_\alpha$ via a synchronous drain and a timer channel with off-option, i.e., a timer channel that allows the timer to be stopped at any point in time before the expiration of its delay. In the picture below, this timer channel is depicted by an arrow with a circle labeled with the delay t in its middle.



A compositional approach similar to the one we suggest here can also be used to provide "Reo-implementations" for processes specified in terms of CCS- or CSP-like process algebras. In fact, some of the typical operators are already included in regular expressions (CCS-like nondeterminism corresponds to union, sequential composition to concatenation, and ε to, e.g., the CCS-process nil). Parallel composition with CCS- or CSP-like synchronization can be realized by establishing appropriate synchronous channels and Reo's join operator. A LOTOS-like disrupt operator $P[>Q$ can be obtained using a Reo component that realizes a switch; this switch is initially "on" and synchronizes with P as long as it is "on" but is turned "off" by Q's first activity (the inhibitor circuit in [2] can be used to construct this switch from our set of primitive channels).

Although the construction presented here is not overly complicated, it can and should still be simplified further and made more efficient. Parts of such considerations have already been sketched in the present paper. In our future work, we will investigate further optimizations and the design of an alternative synthesis algorithm that goes directly from automata to Reo circuits without having the regular expressions as an intermediate step. Furthermore, dynamic reconfiguration of connector circuits is an inherent aspect of Reo that we plan to cover in our future work.

References

1. F. Arbab. Abstract behavior types: A foundation model for components and their composition. In *[11]*, pages 33–70, 2003.
2. F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):1–38, 2004.
3. F. Arbab, C. Baier, F. de Boer, and J. Rutten. Models and temporal logics for timed component connectors. In *Proc. SEFM'04*. IEEE CS Press, 2004.
4. F. Arbab, C. Baier, J.J.M.M. Rutten, and M. Sirjani. Modeling component connectors in reo by constraint automata. In *FOCLASA'03*, volume 97 of *ENTCS*, pages 25–41, 2004. Full version see http://web.informatik.uni-bonn.de/I/baier/publikationen.html.
5. F. Arbab and J.J.M.M. Rutten. A coinductive calculus of component connectors. In *Recent Trends in Algebraic Development Techniques, Proc. 16th Int. Workshop on Algebraic Development Techniques (WADT 2002)*, volume 2755 of *LNCS*, pages 35–56, 2003.
6. P.C. Attie and E.A. Emerson. Synthesis of concurrent systems with many similar sequential processes. In *Proc. POPL*, ACM Press, pages 191–201, 1989.
7. M. Broy and K. Stolen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement*. Springer-Verlag, 2000.
8. A. Church. Logic, arithmetic and automata. In *Proc. Int. Congress of Mathematicians*, pages 23–35. Institut Mittag-Leffler, 1962.
9. CIM. http://www.almende.com/cim/.
10. D. Clarke, D. Costa, and F. Arbab. Modeling coordination in biological systems. In *Proc. of the Int. Symposium on Leveraging Applications of Formal Methods (ISoLA 2004)*, 2004.
11. N. Diakov and F. Arbab. Compositional construction of web services using Reo. In *Proc. International Workshop on Web Services: Modeling, Architecture and Infrastructure (ICEIS 2004), Porto, Portugal, April 13-14*, 2004.
12. E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronous skeleton. *Science of Programming*, 2:241–266, 1982.
13. T. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. In *Proc. 30th Int. Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2719 of *LNCS*, pages 886–902, 2003.
14. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Language, and Computation*. Addison–Wesley, 2nd edition edition, 2001.
15. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6:68–93, 1984.
16. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th Symposium on Principles of Programming Languages*, pages 179–190. ACM Press, 1989.
17. J.J.M.M. Rutten. Component connectors. In *[?]*, chapter 5, pages 73–87. 2004.
18. W. Thomas. On the synthesis of strategies in infinite games. In *Proc. of the 12th Annual Symp. on Theoretical Aspects of Computer Science*, volume 900 of *LNCS*, pages 1–13, 1995.
19. M. Vardi. An automata-theoretic approach to fair realizability and synthesis. In *Proc. CAV*, volume 939 of *LNCS*, pages 267–278, 1995.
20. Z. Zlatev, N. Diakov, and S. Pokraev. Construction of negotiation protocols for E-Commerce applications. *ACM SIGecom Exchanges*, 5(2):11–22, November 2004.

Tagged Sets: A Secure and Transparent Coordination Medium

Manuel Oriol and Michael Hicks

University of Maryland, College Park MD 20742, USA
{oriol, mwh}@cs.umd.edu
http://www.cs.umd.edu/~{oriol, mwh}

Abstract. A simple and effective way of coordinating distributed, mobile, and parallel applications is to use a virtual shared memory (VSM), such as a Linda tuple-space. In this paper, we propose a new kind of VSM, called a *tagged set*. Each element in the VSM is a value with an associated tag, and values are read or removed from the VSM by matching the tag. Tagged sets exhibit three properties useful for VSMs:

1. *Ease of use.* A tagged value naturally corresponds to the notion that data has certain attributes, expressed by the tag, which can be used for later retrieval.
2. *Flexibility.* Tags are implemented as propositional logic formulae, and selection as logical implication, so the resulting system is quite powerful. Tagged sets naturally support a variety of applications, such as shared data repositories (e.g., for media or e-mail), message passing, and publish/subscribe algorithms; they are powerful enough to encode existing VSMs, such as Linda spaces.
3. *Security.* Our notion of tags naturally corresponds to keys, or capabilities: a user may not select data in the set unless she presents a legal key or keys. Normal tags correspond to symmetric keys, and we introduce *asymmetric tags* that correspond to public and private key pairs. Treating tags as keys permits users to easily specify protection criteria for data at a fine granularity.

This paper motivates our approach, sketches its basic theory, and places it in the context of other data management strategies.

1 Introduction

Computer users require means to store, share, retrieve, and compute data to perform a myriad of tasks. Currently, these means are provided in different ways in different settings, ranging from relational databases to file systems to individual applications. To be useful, any data management approach must answer basic questions concerning organization and security:

1. Is the data organized so that relevant information can be easily found? Is the organizational model easy to use and understand?
2. Is the data protected from malicious tampering? Are the policies for doing so flexible and easy to use?

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 252–267, 2005.

Relational databases are extremely flexible and optimized for concurrency, fault-tolerance, and throughput. However, they can be difficult to use, particularly in setting up and managing schemas. File systems are easy to understand, and support flexible and intuitive security policies, but have a limited organizational capacity. Linda-style tuple spaces [4], and more generally *virtual shared memories* (VSMs), have a simple and effective organizational strategy, but typical Linda spaces have limited support for security.

In this paper, we propose to manage data as *tagged values* forming part of a *tagged set*. Our approach is inspired by the simplicity and power of the many applications that use tagging as their organizational mechanism, including Google Mail (GMail)[1] and the iLife suite (iTunes, iPhoto, etc.).[2]

A tagged value is merely some data with attached meta-information specified as a tag. Tags are typically used to organize data. For example, iTunes uses the notion of a playlist to organize songs. A playlist is essentially a kind of tag, with each song tagged with the playlist (or playlists) it belongs to, and perhaps the order in which it should played for a given list. A photo album in iPhoto is a similar idea. A file system can also be viewed as tagging system by considering each directory name as a tag; a file "stored" in some directory d is tagged with d. The richer the language for tags, the more organizational traits one can express. In our approach, we encode tags as propositional logic formulae; selecting tagged data from a set is done by logical implication. This approach is powerful enough to easily construct the above examples, as well as to encode more structured repositories, like Linda-style tuple spaces.

In our system, tags serve not only to organize data, but also to protect it from unauthorized access. A tag corresponds naturally to the idea of a key (or capability). A user may not select data in a tagged set unless she presents the keys that protect it. Normal tags correspond to symmetric keys, and we introduce *asymmetric tags* that correspond to public and private key pairs. Treating tags as keys permits users to easily specify protection criteria for data at a fine granularity. Like a file system, individual tagged values can have widely differing access policies. We illustrate this idea by encoding a secure GMail, and extending the Linda-space encoding to incorporate security tags.

Treating tags as keys naturally lends itself to a distributed setting, which is important if tagged sets are to be used as a coordination medium between cooperating applications. By literally using tags as cryptographic keys, we can encrypt data to ensure it can only be read by the appropriate key holder. (As expected, with asymmetric tags we can do this without requiring the host of the tagged set know a user's private tag/key.)

The contributions of this paper are as follows:

- We present a simple formalism for tagged sets (Section 2). The key novelty of our approach is the use of propositional logic as the language of tags, and logical implication as the means to select tagged data.

[1] `http://www.gmail.com`
[2] `http://www.apple.com/ilife/`

- We show how tags can be treated as keys in order to protect data at a fine granularity (Section 3). We prove a confidentiality theorem that loosely states that one cannot select data protected by some tag t unless he is in possession of that tag. Tags can be used as cryptographic keys, both symmetric and asymmetric, for secure sharing over a distributed medium.
- We show how tagged sets compare to related approaches in terms of security, flexibility, and performance (Section 4).

2 Tagged Sets

At the most basic level, data stored within a repository can be *tagged* with attributes that describe the data. More formally, a repository is a multi-set of pairs $\langle \tau, v \rangle$, where each pair consists of a *tag* τ and a *value* v. We use the $\{\!|\cdot|\!\}$ notation to clarify our use of multi-sets (which can contain more than one copy of the same element).

As an example, consider an audio repository S containing 3 clips ($clip_1$, $clip_2$, and $clip_3$), each of which is tagged to indicate its genre, drawing from topics Jazz, Classical, and Blues. If each clip falls squarely under one genre, we can tag them as such:

$$S_0 = \{\!|\langle \mathsf{Jazz}, clip_1 \rangle, \langle \mathsf{Classical}, clip_2 \rangle, \langle \mathsf{Blues}, clip_3 \rangle|\!\}$$

Naturally, a clip could be described by more than one genre. For example, if clip $clip_1$ is in genres Jazz and Blues, we could set up the repository as:

$$S = \{\!|\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1 \rangle, \langle \mathsf{Classical}, clip_2 \rangle, \langle \mathsf{Blues}, clip_3 \rangle|\!\}$$

To select the clips belonging to a particular genre, we perform a *selection* operation (designated $\downarrow$) on the repository. For example, to select the clips in genre Blues, we would have:

$$S \downarrow \mathsf{Blues} = \{\!|\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1 \rangle, \langle \mathsf{Blues}, clip_3 \rangle|\!\} \tag{1}$$

2.1 Selection as Logical Implication

We naturally think of selection as a kind of matching: to select with tag Jazz yields those elements whose tags contain Jazz. However, by considering the selection tag and the data tags as propositions, we can view selection more generally as a kind of implication: selecting tag t in set S yields those elements in S whose tags τ are *implied* by t.

This notion is made precise in Figure 1, which gives the syntax and semantics of *tagged sets*. A tagged set S represents a multi-set of tagged values. Sets are defined by set literals $\{\!|\langle \tau, v \rangle|\!\}$, possibly modified by operators $\cup$, $\downarrow$, $\downarrow_n$, $-$, and $-_n$, discussed below. Values v in these sets are drawn from the countably-infinite set V; their exact makeup is not important for our purposes. Tags are constructed from tag literals t (drawn from the countably-infinite set T), $\top$ (representing "all tags"), and standard operators $\vee$ ("or") and $\wedge$ ("and"). (We do not have $\neg$ or $\bot$

Syntax:

tag literals	$\mathtt{t} \in$	T
values	$v \in$	V
tags	$\tau ::=$	$\mathtt{t} \mid \tau \vee \tau \mid \tau \wedge \tau \mid \top$
tagged set	$S ::=$	$\emptyset \mid \{\!\mid\langle \tau, v\rangle \mid\!\} \mid S \cup S \mid S \downarrow \tau \mid S - \tau \mid S \downarrow_n \tau \mid S -_n \tau$

Semantics:

$\mathcal{D}[\![\cdot]\!] : S \to \mathcal{P}(prop \times V)$

$\mathcal{D}[\![\emptyset]\!]$	$=$	$\emptyset$
$\mathcal{D}[\![\{\!\mid\langle \tau, v\rangle \mid\!\}]\!]$	$=$	$\{\!\mid\langle \tau, v\rangle \mid\!\}$
$\mathcal{D}[\![S_1 \cup S_2]\!]$	$=$	$\mathcal{D}[\![S_1]\!] \cup \mathcal{D}[\![S_2]\!]$
$\mathcal{D}[\![S \downarrow \tau]\!]$	$=$	$\{\!\mid\langle \tau', v\rangle \in \mathcal{D}[\![S]\!] \mid \tau \vdash \tau' \mid\!\}$
$\mathcal{D}[\![S - \tau]\!]$	$=$	$\mathcal{D}[\![S]\!] - \mathcal{D}[\![S \downarrow \tau]\!]$
$\mathcal{D}[\![S \downarrow_n \tau]\!]$	$=$	s iff $s \subseteq \mathcal{D}[\![S \downarrow \tau]\!]$ and $\lvert s\rvert = n$
$\mathcal{D}[\![S -_n \tau]\!]$	$=$	s iff $s \subseteq \mathcal{D}[\![S]\!]$ and $\lvert\mathcal{D}[\![S]\!] - s\rvert = n$

Fig. 1. Syntax and denotational semantics of Tagged Sets

as they would allow selections to violate a useful notion of confidentiality that we introduce in the next section.)

The semantics is given as a *semantic function* $\mathcal{D}[\![\cdot]\!]$ which maps the syntactic notion of tagged set S to a mathematical multi-set containing pairs of propositions and values. As described above, $S \downarrow \tau$ denotes the set whose elements are contained in S, but whose tags are implied by the selection tag τ, following the rules of propositional logic. For example, in (1) above we have

$$S \downarrow \mathsf{Blues} = \{\!\mid\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle, \langle \mathsf{Blues}, clip_3\rangle \mid\!\}$$
since $\mathsf{Blues} \vdash \mathsf{Jazz} \vee \mathsf{Blues}$,
$\mathsf{Blues} \nvdash \mathsf{Classical}$ and
$\mathsf{Blues} \vdash \mathsf{Blues}$

The inference rules for deriving judgments $\tau \vdash \tau'$ are standard; they are presented with one extension in Figure 4 in the next section.

The syntax $S_1 \cup S_2$ denotes the tagged set that results from the combination of tagged sets S_1 and S_2 (using multi-set union). $S - \tau$ denotes those tagged values not implied by the selection tag. Finally, one can limit the results of a selection or subtraction to n elements using the $\downarrow_n$ and $-_n$ operators, respectively. The actual contents of the defined set are non-deterministically chosen.

We can illustrate $\cup$ and $-$ with some additional examples. To define a refinement of S that covers genres Jazz *or* Classical could be done with two selections, and taking the union of the results:

$$(S \downarrow \mathsf{Jazz}) \cup (S \downarrow \mathsf{Classical}) = \{\!\mid\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle, \langle \mathsf{Classical}, clip_2\rangle \mid\!\}$$

To select those documents that cover both genres Jazz *and* Blues, we can do one of two things:

$$(S \downarrow \mathsf{Jazz}) \downarrow \mathsf{Blues} = \{\!\mid\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle \mid\!\}$$
$$S \downarrow \mathsf{Jazz} \vee \mathsf{Blues} = \{\!\mid\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle \mid\!\}$$

To select those documents that cover genre Blues but *not* topic Jazz, we perform a selection followed by a subtraction:

$$(S \downarrow \mathsf{Blues}) - \mathsf{Jazz} = \{\!|\langle \mathsf{Blues}, clip_3 \rangle|\!\}$$

2.2 Playlists as Ordered Tuples

So far, we have not considered tags defined with $\wedge$. These are interesting because they effectively *restrict* selections: if a value has tag $\mathsf{t}_1 \wedge \mathsf{t}_2$, then it cannot be selected with either t_1 or t_2 alone: $\mathsf{t}_1 \not\vdash \mathsf{t}_1 \wedge \mathsf{t}_2$ and $\mathsf{t}_2 \not\vdash \mathsf{t}_1 \wedge \mathsf{t}_2$.

We can use $\wedge$ tags to extend our audio repository with *playlists*. A playlist is essentially a tuple whose first element designates the first clip to be played, whose second element designates the second clip, etc. Clips can belong to more than one playlist. With tagged sets, we can designate clips $clip_1$, $clip_2$, and $clip_3$ as tracks one, two, and three, respectively, of playlist Favorites by defining repository S_p as follows:

$$S_p = \{\!|\langle \mathsf{Favorites} \wedge 1, clip_1 \rangle, \langle \mathsf{Favorites} \wedge 2, clip_2 \rangle, \langle \mathsf{Favorites} \wedge 3, clip_3 \rangle|\!\}$$

To play the first track of Favorites, we select it with Favorites $\wedge$ 1; to play the second we select with Favorites $\wedge$ 2, and so on:

$$\begin{array}{l} S_p \downarrow \mathsf{Favorites} \wedge 1 = \{\!|\langle \mathsf{Favorites} \wedge 1, clip_1 \rangle|\!\} \\ S_p \downarrow \mathsf{Favorites} \wedge 2 = \{\!|\langle \mathsf{Favorites} \wedge 2, clip_2 \rangle|\!\} \end{array} \tag{2}$$

To permit selecting *all* songs in a playlist, we can store the clips using a special tag Any:

$$\begin{array}{rl} S'_p = \{\!| & \langle \mathsf{Favorites} \wedge (1 \vee \mathsf{Any}), clip_1 \rangle, \langle \mathsf{Favorites} \wedge (2 \vee \mathsf{Any}), clip_2 \rangle, \\ & \langle \mathsf{Favorites} \wedge (3 \vee \mathsf{Any}), clip_3 \rangle|\!\} \end{array}$$

To select all of the songs in playlist Favorites, we simply do $S'_p \downarrow \mathsf{Favorites} \wedge \mathsf{Any}$. Of course, we can continue to organize songs by genre as well as by playlist:

$$\begin{array}{rl} S'_p = \{\!| & \langle (\mathsf{Favorites} \wedge (1 \vee \mathsf{Any})) \vee (\mathsf{Jazz} \vee \mathsf{Blues}), clip_1 \rangle, \\ & \langle (\mathsf{Favorites} \wedge (2 \vee \mathsf{Any})) \vee \mathsf{Classical}, clip_2 \rangle, \\ & \langle (\mathsf{Favorites} \wedge (3 \vee \mathsf{Any})) \vee \mathsf{Blues}, clip_3 \rangle|\!\} \end{array}$$

2.3 Tuple Sets with Linda-Style Matching

We can formalize this basic encoding of tuples to include matching as in Linda-style tuple spaces [4]. Consider the syntax of a simple language of *tuple sets* shown in Figure 2. The basic operations on tuple sets T are similar to those on tagged sets, but rather than performing selections based on a tag, the user provides a *pattern* p. This pattern consists of either a value v or a wildcard ? which matches any value. Subtraction with $T - p$ is as with tagged sets: it defines the set T' with all elements in T removed that match p.

Syntax:

tuple	$u ::= (v) \mid (v,v) \mid (v,v,v) \mid \ldots$
pattern var	$a ::= v \mid \ ?$
pattern	$p ::= (a) \mid (a,a) \mid (a,a,a) \mid \ldots$
tuple set	$T ::= \emptyset \mid \{\!\{u\}\!\} \mid T \cup T \mid T \downarrow p \mid T - p \mid T \downarrow_n p \mid T -_n p$

Semantics:

$$
\begin{array}{lcl}
\mathcal{L}[\![\emptyset]\!] & = & \emptyset \\
\mathcal{L}[\![\{\!\{(v_1,\ldots,v_n)\}\!\}]\!] & = & \{\!\{\langle \underline{n} \wedge \bigvee_{1\le i\le n}(i \wedge v_i), (v_1,...,v_n)\rangle\}\!\} \\
\mathcal{L}[\![T_1 \cup T_2]\!] & = & \mathcal{L}[\![T_1]\!] \cup \mathcal{L}[\![T_2]\!] \\
\mathcal{L}[\![T \downarrow p]\!] & = & \mathcal{L}[\![T]\!] \downarrow \mathcal{T}[\![p]\!] \\
\mathcal{L}[\![T - p]\!] & = & \mathcal{L}[\![T]\!] - \mathcal{T}[\![p]\!] \\
\mathcal{L}[\![T \downarrow_n p]\!] & = & \mathcal{L}[\![T]\!] \downarrow_n \mathcal{T}[\![p]\!] \\
\mathcal{L}[\![T -_n p]\!] & = & \mathcal{L}[\![T]\!] -_n \mathcal{T}[\![p]\!] \\
\\
\mathcal{T}[\![(a_1,\ldots,a_n)]\!] & = & \underline{n} \wedge \bigvee_{1\le i\le n}(i \wedge a_i) \text{ where } a_i \neq \ ?
\end{array}
$$

Fig. 2. Syntax of Linda-style tuples, semantics via Tagged Sets

As an example, say we have the following tuple set which mentions the birthdays of Alice and Bob:

$$T = \{\!\{(\text{“}birthday\text{”}, \text{“}alice\text{”}, 10, 29, 1991), (\text{“}birthday\text{”}, \text{“}bob\text{”}, 10, 4, 1993)\}\!\}$$

If we wanted to select all birthday records, we could do:

$$T \downarrow (\text{“}birthday\text{”}, ?, ?, ?, ?) = \{\!\{ \begin{array}{l}(\text{“}birthday\text{”}, \text{“}alice\text{”}, 10, 29, 1991), \\ (\text{“}birthday\text{”}, \text{“}bob\text{”}, 10, 4, 1993)\}\!\}\end{array} \quad (3)$$

If we wanted only Alice's birthday, we could do

$$T \downarrow (\text{“}birthday\text{”}, \text{“}alice\text{”}, ?, ?, ?) = \{\!\{(\text{“}birthday\text{”}, \text{“}alice\text{”}, 10, 29, 1991)\}\!\} \quad (4)$$

Conversely, we could define the set with an arbitrary birthday element removed:

$$\begin{array}{rcl} T -_1 (\text{“}birthday\text{”}, ?, ?, ?, ?) & = & \{\!\{(\text{“}birthday\text{”}, \text{“}bob\text{”}, 10, 4, 1993)\}\!\} \\ & \text{or} & \{\!\{(\text{“}birthday\text{”}, \text{“}alice\text{”}, 10, 29, 1991)\}\!\} \end{array} \quad (5)$$

The selections in Examples (3) and (4) are similar to what is possible with the `read(-)` operator for Linda-spaces. Example (5) is like the Linda-space `in(-)`, which removes a single tuple from the space (as our language is declarative, we actually define a new tuple set which lacks an element present in the original). Section 3.1 presents communication commands which when combined with these operators can be used to build traditional Linda spaces.

Tuple sets and their operations can be encoded using tagged sets. Shown in Figure 2, the translation function $\mathcal{L}[\![\cdot]\!]$ maps tuple sets T to tagged sets S, employing auxiliary function $\mathcal{T}[\![\cdot]\!]$ to map patterns p to tags τ. In the tagged set, the tuple is stored as the value part of the tagged value (i.e., it is in V),

and the tag encodes its structure. The first part of the tag is the tuple length $\underline{n}$; to select a tuple of length $\underline{n}$ one must provide this length as a tag. The second part is a union of tags, one for each element in the tuple. As with playlists, these tags encode the position of the element i as tag i. In addition, we include the element v itself as tag, so that we can match literal values present in patterns. The resulting element tag is thus $i \wedge v$. Selection patterns are encoded similarly, except that when a ? appears in a pattern, it does not appear in the tag. This way, it has no bearing on the selection (thus encoding its meaning as a "wild card"). Note that we ignore the possible collision between the tags of indices (i), the tag for indicating size ($\underline{n}$), and integer values used as tags. Addressing this problem would be straightforward.

Here are some examples that illustrate the translation:

Tuple sets	Tagged sets
$\mathcal{L}[\![\{\!\|(v)\|\!\}]\!]$	$= \{\!\|\langle \underline{1} \wedge (1 \wedge v), v\rangle\|\!\}$
$\mathcal{L}[\![\{\!\|(v_1, v_2)\|\!\}]\!]$	$= \{\!\|\langle \underline{2} \wedge ((1 \wedge v_1) \vee (2 \wedge v_2)), (v_1, v_2)\rangle\|\!\}$
$\mathcal{L}[\![T - (v_1, v_2)]\!]$	$= \mathcal{L}[\![T]\!] - \underline{2} \wedge ((1 \wedge v_1) \vee (2 \wedge v_2))$
$\mathcal{L}[\![T \downarrow (?, v_2)]\!]$	$= \mathcal{L}[\![T]\!] \downarrow \underline{2} \wedge (2 \wedge v_2)$

Tagged sets are not rich enough to encode SQL-style or publish/subscribe service queries, mainly because tags can only be used to match set elements; it is not possible to, for example, treat a tag as an integer and then return all tagged values whose tag is "greater than 1." We are compare our approach more closely to these systems in Section 4.

3 Secure Tags

If a tagged set is to be used in a secure, multi-user setting, it should allow users to only reveal their data to others whom they trust. For example, a typical file system labels a file with an access control list, specifying an effective list of users and the operations they can perform on the file (e.g., read, write, delete).

A useful feature of tagged sets is that "user lists," or more properly operation system-style *capabilities*, can be encoded using tags. That is, a tag can be viewed as a key, which means that to select a value, one must produce the key with which it is locked. A value can be locked multiple times (using $\wedge$) requiring the selector produce multiple keys to unlock it. A value may have alternate access points (using $\vee$) which permits unlocking with different key sets. This is similar to Gifford's sealed objects [6].

In the remainder of this section, we show how tags can form the foundation of secure access control of shared data. We present a simple interface for shared tagged sets that ensures confidentiality. Then we show how *asymmetric tags* can be used to support public key-style encryption in tagged sets. Finally, we consider how to provide secure, distributed access to a shared tagged set.

3.1 A Shared Repository

Imagine we wish to define tagged sets that may be shared by processes within an operating system. We extend our presentation so far in two ways. First, we need a way to name a shared tagged set. Second, we must specify a list of commands for manipulating named tagged sets that respects the confidentiality policies of data stored in them; these policies are specified by tags. The syntax and semantics of these changes is shown in Figure 3.

Shared tagged sets STS are tagged sets S extended to include variable names x, whose semantics is simply to look up the tagged set named by x from a global store. We designate this in the semantics $\mathcal{D}[\![x]\!]$ as the function $lookup(x)$; elsewhere we use the function $update(x, s)$ to designate updating the name x in the global store to refer to multi-set s.

Commands cmd can be used by processes to manipulate shared tagged sets: $\text{read}_\tau(STS)$ reads (selects) data from a tagged set STS; $\text{add}_\tau(x, STS)$ adds the set STS to the named tagged set x (using multi-set union $\cup$), and $\text{remove}_\tau(x, STS)$ removes data in set STS from the named tagged set x (using multi-set subtraction). Both add and remove operate by side-effect only, "returning" the empty set $\emptyset$. We assume the implementation of commands is atomic. Linda tuple spaces are essentially the combination of these commands with the tuple encoding presented in Section 2.3.

$$
\begin{array}{lll}
STS & ::= & x \mid \emptyset \mid \{\!|\langle \tau, v \rangle|\!\} \mid STS \cup STS \mid STS \downarrow \tau \mid STS - \tau \\
 & \mid & STS \downarrow_n \tau \mid STS -_n \tau \\
cmd & ::= & \text{read}_\tau(STS) \mid \text{add}_\tau(x, STS) \mid \text{remove}_\tau(x, STS)
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{D}[\![\cdot]\!] : STS \to \mathcal{P}(prop \times V) & & \\
\mathcal{D}[\![x]\!] & = & lookup(x) \\
\mathcal{D}[\![STS]\!] & = & \text{as in Figure 1 otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{C}[\![\cdot]\!] : cmd \to \mathcal{P}(prop \times V) & & \\
\mathcal{C}[\![\text{read}_\tau(STS)]\!] & = & \mathcal{D}[\![STS \downarrow \tau]\!] \\
\mathcal{C}[\![\text{add}_\tau(x, STS)]\!] & = & \text{let } s = \mathcal{D}[\![STS \downarrow \tau]\!] \text{ in} \\
 & & \quad \text{let } \_ = update(x, lookup(x) \cup s) \text{ in } \emptyset \\
\mathcal{C}[\![\text{remove}_\tau(x, STS)]\!] & = & \text{let } s = \mathcal{D}[\![STS \downarrow \tau]\!] \text{ in} \\
 & & \quad \text{let } \_ = update(x, lookup(x) - s) \text{ in } \emptyset
\end{array}
$$

Fig. 3. Commands for manipulating shared Tagged Sets

The security of these commands is based on implication: to read, add, or remove data having tag $\mathtt{t}$ from a shared set, the user must hold a "credential" τ that implies $\mathtt{t}$; the credential τ is presented as a subscript on each operation.[3] Without requiring a credential, a user could use subtraction to extract elements from a shared set for which she did not hold the tags. For example, say that x

[3] While τ can be any tag, it is most useful as a *capability list*, having the form $\mathtt{t}_1 \wedge \ldots \wedge \mathtt{t}_n$.

is bound to $\{\!|\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle, \langle \mathsf{Classical}, clip_2\rangle, \langle \mathsf{Blues}, clip_3\rangle|\!\}$, and the user knows about the tag Classical. The user should thus only be allowed to read the tagged value $\langle \mathsf{Classical}, clip_2\rangle$, since she does not know the names of the tags on the other values. Indeed, $\mathrm{read}_{\mathsf{Classical}}(x - \mathsf{Classical})$ is $\emptyset$, whereas not requiring a credential and just permitting the subtraction would have yielded $\{\!|\langle \mathsf{Jazz} \vee \mathsf{Blues}, clip_1\rangle, \langle \mathsf{Blues}, clip_3\rangle|\!\}$, violating confidentiality. Tag implication has been defined so that only when a tag t appears in credential τ can t be selected. Thus a user must "know about" a tag, and place it in her credential, to be able to unlock values locked with that tag. We formally state and prove this notion of confidentiality in the next subsection.

In this scheme, if the tag t is implied by the credential τ, then any data $\langle \mathsf{t}, v\rangle$ can be added, removed, or read from the tagged set. We could encode richer access rights for better control over these operations. For example, to provide separate read and removal permissions, we could define special tags Read and Remove; a value's tag would be $(\mathsf{Read} \wedge \tau_{read}) \vee (\mathsf{Remove} \wedge \tau_{remove})$. The semantics of $\mathrm{read}_\tau(STS)$ would become $\mathcal{D}[\![STS \downarrow \tau \wedge \mathsf{Read}]\!]$, so as to verify against the τ_{read} part of the tag, and removal would be similar. We would also have to ensure that these special tags neither appear in credentials τ nor clash with normal tags.

3.2 Asymmetric Tags

Tags defined so far essentially correspond to *symmetric keys*: if the user can produce the key, he can acquire the value. We can also easily extend our notion of tag to model *asymmetric keys*, as are provided in public key cryptography.

Asymmetric tags are defined in pairs $(\mathsf{k}, \overline{\mathsf{k}})$. As shown in the top right of Figure 4, asymmetric tags are drawn from the countably-infinite set *Keys* and extend our notion of tags τ. Tagging a value using an asymmetric tag k is equivalent to locking it with an asymmetric key. To select this value requires producing the opposite tag $\overline{\mathsf{k}}$. We do not consider how asymmetric tags are generated; we only assume that if a tagged set contains data locked by some tag k, then it has some way of knowing when the complement tag $\overline{\mathsf{k}}$ is provided during selection. We consider how to do this securely in a distributed setting in the next subsection. By convention, we say $\overline{\mathsf{k}}$ is the private tag and k is the public tag.

The left side of Figure 4 shows the (slightly modified) inference rules for the fragment of propositional logic that we use in our tagging system. As usual, Γ is simply an ordered list of assumptions $\tau_1, \ldots, \tau_n$. The rules employ an additional operator $[\cdot]$ that is the identity on symmetric tags, but the complement for asymmetric ones. The operator is used in the assumptions of the ($\vee$ELIM) and (HYP) rules to enforce that k can only be implied by its complement $\overline{\mathsf{k}}$ and vice versa. This relationship is not transitive: having proven k, there is no way to include it in the assumptions to prove $\overline{\mathsf{k}}$. If there were, it would allow the holder of a public tag k to access data tagged with that tag, rather than only allowing the holder of $\overline{\mathsf{k}}$ to access it.

We can now make our notion of confidentiality precise. We wish to ensure that *cmd* operations do not provide or revoke access to data tagged with keys not contained in the credential τ. This is ensured by the following theorem:

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \wedge \tau_2}\ \wedge\text{INTRO} \qquad \frac{\Gamma \vdash \tau_1 \wedge \tau_2}{\Gamma \vdash \tau_1}\ \wedge\text{ELIM}_1 \qquad \frac{\Gamma \vdash \tau_1 \wedge \tau_2}{\Gamma \vdash \tau_2}\ \wedge\text{ELIM}_2$$

$$\frac{\Gamma \vdash \tau_1 \vee \tau_2 \qquad \Gamma, [\tau_1] \vdash \tau_3 \qquad \Gamma, [\tau_2] \vdash \tau_3}{\Gamma \vdash \tau_3}\ \vee\text{ELIM} \qquad \frac{\Gamma \vdash \tau_1}{\Gamma \vdash \tau_1 \vee \tau_2}\ \vee\text{INTRO}_1 \qquad \frac{\Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \vee \tau_2}\ \vee\text{INTRO}_2$$

$$\frac{}{\Gamma \vdash \top}\ \top\text{ELIM} \qquad \frac{}{\Gamma, [\tau], \Gamma' \vdash \tau}\ \text{HYP}$$

$$\begin{array}{lcl} \mathsf{k}, \overline{\mathsf{k}} & \in & \mathit{Keys} \\ \tau & ::= & \ldots \mid \mathsf{k} \mid \overline{\mathsf{k}} \end{array}$$

$$\begin{array}{lcl} [\mathsf{t}] & = & \mathsf{t} \\ [\mathsf{k}] & = & \overline{\mathsf{k}} \\ [\overline{\mathsf{k}}] & = & \mathsf{k} \\ [\tau_1 \vee \tau_2] & = & [\tau_1] \vee [\tau_2] \\ [\tau_1 \wedge \tau_2] & = & [\tau_1] \wedge [\tau_2] \\ [\top] & = & \top \end{array}$$

Fig. 4. Proof system extended with asymmetric tags

Theorem 1 (Confidentiality). *If* $\tau \vdash \mathsf{t}$*, then* $\mathsf{t} \in [\tau]$*.*

In a logical sense, this theorem simply states that to prove t, we have to know about t in the first place; intuitively it "appears" in our assumption τ. In the simplest case, if τ has the form $\mathsf{t_1} \wedge \ldots \wedge \mathsf{t_n}$, then exactly $\mathsf{t_1}, \ldots, \mathsf{t_n}$ appear in τ; only these tags can be proved by τ. Since all operations on shared tagged sets must ultimately be filtered against the credential τ, we ensure that only data whose tags appear in the credential can be manipulated. This theorem is proven by structural induction on derivations $\tau \vdash \mathsf{t}$, aided by some simple lemmas on the $\in$ relation. A full definition of "appears in" ($\in$) is presented in Figure 5.

$$\begin{array}{llll} \mathsf{t} \in \mathsf{t} & & \mathsf{t} \in \tau_1 \vee \tau_2 & \text{iff } \mathsf{t} \in \tau_1 \text{ and } \mathsf{t} \in \tau_2 \\ \mathsf{k} \in \mathsf{k} & & \mathsf{t} \in \tau_1 \wedge \tau_2 & \text{iff } \mathsf{t} \in \tau_1 \text{ or } \mathsf{t} \in \tau_2 \\ \overline{\mathsf{k}} \in \overline{\mathsf{k}} & \tau_1 \vee \tau_2 \in t & & \text{iff } \tau_1 \in t \text{ or } \tau_2 \in t \\ \top \in t & \tau_1 \wedge \tau_2 \in t & & \text{iff } \tau_1 \in t \text{ and } \tau_2 \in t \end{array}$$

Fig. 5. Definition of $t \in t'$

A Secure Mail System. As a simple example of the use of asymmetric tags, consider a secure e-mail system that supports tagging for categorization, as in GMail. When a mail is received for a particular user, it is tagged with her public key. For example, if Daniel sends a private message to Bridget, it is locked with her public key $\mathsf{Bridget}$:

$$S = \{|..., \langle \mathsf{Bridget}, \text{"}\mathit{You\ appear\ to\ have\ forgotten\ ...}\text{"} \rangle, ...|\}$$

To retrieve her messages, Bridget selects using her private key:

$$S \downarrow \overline{\mathsf{Bridget}} = \{|\langle \mathsf{Bridget}, \text{"}\mathit{You\ appear\ to\ have\ forgotten\ ...}\text{"} \rangle|\}$$

A message addressed to many users is simply tagged with their public keys:

$$S = \{|..., \langle \mathsf{Bridget} \vee \mathsf{Mark}, \text{"}\mathit{Come\ home,\ I\ have\ jumpers\ for\ you\ two.}\text{"} \rangle, ...|\}$$

Either $\overline{\mathsf{Mark}}$ or $\overline{\mathsf{Bridget}}$ can be used to select the message.

Users can use the normal tagging system to organize their mail. For example, Bridget could annotate the message from Daniel as a love letter:

$$S' = \{|..., \langle \mathsf{Bridget} \wedge \mathsf{Loveletter}, \text{"}You\ appear\ to\ have\ forgotten...\text{"}\rangle, ...|\}$$

Multiple categories are of course possible, by annotating with tags of the form (e.g.) $\mathsf{Bridget} \wedge (\mathsf{t}_1 \vee \mathsf{t}_2 \vee ... \vee \mathsf{t}_n)$.

Encoding Encrypted Tuples. With asymmetric tags, we can support a simple extension to our tuple language: tagging each tuple with a public key. We can modify the syntax of tuples u and patterns p in Figure 2 to be as follows:

$$\begin{array}{ll} \text{tuple} & u ::= \mathsf{k} : (v_1, v_2, \ldots, v_n) \\ \text{pattern} & p ::= \overline{\mathsf{k}} : (a_1, a_2, \ldots, a_n) \end{array}$$

Tuples now include the public key, and patterns must now specify the private key to select the desired tuple. We modify our semantic functions to take these changes into account:

$$\begin{array}{l} \mathcal{L}[\![\{|(\mathsf{k} : (v_1, v_2, \ldots, v_n))|\}]\!] = \{|\langle \mathsf{k} \wedge (\bigvee_{1 \le i \le n}(v_i)), (v_1, ..., v_n)\rangle|\} \\ \mathcal{T}[\![\overline{\mathsf{k}} : (a_1, a_2, \ldots, a_n)]\!] = \quad (\overline{\mathsf{k}} \wedge (\bigvee_{1 \le i \le n, a_i \neq ?} a_i)) \end{array}$$

While this is a simple extension of the tuple space encoding, we can encode more complex systems, such as SecOS [15] or CryptoKlava [2].

3.3 Distributed Tagged Sets

To use shared tagged sets as a distributed coordination mechanism, we have to protect the repository and the data it contains from malicious tampering. We have two goals. First, tags/keys and the data they protect should not be transmitted in clear text, to prevent snooping. Second, users should not have to present their private tag/key to the database to retrieve data stored with the public key. Here we briefly sketch how these goals might be achieved.

A client can communicate with a server hosting a shared tagged set using a secure channel, in the style of the Secure Socket Layer (SSL)[4]. The client begins by presenting the server with a credential τ of the form $\mathsf{t_1} \wedge \ldots \wedge \mathsf{t_n}$, but rather than sending the actual tags $\mathsf{t_i}$, the client sends a *public name* for each tag. The mapping between the public name and the key must be known by both the client and server; for example, the tag Red could have the name "Red." This credential is sent along with a random integer n_u, both encrypted with the server's public key (to ensure that commands are not redirected to the wrong server). The server maps the names to tags, and then challenges the user to prove he actually holds tags $\mathsf{t_1} \ldots \mathsf{t_n}$ by sending a message containing integers encrypted with each tag, along with a nonce (to prevent replay attacks). The user decrypts these values,

[4] `http://wp.netscape.com/eng/ssl3/`

and from them derives the value n_s. The user sends the decrypted values back to the server, which verifies they are correct.

At this point, both sides have a shared secret (n_u,n_s), and the server is satisfied that the user holds the keys in the claimed credential. Users send commands encrypted with the shared secret, and the server evaluates the commands using the verified credential τ.

Because we are treating tags as keys, the server should be careful about the tags it returns with values selected by a read command. For example, if a shared tagged set contains the value $\langle \mathsf{Bridget} \vee \mathsf{Mark}, \text{"}You\ appear\ to\ have\ forgotten\ ...\text{"} \rangle$, then Bridget can read this value using her public key $\overline{\mathsf{Bridget}}$. However, if the server includes the tag $\mathsf{Bridget} \vee \mathsf{Mark}$ in the result, then Bridget can see Mark's private key! The simplest solution is to simply strip the tags at the server, returning only a set of values (encypted by the shared secret), or else to use the public names of the tags, rather than the tags themselves.

We can avoid the need for a secure channel for read commands by having the server encrypt returned values using their tags. For each tagged value $\langle \tau, v \rangle$ in the returned set, we return the value as the tuple $(clear(\tau), E[\langle \tau, v \rangle])$, where $clear(\tau)$ is a translation of τ with the keys replaced by their names, and where $E[.]$ performs encryption as defined below (similar to Gifford's schema [6]):

$$\begin{array}{ll} E[\langle \top, v \rangle] & \rightarrow v \\ E[\langle \mathsf{t}, v \rangle] & \rightarrow \{v\}_{\mathsf{t}} \\ E[\langle \mathsf{k}, v \rangle] & \rightarrow \{v\}_{\mathsf{k}} \\ E[\langle \tau_1 \wedge \tau_2, v \rangle] & \rightarrow (E[\langle \tau_1, E[\langle \tau_2, v \rangle] \rangle]) \\ E[\langle \tau_1 \vee \tau_2, v \rangle] & \rightarrow (E[\langle \tau_1, v \rangle], E[\langle \tau_2, v \rangle]) \end{array}$$

The notation $\{v\}_{\mathsf{t}}$ denotes encrypting v with key $\mathtt{t}$. The user then maps backwards from the provided tag names to the actual tags to decrypt the returned values.

4 Discussion

Tagged sets aim to organize and share data simply and securely. They bear resemblance to coordination spaces [4], relational databases [13, 7], semi-structured documents [3], and publish/subscribe services [5, 14, 1], but differ chiefly in how data is organized, selected, and secured.

Data Organization and Selection. Coordination spaces (e.g., Linda-spaces) organize data as tuples, which are essentially a fixed-length, ordered list of simple values. Relational databases and publish/subscribe (hereafter *pub/sub*) systems organize data as (unordered) labeled records; an example might be (*what*=`alarm`) $\vee$ (*level*=`5`). For databases, records are stored in tables, while for pub/sub they are periodically published as *events*. Compared to these, tagged sets are relatively *semi-structured*: while events and records typically adhere to a schema, we place no restrictions on the form of tags.

Table 1. Comparison of data selection and locking mechanisms

System	k	Data d ::=	Queries q ::=
Linda	v	$k \mid d \vee d$	$k \mid q \vee q$
Siena/Gryphon	t *op* v	$k \mid d \vee d$	$k \mid q \vee q$
Elvin	t *op* v	$k \mid d \vee d$	$k \mid q \vee q \mid q \wedge q \mid \neg q$
Tagged Sets	t, k	$k \mid d \vee d \mid d \wedge d$	$k \mid q \vee q \mid q \wedge q$
Sealed Objects	k	$k \mid d \vee d \mid d \wedge d$	$k \mid q \wedge q$

As with tagged sets, these systems permit certain records/events/tuples to be selected based on user queries. Linda-based queries were described in subsection 2.3. For pub/sub systems, queries are in the form of *subscriptions*, defined as patterns meant to match subsequently published events. Subscriptions in Siena [5] and Gryphon [1] mimic the structure of events where = can be replaced by a more general operator. For example, the subscription (*what*=`alarm`) $\vee$ (*level* < 7) matches the event presented earlier. Elvin [14] extends these sorts of queries with other logical operators, approximating SQL-style queries used in databases. Compared with tagged sets, coordination spaces are strictly less powerful. Pub/sub systems have both more and less powerful selections: they are more powerful in that they can select based on *data*, whereas tagged sets only select on tags, but Siena and Gryphon are less powerful in that subscriptions can only be specified as a list of attributes, like a record, whereas tagged sets permit more general logical formulae.[5] Elvin and typical databases allow subscriptions/queries to include the $\neg$ (not) operator, which we have avoided for security reasons, but can model using subtraction.

These results are summarized in Table 1 (along with a row for Sealed Objects, described below). Selectable data d and queries over that data q are defined as a combination of elements k. The systems are ordered by expressiveness: Linda-spaces are the most simple, and Elvin (also representing databases) is the most expressive. We have made notation uniform so that in all cases selection is defined by logical entailment: query q selects data d iff $q \vdash d$ (adjusted to take data and relational operators into account for pub/sub systems).[6]

Generally speaking, the more expressive the system, the more expensive the selection algorithm. For example, given a record and a subscription each with m attributes, selection in Siena takes time $O(m)$ (assuming set membership can be implemented in $O(1)$ time); this approach can be further optimized [1, 14]. By contrast, selection in Linda-spaces is cheaper, and selection in Elvin is more expensive. The performance of tagged sets in part depends on how data is encoded. For example, a simple approach (implemented in our current Java

[5] To select on data in a tagged set, we could duplicate the data as part of the tag.

[6] This may be unintuitive at first: a query that says "select those records having both the t1 and t2 tags" would be expressed as $\mathsf{t1} \vee \mathsf{t2}$ since $\mathsf{t1} \vee \mathsf{t2} \vdash \mathsf{t1} \vee \mathsf{t2}$ but $\mathsf{t1} \vee \mathsf{t2} \nvdash \mathsf{t1}$; conversely, if we expressed the query as $\mathsf{t1} \wedge \mathsf{t2}$, then we would have the undesirable result $\mathsf{t1} \wedge \mathsf{t2} \vdash \mathsf{t1}$. See Section 2.1.

prototype) is to store each tagged value in a collection, with each tag τ stored in Conjunctive Normal Form (CNF):[7]

$$\tau_{CNF} ::= c_1 \wedge c_2 \wedge \ldots \wedge c_n \qquad c ::= l_1 \vee l_2 \vee \ldots \vee l_m \qquad l ::= \mathsf{t} \mid \mathsf{k}$$

Given selection tag q and a data tag d, both having the form τ_{CNF}, the implication $q \vdash d$ holds whenever each literal l_i in each of the n conjuncts of d appears in all of the n' conjuncts of q. This can be decided in time $O(n'nm)$, where m is the maximum number of literals in each conjunct of d (again assume $O(1)$ set memberships). Pub/sub events and Siena-style subscriptions are degenerate cases in which n and n' are 1, yielding identical performance. A DNF-oriented encoding would be more expensive in this degenerate case (essentially $O(m^2)$), but would perform better on a complementary set of queries (ones that use $\wedge$ more heavily). A hybrid or adaptive encoding would be useful.

Security. Our notion of tags as keys for symmetric and asymmetric cryptography is similar to Gifford's Sealed Objects [6]. Sealed Objects are values encrypted/locked using a set of keys, which can be either combined with *KeyAnd* and *KeyOr*, analogous to our $\wedge$ and $\vee$. Like us, he supports asymmetric keys (not shown in the Table). Unsealing an encrypted value requires presenting the necessary held keys, specified as a conjunction.

In the original Linda model, security is implied by knowledge of the tuple structure: only a correctly-specified pattern (in particular knowing the arity of the tuple) can select the data [4]. A number of researchers have aimed to provide stronger security guarantees, e.g., to prevent untrusted mobile applications from illegally removing data from a Linda-space [10, 9, 17, 11]. These systems use a variety of access control policies (applying to the entire repository or the actions that might be performed) and verification strategies.

Several Linda-based approaches have similarly proposed to provide both security and easy selection. SecOS [15] provides the ability to match encrypted tuples. SecOS values can only be encrypted by one key, whereas our use of logical formulas provides many possible combinations of keys per value, without inhibiting selection. CryptoKlava [2] encrypts tuple elements. Selection requires an agent to select the tuple to decrypt it, and relies on the "good behavior" of the agents to put back the tuples that they cannot decrypt. The key itself does not play any role in the initial selection. It is similar to the example of encrypted tuples we showed in section 3. Finally, `SecSpaces` [8] extends the operations allowed on Linda spaces to include partitioning the tuple space based on keys (possibly asymmetric) attached to tuples. These keys are treated as with our $\vee$. After the partitioning, no entry is still encrypted when read, similarly to proposal for distributed selection via secure channels.

Databases often provide fine-grained access control, to the granularity of individual table elements (e.g. Oracle 10g [13]). Tagged values need not adhere

[7] As tags are simply propositional formulae, an arbitrary tag can always be converted to CNF before it is stored in a tagged set.

to a schema, so users can specify policies per object, rather than per relation. Treating tags as keys also naturally supports storing or communicating data in encrypted form [7].

In general, security in pub/sub systems is challenging and largely unexplored. Our approach tackles one aspect of the problem: *publication security* [16]. In particular, interpreting tags as cryptographic keys permits fine-grained control over what parts of an event should be encrypted, without requiring point-to-point communication. However, it reduces some "content queries" (e.g., is this event's *alarm* < 7) to "existence queries" (e.g., does this event have an *alarm* attribute), placing more work on subscribers to perform filtering. Opyrchal and Prakash [12] describe a means to implement dynamic access control lists within a pub/sub system. Our approach embeds policy with the data, treating tags as capabilities. Compared to access control lists, capabilities support more decentralized policies, but make revocation more challenging.

5 Conclusions

This article presented tagged sets: a data management approach that relies on tags based on propositional logic to lock and select values. The model is flexible, intuitive, and supports fine-grained access control for individual data, as we have shown with many examples. We believe it is a promising approach to organizing and securing data, and for supporting distributed coordination.

For future work, we plan to explore data encodings and selection strategies, drawing on results from pub/sub systems [1] and database systems. We are interested further clarifying the relationship between tagged sets and databases and pub/sub systems, with the hope of finding a useful and efficient encoding in both directions, to support various applications. We are interested in understanding how tagged sets with cryptographic operations could be implemented by peer-to-peer networks.

Acknowledgements. Oriol is funded by the Swiss National Science Foundation under grant PBGE2-104794, and Hicks under National Science Foundation grant #0346989. The authors thank Jeff Foster and the anonymous referees for helpful comments on drafts of this paper.

References

1. M. K. Aguilera, R. E. Strom, S. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proc. of ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–62, May 1999.
2. L. Bettini and R. D. Nicola. A Java middleware for guaranteeing privacy of distributed tuple spaces. In *Proc. International Workshop on Scientific Engineering for Distributed Java Applications*, pages 175–184, 2003.
3. P. Buneman, A. Deutsch, and W. C. Tan. A deterministic model for semi-structured data. In *Proc. of the 1999 Intl. Workshop on Query Processing for Semi-Structured Data and Non-Standard Data Formats*, Jan. 1999.

4. N. Carriero and D. Gelernter. Applications experience with Linda. *ACM SIGPLAN Notices*, 23(9):173–187, Sept. 1988.
5. A. Carzaniga, D. S. Rosenblum, and W. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. of ACM Symposium on Principles of Distributed Computing (PODC)*, pages 219–227, July 2000.
6. D. K. Gifford and A. K. Jones. Cryptographic sealing for information security and authentication. *Communications of the ACM*, 25(4):274–286, Apr. 1982.
7. J. He and M. Wang. Cryptography and relational database management systems. In *International Database Engineering and Application Symposium*, pages 273–284, 2001.
8. R. Lucchi and G. Zavattaro. WSSecSpaces: a secure data-driven coordination service for web services applications. In *Proc. of ACM symposium on Applied Computing (SAC)*, pages 487–491, 2004.
9. N. H. Minsky, Y. M. Minksy, and U. Ungureanu. Making tuple space safe for heterogeneous distributed systems. In *Proceedings of SAC '2000*, pages 218–226, 2000.
10. R. D. Nicola, G. Ferrari, and P. Pugliese. Programming access control: The KLAIM Experience. *Lecture Notes in Computer Science*, 1877:48–??, 2000.
11. A. Omicini and F. Zambonelli. Tuple centres for the coordination of Internet agents. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99)*, pages 183–190, San Antonio (TX), Feb. 28 - Mar. 2 1999. ACM Press. Track on Coordination Models, Languages and Applications.
12. L. Opyrchal and A. Prakash. Secure distribution of events in content-based publish subscribe systems. In *Proc. of USENIX Security Symposium*, 2001.
13. Oracle. Oracle database 10g security and identity management. Technical report, Dec. 2003.
14. B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, Brisbane, Australia, September 1997.
15. J. Vitek, C. Bryce, and M. Oriol. Coordinating processes with secure spaces. *Science of Computer Programming*, 46:163–193, January-February 2003.
16. C. Wang, A. Carzaniga, D. Evans, and A. Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, page 303. IEEE Computer Society, 2002.
17. A. Wood. Coordination with Attributes. In P. Ciancarini and A. Wolf, editors, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, volume 1594 of *Lecture Notes in Computer Science*, pages 21–36, Amsterdam, Netherland, Apr. 1999. Springer-Verlag, Berlin.

Time-Aware Coordination in ReSpecT

Andrea Omicini, Alessandro Ricci, and Mirko Viroli

DEIS, Alma Mater Studiorum, Università di Bologna,
via Venezia 52, 47023 Cesena, Italy
{andrea.omicini, a.ricci}@unibo.it, mviroli@deis.unibo.it

Abstract. Tuple centres allow for dynamic programming of the coordination media: coordination laws are expressed and enforced as the behaviour specification of tuple centres, and can change over time. Since time is essential in a large number of coordination problems and patterns (involving timeouts, obligations, commitments), coordination laws should be expressive enough to capture and govern time-related issues.

Along this line, in this paper we discuss how tuple centres and the ReSpecT language for programming logic tuple centres can be extended to catch with time, and to support the definition and enforcement of time-aware coordination policies. Some examples are provided to demonstrate the expressiveness of the ReSpecT language to model timed coordination primitives and laws.

1 Introduction

Coordination artifacts are general-purpose run-time abstractions embedded in a MAS (multi-agent system) coordination infrastructure [1, 2], and meant to provide agents with *coordination as a service* [3]. In particular, coordination artifacts aim at automating specific coordination tasks, encapsulated outside the agents, and featuring relevant engineering properties such as predictability, inspectability and malleability of behaviour [1].

The reference coordination model supporting the notion of coordination artifact is TuCSoN [4, 5]. TuCSoN *tuple centres* populate network nodes and play the role of coordination artifacts. Tuple centres are LINDA-like tuple spaces [6], whose reactive behaviour can be programmed using the logic-based, Turing-complete language ReSpecT [7]. By this language, tuple centres can encapsulate any coordination task, from simple synchronisation policies up to complex workflows [8, 9].

However, in most application scenarios characterised by a high degree of openness and dynamism, coordination tasks need to be time dependent. On the one hand, handling time is necessary to specify (and enforce) given levels of liveness and quality of service: for instance, an agent could be required to interact with a coordination artifact at a given minimum / maximum frequency. On the other hand, temporal properties are also fundamental in the agent-artifact contract: for instance, an agent could commit to accomplish a task before a given timeout expires, or could require the artifact to provide a response within a given time.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 268–282, 2005.

The expressive need for timed coordination policies already emerged in the field of distributed systems as well. For instance, in JavaSpaces [10] primitives `read` and `take` — looking for tuples in the same way as `rd` and `in` in LINDA — come with a timeout value: when the timeout expires without a matching tuple is found, a failure result is returned. Similarly, tuples can be equipped with a *lease* time when they are inserted in the space: as soon as the lease expires, the tuple is automatically removed. All these primitives, and others based on time, can actually be used as the basis for structuring more complex coordination scenarios, such as e.g. auctions and negotiations protocols including time-based guarantees and constraints. However, they are typically too specific solutions to capture all the time-related coordination problems, which instead require a general and comprehensive model for time in coordination.

Along this line, in this paper we discuss how the basic ReSpecT tuple-centre model has been extended to support the definition and enaction of time-aware coordination policies. Section 2 discusses the general concept of time-aware coordination artifacts, and describes how tuple centres and ReSpecT can be extended accordingly to deal with time. Section 3 exemplifies the approach by showing how the dining-philosopher problem can be modelled in ReSpecT, and then extended with time constraints. Section 4 briefly discusses the expressiveness of the extended ReSpecT tuple-centre model, by showing how it can be used to express some well-known temporal features as found in a number of well-known coordination models (such as tuple leasing and timed requests). Section 5 gives some clues of the model implementation. Finally, Section 6 considers related works, and Section 7 provides for conclusions.

2 Timed Coordination Artifacts

In order to represent and enforce timed coordination laws within a coordination artifacts, some conceptual and practical pre-conditions have to be satisfied, and some issues need to be properly addressed.

First of all, time has to be an integral part of the ontology of a coordination artifact. Generally speaking, time can be local or global (if it refers to the global system time, or to the local artifact time), relative or absolute (if it assumes as zero the starting time of the artifact, or uses a standard time convention like date-time), continuous or discrete. Typically, local and relative time are the most natural reference for coordination artifacts in distributed systems, since it can be always be defined and used with no conceptual difficulties. Global and absolute time can be defined conventionally / pragmatically from there — for instance, global system time is the time of a specific artifact (along with some practical methods to extract time from there, and define some notion of simultaneity), absolute time is obtained by properly labelling time 0 of the artifact, and then using relative time as a delta. Finally, discrete time is the obvious choice for any computational machine. As a result, a coordination artifact can label any event (either incoming or outgoing) with its own (local, relative and discrete) time, which is then amenable to be used as a unique label within the coordination

artifact, under the simple hypotheses that it works with a single flow of control, and has a fine-enough-grained time scale.

Correspondingly, a coordination artifact should allow coordination laws to talk about time. A range of predicates / functions has then to be provided as the syntactic sugar to access the time information (label) featured by any event to be handled (the time when the action that produced the event has affected the artifact, and the current time of the computation within the artifact), and to perform simple computations over time (comparing time points and / or intervals).

Of course, time has to be embedded into the coordination-artifact working cycle. So, some notion of *time event* has to be introduced, which triggers some time-related computation within the artifact. In fact, it is not enough to allow the time of an event to be accessed: time-related laws like "the answer should come within 3 minutes after the question, otherwise some penalty will be inflicted" cannot be expressed only referring to actions actually performed — since they contemplate the case of no action performed, the corresponding law cannot refer to a reaction to no action. So, time events (as events triggering some behaviour of the coordination artifact, conceptually corresponding to the passage of time) have to be autonomously generated by the coordination artifact in order to suitably handle time-related laws, within the normal working cycle of the artifact. Quite obviously, attention should be paid on the one hand not to overload the cycle, on the other hand not to be too coarse in time intervals.

Along this line, the ability to capture time events and to react appropriately is another obvious capability required to a timed coordination artifact. Time-aware coordination laws can then be enforced, that can specify / constraint behaviours that depend on time, and can suitably relate time with the evolution of the coordination (artifact) state.

Finally, talking about time naturally recalls the dynamics that is typically featured by the coordination laws encapsulated within coordination artifacts. So, a fundamental complement to the ability of the artifact to specify and enact time-related coordination policies is the ability to modify the coordination specification over time, during the "active life" of the artifact, and possibly depending on some time-aware behaviour of the artifact itself. Correspondingly, it should be possible to express how to add a new coordination law, and how to remove an old one, so as to adapt the artifact behaviour (and the coordination altogether) to the passage of time — or to the change, more generally.

2.1 Timed Tuple Centres

Tuple centres are introduced in [7] as coordination artifacts meant at engineering coordination activities in MASs. Technically, a tuple centre is a *programmable* tuple space, i.e. a tuple space whose reactive behaviour to communication events can be programmed so as to specify and enact any coordination policy [7]. Tuple centres can be thought then as general purpose coordination artifacts, which can be suitably forged in order to provide specific coordination services. The tuple centre model is not bound to any specific model / language for behaviour spec-

ification or to a specific communication language: these aspects are defined by specific instances of the model. An example, discussed in next subsection, is given by ReSpecT tuple centres [11], which adopt logic tuples as the communication language, and the ReSpecT language for tuple centre behaviour specification. Independently of the specific language adopted, the tuple centre behaviour is meant to be specified in terms of reactions to (communication) events occurring in the artifact. So, the core idea behind tuple centres (and coordination artifacts, more generally) is to have first-class coordination abstractions which are powerful enough to encapsulate and enforce at execution time the coordination laws required to support MAS activities. This does not happen, for instance, in basic LINDA-like models, where complex coordination activities surpassing the limited expressive power of tuple space coordination force the global logic of coordination to be spread among individual agents [7]. As coordination artifacts, tuple centres have a usage interface, composed by the basic LINDA primitives, plus two primitives — `set_spec` and `get_spec` — for setting and reading the tuple centre behaviour specification. As coordination artifacts, tuple centres also feature inspectability and malleability properties, i.e. their coordinating behaviour can be inspected and changed dynamically, at execution time.

Timed tuple centres extend tuple centres with the temporal framework depicted above for timed coordination artifacts. First, the notion of *current time* for a tuple centre is introduced as a local, relative and discrete time. Conceptually, tuple-centre time is generated by an inner clock owned by the tuple centre: no relationships can be established in principle between the current time of two different tuple centres. The current time of a timed tuple centre is zero when the tuple centre is actually created by the infrastructure at run time. Absolute time is available, conventionally computed by suitably adding the (absolute) tuple-centre creation time (as provided by the infrastructure) to the current time.

With respect to the formal model defined in [7], a *time transition* is introduced in the basic tuple centre working cycle, in addition to the existing *listening*, *speaking*, and *reacting* transitions. The time transition is meant to have priority with respect to all the other transitions, including the reacting one. Conceptually, the time transition is executed at each tick of the tuple centre clock — as reacting to the generation of a *time event* for each tick.[1]

Then, similarly to communication events, it is possible to specify reactions triggered by time events (*timed reactions*). Timed reactions follow the same semantics of other reactions: once triggered, they are placed in the triggered-reaction set and then executed, atomically, in a non-deterministic order. Since at a given time, only one time event can occur, each timed reaction is executed only once.

As a result, a timed tuple centre can be programmed to react to the passing of time, so as to enforce time-aware coordination policies.

[1] In practice, then, the time transition needs to be executed only when the tuple centre specification actually contains triggerable timed reactions, according to a simple mechanism sketched in Section 5.

2.2 Timed ReSpecT

ReSpecT tuple centres are tuple centres based on first-order logic, adopted both for the communication language (logic tuples), and for the behaviour specification language (ReSpecT) [11]. Basically, reactions in ReSpecT are defined as Prolog-like relations of the form

```
reaction(Head, Body).
```

which specify the list of the operations to be executed (the *Body* of the reaction) when a certain communication event occurs (represented by the reaction *Head*). Such operations make it possible to inspect and change current communication and coordination state, for instance by inserting / reading / removing tuples from the tuple set (see [11] for details). Operations can trigger new reactions. If just only one of the operations invoked in the body of the reaction fails, the entire reaction fails atomically, rolling back any change possibly done by previous operations successfully executed by the same reaction.

According to the timed-tuple-centre model described in Subsection 2.1, the ReSpecT language is extended with time *(i)* by introducing some temporal predicates to get information about both tuple-centre and event time, and *(ii)* by making it possible to specify reactions on the occurrence of time events. The temporal predicates introduced are the following:[2]

- `current_time(?Time)` This predicate succeeds if *Time* (typically a variable) unifies with the current tuple-centre time. As an example, the reaction specification tuple

  ```
  reaction(in(p(X)),(current_time(Time),out_r(request_log(Time,p(X))))).
  ```

  inserts a new tuple (`request_log`) with timing information each time a request to retrieve a tuple `p(X)` is executed, thus implementing the temporal log of a specific sort of request.
- `event_time(?Time)` This predicate succeeds if *Time* unifies with the tuple-centre time when the original communication event triggering the reaction occurred.
- `before(@Time)`, `after(@Time)`, `between(@MinTime,@MaxTime)` These predicates succeeds if the current tuple-centre time is respectively less than, greater than, and between the specified temporal arguments.

Reactions to time events are specified analogously to ordinary reactions:

```
reaction(time(Time), Body).
```

where *Time* is a ground term. The intended semantics is the following: as soon as the tuple-centre time reaches the *Time* value — so, the time event `time(Time)` is conceptually generated — all the reactions whose head matches the event time can be triggered, and their *Body* inserted in the triggered-reaction set. As a simple example, consider the following specification:

[2] A Prolog-like notation is adopted for describing the modality of arguments: + is used for specifying input argument, - output argument, ? input/output argument, @ input argument which must be fully instantiated.

```
reaction(time(TimeAlarm),( out_r(alarm(TimeAlarm)) )).
```

When the tuple-centre current time reaches the `TimeAlarm` value (which must be instantiated to some numeric value), the reaction can be triggered, and its subsequent execution causes the insertion of the tuple `alarm` in the tuple centre.

Given that a timed reaction is conceptually triggered when the tuple-centre current time exactly matches the time specification in its head, each timed reaction is executed at most *once*: correspondingly, any triggered timed-reaction is automatically consumed after its execution, and so removed from the specification. Then, timed-reaction execution follows the same atomic semantics of normal reactions [11].

In ReSpecT tuple centres, it is possible to add and remove time reactions dynamically by exploiting the self-modifying specification predicates defined in ReSpecT: `out_r_spec`, `in_r_spec` and `rd_r_spec` predicates, which are used in to add, remove, and read reactions in general. In particular, the effect of an `out_r_spec(H,B)` is to add a reaction `reaction(H,B)` to the current specification, while `in_r_spec(H,B)` / `rd_r_spec(H,B)` removes / reads a reaction whose head and body match with H and B, respectively. This makes it possible to add and remove also timed-reaction specifications dynamically, by need.

For instance, the following specification

```
(1) reaction(out(clockStart), ( in_r(clockStart),
      current_time(StartTime), out_r(tick(StartTime)) ) ).

(2) reaction(out_r(tick(ClockTime)),(
      in_r(tick(ClockTime)), rd_r(delta_time(DeltaTime)),
      NewClockTime is ClockTime + DeltaTime,
      // any activity to be done at each clock goes here
      out_r_spec( time(NewClockTime), out_r(tick(NewClockTime)) ))).

(3) reaction(out(clockStop), ( in_r(clockStop),
      in_r_spec( time(ClockTime), out_r(tick(ClockTime)) ))).
```

defines a clock that starts when the tuple `clockStart` is first inserted in the timed tuple centre (reaction 1), cycles every `delta_time(@DeltaTime)` milliseconds (reaction 2), and stops when the tuple `clockStop` is finally inserted in the tuple centre (reaction 3).

3 An Example: Dining with Time Constraints

As a main example to show the effectiveness of the approach, we consider here an extension to the well-known *dining philosopher* problem, tackling the time issue. The dining philosopher is a classical problem used for evaluating the expressiveness of coordination languages in the context of concurrent systems [12]. In this problem, a number of philosophers eat at the same round table, using shared chopsticks. Philosophers alternate thinking with eating: two chopsticks are required to eat, no chopstick to think. Each philosopher shares the two chopsticks

on his left and right sides, respectively with the philosophers on his left and right sides. Coordination here is mostly needed to avoid deadlock, which can happen if each philosopher has taken a chopstick and is waiting for the other one, which is in turn taken by another waiting philosopher. In spite of its almost trivial formulation, the dining philosophers problem is generally used as an archetype for non-trivial resource-access policies.

The solution to the problem via ReSpecT consists in using a tuple centre — we call it `table` — for encapsulating the coordination policy required to decouple agent requests from the actual requests of resources — specifically, to encapsulate the management of chopsticks (for details refer to [7]). From the agent viewpoint, each philosopher *(i)* gets the two chopsticks needed by retrieving a tuple `chops(C1,C2)`, *(ii)* eats for a certain amount of time, *(iii)* provides back the chopsticks by inserting the tuple `chops(C1,C2)` in the tuple centre, and *(iv)* finally starts thinking until the next dining cycle. A process-algebraic-like description of this interactive behaviour is the following:

```
PHILO(C1,C2)::=
  THINK.table?in(chops(C1,C2)).EAT.table?out(chops(C1,C2)).PHILO(C1,C2)
```

The main point here is that philosophers do not need to worry about how to coordinate themselves, or how the resources are represented: they simply need to know which specific chopstick pair to ask for, and then they can focus on their main tasks (thinking and eating).

The tuple centre `table` is used as a coordination artifact to help their collective activity. Chopsticks are represented by `chop(`*N*`)` tuples, with *N* between 1 and the number of philosophers. Philosophers directly deal with couples of chopsticks (`chops/2` tuples). The tuple centre is programmed with the ReSpecT specification shown in Table 1 (top). Generally speaking, the coordinating behaviour accounts for mediating the representation of the resources (`chops/2` vs. `chop/1` tuples), and most importantly for avoiding deadlocks among the agents. In particular, if a philosopher requests a couple of available chopsticks, the request is reified by inserting a tuple `required` in the tuple centre (reaction 1). As this tuple is inserted, and if both the chopsticks are available, they are removed and a `chops` tuple is released to the agent (reaction 2). When the agent request is satisfied, the tuple `required` representing the pending agent request is removed (reaction 3). Then, when a philosopher inserts back the tuple representing the couple of chopsticks, the artifact reacts in order to mediate between the different chopsticks representations, by removing the `chops(`*C1*`,`*C2*`)` tuple (reaction 5) and inserting two separated chopsticks `chop(`*C1*`)` and `chop(`*C2*`)` (reaction 4). As a single chopstick is inserted, a control is made to check if such a chopstick is required by a pending agent request (`required` tuple) and — at the same time — if the other chopstick that appears in the pending agent request is available (reactions 6 and 7). In case, both chopsticks are removed, and the pending agent request is satisfied by producing a suitable `chops` tuple.

The basic formulation of the dining philosopher problem focuses on the deadlock issue. However, another relevant aspect for a correct collective behaviour of a MAS is fairness in the use of resources: once acquired the chopsticks, philosophers are meant to release them back, sooner or later. If a philosopher inciden-

Table 1. Timed ReSpecT specification for coordinating dining philosophers: (*top*) without maximum eating time constraints, (*bottom*) adaptation / extension to deal with timing constraints. In particular, reaction 8 is added to the specification, and reaction 4 is replaced by a new version (reaction 4')

```
1  reaction(in(chops(C1,C2)), (
     pre, out_r(required(C1,C2)))).

2  reaction(out_r(required(C1,C2)),(
     in_r(chop(C1)),in_r(chop(C2)),out_r(chops(C1,C2)))).

3  reaction(in(chops(C1,C2)), (
     post, in_r(required(C1,C2)))).

4  reaction(out(chops(C1,C2)), (
     out_r(chop(C1)),out_r(chop(C2)))).

5  reaction(out(chops(C1,C2)), (
     in_r(chops(C1,C2)))).

6  reaction(out_r(chop(C1)), (
    rd_r(required(C1,C)),in_r(chop(C1)),in_r(chop(C)),out_r(chops(C1,C)))).

7  reaction(out_r(chop(C2)), (
   rd_r(required(C,C2)), in_ r(chop(C)),inr(chop(C2)),out_r(chops(C,C2)))).
```

```
4' reaction(out(chops(C1,C2)), (
     in_r(used(C1,C2,T)),
     out_r(chop(C1)),out_r(chop(C2)))).

8  reaction(in(chops(C1,C2)), (
     post, current_time(T), rd_r(max_eating_time(Max)), T1 is T + Max,
     out_r(used(C1,C2,T)),
     out_r_spec(time(T1),(
          in_r(used(C1,C2,T)), out_r(chop(C1)),out_r(chop(C2)))))).
```

tally dies or refuses to release back the chopsticks, some philosophers can die by starvation and the coordination activity is compromised. So, a further issue that the coordination policy should capture is how to impose a constraint over the maximum time which philosophers can take to eat (i.e., to use the resources). Whenever such a constraint is violated, chopsticks released to philosophers are considered no more valid (no more usable), and new valid copies are re-created in the tuple centre, so as to allow the other philosophers to use them. If the

agent autonomy is to be preserved, such a coordinating behaviour should be obtained without forcing any individual agent behaviour: instead, this should be achieved by instrumenting the coordination artifact with suitable time-aware coordination laws.

Such a behaviour can straightforwardly be implemented with the ReSpecT model extended with time. By exhibiting the typical incremental nature of ReSpecT specifications, the previous (non-timed) specification is almost entirely reused, with only one minor change and one extension (reactions 4' and 8, respectively), as described in the bottom part of Table 1: the logic of the previous solution is mostly kept, and only the time-related aspects are specifically addressed by the two new reactions. Precisely, the behaviour is obtained by installing a timed reaction implementing a timeout every time a couple of chopsticks is retrieved by a philosopher (reaction 8), and keeping track of the chopstick currently in use by means of tuple `used`. The timed reaction is triggered after *Max* time units, where *Max* is the maximum eating time, stored in the `max_eating_time` tuple. When a philosopher inserts back the couple of chopsticks on time, the `used` tuple is successfully removed, along with the corresponding timed reaction, and the individual chopsticks are inserted back as in the non-timed case (compare reaction 4' with 4). If an installed timed reaction is triggered, it means that eating time for a philosopher has expired: then, tuple `used` is removed and the corresponding individual chopsticks (`chop` tuples) are re-created. When (if) a philosopher inserts back the couple of chopstick out of time, the related `used` tuple is no longer found, and the individual chopsticks are not inserted back (reaction 4' fails).

It is worth noting that keeping track of the maximum eating time as a tuple (`max_eating_time` in the example) makes it possible to easily change it dynamically, while the activity is running. This can be very useful for instance in scenarios where this time need to be adapted (at run time) according to the workload and, more generally, to environmental changes affecting the system.

4 Other Examples

In this section we describe how the extended model can be used to realise some other well-known coordination patterns based on the notion of time, namely timed requests and tuple leasing.

It is worth noting, however, that our point here is not to show that timed requests, or tuple leasing, can be expressed better by Timed ReSpecT than by the specific timed-primitives provided by JavaSpaces [10] and TSpaces [13]: those, in fact, are not general-purpose approaches, and can then address only a limited range of time-related coordination problems. Instead, the most relevant point here is the *generality* of our approach: here, in fact, the same simple model is shown to be capable to express time-based coordination policies of different kinds.

So, even the simple dining-philosophers problem extended with time constraints discussed in Section 3 can not be solved (at least, not straightfor-

wardly) with the timed primitives of JavaSpaces and TSpaces. Instead, the Timed ReSpecT model discussed above proves to be general enough to easily express timed requests and tuple leasing, as well as the timed dining-philosophers example.

4.1 Timed Requests

In this first example we model a timed `in` primitive, i.e. an `in` request that blocks only for an a-priori limited amount of time. An agent issues a timed `in` by

Table 2. Timed requests modelled using Timed ReSpecT

```
1  reaction(in(timed(MaxTime,Tuple,_)),(
     pre, out_r(required(MaxTime,Tuple)) )).

2  reaction(out_r(required(MaxTime,Tuple)),(
     in_r(Tuple),
     in_r(required(MaxTime,Tuple)),out_r(timed(MaxTime,Tuple,yes))
     ;
     current_time(Time), Timeout is MaxTime + Time,
     out_r_spec(time(Timeout),(
         in_r(required(MaxTime,Tuple)),
         out_r(timed(MaxTime,Tuple,no)))) )).

3  reaction(out(Tuple),(
     in_r(required(MaxTime,Tuple)),out_r(timed(MaxTime,Tuple,yes)))).
```

executing primitive `in(timed(@Time,?Template,-Res))`. If a tuple matching *Template* is inserted within *Time* units of time, the requested tuple is removed and returned to the agent via unification with *Template*, with *Res* bound to the `yes` atom. Conversely, if no matching tuples are inserted within the specified *Time*, the request is unblocked by producing a suitable `timed` tuple with *Template* untouched and *Res* bound to `no`, which is then returned to the agent. Table 2 reports the Timed ReSpecT specification that implements the behaviour of this new primitive. When a timed `in` operation is issued, the request is reified by inserting a `required` tuple in the tuple centre (reaction 1). If a tuple matching the request is found, then the agent request is immediately satisfied by inserting back a `timed` tuple reporting a successful result (first part of reaction 2). Conversely, if no tuples are found, a timed reaction is installed, to be triggered after the amount of time specified by the agent request (second part of reaction 2)[3]. If a tuple matching the request is inserted on time (reaction 3),

[3] The `;` operator in ReSpecT has a meaning similar to the Prolog one: `(G1;G2)` succeeds if either `G1` or `G2` succeeds. More precisely, first `G1` is executed: if it fails, `G2` is then executed.

a `timed` tuple reporting a successful result is generated. Otherwise, if the timed reaction is triggered with the request still pending (tuple `required` is still in the tuple centre), a `timed` tuple reporting a negative result is generated, unblocking the agent request.

4.2 Tuples in Leasing

Finally, in this last example we model the notion of *lease*, analogously to the lease notion in models such as JavaSpaces [10] and TSpaces [13]. Tuples can be inserted in the tuple set specifying a lease time, i.e. the maximum amount of time for which they can reside in the tuple centre before automatic removal. The ReSpecT code implementing a simple form of leasing is:

```
reaction( out(leased(Tuple,LeaseTime)),(
   out_r(Tuple),
   current_time(Time), ExpireTime is Time + LeaseTime,
   out_r_spec( time(ExpireTime), in_r(Tuple) ))).
```

An agent inserts a tuple with a lease time by issuing an

```
out(leased(@Time,@Tuple))
```

According to the reaction described above, when a tuple with a lease time is inserted in the tuple centre, a timed reaction is inserted to be triggered when the leasing time has expired. The timed reaction simply removes the tuple in leasing. If the tuple is not found anymore (because it has been removed by some other agent request), the timed-reaction execution has no effect, for it simply fails.

5 Implementation Overview

The basic ReSpecT virtual machine has been designed and realised as a finite state automaton, with transitions through the basic stages (listening, speaking, reacting) as defined in the operational semantics described in [7, 11]. The tuple set, the pending query set, the triggered-reaction set and input/output event queues defined in [7] constitute the main data structures of the virtual machine. A Prolog engine is used for reaction triggering and execution; in particular ReSpecT primitives are implemented as Prolog built-in predicates defined in a library extending the basic engine. The technology is fully Java-based, and has been developed exploiting tuProlog, a Java-based Prolog engine, which is available as an open-source project at the tuProlog web site [14].

In the time-extended model, some new data structures are added:

- a clock, realised as a long-integer counter, holding current tuple-centre time expressed in milliseconds;
- a timed-reaction specification list, which is the list of timed reactions currently defined in the specification. The list is ordered by the time specified in the heads. The list is updated by the `out_r_spec` and `in_r_spec`, when inserting and removing timed reactions, and by the `set_spec` coordination primitive, when setting a specification which includes also time reactions;

– a timer service, triggering a tuple centre virtual machine in idle state at specific time points.

The Prolog library defining ReSpecT predicates has been extended with new predicates implementing the behaviour of the new temporal primitives (predicates).

The basic virtual-machine working-cycle has been extended so as to implement the time transition: at each cycle (that is, after any listening, speaking and reacting transition), current time is updated and the head timed-reaction specification list is checked: if there are timed-reaction specifications whose reaction time is less or equals than current tuple-centre time, they are removed from the list, and their bodies are added to the triggered-reaction set.

Also, to avoid problems due to idleness — when the timed-reaction list is not empty but no timed reaction have to be triggered yet, and there are no external requests to be served, no pending satisfiable pending requests, no triggered reactions to execute — the tuple-centre virtual machine properly configures the timer service, before going idle. In particular, the timer service is programmed so as to trigger the machine at the time point specified by the reaction time of the first timed reaction of the list.

6 Related Works

Outside the specific context of coordination models and languages, the issue of defining suitable languages for specifying the communication and coordination in timed systems have been extensively studied. Examples of such languages are Esterel [15] and LUSTRE [16], both modelling synchronous systems, the former with an imperative style, and the latter based on dataflow. In the coordination literature several approaches have been proposed for extending basic coordination languages with timing capabilities. [17] introduces two notions of time for LINDA-style coordination models, relative time and absolute time, providing for a number of time-related features. Time-outs have been introduced in JavaSpaces [10] and in TSpaces [13], and have been generally formalised by Timed Linda [18].

The Timed ReSpecT approach described in this work differs from these approaches for at least two main reasons. First of all, Timed tuple centres extend the tuple-centre model without altering the basic LINDA model: LINDA primitives are kept unchanged (no change to their semantics, no timed primitives added), and the extension focuses instead on the expressiveness and behaviour of the coordination medium. Also, Timed ReSpecT does not provide agents with specific time capabilities, but — following the philosophy of programmable coordination media [19] — aims instead at instrumenting the model with the general expressiveness required to capture any time-based coordination pattern.

7 Conclusions

The first attempt to enhance ReSpecT with time is reported in [20]. There, however, the syntax and the semantics of the extension (based on the notion

of trap) are quite *ad hoc*, and do not fit well the original ReSpecT model: the examples reported there can be easily compared with the ones in this article to clearly appreciate the differences between the two approaches. Moreover, the contribution provided by this work is meant to be broader, since it generalises over the notion of tuple centre, and extends to the design and development of general-purpose time-aware coordination artifacts in MASs [1].

Our approach aims to be general and expressive enough to allow for the description of a wide range of coordination patterns based on the notion of time, by exploiting medium programmability and the basic time-based mechanisms. An important feature exhibited by our approach is the *encapsulation of coordination*: embedding (specifying, enacting) time-aware policy directly inside the coordination medium promotes modularity of the coordination programs, and then reusability and extensibility. Temporal features have been added with no changes to the usage interface of tuple centres, which is still based on the basic set of Linda-like coordination primitives. Also, the extension has been realised while preserving all the essential properties of the ReSpecT model: in particular, reaction execution is still atomic (both at the system and at the agent levels [7]), and reactions are executed sequentially. Even more, the declarative nature of the reactions, along with the execution model, makes (Timed-)ReSpecT mostly incremental in its specifications, as shown by the example discussed in Section 3.

In the implementation of the model, the issue of the centralised vs. distributed implementation of tuple centres arises. The basic tuple centre model does not necessarily require a centralised implementation *per se*: however, the extension provided in this work — devising out a notion of time for each individual medium — leads quite inevitably to realise tuple centres with a specific spatial location. This is what already happens in the TuCSoN coordination infrastructure, where there can be multiple tuple centres spread over the network, collected and localised in infrastructure nodes. It is worth mentioning that this problem is not caused by our framework, but is somehow inherent in any approach aiming at adding temporal aspects to a coordination model. However, according to our experience in agent-based distributed system design and development, the need for a distributed implementation of an individual coordination medium is an issue of some relevance only for very specific application domains. For most applications, the bottleneck and single point of failure arguments against the use of centralised coordination media can be answered by a suitable design of the MAS and an effective use of the coordination infrastructure. At this level, it is fundamental that a software engineer would know the scale of the coordination artifacts he/she is going to use, and the quality of service (robustness in particular) ensured by the infrastructure.

Even though the model of time used here is not meant to deal with real-time issues in any way, we understand that this work could provide us with a solid grounding for soft and hard real-time agent coordination. In the future, we mean to explore real-time issues by suitably extending (for instance, with time-labelled triggered reactions) the time model presented here.

References

1. Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., Tummolini, L.: Coordination artifacts: Environment-based coordination for intelligent agents. In Jennings, N.R., Sierra, C., Sonenberg, L., Tambe, M., eds.: 3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004). Volume 1., New York, USA, ACM (2004) 286–293
2. Viroli, M., Ricci, A.: Instructions-based semantics of agent mediated interaction. In Jennings, N.R., Sierra, C., Sonenberg, L., Tambe, M., eds.: 3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004). Volume 1., New York, USA, ACM (2004) 102–110
3. Viroli, M., Omicini, A.: Coordination as a service: Ontological and formal foundation. Electronic Notes in Theoretical Computer Science **68** (2003) 1st International Workshop "Foundations of Coordination Languages and Software Architecture" (FOCLASA 2002), Brno, Czech Republic, 24 August 2002. Proceedings.
4. Omicini, A., Zambonelli, F.: Coordination for Internet application development. Autonomous Agents and Multi-Agent Systems **2** (1999) 251–269 Special Issue: Coordination Mechanisms for Web Agents.
5. TuCSoN: Home page. `http://lia.deis.unibo.it/research/TuCSoN/` (2001)
6. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems (TOPLAS) **7** (1985) 80–112
7. Omicini, A., Denti, E.: From tuple spaces to tuple centres. Science of Computer Programming **41** (2001) 277–294
8. Denti, E., Natali, A., Omicini, A.: On the expressive power of a language for programming coordination media. In: 1998 ACM Symposium on Applied Computing (SAC'98), Atlanta, GA, USA, ACM (1998) 169–177 Special Track on Coordination Models, Languages and Applications.
9. Ricci, A., Omicini, A., Viroli, M.: Extending ReSpecT for multiple coordination flows. In Arabnia, H.R., ed.: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02). Volume III., Las Vegas, NV, USA, CSREA Press (2002) 1407–1413
10. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces: Principles, Patterns, and Practice. The Jini Technology Series. Addison-Wesley (1999)
11. Omicini, A., Denti, E.: Formal ReSpecT. Electronic Notes in Theoretical Computer Science **48** (2001) 179–196 Declarative Programming – Selected Papers from AGP 2000, La Habana, Cuba, 4–6 December 2000.
12. Dijkstra, E.: Co-operating Sequential Processes. Academic Press, London (1965)
13. Wyckoff, P., McLaughry, S.W., Lehman, T.J., Ford, D.A.: T Spaces. IBM Journal of Research and Development **37** (1998) 454–474
14. tuProlog: Home page. `http://lia.deis.unibo.it/research/tuProlog/` (2001)
15. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152
16. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: a declarative language for real-time programming. In: 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM Press (1987) 178–188
17. Jacquet, J.M., De Bosschere, K., Brogi, A.: On timed coordination languages. In Porto, A., Roman, G.C., eds.: Coordination Languages and Models. Volume 1906 of LNCS., Springer-Verlag (2000) 81–98 4th International Conference (COORDINATION 2000), Limassol, Cyprus, 11–13 September 2000. Proceedings.
18. de Boer, F., Gabbrielli, M., Meo, M.C.: A Timed Linda language and its denotational semantics. Fundamenta Informaticae **63** (2004) 309–330

19. Denti, E., Natali, A., Omicini, A.: Programmable coordination media. In Garlan, D., Le Métayer, D., eds.: Coordination Languages and Models. Volume 1282 of LNCS., Springer-Verlag (1997) 274–288 2nd International Conference (COORDINATION'97), Berlin, Germany, 1–3 September 1997. Proceedings.
20. Ricci, A., Viroli, M.: A timed extension of ReSpecT. In: 2005 ACM Symposium on Applied Computing (SAC 2005), Santa Fe, NM, USA, ACM (2005) 420–427 Special Track on Coordination Models, Languages and Applications.

Transactional Aspects in Semantic Based Discovery of Services

Laura Bocchi, Paolo Ciancarini, and Davide Rossi

Dipartimento di Science dell'Informazione, University of Bologna,
Mura Anteo Zamboni 7, 40127 Bologna, Italy
{bocchi, cianca, rossi}@cs.unibo.it

Abstract. In a Service Oriented Architecture (SOA), services may need to dynamically discover non-functional properties of possible other services to cooperate with. Among these non-functional properties, transactional support is particularly relevant to enable coordination. In this paper we model the transactional support of Web services in a machine readable format (using OWL-S); in our model transactional support can be defined as negotiable thus requiring a run time multi step interaction among services to agree on the supported transaction type. We use the Business Transaction Protocol (BTP), a distributed transaction protocol, to carry out this negotiation. Specifically, we use an implementation of the bidding negotiation in BTP with the asynchronous pi calculus in order to provide a formal framework for these coordination issues.

1 Introduction

A Service Oriented Architecture (SOA) is '*a set of components which can be invoked, and whose interface descriptions can be published and discovered*' [1]. In this context, components are network addressable services with a well defined interface that have stateless connections and use standards communication protocols. However when complex coordination patterns involve multiple services, just like in many real world scenarios, stateful collaboration is often required. For example, in the e-business context it might be crucial to specify a precise order and causality of the service invocation (e.g., a purchase should happen after a payment).

In the Web Service scenario a number of standards have been proposed to coordinate correlated interactions among services, thus providing statefulness. Some of them extend the base Web Service properties by means of information inside the SOAP message headers. The additional features mainly concern Quality of Service issues (e.g., distributed transaction protocols as the Business Transaction Protocol [2] and WS-BusinessActivity [3]). Other standards regard the coordination of different services with workflow related technologies through the definition and management of business processes (e.g., orchestration/choreography languages such as BPEL4WS [4]).

In the context of Service Oriented Architectures (SOAs), components can be statically bound or they can dynamically search for other components to coop-

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 283–297, 2005.

erate with. In some cases the aspects that characterize a service can be known just at run time (e.g., the load of a system) or they are critical for the client service. This requires a multi-step interaction among the stakeholders in order to achieve an agreement on the provided features. This coordination issue is typically referred to as *negotiation* [5]. Negotiation is a topic of particular interest in the field of Multi Agent Systems (MAS) [6, 7, 8, 9]. Bidding protocols are a type of negotiation protocols that is suitable to represent this scenario. The intuition of bidding protocols is that a single coordinator asks for an offer to a number of participants, each makes an offer and the coordinator chooses one of the requests. A classic protocol is the Contract Net Protocol (CNP)[10], proposed in the scenario of distributed problem solving, where some loosely coupled distributed knowledge-sources (KS) have to find a cooperative solution to a problem, in a decentralized way. Some efforts are addressing the definition of standard technologies for negotiation in the Web Service scenario. In [11], a declarative XML language (WS-Negotiation) for Web Services providers and requestors, and a Service Level Agreement (SLA) template model are proposed. Negotiation is a issue of interest also in the Grid community: the Grid Resource Allocation Agreement Protocol (GRAAP) Working Group of the Grid Global Forum (GGF) [12] is currently working on a proposal, WS-Agreement [13], addressing message format in a negotiation being neutral with respect to the underlying negotiation protocol.

A key issue to enable automated service composition is to make the service descriptions machine readable. It is important to suitably define which aspects of a service are captured by the description and which kind of language is used to express them. In [14] three approaches are discussed: text based (searching is typically done by pattern matching), frame based (properties of a service are expressed as couples attribute-value), and ontology based. An ontology expresses a range of concepts and the relationship among them; it can be represented by a XML file and it is typically used to describe semantic aspects of a resource in a machine readable way. In [14] the advantages of using ontologies are remarked, basing on a comparison of the different approaches that consider the provided recall (i.e., absence of false negative) and precision (i.e., absence of false positive) in the search process. The World Wide Web Consortium (W3C) recommends the Web Ontology Language (OWL) [15] definition of ontologies on the Web. OWL-S [16] is an OWL based ontology for the description of services that has been created as a part of the DARPA Agent Markup Language Program (DAML) [17].

In the last years the challenges of e-science and e-business have been both addressed with the use of SOA instances: the Web Service Architecture [18] (WSA) and the Open Grid Service Architecture (OGSA) [19] respectively. Both WSA and OGSA present a link to Web Service technologies; this allows Grid related research to benefit from research in the generic scenario of service discovery in loosely coupled environment using Web Service technologies. The ongoing convergence between Grid and Web technologies has been represented in [20] within the same OWL-S ontology that has been extended with some Grid concepts.

In this paper, we put the focus on the problem of dynamic automated service discovery: a resource broker has to choose the most suitable service on the basis of a service request and a set of service descriptions. Service descriptions can be achieved from an information repository, for example an implementation of UDDI [21].

Our first contribution is a categorization for types of transactional support in loosely coupled environments that we express within the OWL-S ontology. In the last few years a considerable interest has been provoked by the adaptation of the classic concepts related with transactions to loosely coupled environments. The need of clarifying the exact meaning of the emerging concepts about transactions in the Web Service scenario triggered many efforts, mainly in the context of formal methods [22, 23, 24]. We propose an extension of OWL-S that includes the characterization of transaction types within service descriptions, and expresses the possibility of nesting one transaction type inside another. The principal benefit is to enable a resource broker to search for a service on the basis of that particular type of non-functional preference.

The second contribution of this paper, is to consider a negotiation scenario involving the proposed OWL-S extension. The service description we propose considers the possibility that a piece of information is statically undefined (i.e., declared as *negotiable*) and that a run time negotiation is required to determine it. We present a motivating scenario where a simple instance of negotiation is enacted using a distributed transaction protocol: an implementation of BTP with the asynchronous pi calculus [25], that was presented in [26]. Some considerations rise form this exercise about the suitability of BTP and the considered implementation of BTP to represent bidding.

In Section 2 we describe the evolution of concepts related to transactions toward their adaptation to loosely coupled environments. In Section 3 we present a classification of transaction types that we represent as a non-functional parameter in OWL-S. Section 4 describes a possible use of the extended ontology by a resource broker and a simple negotiation using BTP. Section 5 outlines conclusions and future works.

2 Transactions: An Overview of Emerging Concepts

A transaction, in its classic definition [27], is a transformation of state that possibly comprehends more simple actions satisfying three properties: Atomicity, Consistency and Durability. In literature it is often recalled a fourth property, Isolation. *Atomicity* is the 'all or nothing' property, *Consistency* assures the correctness of state transformation, *Isolation* requires the non interference between the executions of multiple simultaneous transactions, and *Durability* states that a transaction can not be abrogated once committed. Classic transactions are also referred to as *ACID transactions* to recall the four properties they satisfy. In 2.1 we discuss how ACID transactions, as they have been defined for tightly coupled environments, are not suitable for loosely coupled environments (i.e., SOAs). In 2.2 we describe the emerging concepts of transaction in the Web Service scenario.

2.1 Limits of ACID Transactions in Loosely Coupled Environments: The Notion of Compensation

Within a SOA where services are loosely coupled and not always trusted, standard ACID transactions can turn out to be too strict. We refer specifically to the properties of Isolation and Atomicity. Isolation is usually enforced locking the resources used by each activity until the transaction commits. Atomicity is typically assured by the fact that each participant makes temporary changes to the system till a global outcome is achieved among all the participants. Temporary changes have the effect of blocking the access of the involved resources until the end of the protocol (i.e., the achievement of an agreement). In case of negative outcome each part can undo all the temporary changes to the system (rollback) assuring that eventually all the actions enclosed in a transaction are done, or none is.

Sagas. The evidence that in some scenarios it is important to model transactions composed by sub-transactions underlined the need of weakening the notion of rollback. This idea has been introduced in the context of *data processing applications* [28, 29] with the *Sagas* concept. A Saga is composed by a set of transactions that have to be executed as an overall transaction. In a nested transaction the sub-transactions would have to block their resources until the overall outcome is decided. The overall outcome is decided only when all the enclosed transactions have performed the temporary changes and communicated a vote, that can take long time.

The Notion of Compensation. These considerations led to the definition of *compensation*. The compensation C of a transaction T is itself a transaction that can be executed after T's completion. This implies that T neither blocks its resources nor performs temporary changes to the system: the actions executed by T are immediately visible and actual. If an abort occurs in an outer environment, then C is executed to possibly undo the effects of T. It is important to notice that, while the rollback is an integrant part of the transaction execution, the compensation is an independent transaction executed afterwards and externally to the scope of T. The capability of C to undo all the previously performed actions is relative to the particular context: in some cases the compensation is not able to undo all the effects of T, as for instance in the case of data deletion or e-mail sending.

2.2 Long Running Transactions: Local and Global Perspectives

Web Services led to a renewed interest in that notion of weaker transaction, that is typically referred to as *long running transaction*.

In the Web Service scenario, a transaction might enclose resources belonging to different companies, it might be nested and lasts long periods of time. This could make resources block unfeasible. Also, the concept of compensation is particularly suitable to model typical e-business scenarios. For example a provider

might decide that rollback will not cancel all the operations carried out. To cancel an airplane booking, for instance, may lead to the payment of a fee.

Long running transactions are supported, in a mainly local perspective, by the languages for service orchestration/choreograohy such as BPEL4WS [4]. Distributed transaction protocols, such as BTP [2] and WS-BusinessActivity [3], also support the notion of compensation.

In some of these protocols transaction managers present a even more flexible behavior. In particular, BTP introduces two possible types of long running transactions, categorized by the respective transaction manager behavior: *atoms* and *cohesions*. An atom is a transaction that commits if and only if all its sub-entities

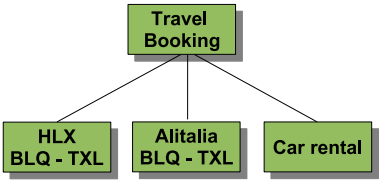


Fig. 1. An example of cohesion: *travel booking* tries to book different alternative flights (the same path with two different airlines) and to rent a car. *Travel booking* succeeds if at least one of the airlines has available flights. Car rental is not necessary. In that case both airlines have available flights, *travel booking* commits but rejects the reservation of the most expensive flight

commit: it commits only if all its sub-entities are able to commit, and in case of commit all its sub-entities commit. A cohesion can decide to commit even if some of its sub-entities are unable to commit. Furthermore in case of commit, a cohesion can decide to reject the commitment of some of its sub-entities, causing their failure. Figure 1 illustrates a travel booking service implemented by a cohesion that encloses multiple distributed and multi-organizational transactions.

3 Expressing a Classification of Transactional Types with OWL-S

In this section transactions are represented in OWL-S. The aim is to enable a requester to consider this information when performing the automated choice of the service to bind with. In 3.1 we introduce a classification for types of transactions and we express it as a non-functional feature inside OWL-S. In 3.2 we include transactions as a construct to define the business processes. This enables the requester to perform a more fine grained reasoning about the behavior of the service.

3.1 Transactions as Quality of Service Parameters

In the OWL-S ontology the generic features of services are described in the class *ServiceProfile* illustrated in Fig. 2. *ServiceProfile* provides an overview of the service characteristics and describes both its functional and non-functional aspects. The functional description consists mainly in the enumeration of its input/output parameters and its pre/post conditions. Non-functional aspects

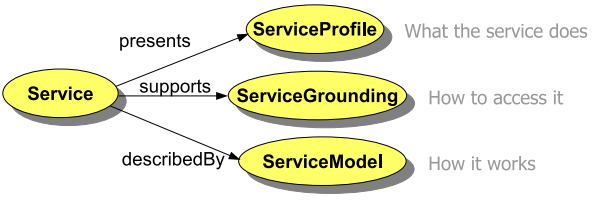


Fig. 2. Main classes of the OWL-based Web Service ontology [16]

refer to preference attributes that are used for evaluating the services, such as the Quality of Service. We represent transactions as a non-functional parameter that can be expressed in a subclass of *ServiceProfile*, nominally *ServiceParameter*. We extend *ServiceParameter* with the subclass *TransactionalSupport*, as illustrated in Fig. 3.

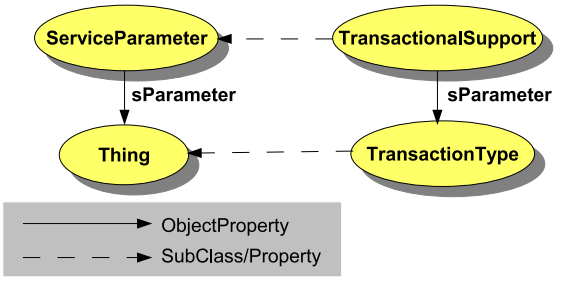


Fig. 3. The class *ServiceParameter* enables to express non-functional aspects of a service. We represent transactional support in the OWL-S ontology as a subclass *TransactionalSupport* of *ServiceParameter*

TransactionType represents a categorization of transactions in the Web Service scenario. In [27] the types of action enclosed within a transaction are classified as:

Unprotected actions need not to be undone. An example is an operation on temporary files. Its effects do not affect the system externally to the scope of the transaction that encloses it.

Protected actions can and must be undone. Conventional database operations are an example of protected actions. It need to be undone with an absolute rollback in case of failure of the overall transaction.

Real actions cannot be undone. An example is an action performed on a real device such as a cash dispenser.

A transaction, in its more general significance can enclose simple actions and other transactions (considered as higher level actions). Also transactions can be classified up to the list defined above (unprotected, protected or real). In the rest of the paper we refer to both as transactions if not specified otherwise. We generalize the classification above, keeping into account long running transactions, here defined *semi-protected* transactions, and *negotiable* transactions.

A semi-protected transaction tries to undo (i.e., compensates) its sub-transactions; it does not grant an absolute rollback. We introduce semi-protected transactions in the classification to distinguish in a more fine grained way between protected and real actions. An action corresponding to a compensating transaction is not classifiable as protected since it does not provide an absolute rollback. At the same time, classifying it as real could also be too restrictive.

The notion of negotiable transaction is crucial when a service is not able to statically describe the provided type of transactional support. This could happen because the transactional support depends on characteristics that change with time. For example a service might rely on a number of sub-services that are searched, discovered and bound at run time. Also, the nature of a transaction depends in part on the nature of its enclosed transactions (e.g., a transaction that encloses real transactions can not be categorized as protected). As a consequence, the transactional support of a service may depend on those of the sub-services that it discovers to bind with.

Table 1. Transaction types in the WSA and their relations. The cell a_{ij} represents the possibility of nesting the transaction type j within the transaction type i

out\in	unprotected	protected	semi-protected	real	negotiable
unprotected	yes	no	no	no	yes
protected	yes	yes	no	no	yes
semi-protected	yes	yes	yes	yes	yes
real	yes	yes	yes	yes	yes
negotiable	yes	yes	yes	yes	yes

In Table 1 we present the categorization of transactions in the Web scenario together with the less restrictive relation representing the possible nesting relations between them. For instance, the first line represents the possibility of nesting other types of transaction within an unprotected transaction: only unprotected and negotiable transactions can be nested within an unprotected one. A requester can define a more restrictive relation up to its internal policies (e.g., not allowing protected transactions to enclose negotiable transactions).

In Fig. 4 we illustrate how we include the table presented above in the OWL-S ontology. The class *TransactionType* is extended by the subclasses *Unprotected*, *Protected*, *SemiProtected*, *Real* ad *Negotiable*. The subclasses represent a partition of *TransactionType*. The relation of possible nesting described in Fig. 1 can be described with the relations (*ObjectProperty*) *CanEnclose* and *CannotEnclose*.

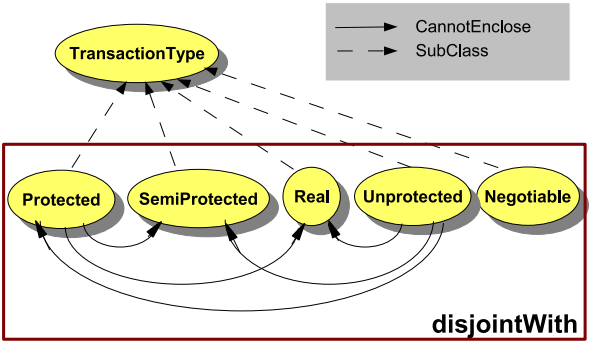


Fig. 4. Classification of transaction types and their possible nesting. For simplicity only the *CannotEnclose* property is illustrated here

3.2 Transactions in the OWL-S Process

Introducing semi-protected transactions does not add, per se, much information. Further interaction could be required between a resource broker and a service in order to establish up to witch degree the transaction is real and protected (e.g., to find an agreement on the compensation provided by the service). Possibly information about the abstract process of the service can be considered to this aim. We introduce an orthogonal characterization that describes transaction types from the point of view of a process. In fact we describe what relation has a transaction manger with its sub-transactions in the computation/coordination process perform to achieve the global outcome.

Atom it commits if and only if all the enclosed entities commit.

Cohesion it can commit also if some of its subparts are not *ready to commit* and in case of commit it can reject some of its subparts.

Atomic Transaction an Atom satisfying the ACID properties.

The process of a service is described in the subclass *Process* of *ServiceModel* (see Fig. 2 and Fig. 5). There are three classes that represent a process: *SimpleProcess*, *AtomicProcess* and *CompositeProcess*. *SimpleProcess* introduces a level of abstraction to refer to the less general concepts (i.e., *AtomicProcess* and *CompositeProcess*), and it is not directly callable. *AtomicProcess* is the basic unit of implementation. With the class *CompositeProcess*, OWL-S enables the description of processes that are composite by a set of other processes by means of a

number of constructs (sequence, split, split-join, parallel and choice) represented by the class *ControlConstruct*. We introduce *Transaction* as a control construct, as illustrated in Fig. 5.

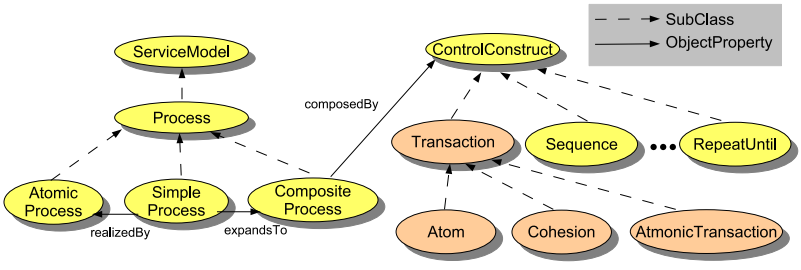


Fig. 5. The class *Process* is a subclass of *ServiceModel* (see Fig. 2). A *CompositeProcess* is composed by other (composite or non-composite) processes by using control constructs as sequence, parallel execution, etc. Transaction is introduced as *ControlConstruct*

A further discussion about negotiation protocols and reasoning on the process of a service, in order to have a more fine grained idea on the compensation policy, is out of the scope of this work. We limit ourself to introduce the base classification; a future work will expand such a base.

4 A Motivating Scenario Involving Negotiation

Let r be a resource broker searching for a service that grants a transactional support t. The required transactional features are expressed by the type $t \in \{Unprotected, Protected, SemiProtected, Real, Negotiable\}$. We limit our focus to the search for a single service. The problem of searching for a set of n independent services would be reducible to n instances of the problem with one service. Complexity would be added if searching for n services cooperating within a workflow schedule. The resource broker evaluates a range of available services descriptions $s_1^1, \ldots, s_1^n$ that are ontologies stored in a *Information Service*. Their supported transaction types $t_1^1, \ldots, t_1^n$ are expressed in their respective ontology (precisely by a particular subclass of *TransactionType*).

We suppose that a part of the services under evaluation have transaction types $t_1^1, \ldots, t_1^m$ for some $m < n$ statically defined (that is not *Negotiable*). This information can be directly used in the internal choice algorithms of the resource broker, in particular we suppose that r eliminates from the evaluation process all the services s_1^i such that it exists a relation *CannotEnclose* from t to t_1^i. We remark that r can define its personal relations as a restriction of those expressed in Fig. 4 between the subclasses of *TransactionType*.

On the other hand $s_1^{m+1}, \ldots, s_1^n$ have *Negotiable* transaction types. An agreement between r and $s_1^{m+1}, \ldots, s_1^n$ is required, involving a multi-step interaction.

The piece of information achieved by the interaction consists in the transactional types that can be supported by $s_1^{m+1}, \ldots, s_1^n$ in the contingency.

In 4.1 we briefly describe an implementation of BTP with the asynchronous pi calculus. The reader can refer to [26] for a deeper treatment. In 4.2 we use the implementation described in 4.1 to define the interaction pattern between r and $s_1^{m+1}, \ldots, s_1^n$.

4.1 A BTP Implementation with the Asynchronous Pi Calculus

In [26] we studied the problem of coordinating distributed business transactions and we formally modeled the behavior of atom and cohesion. We recall from section 2.2 that a cohesion is a transaction that needs only some of its sub-transactions to succeed, while an atom requires them all to succeed. A set

Table 2. The asynchronous pi calculus

Terms P and contexts C in the asynchronous pi calculus are as follows. In $u(\widetilde{x})$ the names $\widetilde{x}$ are bound, as is x in $\nu x.P$. We identify terms up to alpha-renaming of bound names.

$$\begin{array}{rcl} P & ::= & \mathsf{0} \mid \overline{u}\,\widetilde{x} \mid u(\widetilde{x}).P \mid P|P \mid \nu x.P \mid\ !P \\ C & ::= & \_ \mid u(\widetilde{x}).C \mid P|C \mid C|P \mid \nu x.C \mid\ !C \end{array}$$

Labeled transitions are as follows, where labels μ range over $u(\widetilde{x})$, $\nu\tilde{z}.\overline{u}\,\tilde{x}$ and τ.

$$\overline{u}\,\widetilde{x} \xrightarrow{\overline{u}\,\widetilde{x}} \mathsf{0} \quad (\textsc{out}) \qquad u(\widetilde{x}).P \xrightarrow{u(\widetilde{x})} P \quad (\textsc{in}) \qquad \frac{P \mid !P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'} \quad (\textsc{rep})$$

$$\frac{P \xrightarrow{\mu} P' \quad x \notin \mu}{\nu x.P \xrightarrow{\mu} \nu x.P'} \quad (\textsc{res}) \qquad \frac{P \xrightarrow{\nu\tilde{z}.\overline{u}\,\tilde{y}} P' \quad x \neq u, x \in \tilde{y} \backslash \tilde{z}}{\nu x.P \xrightarrow{\nu\tilde{z}x.\overline{u}\,\tilde{y}} P'} \quad (\textsc{open})$$

$$\frac{P \xrightarrow{\mu} P' \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\mu} P' \mid Q} \quad (\textsc{par}) \qquad \frac{P \xrightarrow{\nu\tilde{z}.\overline{u}\,\tilde{y}} P' \quad Q \xrightarrow{u(\tilde{x})} Q' \quad \tilde{z} \cap \mathrm{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\tau} \nu\tilde{z}.(P' \mid Q'\{\tilde{y}/\tilde{x}\})} \quad (\textsc{com})$$

Notation We write $\widetilde{x}_{\mathcal{C}}$ for an arbitrary sequence $x_1, \ldots, x_n$ of the elements in set $\mathcal{C}$. We also use these syntactic sugars:

$$\begin{array}{rcll} x.P & = & x().P & \text{(empty input)} \\ \widetilde{x}.P & = & x_1.\ldots.x_n.P & \text{(sequence input)} \\ \nu\widetilde{x}.P & = & \nu x_1.\ldots.\nu x_n.P & \text{(sequence restriction)} \\ P \oplus Q & = & \nu c.(\overline{c}\,|c.P|c.Q),\ c \text{ fresh} & \text{(nondeterministic choice)} \\ x[P,Q] & = & \nu u, v.(\overline{x}\,u, v|u.P|v.Q),\ u, v \text{ fresh} & \text{(selection)} \\ \overline{x}\,\mathsf{left} & = & x(u,v).\overline{u} & \\ \overline{x}\,\mathsf{right} & = & x(u,v).\overline{v} & \end{array}$$

of transactions that compose a complex activity, with an arbitrary number of nesting levels, can be coordinated by means of precise message patterns. Each transaction has a transaction manager associated, to manage the message pattern. [26] provides a distributed implementation of a transaction manager with the asynchronous pi calculus [25]. The asynchronous pi calculus and particular notation used in [26] have been summarized in Table 2 for reader convenience.

Cohesion is modeled here as an entity able to flexibly specify the relationship with its children. To model a cohesion, the transaction manager creates two partitions of the set of all its sub-transactions. Let $C(r)$ be the set of the children of the cohesion r.

- necessary/unnecessary: $C(r)$ is partitioned into $N(i)$ and $U(i)$ with the meaning that success of all $N(r)$ is necessary for r to succeed, while $U(r)$ are unnecessary.
- accept/reject: $C(r)$ is partitioned into $A(r)$ and $R(r)$ where all of $A(r)$ are accepted, while all of $R(r)$ are rejected (undone) in the case r succeeds.

A cohesion with $C(i) = N(i)$ and $A(i) = C(i)$ is a special case that behaves as an atom. A *two phase commit protocol* is used first to assemble the 'votes' of nested transactions (i.e., whether or not they succeeded), and second to inform them all of the consensus decision. An implementation of the two phase commit protocol with the asynchronous pi calculus is presented in [30] where unreliability is taken into account (message loss and node failure).

Table 3. The implementation of the transaction manager [26]

$$T_i = \nu a_i, ms_i, vs_i, \widetilde{m}_{C(i)}, \widetilde{v}_{C(i)}, \widetilde{d}_{C(i)}.(T_i.\text{sv} \mid T_i.\text{m} \mid T_i.\text{col} \mid \prod_{c \in C(i)} T_c)$$

$$T_i.\text{sv} = \overline{vs}_i left \oplus \overline{vs}_i right$$

$$T_i.\text{m} = vs_i[\overline{ms}_i, \overline{a}_i] \mid \prod_{c \in N(i)} v_c[\overline{m}_c, \overline{a}_i] \mid \prod_{c \in U(i)} v_c[\overline{m}_c, \overline{m}_c]$$

$$T_i.\text{col} = a_i.(\overline{v}_i right \mid T_i.\text{fail}) \mid \widetilde{m}_{C(i)}.ms_i.(\overline{v}_i left \mid d_i[T_i.\text{ok}, T_i.\text{fail}])$$

$$T_i.\text{ok} = \overline{ok}_i \mid \prod_{c \in A(i)} \overline{d}_c left \mid \prod_{c \in R(i)} \overline{d}_c right$$

$$T_i.\text{fail} = \overline{abort}_i \mid \prod_{c \in C(i)} \overline{d}_c right$$

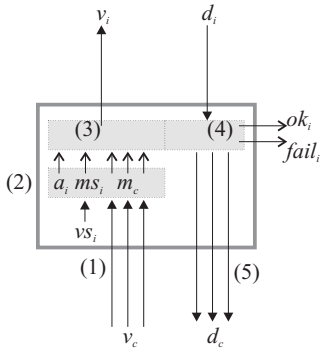


Fig. 6. Message exchange from the perspective of the transaction manager T_i [26]

In [26] the two phase commit protocol is modeled considering an arbitrary number of level nesting: any transaction manager has to manage the interactions with both its top level and its bottom level in the nesting hierarchy. Also, the modeled transaction manager captures both cohesion and atom behavior. Table 3 describes the implementation of the transaction manager presented in [26]. Fig. 6 illustrates the behavior of the transaction manager.

Referring to Fig. 6: **(1)** the transaction i itself makes a non-deterministic 'self' vote vs_i. Also, all of the children c make their votes v_c. **(2)** Each vote is transformed into an 'internal message' m. The purpose of this translation is to separate necessary child votes $N(i)$ from unnecessary votes $U(i)$. Each internal message m_c means that the child either voted success (left), or it was unnecessary. But if a child should vote failure (right) and was necessary, it will make an 'abort' signal a_i instead. **(3)** If all internal messages ms_i/m_c arrive, then the transaction i as a whole can succeed, and so indicates success (left) to its parent over the channel v_i. But if even one abort message a_i was received, then the transaction as a whole fails, and so it indicates failure (right). **(4)** Eventually the parent p will know whether to accept i, or to abort/undo it. This decision is communicated to i via the 'decision' channel d_i, and so determines i's final state. The transaction i can indicate its final state via the messages $ok_i/abort_i$. **(5)** Finally, the decision is propagated down to all the children c. The accepted children, those in $A(i)$, will be told the same decision as i received. The rejected children, those in $R(i)$, will be told to abort/undo regardless.

Correctness of the protocol has been proved in [26] up to the notions of *Durability*, *Eventuality* (eventually each node reaches an outcome) and *Local Atomicity*. Of particular interest is the property of local atomicity: it is a weaker, more general version of atomicity granted by cohesions: if one transaction fails, then all its children fail.

4.2 Enabling Dynamic Properties Discovery with BTP

We suppose that the resource broker performs a ranking of the services $s_1^1, \ldots, s_1^n$ based on other already known preference parameters. Ranking is expressed by

the permutation function f of $\{1,\ldots,n\}$. Let us assume that all the $s_1^1,\ldots,s_1^m$ (i.e., the services whose transaction type is not negotiable) provide a suitable transactional support. To complete the ranking, an interaction has to be performed with services in order to delete those of the $s_1^{m+1},\ldots,s_1^n$ that do not provide enough transactional support.

Let us build a table with n rows: one for each evaluated service. The implementation of BTP that we considered does not take into account run time decisions (that is why we use the table). Virtually we think n versions of the protocol to be run in parallel. This simulates the fact that r can partition $C(r)$ in necessary/unnecessary at run time depending on information obtained in the first phase of the protocol. Each row of the table below represents an instance of the protocol. In each instance of the protocol we consider only one service, say s_1^j.

- $\forall j : j > m$ we model that if s_1^j can provide suitable transactional support than is chosen, independently from the behavior of the other services. This is expressed by the partitions of $C(r)$: $N(r) = \{s_1^j\}$ and $A(r) = \{s_1^j\}$ (all the other services are unnecessary and rejected).
- $\forall j : j \leq m$, s_1^j is not involved in the negotiation. In this case we consider the trivial instance of the protocol where partition $C(r)$ as follows: $N(r) = A(r) = \emptyset$ (all the services involved in the negotiation are rejected).

row	*If all of these succeed...*	*then accept these...*	*and undo these*
$1\ldots m$	$\emptyset$	$\emptyset$	$C(r)$
$m+1$	$\{s_1^{m+1}\}$	$\{s_1^{m+1}\}$	$\{C(r) \setminus \{s_1^{m+1}\}\}$
$\ldots$	$\ldots$	$\ldots$	$\ldots$
n	$\{s_1^n\}$	$\{s_1^n\}$	$\{C(r) \setminus \{s_1^n\}\}$

Finally we order rows up to the ranking function f. The protocol described in [26] considers a single row of the table, rather than multiple rows in each protocol specification. To handle multiple rows, something like Join patterns [31] might be used. Here we simply illustrate a possible use of the protocol to handle complex choices: in fact $n-(m+1)$ instances of the protocol are run in parallel, when all the information $(t_1^{m+1},\ldots,t_1^n)$ is available to r then a choice is made depending on the ranking function. The rows of the table are considered in the order expressed by f. It is chosen the service relative to the first row in the ranking order with successful outcome.

5 Conclusion and Future Works

In a bidding protocol the evaluation of the participants proposals has to be done at run time. BTP allows a run time evaluation: 'The determination of the Confirm-set is made by the controlling application, but is affected by events from the Inferiors themselves'[2]. The protocol we considered defines the sets of necessary and accepted sub-transactions statically. We modeled this using multiple entries of a table. Including run time decision in the protocol implementation would not allow to separate the protocol level and the application level information, but would possibly lead to a more elegant representation.

As future work, we are extending [26] with the management of unreliability, in particular managing message loss with the use of timers as those described in [30]. In a general bidding scenario a coordinator does not necessarily wait that all the participants send a feedback: it can decide a deadline and evaluate the participants that answered before the expiration. In [26] the coordinator waits the votes from all the participants. The use of timers would possibly enable the representation of this behavior.

References

1. Booth D., Haas H., and Brown A. Web Services Glossary. Technical report, World Wide Web Consortium (W3C), 2004. http://www.w3.org/TR/ws-gloss/.
2. OASIS. Business Transaction Protocol. http://www.oasis-open.org/committees/download.php/1184/20020603.BTP_cttee_spec_1.0.pdf, 2002.
3. Cabrera L. F., Copeland G., Freund T., Klein J., Langworthy D., Leymann F., Orchard D., Robinson I., Storey T., and Thatte S. Web Services Business Activity Framework (WS-BusinessActivity). http://eu-datagrid.web.cern.ch/eu-datagrid/.
4. Curbera F., Goland Y., Klein J., Leymann F., Roller D., Thatte S., and Weerawarana S. Business Process Execution Language for Web Services (BPEL4WS 1.0). Technical report.
5. R. Davis and R. G. Smith. Negotiation as a Metaphor for Distributed Problem Solving. In *Readings in Distributed Artificial Intelligence*, pages 333–356. Morgan Kaufmann Publishers Inc., 1988.
6. S. Kraus. *Strategic Negotiation in Multi-Agent Environments.* MIT Press, Cambridge, MA, 2000.
7. B. Laasri, H. Laasri, S. Lander, and V. Lesser. A Generic Model for Intelligent Negotiating Agents. *International Journal on Intelligent Cooperative Information Systems*, 1(2):291–317, January 1992.
8. N. Maudet. Negotiating dialogue games. *Journal of autonoumous agents and multi-agent systems*, 7(2):229–233, November 2003.
9. S. Kraus, J. Wilkenfeld, and G. Zlotkin. Multiagent negotiation under time constraints. *Artificial Intelligence*, 75(2):297–345, 1995.
10. R. G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. In *Readings in Distributed Artificial Intelligence*, pages 357–366. Morgan Kaufmann Publishers Inc., 1988.
11. Hung C. K., Li H., and Jeng J. WS-Negotiation: An overview of research issues. In *Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS'04)*. IEEE Computer, 2004.
12. Global Grid Forum. http://www.ggf.org/.
13. Andrieux A., Czajkowski K., Dan A., Keahey K., Ludwig H., Pruyne J., Rofrano J., Tuecke S., and Xu M. Web Services Agreement Specification (WS-Agreement). https://forge.gridforum.org/projects/graap-wg/document/WS-AgreementSpecification/en/6.
14. Klein M. and Bernstein A. Searching for Services on the Semantic Web Using Process Ontologies. In Isabel Cruz, Stefan Decker, Jerome Euzenat, and Deborah McGuinness, editors, *The Emerging Semantic Web - Selected papers from the first Semantic Web Working Symposium*, pages 159–172. IOS press, Amsterdam, 2002.

15. Bechhofer S., Harmelen F., Hendler J., Horrocks D., McGuinnes I., Patel-Schneider P., and Stein L.A. OWL Web ontology language reference. Technical report, W3C, 2004.
16. OWL-S 1.0 rel. 1.0. http://www.daml.org/services/owl-s/1.0/.
17. The DARPA Agent Markup Language. http://www.daml.org.
18. Booth D., Haas H., McCabe F., Newcomer E., Champion M., Ferris C., and Orchard D. Web Service Architecture. Technical report, World Wide Web Consortium (W3C), 2004. http://www.w3.org/TR/ws-arch/.
19. Foster I., Kesselman C., and Tuecke S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.
20. Bocchi L., Ciancarini P., Moretti R., Presutti V., and Rossi D. An OWL-S Based Approach to Express Grid Services Coordination. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC 2005)*. ACM, 2005. To appear.
21. UDDI. http://www.uddi.org.
22. Bocchi L., Laneve C., and Zavattaro G. A calculus for long running transactions. In Elie Najm, Uwe Nestmann, and Perdita Stevens, editors, *Proceedings of the 6th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2003.
23. Bruni R., Melgratti H., and Montanari U. Nested commits for mobile calculi: extending join. In *Proceedings of 3rd IFIP International Conference on Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics (IFIP TCS 2004)*, pages 563–576, 2004.
24. Laneve C. and Zavattaro G. Foundations of Web Transactions. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS 2005)*, Lecture Notes in Computer Science, 2005. To appear.
25. Milner R. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1989.
26. Bocchi L. Compositional Nested Long Running Transactions. In Michel Wermelinger and Tiziana Margaria, editors, *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE 2004)*, volume 2984 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2004.
27. Gray J. The Transaction Concept: Virtues and Limitations. In *Proceedings of Very Large Data Bases*, pages 179–201, 1981.
28. Garcia-Molina H. and Salem K. Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM Press, 1987.
29. Garcia-Molina H., Gawlick D., Klein J., Kleissner K., and Salem K. Modeling Long-Running Activities as Nested Sagas. *Data Eng.*, 14(1):14–18, 1991.
30. Berger M. and Honda K. The Two-Phase Commitment Protocol in an Extended Pi-calculus. *Electronic Notes in Theoretical Computer Science*, 39(1):105–130, 2003.
31. Le Fessant F. and Maranget L. Compiling Join-Patterns. In *Proceedings of High-Level Concurrent Languages (HLCL '98)*, volume 16 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers.

Author Index