Rocco De Nicola
Christine Julien (Eds.)

LNCS 7890

Coordination Models and Languages

15th International Conference, COORDINATION 2013
Held as Part of the 8th International Federated Conference
on Distributed Computing Techniques, DisCoTec 2013
Florence, Italy, June 2013, Proceedings

ifip

Lecture Notes in Computer Science 7890

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

Rocco De Nicola Christine Julien (Eds.)

Coordination Models and Languages

15th International Conference, COORDINATION 2013
Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013
Florence, Italy, June 3-5, 2013
Proceedings

Springer

Volume Editors

Rocco De Nicola
IMT - Institute for Advanced Studies Lucca
Piazza San Ponziano 6, 55100 Lucca, Italy
E-mail: rocco.denicola@imtlucca.it

Christine Julien
The University of Texas at Austin
One University Station, C5000, Austin, TX 78712, USA
E-mail: c.julien@mail.utexas.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-38492-9 e-ISBN 978-3-642-38493-6
DOI 10.1007/978-3-642-38493-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013938179

CR Subject Classification (1998): D.2, D.4, F.1, F.3, C.2, C.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

Printed on acid-free paper

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

Foreword

In 2013 the 8th International Federated Conference on Distributed Computing Techniques (DisCoTec) took place in Florence, Italy, during June 3–6. It was hosted and organised by Università di Firenze. The DisCoTec series of federated conferences, one of the major events sponsored by the International Federation for Information processing (IFIP), included three conferences:

- The 15th International Conference on Coordination Models and Languages (Coordination)
- The 13th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)
- The 2013 IFIP Joint International Conference on Formal Techniques for Distributed Systems (33rd FORTE / 15th FMOODS)

Together, these conferences cover the complete spectrum of distributed computing subjects ranging from theoretical foundations to formal specification techniques to systems research issues.

Each of the first three days of the federated event began with a plenary speaker nominated by one of the conferences. The three invited speakers were: Tevfik Bultan, Department of Computer Science at the University of California, Santa Barbara, UCSB; Gian Pietro Picco, Department of Information Engineering and Computer Science at the University of Trento, Italy; and Roberto Baldoni, Department of Computer, Control and Management Engineering "Antonio Ruberti", Università degli Studi di Roma "La Sapienza", Italy. In addition, on the second day, there was a joint technical session consisting of one paper from each of the conferences. There were also three satellite events:

1. The 4th International Workshop on Interactions between Computer Science and Biology (CS2BIO) with keynote talks by Giuseppe Longo (ENS Paris, France) and Mario Rasetti (ISI Foundation, Italy)
2. The 6th Workshop on Interaction and Concurrency Experience (ICE) with keynote lectures by Davide Sangiorgi (Università di Bologna, Italy) and Damien Pous (ENS Lyon, France)
3. The 9th International Workshop on Automated Specification and Verification of Web Systems (WWV) with keynote talks by Gerhard Friedrich (Universität Klagenfurt, Austria) and François Taïani (Université de Rennes 1, France)

I believe that this program offered each participant an interesting and stimulating event. I would like to thank the Program Committee Chairs of each conference and workshop for their effort. Moreover, organizing DisCoTec 2013 was only possible thanks to the dedicated work of the Publicity Chair Francesco Tiezzi (IMT Lucca, Italy), the Workshop Chair Rosario Pugliese (Università

di Firenze, Italy), and members of the Organizing Committee from Università di Firenze: Luca Cesari, Andrea Margheri, Massimiliano Masi, Simona Rinaldi, and Betti Venneri. To conclude I want to thank the International Federation for Information Processing (IFIP) and Università di Firenze for their sponsorship.

June 2013 Michele Loreti

Preface

This volume contains the papers presented at Coordination 2013: the 15th International Conference on Coordination Models and Languages held in Florence during June 3–5. The conference focused on the design and implementation of models and technologies for collaboration and coordination in concurrent, distributed, and socio-technical systems, including both practical and foundational models, runtime systems, and related verification and analysis techniques.

The Program Committee (PC) of Coordination 2012 consisted of 25 top researchers from 13 different countries. We received more than 50 abstracts that materialized in a total of 42 submissions out of which the PC selected 17 papers for inclusion in the program. Each submission was reviewed by at least three independent referees; papers were selected based on their quality, originality, contribution, clarity of presentation, and relevance to the conference topics. The review process included an in-depth discussion phase, during which the merits of all papers were discussed by the committee. The process culminated in a shepherding phase whereby some of the authors received active guidance by one member of the PC in order to produce a high-quality final version. The selected papers constituted a program covering a varied range of topics including coordination of social collaboration processes, coordination of mobile systems in peer-to-peer and ad-hoc networks, programming and reasoning about distributed and concurrent software, types, contracts, synchronization, coordination patterns, and families of distributed systems. The program was further enhanced by an invited talk by Gian Pietro Picco from the University of Trento entitled "Of Tuples, Towers, Tunnels, and Wireless Sensor Networks."

The success of Coordination 2013 was due to the dedication of many people. We thank the authors for submitting high-quality papers and the Program Committee (and their sub-reviewers) for their careful reviews and lengthy discussions during the final selection process. We thank Francesco Tiezzi from IMT Lucca, who acted as the Publicity Chair of Coordination 2013. We thank the providers of the EasyChair conference management system, which was used to run the review process and to facilitate the preparation of the proceedings. Finally, we thank the Distributed Computing Techniques Organizing Committee (led by Michele Loreti) for their contribution in making the logistic aspects of Coordination 2012 a success.

June 2013

Rocco De Nicola
Christine Julien

Organization

Steering Committee

Farhad Arbab	CWI and Leiden University, The Netherlands (Chair)
Gul Agha	University of Illinois at Urbana-Champaign, USA
Dave Clarke	KU Leuven, Belgium
Jean-Marie Jacquet	University of Namur, Belgium
Wolfgang De Meuter	Vrije Universiteit Brussel, Belgium
Rocco De Nicola	IMT - Institute for Advanced Studies Lucca, Italy
Marjan Sirjani	Reykjavik University, Iceland
Carolyn Talcott	SRI, USA
Vasco T. Vasconcelos	University of Lisbon, Portugal
Gianluigi Zavattaro	Università di Bologna, Italy

Program Committee

Gul Agha	University of Illinois at Urbana-Champaign, USA
Saddek Bensalem	VERIMAG Laboratory, France
Borzoo Bonakdarpour	The University of Waterloo, Canada
Giacomo Cabri	Università di Modena e Reggio Emilia, Italy
Paolo Ciancarini	Università di Bologna, Italy
Dave Clarke	KU Leuven, Belgium
Frank De Boer	CWI, The Netherlands
Wolfgang De Meuter	Vrije Universiteit Brussel, Belgium
Rocco De Nicola	IMT - Institute for Advanced Studies Lucca, Italy
José Luiz Fiadeiro	Royal Holloway, University of London, UK
Chien-Liang Fok	The University of Texas at Austin, USA
Chris Hankin	Imperial College London, UK
Raymond Hu	Imperial College London, UK
Christine Julien	The University of Texas at Austin, USA
K.R. Jayaram	HP Labs, USA
Eva Kühn	Vienna University of Technology, Austria
Mieke Massink	ISTI-CNR, Pisa, Italy
Jamie Payton	The University of North Carolina at Charlotte, USA
Rosario Pugliese	Università di Firenze, Italy
Nikola Serbedzija	Fraunhofer FOKUS, Germany
Marjan Sirjani	Reykjavik University, Iceland

Robert Tolksdorf	Freie Universität Berlin, Germany
Emilio Tuosto	University of Leicester, UK
Vasco T. Vasconcelos	University of Lisbon, Portugal
Mirko Viroli	Università di Bologna, Italy

Additional Reviewers

Bocchi, Laura	Marek, Alexander
Bourgos, Paraskevas	Marques, Eduardo R.B.
Bruni, Roberto	Marr, Stefan
Capodieci, Nicola	Martins, Francisco
Charalambides, Minas	Melgratti, Hernan
Ciancia, Vincenzo	Mezzina, Claudio Antares
Craß, Stefan	Mostrous, Dimitris
Creissac Campos, José	Nobakht, Behrooz
De Wael, Mattias	Poplavko, Peter
Dinges, Peter	Proenca, Jose
Fabiunke, Marko	Puviani, Mariachiara
Harnie, Dries	Quilbeuf, Jean
Jafari, Ali	Ridge, Tom
Jaghoori, Mohammad Mahdi	Sabouri, Hamideh
Jongmans, Sung-Shik T.Q.	Scholliers, Christophe
Khamespanah, Ehsan	Senni, Valerio
Kock, Gerd	Sesum-Cavic, Vesna
Lanese, Ivan	Tiezzi, Francesco
Latella, Diego	Vandriessche, Yves

Of Tuples, Towers, Tunnels, and Wireless Sensor Networks

Gian Pietro Picco

Department of Information Engineering and Computer Science (DISI),
University of Trento, Italy
gianpietro.picco@unitn.it

Wireless sensor networks (WSNs) have been around for more than a decade. They are distributed systems made of tiny, resource-scarce, often battery-powered devices that cooperate toward distributed monitoring and control. Their small size, autonomy, and flexibility has placed them at the forefront of pervasive computing scenarios. Yet, their programming is still largely carried out by using directly the low-level primitives provided by the operating system. This approach steers the programmer away from the application, hampers reusability and decoupling, and ultimately makes development unnecessarily complex.

In this talk, we report our research efforts in simplifying WSN programming through the notion of tuple space, embodied in a middleware called TeenyLime [1]. As the name implies, TeenyLime borrows the transiently shared tuple space model introduced by the Lime middleware for mobile ad hoc networks, but also deeply revisits it to match the peculiar characteristics of WSNs, e.g., to deal with the limited resources of WSN nodes and to provide additional visibility and control on the lower levels of the stack. TeenyLime was indeed designed as a thin veneer atop the basic OS communication facilities, to support the development of both the application logic and system tasks such as routing.

TeenyLime and its particular incarnation of the tuple space concept proved successful in developing several *real-world* applications where the WSN was deployed for a long time, in an operational setting, and most importantly to fulfill the needs of real users. This talk reports on two of these experiences: the structural health monitoring of a medieval tower [2], and the closed-loop control of adaptive lighting of a road tunnel [3]. Facing real-world challenges forced a redesign of the TeenyLime implementation, but left its original abstractions essentially unaltered. As expected, the higher level of abstraction provided by TeenyLime w.r.t. using directly the OS primitives resulted in a significant reduction of the source code size, hinting at a lower burden on the programmer. Less expected, it also resulted in a smaller binary size, therefore enabling one to pack more functionality on the resource-scarce WSN nodes. Ultimately, these experiences clearly showed that it is possible to simplify the chore of programming WSN applications without sacrificing their performance and efficiency.

TeenyLime is available as open source at `teenylime.sourceforge.net`.

References

1. Costa, P., Mottola, L., Murphy, A.L., Picco, G.P.: Programming Wireless Sensor Networks with the TeenyLime Middleware. In: Cerqueira, R., Campbell, R.H. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 429–449. Springer, Heidelberg (2007)
2. Ceriotti, M., Mottola, L., Picco, G.P., Murphy, A.L., Guna, S., Corrà, M., Pozzi, M., Zonta, D., Zanon, P.: Monitoring Heritage Buildings with Wireless Sensor Networks: The Torre Aquila Deployment. In: Proc. of the 8th ACM/IEEE Int. Conf. on Information Processing in Sensor Networks (IPSN) (2009)
3. Ceriotti, M., Corrà, M., D'Orazio, L., Doriguzzi, R., Facchin, D., Guna, S., Jesi, G.P., Cigno, R.L., Mottola, L., Murphy, A.L., Pescalli, M., Picco, G.P., Pregnolato, D., Torghele, C.: Is There Light at the Ends of the Tunnel? Wireless Sensor Networks for Adaptive Lighting in Road Tunnels. In: Proc. of the 10th ACM/IEEE Int. Conf. on Information Processing in Sensor Networks (IPSN) (2011)

Table of Contents

Stochastic Process Algebra and Stability Analysis of Collective Systems

Luca Bortolussi[1,2,⋆], Diego Latella[2], and Mieke Massink[2]

[1] Dipartimento di Matematica e Geoscienze, University of Trieste, Italy
[2] Istituto di Scienza e Tecnologie dell'Informazione 'A. Faedo', CNR, Italy

Abstract. Collective systems consist of large numbers of agents that coordinate through local behaviour, adapt to their environment and possibly give rise to emergent phenomena. Their formal analysis requires advanced scalable mathematical approximation techniques. We show how Stochastic Process Algebra (SPA) can be combined with numeric analysis tools for the analysis of emergent behavioural aspects of such systems. The approach is based on an automatic transformation of SPA models into ordinary differential equations in a format in which numeric and symbolic computing environments can be used to perform stability analysis of the system. The potential of the approach is illustrated by a crowd dynamics scenario in which various forms of behavioural and topological asymmetry are introduced. These are cases in which analytical approaches to stability analysis are in general not feasible. The analysis also shows some surprising aspects of the crowd model itself.

Keywords: Fluid flow, process algebra, crowd dynamics, self-organisation.

1 Introduction

A key factor to allow modern cities to reach or maintain a good and sustainable quality of life for their increasingly numerous inhabitants is the development of systems that are relying on a much more decentralised and distributed design that is adapting itself to dynamically changing circumstances [19]. Examples of such future systems are electricity grids that can cope with many local electricity producers and consumers, and a decentralised organisation of transportation and information. Such large scale collective adaptive systems rely on the continuous feedback between vast numbers of participants of different kinds, and, as is well-known, can be expected to show complex dynamic and emergent behaviour or perhaps even exploit such behaviour [8]. Also the formal analysis of such systems poses many new research challenges.

Process algebras have been specifically designed for the compositional and high-level modelling and analysis of distributed concurrent systems. Recently some of them, in particular PEPA [10] and Bio-PEPA [7], have been provided with a fluid semantics based on ordinary differential equations (ODE) [11] providing a scalable approach to the analysis of agent coordination in large collective

⋆ Work partially supported by the project "FRA-UniTS".

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 1–15, 2013.

systems. In this paper we further exploit this development to study stability aspects of collective adaptive dynamic systems in a symbolic and numeric way. Stability analysis provides important information about the predictability of dynamic systems and their sensitivity to parameter values. Numeric stability analysis is of particular interest for the analysis of distributed adaptive strategies that are applied in *asymmetric* situations. Such situations occur naturally in many real-world, natural or designed, systems. An analytical approach to stability analysis in the presence of asymmetry is infeasible in most cases. To the best of our knowledge this is the first time that a method is proposed for stability analysis that starts from a stochastic process algebra specification of agent coordination in a collective dynamic system. The use of a process algebra greatly facilitates the modelling of variants of a system at a high level. Designers can then focus on the coordination strategies instead of having to manipulate the underlying, possibly large set of non-compositional, ODEs in ad-hoc ways.

We illustrate the approach and related tool chain by analysing a new variant of a collective model of spontaneous self-organisation of drinking parties in the squares of cities in Spain, also known as "El Botellón" [21]. In this variant the parameter of the system is the *level of socialisation*, i.e. the average number of friends people have, instead of the the probability to find a partner to chat with[1]. The model shows a number of surprising behavioural aspects. However, the main contribution of the paper is to illustrate and explore what stochastic process algebra can offer to provide high-level models of coordination in large scale collective systems, in combination with well-established numeric and symbolic analysis techniques for a systematic stability analysis of such systems. In particular in case of models showing various forms of asymmetry.

Related work can be found in several directions. In [15] the original model of Rowe and Gomez was analysed with Bio-PEPA. Both empirical and analytical justification was provided for the good correspondence found between stochastic simulation results and the Bio-PEPA based fluid flow approximation. Moreover, a comparison of analytical results with those obtained numerically via Bio-PEPA was provided. For this reason the focus was on symmetric models that can be handled analytically. It does not include a systematic study of the stability aspects of the model. In [5] a variant based on the socialisation level has been analysed in an analytical and numerical way. The study addresses a stability analysis of a *symmetric* model limited to three squares, but it contains no stability analysis of *asymmetric* models. Recently, the use of Bio-PEPA has also been explored in the field of swarm robotics, where it was used to model a swarm decision making strategy [16,17]. This collective decision-making strategy has been used as a benchmark for the application of stochastic process algebra, and in particular Bio-PEPA with locations, in this new field of application. It was shown that important aspects of swarm robotics can be addressed such as cooperation and space-time characteristics, but also emergent behaviour.

In the following we first introduce the crowd model and Bio-PEPA followed by a description of the tool pipeline and its application.

[1] Studied in the original work by Row and Gomez [21] and in [15].

2 A Socialisation Level Based Crowd Dynamics Model

Rowe and Gomez [21] analyse the movement of crowds between four city squares, connected in a ring by streets, using an agent based approach. The movement of individual people is simulated by agents following a simple set of rules. At every step each agent tries to find a partner to chat with. If this succeeds it stays where it is; else, it moves randomly to an adjacent square. It is assumed that the probability of the latter is $(1-c)^{p_i-1}$ when this agent is at square i, and $p_i > 0$ is the number of agents currently at square i. The parameter c (representing the *chat probability*, $0 \leq c \leq 1$) is the probability that an agent finds a partner to talk to and thus remains in the square. Rowe and Gomez showed that the emergence of crowds in their model is directly linked to a critical threshold value of the chat probability. If the value is below the threshold, the population remains evenly distributed over the squares, while walking around. If the value is above the threshold, the population tends to gather into one or a few squares. However, the probability to meet a friend in a crowded city is in general not the same as the probability to find a friend when it is less crowded. People tend to have a fixed number of friends given a city population, and the larger the number of people walking around, the more of them will turn out not to be one of your friends. This consideration leads to an alternative crowd model, introduced in [5], in which the chat probability is defined as $c = s/N$, where N is the size of the population and s is the level of socialisation of the population, i.e. the average number of friends that people have. Using this alternative definition of c and an $n \times n$ routing matrix Q for n squares, i.e. $Q_{i,j}$ is the probability that a person moves to square j, given that she decided to leave square i, the ODE for population level N of this model is:

$$\frac{dx_i}{dt} = -x_i(1-s/N)^{Nx_i-1} + \sum_j x_j(1-s/N)^{Nx_j-1}Q_{j,i} \tag{1}$$

where x_i denotes the fraction of the population that is in location i. Here we assume that Q is symmetric, i.e. $Q_{i,j} = Q_{j,i}$, so Q is a stochastic and symmetric matrix. It is further assumed that Q is irreducible (this is not a limitation since otherwise the city can be split into its connected components.) The above ODE is also the fluid flow interpretation of a Bio-PEPA model of this scenario that will be presented in Section 4 with the only difference that the latter is defined directly on the population sizes itself and not on their fractions. The basic symmetric model has also an interesting fluid limit, i.e. an ODE model resulting from letting $N \to \infty$. For N going to ∞ one obtains (see [5] for an analytical derivation):

$$\frac{dx_i}{dt} = -x_i e^{-sx_i} + \sum_j x_j e^{-sx_j} Q_{j,i} \tag{2}$$

This is a non linear ODE, that has all its solutions in the unit simplex $\Delta_n = \{x \in \mathbb{R}^n : x_i \geq 0 \text{ and } \sum_i x_i = 1\}$ if the initial condition is in Δ_n.

3 Bio-PEPA and Fluid Flow Analysis

Before presenting a process algebra model of the crowd scenario we briefly recall Bio-PEPA [7], a language that has originally been developed for the modelling and analysis of biochemical systems. The main components of a Bio-PEPA system are the *"species" components*, describing the behaviour of individual entities of a species, and the *model component*, describing the interactions between the various species. The initial amounts of each type of entity or species are given in the model component. The syntax of the Bio-PEPA components is defined as:

$$S ::= (\alpha, \kappa)\ \mathtt{op}\ S \mid S + S \mid C \text{ with } \mathtt{op} = \downarrow \mid \uparrow \mid \oplus \mid \ominus \mid \odot \text{ and } P ::= P \underset{\mathcal{L}}{\bowtie} P \mid S(x)$$

where S is a *species component* and P is a *model component*. In the prefix term $(\alpha, \kappa)\ \mathtt{op}\ S$, κ specifies the multiples of an entity of species S involved in an occurring action α[2]. The *prefix combinator* "op" represents the role of S in the action, or conversely the impact that the action has on the species. Specifically, $\downarrow$ indicates a reduction of the population involved in the action, $\uparrow$ indicates an increase as a result of the action. The operators $\oplus$, $\ominus$ and $\odot$ play a role in an action without leading to increments or decrements in the involved populations and have a defined meaning in the biochemical context. Except from $\odot$, that will play a role in the shorthand notation introduced below, we will not need these operators in this paper. The operator "+" expresses the choice between possible actions, and the constant C is defined by an equation $C{=}S$. The process $P \underset{\mathcal{L}}{\bowtie} Q$ denotes synchronisation between components P and Q, the set $\mathcal{L}$ determines those actions on which the components P and Q are forced to synchronise, with $\underset{*}{\bowtie}$ denoting a synchronisation on all common actions. In $S(x)$, the parameter $x \in \mathbb{R}$ represents the initial amount of the species. The frequency with which an action occurs is defined by its (functional) rate. This rate is the parameter of a negative exponential distribution. Its value may be a function of the size of the populations involved in the interaction.

Bio-PEPA comes with a notion of discrete locations that may contain species. A Bio-PEPA *system* with *locations* consists of a set of species components, also called sequential processes, a model component, and a context (locations, functional rates, parameters, etc.). The prefix term $(\alpha, \kappa)\ \mathtt{op}\ S@l$ is used to specify that the action is performed by S in location l. The notation $\alpha[I \to J] \odot S$ is a shorthand for the pair of interactions $(\alpha, 1)\downarrow S@I$ and $(\alpha, 1)\uparrow S@J$ that synchronise on action α. This shorthand is very convenient when modelling agents migrating from one location to another as we will see in the next section. Bio-PEPA is given an operational semantics, based on Continuous Time Markov Chains (CTMCs), and a fluid semantics, based on ordinary differential equations (ODE) [7]. The Bio-PEPA language is supported by a suite of software tools which automatically process Bio-PEPA models and generate internal representations suitable for different types of analysis [7,6]. These tools include mappings from Bio-PEPA to differential equations (supporting a fluid flow approximation), stochastic simulation models [9], and PRISM models [14].

[2] The default value of κ is 1 in which case we simply write α instead of (α, κ).

4 Bio-PEPA Crowd Model

In this section we define a Bio-PEPA specification of the crowds scenario presented in Section 2. Let us consider a small ring topology with 4 city squares in a 2×2 grid, denoting them by 00, 01, 10 and 11, allowing bi-directional movement between squares. In Bio-PEPA the city squares are modelled as locations called $sq00, sq01$, $sq10$ and $sq11$. Parameter c defines the chat-probability and parameter d the degree or number of streets connected to a square. In the symmetric topology $d = 2$ for each square. The chat-probability is defined as the fraction of the socialisation factor s w.r.t. the total population N, i.e. $c = s/N$. The actions modelling agents moving *f*rom square X *t*o square Y are denoted by $fXtY$. The associated functional rate for *f00t01* with $P@sq00$ denoting the population in $sq00$ at time t is defined as:

$$f00t01 = (P@sq00 * (1-c)^{(P@sq00-1)})/d;$$

the other rates are defined similarly. The behaviour of a typical agent moving between squares is modelled by sequential component P. For example, $f00t01[sq00 \rightarrow sq01] \odot P$ models that an agent present in $sq00$ moves to $sq01$ according to the functional rate defined for the action *f00t01*.

$$\begin{aligned} P = {} & f00t01[sq00 \rightarrow sq01] \odot P + f01t00[sq01 \rightarrow sq00] \odot P + \\ & f00t10[sq00 \rightarrow sq10] \odot P + f10t00[sq10 \rightarrow sq00] \odot P + \\ & f01t11[sq01 \rightarrow sq11] \odot P + f11t01[sq11 \rightarrow sq01] \odot P + \\ & f01t11[sq10 \rightarrow sq11] \odot P + f11t10[sq11 \rightarrow sq10] \odot P; \end{aligned}$$

Finally, the model component defines the initial conditions of the system, i.e. in which squares the agents are located initially, and the relative synchronisation pattern. If, initially, there are 1000 agents in $sq00$ this is expressed by $P@sq00[1000]$. The fact that moving agents need to synchronise follows from the definition of the shorthand operator $\rightarrow$.

$$((P@sq00[1000] \underset{*}{\bowtie} P@sq01[0]) \underset{*}{\bowtie} (P@sq10[0])) \underset{*}{\bowtie} (P@sq11[0])$$

The total number of agents $P@sq00 + P@sq10 + P@sq01 + P@sq11$ is invariant and amounts to 1000 in this specific case. The occupancy measure, i.e. the fraction of the population in $sq00$ can be defined as $Psq00 = P@sq00/N$ and similarly for the other squares. The fluid semantics of Bio-PEPA leads to an ODE that is very similar to Eq. (1) except that it is defined on the actual population sizes rather than their fractions:

$$\frac{dP@sq_i}{dt} = -P@sq_i(1-s/N)^{P@sq_i-1} + \sum_j P@sq_j(1-s/N)^{P@sq_j-1}Q_{j,i} \quad (3)$$

Using the Bio-PEPA plugin tool suite a first insight in the behaviour of the above model for different values of the socialisation factor s can be obtained using e.g. stochastic simulation [9] or one of the built-in ODE solvers. For example,

for $s = 5$ an interesting so called 'metastable' behaviour can be observed in ODE trajectories such as the one shown in Fig. 2(a), where the fractions of the population present in squares $sq00$ through $sq11$, are denoted by x_i, for $i \in \{1, .., 4\}$. Until time 150 just over 40% of the population is in each of the non-adjacent squares x_1 and x_3, and slightly less than 10% in each of the remaining squares. Then suddenly this situation changes and square x_1 gets almost all of the population. This is just one example of the typical kind of behaviours that may occur in non-linear systems such as these.

To get a more complete overview of potential emergent behaviour of a collective dynamic system it may be useful to construct a bifurcation diagram of the system. This is a diagram that shows for each value of a selected parameter of the model its stationary points for that value. For each stationary point it also shows whether it is stable or unstable, i.e. whether a system would remain in a state forever once it is reached, or whether it could still move on from there reaching other states. The selected parameter of interest in our case is the socialisation value s. Fig. 1 shows the bifurcation diagram for square x_1 of the crowds model with four squares and for s ranging from 2.75 to 6. For example, for $s = 3$ we see that the model has one stable stationary point with value 0.25. This means that for $s = 3$ the system ends up in a stable state in which 25% of the population is expected to be in square x_1.

In fact, the model has a stationary point at 0.25 for all values of s considered, i.e. the vector of four squares $\mathbf{x}_{sym} = \left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right)$ is always a stationary point, but it is not stable for all values of s; its stability changes at $s = 4$. The stability of this equilibrium has been analysed in an analytical way in [5] for a model with an arbitrary number n of squares, but with symmetric routing matrix Q. There it was proven that the stability status of this stationary point changes from stable into unstable when s becomes equal to n. It is stable for models in which $s < n$ and unstable for those in which $s > n$. However, in [5] no analytic results have been given for the other stationary points, due to the difficulty in identifying them in general. Stability of all fixed points has been discussed in [5] only for a symmetric, fully connected, model with three squares. Such difficulties are also common for models that show irregular structure or other forms of asymmetry. When investigating the behaviour of the model in more detail using stochastic simulation (not shown) it turns out that in the model with 4 squares and $s < 3.25$ the population is migrating between squares in such a way that it is evenly distributed over the squares. This means that at any point in time one would see approximately 25% of the population in each of the squares. However, this situation changes for $s > 3.25$. Stochastic simulation shows that in that case most of the population eventually gathers at random in one of the squares.

In the following we propose an analysis pipeline, starting from a Bio-PEPA specification, that can be used to compute the stationary points *numerically* instead of analytically for various values of s, such that a bifurcation diagram of the kind shown in Fig. 1 can be generated. As we can see in the figure, the change

of stability of $\mathbf{x}_{sym}$ is correctly identified by the numerical method, which also predicts other stationary points, corresponding to three different configurations:

1. Most agents in a single square. This is the top branch in the diagram. These are stable stationary points, but emerge for s greater than 3.5, approximately. For instance, for $s = 5$, in this case 98% of agents are in a single square.
2. Agents evenly split between two opposite squares. This is an unstable stationary point, emerging for $s > 4$, which is in fact a saddle node [3]. For $s = 5$, this configuration corresponds to 43% of agents in two opposite squares (for instance, $sq00$ and $sq11$), and 7% in the remaining ones. Notice that this information can be deduced by observing the numeric values of fixed points.
3. Agents evenly split between three adjacent squares. This is again an unstable equilibrium of the system. For $s = 5$, we have 29% of agents in each of the three adjacent squares and 0.13% in the remaining one.

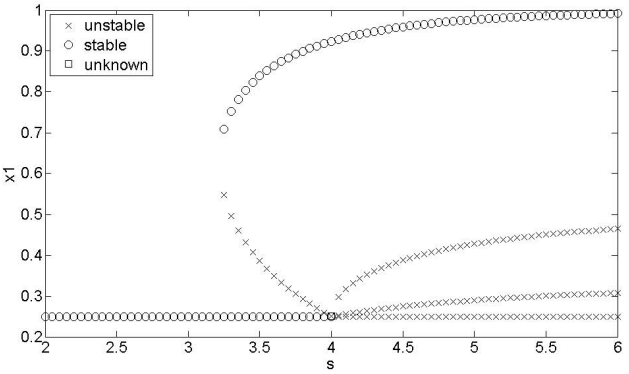
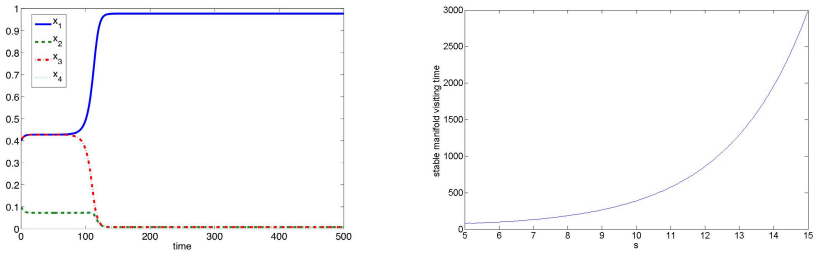


Fig. 1. Bifurcation diagram for the symmetric 4 square model

Interestingly, configuration (2) above, despite being a saddle node, hence unstable, has a quite strong stable manifold. The effect is that solutions starting nearby the stable manifold (e.g. close to the plane $x_1 = x_3$) remain for some time close to it, before escaping to one of the stable fixed points. This gives rise to the typical metastable behaviour that is shown in the ODE trajectory in Fig. 2(a). As could be expected, the higher the socialisation coefficient s, the more intense is the attractive behaviour of the stable manifold, hence the longer solutions of the ODE of the model, that start from initial values in the neighbourhood of such a point, remain attracted to it. This can be seen in Figure 2(b), where the time spent nearby the stable manifold is shown as a function of s. In Section 6 we will apply the analysis pipeline proposed in the next section to several variants

[3] Unstable stationary points come in different kinds. One of these kinds is called a 'saddle node', informally it is attracting from two opposite sides and repellent from the two other sides of the point.



(a) Metastable trajectory (b) Metastable equilibrium visiting time

Fig. 2. (2(a)) A trajectory for $s = 5$ showing metastable behaviour. The initial conditions are $x_1 = 0.4 + \delta$, $x_2 = 0.1$, $x_3 = 0.4 - \delta$, $x_4 = 0.1$, with δ small. (2(b)) Time spent close to the metastable equilibrium, as a function of s. Initial conditions are as above, with $\delta = 10^{-7}$.

of the crowd model. Each of these variants is characterised by a different form of asymmetry e.g. caused by squares with different attractivity, or by asymmetry in topology.

5 Numeric/Symbolic Stability Analysis

Before discussing the tool pipeline, we briefly sketch the methods we are using in Octave/Matlab to investigate the stability behaviour of fixed points and to generate bifurcation diagrams for different asymmetric variants of the crowd model that will be presented in next section. The idea, as discussed previously, is to combine the features of stochastic process algebras as modelling languages, Bio-PEPA specifically, with the numerical algorithms and analysis routines of a platform like Matlab [18] or Octave [1].

Assume that we have obtained an ODE function in the input format of Matlab or Octave, i.e. an m-file. Such an ODE function models the changes in the behaviour of the system over time. The idea is then to use numerical routines to look for the set of zeros of this function, i.e. points for which the system is stationary, for a fixed set of parameters (e.g. in our case for the socialisation coefficient s). More precisely, we can use the Octave/Matlab method *fsolve* which looks for zeros of non-linear equations and which incorporates different solvers (in our case we used the Levenberg-Marquardt algorithm [20]). As *fsolve* finds one zero, depending on the initial values, we run the algorithm many times to look for more zeros, starting from random initial coordinates (and from some predefined fixed points, like the unit vectors). In order to avoid considering all possible symmetric solutions of the problem, we post-process the set of zeros found in this way, to keep only one zero among those equivalent under symmetry. For instance, in a fully symmetric model with ring topology as the one in Section 4, there is a zero which puts almost all the population in one square, for each square. The post-processing then removes all but one of those zeros. Finally, for each

such stationary point, we investigate its stability by computing numerically the Jacobian matrix and its eigenvalues, according to standard methods in dynamic systems theory [22,13]. After discarding one zero eigenvalue (which is always present due to the conservation of the total number of agents), we check whether the real part of the remaining ones is positive or negative. If all eigenvalues have a negative real part (and are sufficiently far away from zero to account for numerical errors), we declare the point stable, otherwise we mark it unstable. If there is an eigenvalue too close to zero, we mark its status as unknown. In order to generate bifurcation diagrams, we perform this operation for a range of values of the parameter of interest, e.g. in this case the socialisation coefficient s.

Tool Pipeline. In order to link the above described procedure to Bio-PEPA, we export the model in the SBML format [4]. Such an export is already available via the Bio-PEPA plugin tool suite [6]. Both Matlab and Octave have a toolbox importing from SBML files [12], generating an m-file computing the vector field (ODE) corresponding to the fluid semantics of Bio-PEPA [7]. Once an m-file has been obtained, it can be used within the routines explained above by creating a function handle. However, following the above procedure, we have not yet obtained the limit model discussed in Section 2, as from Bio-PEPA we export the N-dependent model (see Sect. 4). Even if the two sets of ODE will produce very similar solutions when N is large enough, working with the limit model would be preferable, as this seems to reduce the numerical errors caused by the exponentiation. A limit model can be obtained by exploiting a computer algebra system such as the symbolic toolbox in Matlab [2] or the open source software Maxima [3]. This requires that the Bio-PEPA specification is exported to an m-file containing a symbolic definition of the ODE equations in Matlab or an equivalent format suitable for use with Maxima. This is not very difficult to automatise. Once this operation is performed, the symbolic calculus routines can be exploited to compute the limit of the vector field and to compute the Jacobian matrix symbolically, increasing the precision of the method and speeding up the numerical analysis phase. The resulting functions can be either exported from Maxima to Matlab/Octave by a suitable script that generates an appropriate m-file, or by converting a symbolic function into a numerical one in Matlab.

6 Results

In this section we present some results for variants of the basic crowd model enriching it either by adding new behaviours or by breaking the symmetry between squares. Due to space limitations, we will mainly report on results for a model with four squares connected in a ring topology. However, we have also obtained interesting results for a larger a model with 9 squares disposed in a 3 $\times$ 3 grid-like topology. The modifications in the basic model are essentially of four types:

- Breaking the symmetry between squares (in a symmetric topology like the ring one) by assigning to each square an uneven attractiveness coefficient. Each agent then chooses the next square to go according to their relative attractiveness.
- Breaking the symmetry in the routing, by assuming that certain connections between squares can be crossed only in a given direction, i.e. by introducing one way pedestrian streets (which can be enforced for instance by the presence of police).
- Breaking the symmetry in the topology, by having different degrees of connection between different squares, e.g. the 9 squares model (not shown).
- Breaking the symmetry in the behaviour of agents, assuming that an agent first decides if she wants either to leave the square or look for somebody to chat to. In this second case, we assume that she leaves the square with the same probability as in the standard model.

In particular, in each of these cases, we will discuss the stability of stationary points as a function of the socialisation parameter s, or of other parameters, like attractiveness or the probability to leave.

As discussed in Section 5, the results always consider the limit fluid model (see Eq.(2)), for a population going to infinity. Due to convergence of vector fields, the results for the limit model can be expected to be the same as those of the ODE with explicit dependence on N, if N is reasonably large. In particular, we have seen basically no differences between results for the limit model and those for a model with a total population of $N = 1000$.

4 Square Model with Attractive Squares. The first kind of asymmetric model we consider is one in which an attractiveness coefficient is assigned to each square. Higher attractiveness is modelled by a higher value of the coefficient. When agents decide to leave a square, their decision to which of the adjacent squares to go is now proportional to the relative attractiveness of the adjacent squares. In particular, we consider a situation in which only one square (by convention, square $sq00$), has a higher attractiveness than the others. Hence, the attractiveness coefficient of square $sq00$ is equal to $a \geq 1$, while that of the other squares is set to 1. In Bio-PEPA this is modeled by replacing the transition rates in the symmetric model of Sect. 4 by the following ones, assuming attractiveness coefficients $a00$ for square $sq00$, $a01$ for square $sq01$ and so on, e.g.:

$$f00t01 = (P@sq00 * (1 - c)^{(P@sq00-1)}) * (a01/(a01 + a10));$$

In Fig. 3 we show a bifurcation diagram as a function of the socialisation factor s, for the fraction of people in the first square, x_1, for a model with $a = 3$. As we can see, the solution in which the more attractive $sq00$ gets the larger amount of people is always stable. However, for s around 3.5, a new set of stationary points appears, with a stable and an unstable branch. The stable branch corresponds to situations in which most of the people stay in one of the two squares adjacent to the attractive one. This stable equilibrium is quite surprising, as one would

expect that people always move towards the attractive square. Note that this happens for $s < 4$, i.e. the predicted threshold for s in the symmetric model.

The other stable equilibrium that emerges around $s = 4.5$ corresponds to the situation in which all agents are in the square opposite to the attractive one. Hence, even in the presence of asymmetric attractiveness, we may still obtain a pattern of emergent behaviour in which all people are gathered at a random single square. So, contrary to intuition, for sufficiently large values of s, people may concentrate also in non-attractive squares. Furthermore, as expected, we loose the symmetric equilibrium in which agents are uniformly distributed over the squares and find no replacement for it.

4 Square Model with One-Way Streets. In this model we assume that streets between squares are unidirectional rather than bi-directional. In particular, we assume that the ring can be traversed only clockwise, with no square having a higher attractiveness than others. This model is still ergodic, and moreover the four squares still behave symmetrically. However, the routing matrix is no longer symmetric. In the Bio-PEPA model this can be easily obtained by removing the related directions of movement and their corresponding rate definitions.

The pattern of the bifurcation diagram we obtain is very similar to the one for the symmetric model in Fig. 1. Indeed, the only difference is that now some eigenvalues of the Jacobian in steady states have imaginary values, suggesting that we may in fact have stable and unstable foci rather than single nodes [22], i.e. convergence to the fixed point happens by damped oscillations. In any case, this effect is very weak, and cannot be observed at the resolution scale at which we plot trajectories.

4 Square Model with Independent Leaving Probability. In this model we change the behaviour of single agents. In particular, we assume that each agent first chooses if it wants to leave the square (with a "boredom" probability p) or

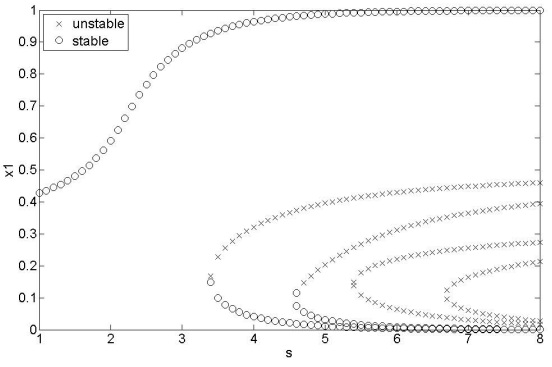


Fig. 3. Bifurcation diagram for the 4 square model with asymmetric attractiveness, as a function of s, holding a fixed to 3

look for another one to chat with. In the latter case, it behaves like in the original model. In Bio-PEPA this is modeled by replacing the transition rates in the model with attractiveness by:

$$f00t01 = (1-p) * (P@sq00 * (1-c)^{(P@sq00-1)}) * (a01/(a01+a10)) + p * (P@sq00);$$

Note that in case all squares have an attractiveness coefficient equal to 1 we can study the effect of the boredom probability separately from the effect of attractiveness of squares. This is what is assumed in the following.

The model with independent leaving probability has a second parameter in addition to s, namely the boredom probability p. Intuitively, this probability should influence the overall behaviour of the model quite radically. If it is large enough (maybe in case of a meeting of eremites), then we do not expect people to gather in a single square, but rather to see them uniformly distributed over the four squares while moving between them. This indeed happens for $p = 0.15$ and for any value of s considered. On the other hand, for small values of p, the behaviour manifested by the system becomes much more complicated, as shown in Fig. 4. In this case, for $p = 0.05$, we can observe a frequent change in the stability status of fixed points, due to bifurcation events in which stationary points split, change stability status, and move towards other fixed points, generating a cascade of bifurcations.

The most interesting feature is the stability of the symmetric equilibrium. Initially it is stable, but then, as s increases, it undergoes a bifurcation event and becomes unstable (for s around 5 here), as in the original model. At the same time, a new stable branch emerges, corresponding to the emergent behaviour in which most people are concentrated in a single square. However, for s around 12, this stable behaviour undergoes a new bifurcation event and becomes unstable. Furthermore, as s increases further, this unstable equilibrium approaches the symmetric one and hits it around $s = 16$. When this happens, the unstable symmetric equilibrium becomes stable again, and as s increases even further, it becomes the only (stable) fixed point. This is a counter-intuitive behaviour of the model: as the socialisation factor increases, instead of obtaining a higher probability of people concentrating in a single square, exactly the opposite effect emerges.

The assumption that agents can leave a square with a small fixed probability, no matter whether friends are present or not, is quite reasonable. For example one may receive a text message from a friend in another square or have other things to do. The behaviour of this model shows that, when the socialisation coefficient is large enough (in reality, for a population of a few thousand people, we can expect a value of s well above 30), the chat probability mechanism is not able to fully explain the emergent behaviour of 'El Botellon'. Other mechanisms have to be taken into account and also their potential interference. Among these are most likely some asymmetry breaking phenomena such as the different degrees of attractiveness of the squares.

To obtain deeper insight in why we observe the effect shown in Fig. 4, we can compare the total exit rate of an agent from a square for the symmetric model and the model with boredom probabilities. In particular, we compare these rates as a function of s, for a fixed fraction α of the total population

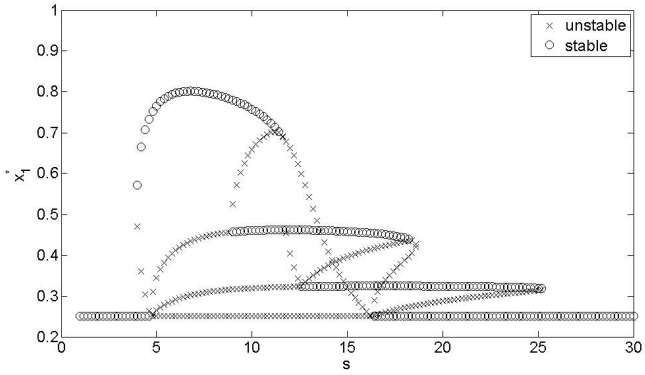


Fig. 4. Bifurcation diagram for the 4 square model with boredom probability $p = 0.05$

in the square. In the symmetric case, we obtain $e^{-\alpha s}$ (see Eq.(2)), which is a decreasing function of s, exponentially approaching zero for large values of s. Hence, for a sufficiently large value of s, the rate of leaving a square becomes very small: if there are many agents in the square (α close to 1), then we can expect nobody will move, as the exit rate from the crowded square will be much smaller than the exit rate of the other squares. On the other hand, for the model with boredom probability, the probability at which a *single* agent leaves a square is $p + (1 - p)e^{-\alpha s}$, which for large s converges to p. This means that for large s, the effect of the chat probability to remain in a square is negligible compared to the effect of the boredom probability with which agents leave a square. This, in turn, implies that the ODE we obtain for large s is essentially a set of linear ODE with a small non-linear perturbation term, hence they converge to the unique equilibrium of the linear system, which is the symmetric one in which the population is uniformly distributed over the squares.

In order to break this effect, we may want to find out whether the introduction of asymmetric attractiveness can counterbalance the disruptive effect of boredom probability on the emergent behaviour of the model. Fig. 6 (left) shows the bifurcation diagram for the model which $sq00$ is three times as attractive as the other squares, $a = 3$, and boredom probability with $p = 0.05$. We observe the same pattern as in the symmetric model: for large enough values of s, the system converges to the perturbed symmetric equilibrium (due to asymmetric attractiveness). An additional variation can be to consider a boredom probability which is inversely proportional to the attractiveness of squares, for instance equal to p/a^2, in order to take into account the effect that it is more unlikely that people just leave a square where interesting events are going on. In this case, in the presence of the compensating effect of boredom probability for large s, we can enlarge the range of s for which we observe an emergent party (see Fig. 6 (right)). So, a combination of effects of the attractiveness on the choice of the next square and on the boredom probability can still be used to explain the emergent behaviour of 'El Botellon'.

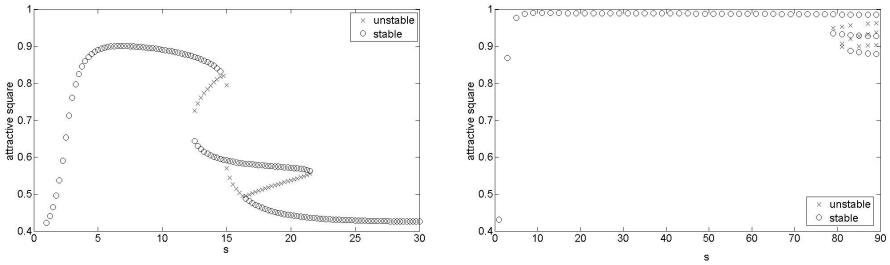


Fig. 5. Bifurcation diagram for the attractive square in 4 square model with boredom probability. for $p = 0.05$ and attractiveness equal to 3 (left), and attractiveness dependent p (right).

7 Discussion and Further Work

The engineering of large scale collective dynamic systems is a relatively new domain of software engineering and requires effective and scalable formal analysis methods. We illustrated how the exploitation of the combination of process algebras, designed specifically for concurrent systems, and techniques typical of dynamic systems analysis, such as stability analysis, could provide valuable tools for such engineering approaches. In this paper we proposed a tool pipeline that, starting from a Bio-PEPA specification of non-linear asymmetric collective coordination, can produce bifurcation diagrams. These can be used to analyse the effect of potentially interfering coordination mechanisms on the stability properties of a system. We have illustrated the combined numeric and symbolic approach on a number of variants of a model of crowd dynamics that represents various kinds of *asymmetry*. The automatisation of the approach is feasible and part of future work. An issue that needs further investigation is the scalability of the method. Fluid ODEs are independent on the population size, but not on the number of states of the agents. Finding numerically all the zeros of a vector field can be challenging in large dimensions. One approach would be to parallelise the code and use more efficient zero finding numerical routines. An alternative and more promising direction is to exploit the formal nature of process algebras to reduce the agent's state space by using behavioural equivalences or abstract interpretation. Furthermore we plan to integrate the approach with the other already available analysis tools for Bio-PEPA, such as fluid flow analysis and stochastic simulation, via the Bio-PEPA tool suite.

Acknowledgments. This research has been partially funded by the EU-IP project ASCENS (nr. 257414), the EU-FET project QUANTICOL (nr. 600708) and the Italian MIUR-PRIN project CINA.

References

1. GNU Octave, http://www.gnu.org/software/octave/
2. Matlab symbolic toolbox, http://www.mathworks.it/products/symbolic/
3. Maxima, a computer algebra system, http://maxima.sourceforge.net/
4. SBML: Systems Biology Markup Language, http://sbml.org
5. Bortolussi, L., Le Boudec, J.Y., Latella, D., Massink, M.: Revisiting the limit behaviour of El Botellon. Tech. Rep. EPFL-REPORT-179935, École Polytechnique Fédérale de Lausanne - INFOSCIENCE (July 2012)
6. Ciocchetta, F., Duguid, A., Gilmore, S., Guerriero, M.L., Hillston, J.: The Bio-PEPA Tool Suite. In: Proc. of the 6th Int. Conf. on Quantitative Evaluation of SysTems (QEST 2009), pp. 309–310 (2009)
7. Ciocchetta, F., Hillston, J.: Bio-PEPA: A framework for the modelling and analysis of biological systems. TCS 410(33-34), 3065–3084 (2009)
8. Frei, R., Di Marzo Serugendo, G.: Advances in complexity engineering. International Journal of Bio-Inspired Computation 3, 199–212 (2011)
9. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. The Journal of Physical Chemistry 81(25), 2340–2361 (1977)
10. Hillston, J.: A compositional approach to performance modelling, distinguished Dissertation in Computer Science. Cambridge University Press (1996)
11. Hillston, J.: Fluid flow approximation of PEPA models. In: Proceedings of QEST 2005, pp. 33–43. IEEE Computer Society (2005)
12. Keating, S.M., Bornstein, B.J., Finney, A., Hucka, M.: SBMLToolbox: an SBML toolbox for MATLAB users. Bioinformatics 22(10), 1275–1277 (2006), http://sbml.org/Software/SBMLToolbox
13. Khalil, H.K.: Nonlinear systems. MacMillan Pub. Co. (1992)
14. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic model checking for performance and reliability analysis. ACM SIGMETRICS Performance Evaluation Review (2009)
15. Massink, M., Latella, D., Bracciali, A., Hillston, J.: Modelling non-linear crowd dynamics in Bio-PEPA. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 96–110. Springer, Heidelberg (2011)
16. Massink, M., Brambilla, M., Latella, D., Dorigo, M., Birattari, M.: Analysing robot swarm decision-making with Bio-PEPA. In: Dorigo, M., Birattari, M., Blum, C., Christensen, A.L., Engelbrecht, A.P., Groß, R., Stützle, T. (eds.) ANTS 2012. LNCS, vol. 7461, pp. 25–36. Springer, Heidelberg (2012)
17. Massink, M., Latella, D.: Fluid analysis of foraging ants. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 152–165. Springer, Heidelberg (2012)
18. MATLAB: v. 7.10.0 (R2010a). The MathWorks Inc., Natick(2010)
19. Naphade, M., Banavar, G., Harrison, C., Paraszczak, J., Morris, R.: Smarter cities and their innovation challenges. Computer 44(6), 32–39 (2011)
20. Nocedal, J., Wright, S.J.: Numerical Optimization, 2nd edn. Springer (2006)
21. Rowe, J.E., Gomez, R.: El Botellón: Modeling the movement of crowds in a city. Complex Systems 14, 363–370 (2003)
22. Strogatz, S.H.: Non-linear dynamics and chaos: with applications to physics, biology, chemistry, and engineering. Perseus Books Publishing (1994)

Modelling MAC-Layer Communications in Wireless Systems
(Extended Abstract)

Andrea Cerone[1], Matthew Hennessy[1,⋆], and Massimo Merro[2,⋆⋆]

[1] Department of Computer Science and Statistics, Trinity College Dublin, Ireland
{acerone,Matthew.Hennessy}@scss.tcd.ie
[2] Dipartimento di Informatica,
Università degli Studi di Verona, Italy
massimo.merro@univr.it

Abstract. We present a timed broadcast process calculus for wireless networks at the MAC-sublayer where time-dependent communications are exposed to collisions. We define a reduction semantics for our calculus which leads to a contextual equivalence for comparing the external behaviour of wireless networks. Further, we construct an extensional LTS (labelled transition system) which models the activities of stations that can be directly observed by the external environment. Standard bisimulations in this novel LTS provide a sound proof method for proving that two systems are contextually equivalent. In addition, the main contribution of the paper is that our proof technique is also complete for a large class of systems.

1 Introduction

Wireless networks are becoming increasingly pervasive with applications across many domains, [19,1]. They are also becoming increasingly complex, with their behaviour depending on ever more sophisticated protocols. There are different levels of abstraction at which these can be defined and implemented, from the very basic level in which the communication primitives consist of sending and receiving electromagnetic signals, to the higher level where the basic primitives allow the set up of connections and exchange of data between two nodes in a wireless system [23].

Assuring the correctness of the behaviour of a wireless network has always been difficult. Several approaches have been proposed to address this issue for networks described at a high level [16,13,6,5,22,11,2,3]; these typically allow the formal description of protocols at the *network layer* of the *TCP/IP* reference model [23]. However there are few frameworks in the literature which consider networks described at the MAC-Sublayer of the *TCP/IP* reference model [12,14]. This is the topic of the current paper. We propose a process calculus for describing and verifying wireless networks at the *MAC-Sublayer* of the *TCP/IP* reference model.

⋆ Supported by SFI project SFI 06 IN.1 1898.
⋆⋆ Author partially supported by the PRIN 2010-2011 national project “Security Horizons”.

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 16–30, 2013.

This calculus, called the Calculus of Collision-prone Communicating Processes (CCCP), has been largely inspired by TCWS [14]; in particular CCCP inherits its communication features but simplifies considerably the syntax, the reduction semantics, the notion of observation, and as we will see the behavioural theory. In CCCP a wireless system is considered to be a collection of wireless stations which transmit and receive messages. The transmission of messages is broadcast, and it is time-consuming; the transmission of a message v can require several time slots (or instants). In addition, wireless stations in our calculus are sensitive to collisions; if two different stations are transmitting a value over a channel c at the same time slot a collision occurs, and the content of the messages originally being transmitted is lost.

More specifically, in CCCP a state of a wireless network (or simply network, or system) will be described by a *configuration* of the form $\Gamma \triangleright W$ where W describes the code running at individual wireless stations and Γ represents the communication state of channels. At any given point of time there will be *exposed* communication channels, that is channels containing messages (or values) in transmission; this information will be recorded in Γ.

Such systems evolve by the broadcast of messages between stations, the passage of time, or some other internal activity, such as the occurrence of collisions and their consequences. One of the topics of the paper is to capture formally these complex evolutions, by defining a *reduction semantics*, whose judgments take the form $\Gamma_1 \triangleright W_1 \twoheadrightarrow \Gamma_2 \triangleright W_2$. The reduction semantics satisfies some desirable properties such as *time determinism*, *patience* and *maximal progress* [17,9,25].

However the main aim of the paper is to develop a behavioural theory of wireless networks. To this end we need a formal notion of when two such systems are indistinguishable from the point of view of users. Having a reduction semantics it is now straightforward to adapt a standard notion of *contextual equivalence*: $\Gamma_1 \triangleright W_1 \simeq \Gamma_2 \triangleright W_2$. Intuitively this means that either system, $\Gamma_1 \triangleright W_1$ or $\Gamma_2 \triangleright W_2$, can be replaced by the other in a larger system without changing the observable behaviour of the overall system. Formally we use the approach of [10], often called *reduction barbed congruence*; the only parameter in the definition is the choice of primitive observation or *barb*. Our choice is natural for wireless systems: the ability to transmit on an idle channel, that is a channel with no active transmissions.

As explained in papers such as [20,7], contextual equivalences are determined by so-called *extensional actions*, that is the set of minimal observable interactions which a system can have with its external environment. For CCCP determining these actions is non-trivial. Although values can be transmitted and received on channels, the presence of collisions means that these are not necessarily observable. In fact the important point is not the transmission of a value, but its successful delivery. Also, although the basic notion of observation on systems does not involve the recording of the passage of time, this has to be taken into account extensionally in order to gain a proper extensional account of systems.

The extensional semantics determines an LTS (labelled transition system) over configurations, which in turn gives rise to the standard notion of (weak) bisimulation equivalence between configurations. This gives a powerful co-inductive proof technique: to show that two systems are behaviourally equivalent it is sufficient to exhibit a witness bisimulation which contains them.

One result of this paper is that weak bisimulation in the extensional LTS is sound with respect to the touchstone contextual equivalence: if two systems are related by some bisimulation in the extensional LTS then they are contextually equivalent. However, the main contribution is that completeness holds for a large class of networks, called *well-formed*. If two such networks are contextually equivalent then there is some bisimulation, based on our novel extensional actions, which contains them. In [14], a sound but not complete bisimulation based proof method is developed for (a different form of) reduction barbed congruence. Here, by simplifying the calculus and isolating novel extensional actions we obtain both soundness and completeness.

The rest of the paper is organised as follows: in Section 2 we define the syntax which we will use for modelling wireless networks. The reduction semantics is given in Section 3 from which we develop in the same section our notion of reduction barbed congruence. In Section 4 we define the extensional semantics of networks, and the (weak) bisimulation equivalence it induces. In Section 5 we state the main results of the paper, namely that bisimulation is sound with respect to barbed congruence and, for a large class of systems, it is also complete. Detailed proofs of the results can be found in the associated technical report [4]. The latter also contains an initial case study showing the usefulness of our proof technique. Two particular instances of networks are compared; the first forwards two messages to the external environment using a TDMA modulation technique, the second performs the same task by routing the messages along different stations.

2 The Calculus

Formally we assume a set of channels **Ch**, ranged over by $c, d, \cdots$, and a set of values **Val**, which contains a set of data-variables, ranged over by $x, y, \cdots$ and a special value err; this value will be used to denote faulty transmissions. The set of *closed values*, that is those not containing occurrences of variables, are ranged over by $v, w, \cdots$. We also assume that every closed value $v \in \mathbf{Val}$ has an associated strictly positive integer δ_v, which denotes the number of time slots needed by a wireless station to transmit v.

A channel environment is a mapping $\Gamma : \mathbf{Ch} \to \mathbb{N} \times \mathbf{Val}$. In a configuration $\Gamma \triangleright W$ where $\Gamma(c) = (n, v)$ for some channel c, a wireless station is currently transmitting the value v for the next n time slots. We will use some suggestive notation for channel environments: $\Gamma \vdash_t c : n$ in place of $\Gamma(c) = (n, w)$ for some w, $\Gamma \vdash_v c : w$ in place of $\Gamma(c) = (n, w)$ for some n. If $\Gamma \vdash_t c : 0$ we say that channel c is idle in Γ, and we denote it with $\Gamma \vdash c : \mathbf{idle}$. Otherwise we say that c is exposed in Γ, denoted by $\Gamma \vdash c : \mathbf{exp}$. The channel environment Γ such that $\Gamma \vdash c : \mathbf{idle}$ for every channel c is said to be *stable*.

The syntax for system terms W is given in Table 1, where P ranges over code for programming individual stations, which is also explained in Table 1. A system term W is a collection of individual threads running in parallel, with possibly some channels restricted. Each thread may be either an inactive piece of code P or an active code of the form $c[x].P$. This latter term represents a wireless station which is receiving a value from the channel c; when the value is eventually received the variable x will be replaced with the received value in the code P. The restriction operator $\nu c : (n, v).W$ is non-standard, for a restricted channel has a positive integer and a closed value associated with it; roughly speaking, the term $\nu c : (n, v).W$ corresponds to the term W where

Table 1. CCCP: Syntax

W	$::= P$	station code
	$\mid c[x].P$	active receiver
	$\mid W_1 \mid W_2$	parallel composition
	$\mid \nu c{:}(n, v).W$	channel restriction
P, Q	$::= c\,!\langle u\rangle.P$	broadcast
	$\mid \lfloor c?(x).P \rfloor Q$	receiver with timeout
	$\mid \sigma.P$	delay
	$\mid \tau.P$	internal activity
	$\mid P + Q$	choice
	$\mid [b]P, Q$	matching
	$\mid X$	process variable
	$\mid \mathsf{nil}$	termination
	$\mid \mathsf{fix}\, X.P$	recursion

Channel Environment: $\Gamma : \mathbf{Ch} \to \mathbb{N} \times \mathbf{Val}$

channel c is local to W, and the transmission of value v over channel c will take place for the next n slots of time.

The syntax for station code is based on standard process calculus constructs. The main constructs are time-dependent reception from a channel $\lfloor c?(x).P \rfloor Q$, explicit time delay $\sigma.P$, and broadcast along a channel $c\,!\langle u\rangle.P$. Here u denotes either a data-variable or closed value $v \in \mathbf{Val}$. Of the remaining standard constructs the most notable is matching, $[b]P, Q$ which branches to P or Q, depending on the value of the Boolean expression b. We leave the language of Boolean expressions unspecified, other than saying that it should contain equality tests for values, $u_1 = u_2$. More importantly, it

Table 2. Intensional semantics: transmission

$$\text{(Snd)}\ \frac{}{\Gamma \triangleright c\,!\langle v\rangle.P \xrightarrow{c!v} \sigma^{\delta_v}.P} \qquad \text{(Rcv)}\ \frac{\Gamma \vdash c : \mathbf{idle}}{\Gamma \triangleright \lfloor c?(x).P \rfloor Q \xrightarrow{c?v} c[x].P}$$

$$\text{(RcvIgn)}\ \frac{\neg \mathsf{rcv}(W, c)}{\Gamma \triangleright W \xrightarrow{c?v} W} \qquad \text{(Sync)}\ \frac{\Gamma \triangleright W_1 \xrightarrow{c!v} W_1' \quad \Gamma \triangleright W_2 \xrightarrow{c?v} W_2'}{\Gamma \triangleright W_1 \mid W_2 \xrightarrow{c!v} W_1' \mid W_2'}$$

$$\text{(RcvPar)}\ \frac{\Gamma \triangleright W_1 \xrightarrow{c?v} W_1' \quad \Gamma \triangleright W_2 \xrightarrow{c?v} W_2'}{\Gamma \triangleright W_1 \mid W_2 \xrightarrow{c?v} W_1' \mid W_2'}$$

should also contain the expression $\mathrm{exp}(c)$ for checking if in the current configuration the channel c is exposed, that is it is being used for transmission.

In the construct $\mathsf{fix}\, X.P$ occurrences of the recursion variable X in P are bound; similarly in the terms $\lfloor c?(x).P \rfloor Q$ and $c[x].P$ the data-variable x is bound in P. This gives rise to the standard notions of free and bound variables, α-conversion and capture-avoiding substitution; we assume that all occurrences of variables in system terms are bound and we identify systems up to α-conversion. Moreover we assume that all occurrences of recursion variables are *guarded*; they must occur within either a broadcast, input or time delay prefix, or within an execution branch of a matching construct. We will also omit trailing occurrences of nil, and write $\lfloor c?(x).P \rfloor$ in place of $\lfloor c?(x).P \rfloor \mathsf{nil}$.

Our notion of wireless networks is captured by pairs of the form $\Gamma \triangleright W$, which represent the system term W running in the channel environment Γ. Such pairs are called configurations, and are ranged over by the metavariable C.

3 Reduction Semantics and Contextual Equivalence

The reduction semantics is defined incrementally. We first define the evolution of system terms with respect to a channel environment Γ via a set of SOS rules whose judgments take the form $\Gamma \triangleright W_1 \xrightarrow{\lambda} W_2$. Here λ can take the form $c!v$ denoting a broadcast of value v along channel c, $c?v$ denoting an input of value v being broadcast along channel c, τ denoting an internal activity, or σ, denoting the passage of time. However these actions will also have an effect on the channel environment, which we first describe, using a functional $\mathrm{upd}_\lambda(\cdot) : \mathbf{Env} \to \mathbf{Env}$, where $\mathbf{Env}$ is the set of channel environments.

The channel environment $\mathrm{upd}_\lambda(\Gamma)$ describes the update of the channel environment Γ when the action λ is performed, is defined as follows: for $\lambda = \sigma$ we let

$$\mathrm{upd}_\sigma(\Gamma) \vdash_t c : (n-1) \text{ whenever } \Gamma \vdash_t c : n, \qquad \mathrm{upd}_\sigma(\Gamma) \vdash_v c : w \text{ whenever } \Gamma \vdash_v c : w.$$

For $\lambda = c!v$ we let $\mathrm{upd}_{c!v}(\Gamma)$ be the channel environment such that

$$\mathrm{upd}_{c!v}(\Gamma) \vdash_t c : \begin{cases} \delta_v & \text{if } \Gamma \vdash c : \mathbf{idle} \\ \max(\delta_v, k) & \text{if } \Gamma \vdash c : \mathbf{exp} \end{cases} \qquad \mathrm{upd}_{c!v}(\Gamma) \vdash_v c : \begin{cases} v & \text{if } \Gamma \vdash c : \mathbf{idle} \\ \mathsf{err} & \text{if } \Gamma \vdash c : \mathbf{exp} \end{cases}$$

where $\Gamma \vdash_t c : k$. Finally, we let $\mathrm{upd}_{c?v}(\Gamma) = \mathrm{upd}_{c!v}(\Gamma)$ and $\mathrm{upd}_\tau(\Gamma) = \Gamma$.

Let us describe the intuitive meaning of this definition. When time passes, the time of exposure of each channel decreases by one time unit[1]. The predicates $\mathrm{upd}_{c!v}(\Gamma)$ and $\mathrm{upd}_{c?v}(\Gamma)$ model how collisions are handled in our calculus. When a station begins broadcasting a value v over a channel c this channel becomes exposed for the amount of time required to transmit v, that is δ_v. If the channel is not free a collision happens. As a consequence, the value that will be received by a receiving station, when all transmissions over channel c terminate, is the error value err, and the exposure time is adjusted accordingly.

For the sake of clarity, the inference rules for the evolution of system terms, $\Gamma \triangleright W_1 \xrightarrow{\lambda} W_2$, are split in four tables, each one focusing on a particular form of activity.

[1] For convenience we assume $0 - 1$ to be 0.

Table 2 contains the rules governing transmission. Rule (Snd) models a non-blocking broadcast of message v along channel c. A transmission can fire at any time, independently on the state of the network; the notation σ^{δ_v} represents the time delay operator σ iterated δ_v times. So when the process $c\,!\langle v\rangle.P$ broadcasts it has to wait δ_v time units before the residual P can continue. On the other hand, reception of a message by a time-guarded listener $\lfloor c?(x).P\rfloor Q$ depends on the state of the channel environment. If the channel c is free then rule (Rcv) indicates that reception can start and the listener evolves into the active receiver $c[x].P$.

The rule (RcvIgn) says that if a system can not receive on the channel c then any transmission along it is ignored. Intuitively, the predicate $\mathsf{rcv}(W, c)$ means that W contains among its parallel components at least one non-guarded receiver of the form $\lfloor c?(x).P\rfloor Q$ which is actively awaiting a message. Formally, the predicate $\mathsf{rcv}(W, c)$ is the least predicate such that $\mathsf{rcv}(\lfloor c?(x).P\rfloor Q, c) = \text{true}$ and which satisfies the equations $\mathsf{rcv}(P + Q, c) = \mathsf{rcv}(P, c) \vee \mathsf{rcv}(Q, c)$, $\mathsf{rcv}(W_1 \mid W_2, c) = \mathsf{rcv}(W_1, c) \vee \mathsf{rcv}(W_2, c)$ and $\mathsf{rcv}(\nu d.W, c) = \mathsf{rcv}(W, c)$ if $d \neq c$. The remaining two rules in Table 2 (Sync) and (RcvPar) serve to synchronise parallel stations on the same transmission [8,17,18].

Example 1 (Transmission). Let $C_0 = \Gamma_0 \triangleright W_0$, where $\Gamma_0 \vdash c, d : \mathbf{idle}$ and $W_0 = c!\langle v_0\rangle \mid \lfloor d?(x).\mathsf{nil}\rfloor(\lfloor c?(x).Q\rfloor) \mid \lfloor c?(x).P\rfloor$ where $\delta_{v_0} = 2$.

Using rule (Snd) we can infer $\Gamma_0 \triangleright c!\langle v_0\rangle \xrightarrow{c!v_0} \sigma^2$; this station starts transmitting the value v_0 along channel c. Rule (RcvIgn) can be used to derive the transition $\Gamma_0 \triangleright \lfloor d?(x).\mathsf{nil}\rfloor(\lfloor c?(x).Q\rfloor) \xrightarrow{c?v_0} \lfloor d?(x).\mathsf{nil}\rfloor(\lfloor c?(x).Q\rfloor)$, in which the broadcast of value v_0 along channel c is ignored. On the other hand, Rule (RcvIgn) cannot be applied to the configuration $\Gamma_0 \triangleright \lfloor c?(x).P\rfloor$, since this station is waiting to receive a value on channel c; however we can derive the transition $\Gamma_0 \triangleright \lfloor c?(x).P\rfloor \xrightarrow{c?v_0} c[x].P$ using Rule (Rcv).

We can put the three transitions derived above together using rule (Sync), leading to the transition $C_0 \xrightarrow{c!v} W_1$, where $W_1 = \sigma^2 \mid \lfloor d?(x).\mathsf{nil}\rfloor(\lfloor c?(x).Q\rfloor) \mid c[x].P$. □

The transitions for modelling the passage of time, $\Gamma \triangleright W \xrightarrow{\sigma} W'$, are given in Table 3. In the rules (ActRcv) and (EndRcv) we see that the active receiver $c[x].P$ continues to wait for the transmitted value to make its way through the network; when the allocated transmission time elapses the value is then delivered and the receiver evolves to $\{^w/_x\}P$. The rule (SumTime) is necessary to ensure that the passage of time does not resolve non-deterministic choices. Finally (Timeout) implements the idea that $\lfloor c?(x).P\rfloor Q$ is a time-guarded receptor; when time passes it evolves into the alternative Q. However this only happens if the channel c is not exposed. What happens if it is exposed is explained later in Table 4. Finally, Rule (TimePar) models how σ-actions are derived for collections of threads.

Example 2 (Passage of Time). Let $C_1 = \Gamma_1 \triangleright W_1$, where $\Gamma_1(c) = (2, v_0), \Gamma_1 \vdash d : \mathbf{idle}$ and W_1 is the system term derived in Example 1.

We show how a σ-action can be derived for this configuration. First note that $\Gamma_1 \triangleright \sigma^2 \xrightarrow{\sigma} \sigma$; this transition can be derived using Rule (Sleep). Since d is idle in Γ_1, we can apply Rule (TimeOut) to infer the transition $\Gamma_1 \triangleright \lfloor d?(x).\mathsf{nil}\rfloor(\lfloor c?(x).Q\rfloor) \xrightarrow{\sigma} \lfloor c?(x).Q\rfloor$; time passed before a value could be broadcast along channel d, causing a timeout in the

Table 3. Intensional semantics: timed transitions

$$\text{(TimeNil)}\ \frac{}{\Gamma \triangleright \mathsf{nil} \xrightarrow{\sigma} \mathsf{nil}} \qquad \text{(Sleep)}\ \frac{}{\Gamma \triangleright \sigma.P \xrightarrow{\sigma} P}$$

$$\text{(ActRcv)}\ \frac{\Gamma \vdash_t c : n,\ n > 1}{\Gamma \triangleright c[x].P \xrightarrow{\sigma} c[x].P} \qquad \text{(EndRcv)}\ \frac{\Gamma \vdash_t c : 1,\ \Gamma \vdash_v c : w}{\Gamma \triangleright c[x].P \xrightarrow{\sigma} \{^{w}/_{x}\}P}$$

$$\text{(SumTime)}\ \frac{\Gamma \triangleright P \xrightarrow{\sigma} P' \quad \Gamma \triangleright Q \xrightarrow{\sigma} Q'}{\Gamma \triangleright P + Q \xrightarrow{\sigma} \Gamma' \triangleright P' + Q'} \qquad \text{(Timeout)}\ \frac{\Gamma \vdash c : \mathbf{idle}}{\Gamma \triangleright \lfloor c?(x).P \rfloor Q \xrightarrow{\sigma} Q}$$

$$\text{(TimePar)}\ \frac{\Gamma \triangleright W_1 \xrightarrow{\sigma} W_1' \quad \Gamma \triangleright W_2 \xrightarrow{\sigma} W_2'}{\Gamma \triangleright W_1 \mid W_2 \xrightarrow{\sigma} W_1' \mid W_2'}$$

Table 4. Intensional semantics: internal activity

$$\text{(RcvLate)}\ \frac{\Gamma \vdash c : \mathbf{exp}}{\Gamma \triangleright \lfloor c?(x).P \rfloor Q \xrightarrow{\tau} c[x].\{^{\mathsf{err}}/_{x}\}P} \qquad \text{(Tau)}\ \frac{}{\Gamma \triangleright \tau.P \xrightarrow{\tau} P}$$

$$\text{(Then)}\ \frac{[\![b]\!]_\Gamma = \text{true}}{\Gamma \triangleright [b]P, Q \xrightarrow{\tau} \sigma.P} \qquad \text{(Else)}\ \frac{[\![b]\!]_\Gamma = \text{false}}{\Gamma \triangleright [b]P, Q \xrightarrow{\tau} \sigma.Q}$$

station waiting to receive a value along d. Finally, since $\Gamma_1 \vdash_n c : 2$, we can use Rule (ActRcv) to derive $\Gamma_1 \triangleright c[x].P \xrightarrow{\sigma} c[x].P$.

At this point we can use Rule (TimePar) twice to infer a σ-action performed by C_1. This leads to the transition $C_1 \xrightarrow{\sigma} W_2$, where $W_2 = \sigma \mid \lfloor c?(x).Q \rfloor \mid c[x].P$. □

Table 4 is devoted to internal transitions $\Gamma \triangleright W \xrightarrow{\tau} W'$. Let us first explain rule (RcvLate). Intuitively the process $\lfloor c?(x).P \rfloor Q$ is ready to start receiving a value on an exposed channel c. This means that a transmission is already taking place. Since the process has therefore missed the start of the transmission it will receive an error value. Thus Rule (RcvLate) reflects the fact that in wireless systems a broadcast value cannot be correctly received by a station in the case of a misalignment between the sender and the receiver.

The remaining rules are straightforward except that we use a channel environment dependent evaluation function for Boolean expressions $[\![b]\!]_\Gamma$, because of the presence of the exposure predicate $\mathrm{exp}(c)$ in the Boolean language. However in wireless systems it is not possible to both listen and transmit within the same time unit, as communication is half-duplex, [19]. So in our intensional semantics, in the rules (Then) and (Else), the execution of both branches is delayed of one time unit; this is a slight simplification, as evaluation is delayed even if the Boolean expression does not contain an exposure predicate.

Table 5. Intensional semantics: - structural rules

$$\text{(TauPar)}\ \frac{\Gamma \triangleright W_1 \xrightarrow{\tau} W_1'}{\Gamma \triangleright W_1 \mid W_2 \xrightarrow{\tau} W_1' \mid W_2} \qquad \text{(Rec)}\ \frac{\{\mathsf{fix}\, X.P/X\}P \xrightarrow{\lambda} W}{\Gamma \triangleright \mathsf{fix}\, X.P \xrightarrow{\lambda} W}$$

$$\text{(Sum)}\ \frac{\Gamma \triangleright P \xrightarrow{\lambda} W \quad \lambda \in \{\tau, c!v\}}{\Gamma \triangleright P + Q \xrightarrow{\lambda} W} \qquad \text{(SumRcv)}\ \frac{\Gamma \triangleright P \xrightarrow{c?v} W \quad \mathsf{rcv}(P,c) \quad \Gamma \vdash c : \mathbf{idle}}{\Gamma \triangleright P + Q \xrightarrow{c?v} W}$$

$$\text{(ResI)}\ \frac{\Gamma[c \mapsto (n,v)] \triangleright W \xrightarrow{c!v} W'}{\Gamma \triangleright \nu c{:}(n,v).W \xrightarrow{\tau} \nu c{:}\mathrm{upd}_{c!v}(\Gamma)(c).W'} \qquad \text{(ResV)}\ \frac{\Gamma[c \mapsto (n,v)] \triangleright W \xrightarrow{\lambda} W',\ c \notin \lambda}{\Gamma \triangleright \nu c{:}(n,v).W \xrightarrow{\lambda} \nu c{:}(n,v).W'}$$

Example 3. Let Γ_2 be a channel environment such that $\Gamma_2(c) = (1, v)$, and consider the configuration $C_2 = \Gamma_2 \triangleright W_2$, where W_2 has been defined in Example 2.

Note that this configuration contains an active receiver along the exposed channel c. We can think of such a receiver as a process which missed the synchronisation with a broadcast which has been previously performed along channel c; as a consequence this process is doomed to receive an error value.

This situation is modelled by Rule (RcvLate), which allows us to infer the transition $\Gamma_2 \triangleright \lfloor c?(x).Q \rfloor \xrightarrow{\tau} c[x].\{\mathsf{err}/x\}Q$. As we will see, Rule (TauPar), introduced in Table 5, ensures that τ-actions are propagated to the external environment. This means that the transition derived above allows us to infer the transition $C_2 \xrightarrow{\tau} W_3$, where $W_3 = \sigma \mid c[x].\{\mathsf{err}/x\}Q \mid c[x].P$. □

The final set of rules, in Table 5, are structural. Here we assume that Rules (Sum), (SumRcv) and (SumTime) have a symmetric counterpart. Rules (ResI) and (ResV) show how restricted channels are handled. Intuitively moves from the configuration $\Gamma \triangleright \nu c{:}(n,v).W$ are inherited from the configuration $\Gamma[c \mapsto (n,v)] \triangleright W$; here the channel environment $\Gamma[c \mapsto (n,v)]$ is the same as Γ except that c has associated with it (temporarily) the information (n, v). However if this move mentions the restricted channel c then the inherited move is rendered as an internal action τ, (ResI). Moreover the information associated with the restricted channel in the residual is updated, using the function $\mathrm{upd}_{c!v}(\cdot)$ previously defined.

We are now ready to define the reduction semantics; formally, we let $\Gamma_1 \triangleright W_1 \twoheadrightarrow \Gamma_2 \triangleright W_2$ whenever $\Gamma_1 \triangleright W_1 \xrightarrow{\lambda} W_2$ and $\Gamma_2 = \mathrm{upd}_\lambda(\Gamma_1)$ for some $\lambda = \tau, \sigma, c!v$.

Note that input actions cannot be used to infer reductions for computations; following the approach of [15,21] reductions are defined to model only the internal of a system. In order to distinguish between timed and untimed reductions in $\Gamma_1 \triangleright W_1 \twoheadrightarrow \Gamma_2 \triangleright W_2$ we use $\Gamma_1 \triangleright W_1 \twoheadrightarrow_\sigma \Gamma_2 \triangleright W_2$ if $\Gamma_2 = \mathrm{upd}_\sigma(W_1)$ and $\Gamma_1 \triangleright W_1 \twoheadrightarrow_i \Gamma_2 \triangleright W_2$ if $\Gamma_2 = \mathrm{upd}_\lambda(\Gamma_1)$ for some $\lambda = \tau, c!v$.

Proposition 1 (Maximal Progress and Time Determinism). *Suppose $C \twoheadrightarrow_\sigma C_1$; then $C \twoheadrightarrow_\sigma C_2$ implies $C_1 = C_2$, and $C \not\twoheadrightarrow_i C_3$ for any C_3.*

Example 4. We now show how the transitions we have inferred in the Examples 1-3 can be combined to derive a computation fragment for the configuration C_0 considered in Example 1.

Let $C_i = \Gamma_i \triangleright W_i, i = 0, \cdots, 2$ be as defined in these examples. Note that $\Gamma_1 = \mathrm{upd}_{c!v_0}(\Gamma_0)$ and $\Gamma_2 = \mathrm{upd}_{\sigma}(\Gamma_1)$. We have already shown that $C_0 \xrightarrow{c!v_0} W_1$; this transition, together with the equality $\Gamma_1 = \mathrm{upd}_{c!v_0}(\Gamma_0)$, can be used to infer the reduction $C_0 \rightarrow_i C_1$. A similar argument shows that $C_1 \rightarrow_{\sigma} C_2$. Also if we let C_3 denote $\Gamma_2 \triangleright W_3$ we also have $C_2 \rightarrow_i C_3$ since $\Gamma_2 = \mathrm{upd}_{\tau}(\Gamma_2)$. □

Example 5 (Collisions). Consider the configuration $C = \Gamma \triangleright W$, where $\Gamma \vdash c : \mathbf{idle}$ and $W = c!\langle w_0\rangle \mid c!\langle w_1\rangle \mid \lfloor c?(x).P \rfloor$; here we assume $\delta_{w_0} = \delta_{w_1} = 1$. Using rules (Snd), (RcvIgn), (Rcv) and (Sync) we can infer the transition $\Gamma \triangleright W \xrightarrow{c!w_0} W_1$, where $W_1 = \sigma \mid c!\langle w_1\rangle \mid c[x].P$. Let $\Gamma_1 := \mathrm{upd}_{c!w_0}(\Gamma)$, that is $\Gamma_1(c) = (1, w_0)$. This equality and the transition above lead to the instantaneous reduction $C \rightarrow_i C_1 = \Gamma_1 \triangleright W_1$.

For C_1 we can use the rules (RcvIgn), (Snd) and (Sync) to derive the transition $C_1 \xrightarrow{c!w_1} W_2$, where $W_2 = \sigma \mid \sigma \mid c[x].P$. This transition gives rise to the reduction $C_1 \rightarrow_i C_2 = \Gamma_2 \triangleright W_2$, where $\Gamma_2 = \mathrm{upd}_{c!w_1}(\Gamma_1)$. Note that, since $\Gamma_1 \vdash c : \mathbf{exp}$ we obtain that $\Gamma_2(c) = (1, \mathsf{err})$. The broadcast along a busy channel caused a collision to occur.

Finally, rules (Sleep), (EndRcv) and (TimePar) can be used to infer the transition $C_2 \xrightarrow{\sigma} W_3 = \mathsf{nil} \mid \mathsf{nil} \mid \{\mathsf{err}/x\}P$. Let $\Gamma_3 := \mathrm{upd}_{\sigma}(\Gamma'')$; then the transition above induces the timed reduction $C_2 \rightarrow_{\sigma} C_3 = \Gamma_3 \triangleright W_3$, in which an error is received instead of either of the transmitted values w_0, w_1. □

We now define a contextual equivalence between configurations, following the approach of [10]. This relies on two crucial concepts: a notion of reduction, already been defined, and a notion of minimal observable activity, called a *barb*.

While in other process algebras the basic observable activity is chosen to be an output on a given channel [21,7], for our calculus it is more appropriate to rely on the exposure state of a channel: because of possible collisions transmitted values may never be received. Formally, we say that a configuration $\Gamma \triangleright W$ has a barb on channel c, written $\Gamma \triangleright W \downarrow_c$, whenever $\Gamma \vdash c : \mathbf{exp}$. A configuration $\Gamma \triangleright W$ has a weak barb on c, denoted by $\Gamma \triangleright W \Downarrow_c$, if $\Gamma \triangleright W \rightarrow^* \Gamma' \triangleright W'$ for some $\Gamma' \triangleright W'$ such that $\Gamma' \triangleright W' \downarrow_c$. As we will see, it turns out that using this notion of barb we can observe the content of a message being broadcast only at the end of its transmission. This is in line with the standard theory of wireless networks, in which it is stated that collisions can be observed only at reception time [23,19].

Definition 1. *Let $\mathcal{R}$ be a relation over configurations.*

(1) $\mathcal{R}$ is said to be barb preserving *if $\Gamma_1 \triangleright W_1 \Downarrow_c$ implies $\Gamma_2 \triangleright W_2 \Downarrow_c$, whenever $(\Gamma_1 \triangleright W_1)\ \mathcal{R}\ (\Gamma_2 \triangleright W_2)$.*

(2) It is reduction-closed *if $(\Gamma_1 \triangleright W_1)\ \mathcal{R}\ (\Gamma_2 \triangleright W_2)$ and $\Gamma_1 \triangleright W_1 \rightarrow \Gamma_1' \triangleright W_1'$ imply there is some $\Gamma_2' \triangleright W_2'$ such that $\Gamma_2 \triangleright W_2 \rightarrow^* \Gamma_2' \triangleright W_2'$ and $(\Gamma_1' \triangleright W_1')\ \mathcal{R}\ (\Gamma_2' \triangleright W_2')$.*

(3) It is contextual *if $\Gamma_1 \triangleright W_1\ \mathcal{R}\ \Gamma_2 \triangleright W_2$, implies $\Gamma_1 \triangleright (W_1 \mid W)\ \mathcal{R}\ \Gamma_2 \triangleright (W_2 \mid W)$ for all processes W.* □

Reduction barbed congruence, written $\simeq$, is the largest symmetric relation over configurations which is barb preserving, reduction-closed and contextual.

Table 6. Extensional actions

$$\text{(Input)}\ \frac{\Gamma \triangleright W \xrightarrow{c?v} W'}{\Gamma \triangleright W \overset{c?v}{\longmapsto} \mathrm{upd}_{c?v}(\Gamma) \triangleright W'} \qquad \text{(Time)}\ \frac{\Gamma \triangleright W \xrightarrow{\sigma} W'}{\Gamma \triangleright W \overset{\sigma}{\longmapsto} \mathrm{upd}_{\sigma}(\Gamma) \triangleright W'}$$

$$\text{(Shh)}\ \frac{\Gamma \triangleright W \xrightarrow{c!v} W'}{\Gamma \triangleright W \overset{\tau}{\longmapsto} \mathrm{upd}_{c!v}(\Gamma) \triangleright W'} \qquad \text{(TauExt)}\ \frac{\Gamma \triangleright W \xrightarrow{\tau} W'}{\Gamma \triangleright W \overset{\tau}{\longmapsto} \Gamma \triangleright W'}$$

$$\text{(Deliver)}\ \frac{\Gamma(c) = (1, v) \quad \Gamma \triangleright W \xrightarrow{\sigma} W'}{\Gamma \triangleright W \overset{\gamma(c,v)}{\longmapsto} \mathrm{upd}_{\sigma}(\Gamma) \triangleright W'} \qquad \text{(Idle)}\ \frac{\Gamma \vdash c : \mathbf{idle}}{\Gamma \triangleright W \overset{\iota(c)}{\longmapsto} \Gamma \triangleright W}$$

Example 6. We first give some examples of configurations which are not barbed congruent; here we assume that Γ is the stable environment.

- $\Gamma \triangleright c!\langle v_0\rangle \not\simeq \Gamma \triangleright c!\langle v_1\rangle$; let $T = \lfloor c?(x).[x = v_0]d!\langle ok\rangle \mathsf{nil}, \rfloor$, where $d \neq c$ and ok is an arbitrary value. It is easy to see that $\Gamma \triangleright c!\langle v_0\rangle \mid T \Downarrow_d$, whereas $\Gamma \triangleright c!\langle v_1\rangle \mid T \not\Downarrow_d$.
- $\Gamma \triangleright c!\langle v\rangle \not\simeq \Gamma \triangleright \sigma.c!\langle v\rangle$; let $T = [\exp(c)]d!\langle ok\rangle, \mathsf{nil}$. In this case we have that $\Gamma \triangleright c!\langle v\rangle \mid T \Downarrow_d$, while $\Gamma \triangleright \sigma.c!\langle v\rangle \mid T \not\Downarrow_d$.

On the other hand, consider the configurations $\Gamma \triangleright c!\langle v_0\rangle \mid c!\langle v_1\rangle$ and $\Gamma \triangleright c!\langle \mathsf{err}\rangle$, where $\delta_{v_0} = \delta_{v_1}$ and for the sake of convenience we assume that $\delta_{\mathsf{err}} = \delta_{v_0}$. In both cases a communication along channel c starts, and in both cases the value that will be eventually delivered to some receiving station is err, independently of the behaviour of the external environment. This gives us the intuition that these two configurations are barbed congruent. Later in the paper we will develop the tools that will allow us to prove this statement formally. □

4 Extensional Semantics

In this section we give a co-inductive characterisation of the contextual equivalence $\simeq$ between configurations, using a standard bisimulation equivalence over an extensional LTS, with configurations as nodes, but with a special collection of *extensional actions*; these are defined in Table 6.

Rule (Input) simply states that input actions are observable, as is the passage of time, by Rule (Time). Rule (TauExt) propagates τ-intensional actions to the extensional semantics. Rule (Shh) states that broadcasts are always treated as internal activities in the extensional semantics. This choice reflects the intuition that the content of a message being broadcast cannot be detected immediately; in fact, it cannot be detected until the end of the transmission.

Rule (Idle) introduces a new label $\iota(c)$, parameterized in the channel c, which is not inherited from the intensional semantics. Intuitively this rules states that it is possible to observe whether a channel is exposed. Finally, Rule (Deliver) states that the delivery of a value v along channel c is observable, and it corresponds to a new action whose label is $\gamma(c, v)$. In the following we range over extensional actions by α.

Example 7. Consider the configuration $\Gamma \triangleright c!\langle v\rangle$, where Γ is the stable channel environment. By an application of Rule (Shh) we have the transition $\Gamma \triangleright c!\langle v\rangle \stackrel{\tau}{\longmapsto} \Gamma' \triangleright \sigma^{\delta_v}$, with $\Gamma' \vdash c : \mathbf{exp}$. Furthermore, $\Gamma \triangleright c!\langle v\rangle \stackrel{\iota(c)}{\longmapsto}$ since channel c is idle in Γ. Notice that $\Gamma' \triangleright \sigma^{\delta_v}$ cannot perform a $\iota(c)$ action, and that the extensional semantics gives no information about the value v which has been broadcast.

The extensional semantics endows configurations with the structure of an LTS. Weak extensional actions in this LTS are defined as usual, and the formulation of bisimulations is facilitated by the notation $C \stackrel{\hat{\alpha}}{\Longmapsto} C'$, which is again standard: for $\alpha = \tau$ this denotes $C \longmapsto^* C'$ while for $\alpha \neq \tau$ it is $C \stackrel{\tau}{\longmapsto}^* \stackrel{\alpha}{\longmapsto} \stackrel{\tau}{\longmapsto}^* C'$.

Definition 2 (Bisimulations). *Let $\mathcal{R}$ be a symmetric binary relation over configurations. We say that $\mathcal{R}$ is a (weak) bisimulation if for every extensional action α, whenever $C_1 \mathcal{R} C_2$, then $C_1 \stackrel{\alpha}{\Longmapsto} C_1'$ implies $C_2 \stackrel{\hat{\alpha}}{\Longmapsto} C_2'$ for some C_2' satisfying $C_1' \mathcal{R} C_2'$ We let $\approx$ be the the largest bisimulation.* □

Example 8. Let us consider again the configurations $\Gamma \triangleright W_0 = c!\langle v_0\rangle \mid c!\langle v_1\rangle, \Gamma \triangleright W_1 = c!\langle \mathsf{err}\rangle$ of Example 6. Recall that in this example we assumed that Γ is the stable channel environment; further, $\delta_{v_0} = \delta_{v_1} = \delta_{\mathsf{err}} = k$ for some $k > 0$.

We show that $\Gamma \triangleright W_0 \approx \Gamma \triangleright W_1$ by exhibiting a witness bisimulation $\mathcal{S}$ such that $\Gamma \triangleright W_0 \; \mathcal{S} \; \Gamma \triangleright W_1$. To this end, let us consider the relation

$$\begin{aligned}\mathcal{S} = \{ \; & (\Delta \triangleright W_0, \Delta \triangleright W_1) \quad , (\Delta' \triangleright \sigma^k \mid c!\langle v_1\rangle, \Delta'' \triangleright \sigma^k) \; , (\Delta' \triangleright c!\langle v_0\rangle, \Delta'' \triangleright \sigma^k) \\ & , \quad (\Delta \triangleright \sigma^j \mid \sigma^j, \Delta \triangleright \sigma^j) \mid \Delta' \vdash_t c : n, \Delta''(c) = (n, \mathsf{err}) \text{ for some } n > 0, j \leq k\}\end{aligned}$$

Note that this relation contains an infinite number of pairs of configurations, which differ by the state of channel environments.This is because input actions can affect the channel environment of configurations. It is easy to show that the relation $\mathcal{S}$ is a bisimulation which contains the pair $(\Gamma_0 \triangleright W_0, \Gamma_1 \triangleright W_1)$, therefore $\Gamma \triangleright W_0 \approx \Gamma \triangleright W_1$. □

One essential property of weak bisimulation is that it does not relate configurations which differ by the exposure state of some channel:

Proposition 2. *Suppose $\Gamma_1 \triangleright W_1 \approx \Gamma_2 \triangleright W_2$. Then for any channel c, $\Gamma_1 \vdash c :$ **idle** iff $\Gamma_2 \vdash c :$ **idle**.* □

5 Full Abstraction

The aim of this section is to prove that weak bisimilarity in the extensional semantics is a proof technique which is both sound and complete for reduction barbed congruence.

Theorem 1 (Soundness). *$C_1 \approx C_2$ implies $C_1 \simeq C_2$.*

Proof. It suffices to prove that bisimilarity is reduction-closed, barb preserving and contextual. Reduction closure follows from the definition of bisimulation equivalence. The preservation of barbs follows directly from Proposition 2. The proof of contextuality on the other hand is quite technical, and is addressed in detail in the associated technical

report [4]. One subtlety lies in the definition of τ-extensional actions, which include broadcasts. While broadcasts along exposed do not affect the external environment, and hence cannot affect the external environment, this is not true for broadcasts performed along idle channels. However, we can take advantage of Proposition 2 to show that these extensional τ-actions preserve the contextuality of bisimilar configurations. □

To prove completeness, the converse of Theorem 1, we restrict our attention to the subclass of *well-formed* configurations. Informally $\Gamma \triangleright W$ is well-formed if the system term W does not contain active receivers along idle channels; a wireless station cannot be receiving a value along a channel if there is no value being transmitted along it.

Definition 3 (Well-formedness). *The set of well-formed configurations WNets is the least set such that for all processes P* ***(i)*** $\Gamma \triangleright P \in$ *Wnets,* ***(ii)*** *if* $\Gamma \vdash c : \mathbf{exp}$ *then* $\Gamma \triangleright c[x].P \in$ *WNets,* ***(iii)*** *is closed under parallel composition and* ***(iv)*** *if* $\Gamma[c \mapsto (n, v)] \triangleright W \in$ *WNets then* $\Gamma \triangleright \nu c : (n, v).W \in$ *WNets.* □

By focusing on well-formed configurations we can prove a counterpart of Proposition 2 for our contextual equivalence:

Proposition 3. *Let* $\Gamma_1 \triangleright W_1, \Gamma_2 \triangleright W_2$ *be two well formed configurations such that* $\Gamma_1 \triangleright W_1 \simeq \Gamma_2 \triangleright W_2$. *Then for any channel c,* $\Gamma_1 \vdash c : \mathbf{idle}$ *implies* $\Gamma_2 \vdash c : \mathbf{idle}$. □

Proposition 3 does not hold for ill-formed configurations. For example, let $\Gamma_1 \vdash c : \mathbf{exp}$, $\Gamma_1 \vdash d : \mathbf{idle}$ and $\Gamma_2 \vdash c, d : \mathbf{idle}$ and consider the two configurations $C_1 = \Gamma_1 \triangleright \mathsf{nil} \mid d[x].P$ and $C_2 = \Gamma_2 \triangleright c!\langle v\rangle \mid d[x].P$, neither of which are well-formed; nor do they let time pass, $C_i \not\rightarrow_\sigma$. As a consequence $C_1 \simeq C_2$. However Proposition 2 implies that they are not bisimilar, since they differ on the exposure state of c.

Another essential property of well-formed systems is patience: time can always pass in networks with no instantaneous activities.

Proposition 4 (Patience). *If C is well-formed and* $C \not\rightarrow_i$, *then* $C \rightarrow_\sigma C'$ *for some* C'.□

This means that, if we restrict our attention to well-formed configurations, we can never reach a configuration which is deadlocked; at the very least time can always proceed.

Theorem 2 (Completeness). *On well-formed configurations, reduction barbed congruence implies bisimilarity.*

The proof relies on showing that for each extensional action α it is possible to exhibit a test T_α which determines whether or not a configuration $\Gamma \triangleright W$ can perform the action α. The main idea is to equip the test with some fresh channels; the test T_α is designed so that a configuration $\Gamma \triangleright W \mid T_\alpha$ can reach another one $C' = \Gamma' \triangleright W' \mid T'$, where T' is determined uniquely by the barbs of the introduced fresh channel; these are enabled in $\Gamma' \triangleright T'$, if and only if C can weakly perform the action α.

The tests T_α are defined by performing a case analysis on the extensional action α:

$$
\begin{aligned}
T_\tau &= \mathit{eureka}!\langle \mathit{ok}\rangle \\
T_\sigma &= \sigma.(\tau.\mathit{eureka}!\langle \mathit{ok}\rangle + \mathit{fail}!\langle \mathit{no}\rangle) \\
T_{\gamma(c,v)} &= \nu d{:}(0,\cdot).((c[x].([x{=}v]d!\langle \mathit{ok}\rangle, \mathsf{nil}) + \mathit{fail}!\langle \mathit{no}\rangle) \mid \\
&\quad \mid \sigma^2.[\exp(d)]\mathit{eureka}!\langle \mathit{ok}\rangle, \mathsf{nil} \mid \sigma.\mathit{halt}!\langle \mathit{ok}\rangle) \\
T_{c?_v} &= (c\,!\langle v\rangle.\mathit{eureka}!\langle \mathit{ok}\rangle + \mathit{fail}!\langle \mathit{no}\rangle) \mid \mathit{halt}!\langle \mathit{ok}\rangle \\
T_{\iota(c)} &= ([\exp(c)]\mathsf{nil}, \mathit{eureka}!\langle \mathit{ok}\rangle) + \mathit{fail}!\langle \mathit{no}\rangle \mid \mathit{halt}!\langle \mathit{ok}\rangle
\end{aligned}
$$

where *eureka*, *fail*, *halt* are arbitrary distinct channels and *ok*, *no* are two values such that $\delta_{ok} = \delta_{no} = 1$.

For the sake of simplicity, for any action α we define also the tests T'_α as follows:

$$\begin{array}{ll} T'_\tau = T'_\sigma & = eureka!\langle ok\rangle \\ T'_{\gamma(c,v)} & = \nu d{:}(0,\cdot).(\sigma.d!\langle ok\rangle \mathsf{nil} \mid \sigma.[\exp(d)]eureka!\langle ok\rangle, \mathsf{nil} \mid halt!\langle ok\rangle) \\ T'_{c?v} & = \sigma^{\delta_v}.eureka!\langle ok\rangle \mid halt!\langle ok\rangle \\ T'_{\iota(c)} & = \sigma.eureka!\langle ok\rangle \mid halt!\langle ok\rangle \end{array}$$

Proposition 5 (Distinguishing Contexts). *Let $\Gamma \triangleright W$ be a well-formed configuration, and suppose that the channels* eureka, halt, fail *do not appear free in W, nor they are exposed in Γ. Then for any extensional action α, $\Gamma \triangleright W \stackrel{\alpha}{\Longrightarrow} \Gamma' \triangleright W'$ iff $\Gamma \triangleright W \mid T_\alpha \rightarrow^* \Gamma' \triangleright W' \mid T'_\alpha$.* □

A pleasing property of the tests T'_α is that they can be identified by the (both strong and weak) barbs that they enable in a computation rooted in the configuration $\Gamma \triangleright W \mid T_\alpha$.

Proposition 6 (Uniqueness of successful testing components). *Let $\Gamma \triangleright W$ be a configuration such that* eureka, halt, fail *do not appear free in W, nor they are exposed in Γ. Suppose that $\Gamma \triangleright W \mid T_\alpha \rightarrow^* C'$ for some configuration C' such that*

- *if $\alpha = \tau, \sigma$, then $C' \not\downarrow_{eureka}$, $C' \Downarrow_{eureka}$, $C' \not\Downarrow_{fail}$,*
- *otherwise, $C' \not\downarrow_{eureka}, C' \not\downarrow_{halt}$, $C' \Downarrow_{eureka}, C' \Downarrow_{halt}$, $C' \not\Downarrow_{fail}$.*

Then $C' = \Gamma' \triangleright W' \mid T'_\alpha$ for some configuration $\Gamma' \triangleright W'$. □

Note the use of the fresh channel *halt* when testing some of these actions. This is because of a time mismatch between a process performing the action, and the test used to detect it. For example the weak action $\stackrel{\iota(c)}{\Longrightarrow}$ does not involve the passage of time but the corresponding test uses a branching construct which needs at least one time step to execute. Requiring a weak barb on *halt* in effect prevents the passage of time.

Outline proof of Theorem 2: It is sufficient to show that reduction barbed congruence, $\simeq$, is a bisimulation. As an example suppose $\Gamma_1 \triangleright W_1 \simeq \Gamma_2 \triangleright W_2$ and $\Gamma_1 \triangleright W_1 \stackrel{\gamma(c,v)}{\longmapsto} \Gamma'_1 \triangleright W'_1$. We show how to find a matching move from $\Gamma_2 \triangleright W_2$.

Suppose that $\Gamma_1 \triangleright W_1 \stackrel{\gamma(c,v)}{\longmapsto} \Gamma'_1 \triangleright W'_1$, we need to show that $\Gamma_2 \triangleright W_2 \stackrel{\gamma(c,v)}{\Longrightarrow} \Gamma'_2 \triangleright W'_2$ for some $\Gamma'_2 \triangleright W'_2$ such that $\Gamma'_1 \triangleright W'_1 \simeq \Gamma'_2 \triangleright W'_2$. By Proposition 5 we know that $\Gamma_1 \triangleright W_1 \mid T_{\gamma(c,v)} \rightarrow^* \Gamma'_1 \triangleright W'_1 \mid T'_\alpha$.By the hypothesis it follows that $\Gamma_1 \triangleright W_1 \mid T_{\gamma(c,v)} \simeq \Gamma_2 \triangleright W_2 \mid T_{\gamma(c,v)}$, therefore $\Gamma_2 \triangleright W_2 \mid T_{\gamma(c,v)} \rightarrow^* C_2$ for some $C_2 \simeq \Gamma'_1 \triangleright W'_1 \mid T'_{\gamma(c,v)}$.

Let $C_1 = \Gamma'_1 \triangleright W'_1 \mid T'_{\gamma(c,v)}$. It is easy to check that $C_1 \not\downarrow_{eureka}, C_1 \not\downarrow_{halt}, C_1 \not\Downarrow_{fail}$ and $C_1 \Downarrow_{eureka}, C_1 \Downarrow_{halt}$. By definition of reduction barbed congruence and Proposition 3 we obtain that $C_2 \not\downarrow_{eureka}, C_2 \not\downarrow_{halt}$, $C_2 \Downarrow_{eureka}, C_2 \Downarrow_{halt}$ and $C_2 \not\Downarrow_{fail}$. Proposition 6 ensures then that $C_2 = \Gamma'_2 \triangleright W'_2 \mid T'_{\gamma(c,v)}$ for some Γ'_2, W'_2. An application of Proposition 5 leads to $\Gamma_2 \triangleright W_2 \stackrel{\gamma(c,v)}{\Longrightarrow} \Gamma'_2 \triangleright W'_2$. Now standard process calculi techniques enable us to infer from this that $\Gamma'_1 \triangleright W'_1 \simeq \Gamma'_2 \triangleright W'_2$. □

6 Conclusions and Related Work

In this paper we have given a behavioural theory of wireless systems at the MAC level. We believe that our reduction semantics, given in Section 2, captures much of the subtlety of intensional MAC-level behaviour of wireless systems. We also believe that our behavioural theory is the only one for wireless networks at the MAC-Layer which is both sound and complete. The only other calculus which considers such networks is TCWS from [14] which contains a sound theory; as we have already stated we view CCCP as a simplification of this TCWS, and by using a more refined notion of extensional action we also obtain completeness.

We are aware of only two other papers modelling networks at the MAC-Sublayer level of abstraction, these are [12,24]. They present a calculus CWS which views a network as a collection of nodes distributed over a metric space. [12] contains a reduction and an intensional semantics and the main result is their consistency. In [24], time and node mobility is added.

On the other hand there are numerous papers which consider the problem of modelling networks at a higher level. Here we briefly consider a selection; for a more thorough review see [4].

Nanz and Hankin [16] have introduced an untimed calculus for Mobile Wireless Networks (CBS$^\sharp$), relying on a graph representation of node localities. The main goal of that paper is to present a framework for specification and security analysis of communication protocols for mobile wireless networks. Merro [13] has proposed an untimed process calculus for mobile ad-hoc networks with a labelled characterisation of reduction barbed congruence, while [6] contains a calculus called CMAN, also with mobile ad-hoc networks in mind. Singh, Ramakrishnan and Smolka [22] have proposed the ω-calculus, a conservative extension of the π-calculus. A key feature of the ω-calculus is the separation of a node's communication and computational behaviour from the description of its physical transmission range. Another extension of the π-calculus, which has been used for modelling the LUNAR ad-hoc routing protocol, may be found in [2].

In [3] a calculus is proposed for describing the probabilistic behaviour of wireless networks. There is an explicit representation of the underlying network, in terms of a connectivity graph. However this connectivity graph is static. In contrast Ghassemi et al. [5] have proposed a process algebra called RBPT where topological changes to the connectivity graph are implicitly modelled in the operational semantics rather than in the syntax. Kouzapas and Philippou [11] have developed a theory of confluence for a calculus of dynamic networks and they use their machinery to verify a leader-election algorithm for mobile ad hoc networks.

References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer Networks 38(4), 393–422 (2002)
2. Borgström, J., Huang, S., Johansson, M., Raabjerg, P., Victor, B., Pohjola, J.Å., Parrow, J.: Broadcast psi-calculi with an application to wireless protocols. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 74–89. Springer, Heidelberg (2011)

3. Cerone, A., Hennessy, M.: Modelling probabilistic wireless networks (extended abstract). In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 135–151. Springer, Heidelberg (2012), http://www.scss.tcd.ie/~ceronea/works/ProbabilisticWirelessNetworks.pdf
4. Cerone, A., Hennessy, M., Merro, M.: Modelling mac-layer communications in wireless systems. Technical Report, Trinity College Dublin (2012), https://www.scss.tcd.ie/~acerone/works/CCCP.pdf.
5. Ghassemi, F., Fokkink, W., Movaghar, A.: Equational reasoning on mobile ad hoc networks. Fundamenta Informaticae 105(4), 375–415 (2010)
6. Godskesen, J.C.: A Calculus for Mobile Ad Hoc Networks. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 132–150. Springer, Heidelberg (2007)
7. Hennessy, M.: A distributed Pi-calculus. Cambridge University Press (2007)
8. Hennessy, M., Rathke, J.: Bisimulations for a calculus of broadcasting systems. TCS 200 (1-2), 225–260 (1998)
9. Hennessy, M., Regan, T.: A process algebra for timed systems. IaC 117(2), 221–239 (1995)
10. Honda, K., Yoshida, N.: On reduction-based process semantics. TCS 152(2), 437–486 (1995)
11. Kouzapas, D., Philippou, A.: A process calculus for dynamic networks. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 213–227. Springer, Heidelberg (2011)
12. Lanese, I., Sangiorgi, D.: An operational semantics for a calculus for wireless systems. TCS 411(19), 1928–1948 (2010)
13. Merro, M.: An Observational Theory for Mobile Ad Hoc Networks (full paper). IaC 207(2), 194–208 (2009)
14. Merro, M., Ballardin, F., Sibilio, E.: A timed calculus for wireless systems. TCS 412(47), 6585–6611 (2011)
15. Milner, R.: Communicating and Mobile Systems: The π-calculus. Cambridge University Press (1999)
16. Nanz, S., Hankin, C.: Static analysis of routing protocols for ad-hoc networks. In: ACM SIGPLAN and IFIP WG, vol. 1, pp. 141–152. Citeseer (2004)
17. Nicollin, X., Sifakis, J.: The algebra of timed processes, atp: Theory and application. IaC 114(1), 131–178 (1994)
18. Prasad, K.V.S.: A calculus of broadcasting systems. In: Sannella, D. (ed.) ESOP 1994. LNCS, vol. 788, pp. 285–327. Springer, Heidelberg (1994)
19. Rappaport, T.S.: Wireless communications - principles and practice. Prentice-Hall (1996)
20. Rathke, J., Sobocinski, P.: Deconstructing behavioural theories of mobility. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) Fifth IFIP ICTCScience. IFIP, vol. 273, pp. 507–520. Springer, Boston (2008)
21. Sangiorgi, D., Walker, D.: The Pi-Calculus — A Theory of Mobile Processes. Cambridge University Press (2001)
22. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. SCP 75(6), 440–469 (2010)
23. Tanenbaum, A.S.: Computer Networks, 4th edn. Prentice-Hall International, Inc. (2003)
24. Wang, M., Lu, Y.: A timed calculus for mobile ad hoc networks. arXiv preprint arXiv:1301.0045 (2013)
25. Yi, W.: A Calculus of Real Time Systems. Ph.D Thesis, Chalmers University (1991)

Coordinating Phased Activities while Maintaining Progress

Tiago Cogumbreiro, Francisco Martins, and Vasco Thudichum Vasconcelos

LaSIGE, Faculty of Sciences, University of Lisbon

Abstract. In order to develop reliable applications for parallel machines, programming languages and systems need to provide for flexible parallel programming coordination techniques. Barriers, clocks and phasers constitute promising synchronisation mechanisms, but they exhibit intricate semantics and allow writing programs that can easily deadlock. We present an operational semantics and a type system for a fork/join programming model equipped with a flexible variant of phasers. Our proposal allows for a precise control over the maximum number of synchronisation steps each task can be ahead of others. A type system ensures that programs do not deadlock, even when they use multiple phasers.

1 Introduction

The key to develop scalable parallel applications lies in using coordination mechanisms at the "right" level of abstraction [8]. Rather than re-inventing the wheel with *ad hoc* solutions [20], programmers should resort to off-the-shelf coordination mechanisms present in programming languages and systems. Barriers, in their multiple forms [1,5,6,7,9,11,14] constitute one such coordination mechanism. A barrier allows multiple tasks to synchronise at a single point, in such a way that: a) before synchronisation *no task* has crossed the barrier, and b) after synchronisation *all tasks* have crossed the barrier.

Programs that use a single barrier to coordinate all their tasks do not deadlock. If using a single barrier may reveal itself quite limited in practice, groups of tasks that use multiple barriers may easily deadlock. To address this issue, the X10 programming language [7] proposes clocks, a deadlock-free coordination mechanism. Clocks later inspired primitives in other languages, such as Java [13] (as of version 7), Habanero Java [17] (HJ), and an extension to OpenMP [18].

For some applications, the semantics of traditional barriers is overly inflexible. With this mind, Shirako *et al.* introduced *phasers* [17], a primitive that allows some tasks to cross the barrier before synchronisation, thus relaxing condition a) above. Phasers allow for asymmetric parallel programs, including multiple producer/single consumer applications where, at each iteration, the consumer waits for the producers to cooperate in assembling an item. In a different direction, Albrecht *et al.* proposed a *partial barrier* construct [3] that only requires some tasks to arrive at the barrier, thus relaxing condition b). This form of barriers

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 31–44, 2013.

can be used in applications that synchronise on a subset of all tasks, allowing, *e.g.*, computation to progress quickly even when in presence of slow tasks.

Reasoning about such enriched constructs is usually far from trivial: the semantics is intricate and most languages lack a precise specification (including Java barriers and HJ phasers). In this work we propose a calculus for a fork/join programming model that unifies the various forms of barriers, including X10 clocks [7], HJ phasers (both bounded [16] and unbounded [17]), and Java barriers [13].

Our proposal not only subsumes those of X10, HJ and Java, but further increases the flexibility of the coordination mechanism. We allow tasks to be ahead of others up to a *bounded* number of synchronisation phases, and yet guarantee that well-typed programs are deadlock free. In contrast, HJ allows tasks to be ahead of others by an arbitrary number of phases, which is of limited interest, since fast tasks may eventually exhaust computational resources, such as buffer space.

To summarise, our contributions are:

- a flexible barrier construct that allows for a precise control over the maximum number of phases each task can be ahead of others;
- an operational semantics for a fork/join programming model that captures barrier-like coordination patterns found in X10, HJ, and Java;
- a type system that ensures progress, hence the absence of deadlocks, in addition to the usual type preservation.

The paper is organised as follows. The next section addresses related work. Section 3 presents the syntax and the operational semantics of our language. Thereafter, we introduce the type system and the main results of our work. Section 5 concludes the paper while putting forward lines of future research.

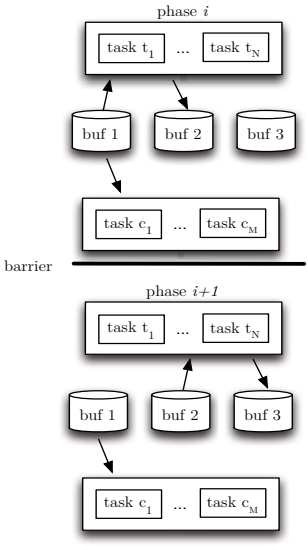
2 Related Work via an Example

Figure 1a sketches a parallel breadth-first search algorithm to find an exit in a labyrinth. The algorithm uses two groups of tasks: one traverses the labyrinth (modelled by a graph) and another inspects the visited nodes for an exit. The sketch was originally implemented in OpenMP by Süß and Leopold [19]. The algorithm proceeds iteratively, handling at each step (or phase) all nodes at the same depth. If a node is an exit, the algorithm terminates; otherwise it computes the node's neighbours and places them in a buffer to be processed in a later phase. The tasks synchronise at the end of each phase.

Figure 1b depicts two execution states for phases i and $i+1$. At phase i, tasks $t_1, \ldots, t_N$ read nodes from buf 1 (that stores the nodes of depth level i), compute their descendants, and then in buf 2 (via function traverseNodes()). Tasks $c_1, \ldots, c_M$ read nodes from buf 1 and look for an exit node (function checkNodesForAnExit()). Both groups of tasks synchronise (at phaser c) and advance to phase $i+1$. Notice that at phase $i+1$ the tasks $c_1, \ldots, c_M$ continue to process buf 1, while the traversal group $t_1, \cdots, t_N$ is handling nodes at depth

```
1  finish (
2    let t = newPhaser 0 in
3    let c = newPhaser 0 in (
4      for (int j=0; j<N; j++)
5        async {c:K, t:0} (
6          while(!exitFound) (
7            traverseNodes();
8            arrive c; arrive t; awaitAll);
9          drop c; drop t);
10     drop t;
11     for (int j=0; j<M; j++)
12       async {c:0} (
13         while(!exitFound) (
14           arrive c; awaitAll;
15           checkNodesForAnExit());
16         drop c)
17     drop c)
18 ) // join task created in lines 5,12
```

(a) Algorithm



(b) Execution diagram (K=2)

Fig. 1. A parallel breadth-first search algorithm

level $i + 1$ (buf 2). This situation is possible due to the bound K assigned to phaser c upon launching the traversal tasks (line 5). Bound K denotes the number of phases the traversal tasks may be ahead of the checker tasks. The number of buffers is equal to K + 1.

The synchronisation of tasks is challenging and involves phasers c and t. Phaser c is used to synchronise traversal tasks with checker tasks, while the traversal tasks further synchronise among themselves at phaser t. The bound c:K in line 5 reads as "phaser b has bound K in the spawned task." On the other hand, bound t:0 (also in line 5) means that the traversal activities must all be in the same phase. This synchronisation scheme ensures that the traversal tasks must process all nodes at the same depth and only after that advance to the next level. The traversal tasks set the pace for the checker tasks with phaser c. In their turn, the checker tasks (lines 12–16) use bound c:0, thus enforcing that they simultaneously process the same depth level and do not overtake the tasks in the traversal group.

Barriers, clocks, and phasers are insufficient for this sort of coordination. Barriers and clocks are too inflexible, since no task can cross the barrier before the others arrive. HJ phasers are too loose, since when not used as barriers, tasks run unconstrained and may overflow buffers. Phasers beams [16] are a step towards this sort of control, but its semantics is only informally described and they do not guarantee deadlock-freedom. Our proposal permits different tasks to specify the maximum number of phases they can be ahead of the slowest.

In contrast, phaser beams define such a number on a per-phaser basis. To unify regular phasers and phaser beams, we also supply an operation that skips waiting on a phase.

Saraswat and Jagadeesan presented a calculus for the X10 language that includes barriers, fork/join, conditional atomic blocks, and hierarchical shared memory [15]. The authors define a small-step operational semantics and claim that X10 programs without conditional atomic blocks do not deadlock, yet no deadlock-freedom theorem is formally proved. Lee and Palsberg present a calculus, called FX10, with two constructs from X10: fork/join and atomic blocks [10]. FX10 is suited for inter-procedural analysis through type inference and includes a formal proof of a fragment of the deadlock theorem stated by Saraswat and Jagadeesan. The type system used to identify may-happen-parallelism is further explored in [2]. Other formal studies on fork/join semantics include [1,4]. Our previous work defines an operational semantics and a type system for a calculus with a fork/join programming model and clocks [12], but does not include a deadlock-freedom theorem.

X10 clocks and HJ phasers are language-based approaches, whereas the Java barrier (also called phaser) is a library-based approach. Features that appear simultaneous in our calculus, in X10, and in HJ are: controlled barrier registration to avoid non-determinism, advancement on multiple barriers, and enforced barrier deregistration before activity termination to avoid barrier-related deadlocks. Features that appear in our calculus and in HJ alone include phaser visibility restricted to finish scopes to avoid deadlocks between phaser and finish; enforced deregistration before terminating a finish to avoid deadlocks between barriers. HJ and X10 automatically deregister from barriers when activities terminate. Instead we require explicit operations on phasers, in order to obtain a clearer operational semantics. Our type system provides enough information to guide a compiler into automatically inserting such operations, if desired. Finally, the ability to specify the maximum number of phases a task may be ahead of the slowest is unique to our language.

3 Syntax and Operational Semantics

Our language, inspired by X10 and HJ, uses activities to organise independent computations and features two coordination mechanisms to control concurrency: phaser and finish. For the sake of simplicity, the language focuses on task coordination, providing very little in the way of describing complex computations.

The syntax of the language is defined in Figure 2. It relies on a base set of variables, ranged over by x, and a set of natural numbers, ranged over by m and n. Bounds B map phaser names to the phaser bounds. We use the standard abbreviation $t; e$ to denote the expression $\textsf{let}\ x = t\ \textsf{in}\ e$ when variable x does not occur in expression e. A term t is transformed into an expression via $\textsf{let}\ x = t\ \textsf{in}\ x$. We describe the language constructs along with the presentation of the operational semantics.

Figure 3 introduces the syntax of a machine state. The run-time system relies on two additional disjoint sets, $\mathcal{H}$ and $\mathcal{L}$. Set $\mathcal{H}$ contains *phaser names*, ranged

e	::=	*Expressions*	t	::=	*Terms*
	v	value		e	expression
	$\mid$ **let** $x = t$ **in** e	local declaration		$\mid$ **newPhaser** v	new phaser, bounded by v
v	::=	*Values*		$\mid$ **drop** v	deregister from phaser v
	x	variable		$\mid$ **arrive** v	arrive at phaser v
	$\mid n$	natural number		$\mid$ **awaitAll**	await on registered phasers
	$\mid$ ()	unit		$\mid$ **skipAll**	skip phase on all phasers
B	::=	*Bounds*		$\mid$ **async** B e	fork an activity
	$\emptyset$	empty		$\mid$ **finish** e	wait for termination
	$\mid B \uplus \{v\colon v\}$	bound			

Fig. 2. The syntax of expressions

S	::= $\langle Q; A\rangle$	*States*	A	::= $\emptyset \mid A \uplus \{l\colon a\}$	*Activity maps*
Q	::= $\emptyset \mid Q \uplus \{h\colon P\}$	*Phaser maps*	a	::= $\langle B; e\rangle \mid \langle S; B; e\rangle$	*Activities*
P	::= $\emptyset \mid P \uplus \{l\colon r\}$	*Phasers*	v	::= $\ldots \mid h$	*Values*
r	::= $\langle n; s\rangle$	*Local views*	s	::= **un** $\mid$ **ar**	*Arrival status*

Fig. 3. The syntax states

over by h. *Activity names*, l, are taken from set $\mathcal{L}$. A state S of a computation comprises a shared *phaser map* Q and an *activity map* A. Each phaser map Q stores the available phasers, mapping phaser names to phasers. A phaser maps activity names to *local views*. A local view consists of the phase n the activity is in and an arrival status s set to **ar** when the activity arrives at the phaser. An activity map A maps activity names l to activities a. There are two kinds of activities: regular activities $\langle B; e\rangle$ consist of the bound for each phaser the activity is registered with, and an expression e; finish activities $\langle S; B; e\rangle$ additionally include a state S comprising the activities spawned within a **finish** instruction. Given any map, say X, we write $\operatorname{dom} X$ for the domain of X and $\operatorname{range} X$ for the co-domain of X. In a map $X \uplus \{x\colon u\}$ we assume x does not occur in $\operatorname{dom} X$.

Figure 4 introduces a small step reduction relation on states, $S_1 \to S_2$, capturing the non-deterministic choice of which activity to evaluate next. Auxiliary functions and predicates are in Figure 5. The phaser creation instruction, **newPhaser** n, evaluates to a fresh phaser name h; it also registers under h the current activity l with a local view composed of bound n and phase 0. All other phaser-related operations evaluate to (). Activities deregister from a phaser h via an expression **drop** h, thus removing the local view from the phaser. Instruction **arrive** h marks the local view of activity l as arrived, **ar**. An activity can only **arrive** once per phase.

$$\frac{h \notin \operatorname{dom} Q}{\begin{array}{l}\langle Q \quad ; A \uplus \{l : \langle B; \mathsf{let}\ x = \mathsf{newPhaser}\ n\ \mathsf{in}\ e\rangle\}\rangle \\ \to \langle Q \uplus \{h : \{l : \langle 0; \mathsf{un}\rangle\}\} ; A \uplus \{l : \langle B \uplus \{h : n\}; \mathsf{let}\ x = h\ \mathsf{in}\ e\rangle\}\rangle\end{array}} \text{(R-PHASER)}$$

$$\begin{array}{l}\langle Q \uplus \{h : (P \uplus \{l : \_\})\} ; A \uplus \{l : \langle B \uplus \{h : \_\}; \mathsf{let}\ x = \mathsf{drop}\ h\ \mathsf{in}\ e\rangle\}\rangle \\ \to \langle Q \uplus \{h : P\} \quad ; A \uplus \{l : \langle B; \mathsf{let}\ x = ()\ \mathsf{in}\ e\rangle\}\rangle\end{array} \text{(R-DROP)}$$

$$\begin{array}{l}\langle Q \uplus \{h : (P \uplus \{l : \langle n; \mathsf{un}\rangle)\}\} ; A \uplus \{l : \langle B; \mathsf{let}\ x = \mathsf{arrive}\ h\ \mathsf{in}\ e\rangle\}\rangle \\ \to \langle Q \uplus \{h : (P \uplus \{l : \langle n; \mathsf{ar}\rangle)\}\} ; A \uplus \{l : \langle B; \mathsf{let}\ x = ()\ \mathsf{in}\ e\rangle\}\rangle\end{array} \text{(R-ARRIVE)}$$

$$\frac{\operatorname{unblocked}(Q, l, B)}{\begin{array}{l}\langle Q \quad ; A \uplus \{l : \langle B; \mathsf{let}\ x = \mathsf{awaitAll}\ \mathsf{in}\ e\rangle\}\rangle \\ \to \langle \operatorname{commit}(l, Q) ; A \uplus \{l : \langle B; \mathsf{let}\ x = ()\ \mathsf{in}\ e\rangle\}\rangle\end{array}} \text{(R-AWAIT)}$$

$$\begin{array}{l}\langle Q \quad ; A \uplus \{l : \langle B; \mathsf{let}\ x = \mathsf{skipAll}\ \mathsf{in}\ e\rangle\}\rangle \\ \to \langle \operatorname{commit}(l, Q) ; A \uplus \{l : \langle B; \mathsf{let}\ x = ()\ \mathsf{in}\ e\rangle\}\rangle\end{array} \text{(R-SKIP)}$$

$$\frac{l_2 \notin \operatorname{dom} A \cup \operatorname{dom} B_1 \cup \operatorname{dom} B_2}{\begin{array}{l}\langle Q \quad ; A \uplus \{l_1 : \langle B_1; \mathsf{let}\ x = \mathsf{async}\ B_2\ e_2\ \mathsf{in}\ e_1\rangle\}\rangle \\ \to \langle \operatorname{copy}(l_1, l_2, \operatorname{dom} B_2, Q) ; A \uplus \{l_1 : \langle B_1; \mathsf{let}\ x = ()\ \mathsf{in}\ e_1\rangle\} \uplus \{l_2 : \langle B_2; e_2\rangle\}\rangle\end{array}} \text{(R-ASYNC)}$$

$$\frac{l_2 \notin \operatorname{dom} A \cup \operatorname{dom} B}{\begin{array}{l}\langle Q; A \uplus \{l_1 : \langle B; \mathsf{let}\ x = \mathsf{finish}\ e_1\ \mathsf{in}\ e_2\rangle\}\rangle \\ \to \langle Q; A \uplus \{l_1 : \langle \langle \emptyset; \{l_2 : \langle \emptyset; e_1\rangle\}\rangle; B; \mathsf{let}\ x = ()\ \mathsf{in}\ e_2\rangle\}\rangle\end{array}} \text{(R-FINISH)}$$

$$\frac{S_1 \to S_2}{\langle Q; A \uplus \{l : \langle S_1; B; e\rangle\}\rangle \to \langle Q; A \uplus \{l : \langle S_2; B; e\rangle\}\rangle} \text{(R-ACTIVITY)}$$

$$\frac{\operatorname{halted}(S)}{\langle Q; A \uplus \{l : \langle S; B; e\rangle\}\rangle \to \langle Q; A \uplus \{l : \langle B; e\rangle\}\rangle} \text{(R-JOIN)}$$

$$\langle Q; A \uplus \{l : \langle B; \mathsf{let}\ x = v\ \mathsf{in}\ e\rangle\}\rangle \to \langle Q; A \uplus \{l : \langle B; e[v/x]\rangle\}\rangle \text{(R-LET)}$$

$$\begin{array}{l}\langle Q; A \uplus \{l : \langle B; \mathsf{let}\ x = (\mathsf{let}\ y = e_1\ \mathsf{in}\ t)\ \mathsf{in}\ e_2\rangle\}\rangle \\ \to \langle Q; A \uplus \{l : \langle B; \mathsf{let}\ y = e_1\ \mathsf{in}\ (\mathsf{let}\ x = t\ \mathsf{in}\ e_2)\rangle\}\rangle\end{array} \text{(R-UNFOLD)}$$

Fig. 4. Small step semantics for states, $S \to S$

Instruction **awaitAll** waits until activity l becomes *unblocked*. The current phase of a phaser is the natural number corresponding to the smallest local view among all activities registered in the phaser (Figure 5). An activity l is unblocked when it has arrived at all phasers *and* each bound allows progress (i.e., the bound is larger than the difference between the current phase and the phaser's phase). For each phaser activity l is registered with, rule R-AWAIT advances the phase and sets the arrival status back to **un**, via function $\operatorname{commit}(l, Q)$. HJ and X10 implicitly arrive at all non-arrived barriers before advancing.

Expression **skipAll** simply advances the phase and does not wait for other activities. This operation can be used to let an activity use a phaser repeatedly without waiting for others. Spawning a new activity with rule R-ASYNC evaluates expression e concurrently, by augmenting the activity map with a new activity $\langle B_2; e\rangle$. For each phase in bounds B_2, we copy the local views

Phase function for local views, $\text{phase}(r) = n$:

$$\text{phase}(\langle n; \mathbf{un} \rangle) = n \qquad \text{phase}(\langle n; \mathbf{ar} \rangle) = n + 1$$

Phase partial function for phasers, $\text{phase}(P) = n$:

$$\text{phase}(P) = \min\{\text{phase}(r) \mid r \in \text{range}\, P\}$$

Unblocked predicate, $\text{unblocked}(Q, l, B)$: $\qquad \text{unblocked}(Q, l, \emptyset)$

$$\frac{\text{unblocked}(Q, l, B) \qquad Q(h) = P \qquad P(l) = \langle n; \mathbf{ar} \rangle \qquad m > n - \text{phase}(P)}{\text{unblocked}(Q, l, B \uplus \{h \colon m\})}$$

Phase commit partial function, $\text{commit}(l, Q) = Q$:

$$\text{commit}(l, Q \uplus \{h \colon P \uplus \{l \colon \langle n; \mathbf{ar} \rangle\}\}) = \text{commit}(l, Q) \uplus \{h \colon P \uplus \{l \colon \langle n + 1; \mathbf{un} \rangle\}\}$$

$$\frac{l \notin \text{dom}\, P}{\text{commit}(l, Q \uplus \{h \colon P\}) = \text{commit}(l, Q) \uplus \{h \colon P\}} \qquad \text{commit}(l, \emptyset) = \emptyset$$

Local view copy partial function, $\text{copy}(l_1, l_2, H, Q) = Q$:

$$\frac{h \in H \qquad P(l_1) = r}{\text{copy}(l_1, l_2, H, Q \uplus \{h \colon P\}) = \text{copy}(l_1, l_2, H, Q) \uplus \{h \colon P \uplus \{l_2 \colon r\}\}}$$

$$\frac{h \notin H}{\text{copy}(l_1, l_2, H, Q \uplus \{h \colon P\}) = \text{copy}(l_1, l_2, H, Q) \uplus \{h \colon P\}} \qquad \text{copy}(l_1, l_2, H, \emptyset) = \emptyset$$

Halted state predicate, $\text{halted}(S)$: $\qquad \text{halted}(\langle\!\langle Q; \{l_1 \colon \langle \emptyset; v_1 \rangle, \ldots, l_n \colon \langle \emptyset; v_n \rangle\}\rangle\!\rangle)$

Fig. 5. Phaser-related functions and predicates

from the spawning activity l_1 to the spawned activity l_2, as captured by function $\text{copy}(l_1, l_2, H, Q)$, where H is a set of phaser names, defined in Figure 5. Spawned activities inherit the arrival statuses, for otherwise, depending on the order of reduction of the spawning and spawned activities, the spawned activity may or may not participate in the synchronisation of the current phase, inducing an undesirable non-determinism in the phaser semantics [12].

Rule R-FINISH evaluates expressions of the form **let** $x =$ **finish** e_1 **in** e_2 by suspending expression **let** $x = ()$ **in** e_2 and by evaluating e_1 in a newly created state. The finish activity (a triple as in Figure 3) holds a state (comprising an empty phaser map and an activity $\langle \emptyset; e_1 \rangle$ that is not registered on any phaser), the current bounds B, and the suspended expression **let** $x = ()$ **in** e_2. Such an activity will then reduce, via rule R-ACTIVITY, until halted (a predicate introduced in Figure 5). Then, the suspended activity resumes execution by means of rule R-JOIN.

The evaluation of let-expressions is standard. Rule R-LET replaces variable x by value v in continuation e. Nested let bindings are unfold with rule R-UNFOLD.

We complete this section with a pair of examples leading to deadlocks, thus motivating the need for the type system in the next section. An activity that, before terminating, does not deregister from every phaser it is registered with will

cause every activity that synchronises with that phaser to deadlock. Consider a program composed of two activities. Activity l1 creates a phaser x (line 2) and in the subsequent line spawns an activity l2 also registered with x; the latter activity does nothing (not even deregisters from phaser x). Activity l1 synchronises and deadlocks at line 5, forever waiting for activity l2 to **arrive**.

```
1  // activity l1
2  let x = newPhaser 0 in
3  async {x:0} (); // forgets drop x
4  arrive x;
5  awaitAll // l1 deadlocks here
```

Listing 1.1. An activity that forgets to deregister from a phaser.

The deadlocked state:

$$\langle\{h\colon\{l_1\colon\langle 0;\mathbf{ar}\rangle, l_2\colon\langle 0;\mathbf{un}\rangle\}\};\ \{l_1\colon\langle\{h\colon 0\};\mathbf{let}\ w=\mathbf{awaitAll}\ \mathbf{in}\ w\rangle,\ l_2\colon\langle\{h\colon 0\};()\rangle\},\}\rangle$$

An activity that forgets to **arrive** before awaiting other activities deadlocks itself and the remaining participants in the synchronisation. In the next program, activity l1 creates a phaser x and spawns activity l2 that simply arrives at x (line 4) and awaits for l1 (line 5). Activity l1 forgets to arrive at x but still awaits (line 6) and therefore deadlocks along with activity l2 (in line 5).

```
1  // activity l1
2  let x = newPhaser 0 in
3  async {x:0} ( // activity l2
4    arrive x;
5    awaitAll); // l2 deadlocks here
6  awaitAll // l1 deadlocks here
```

Listing 1.2. An activity that forgets to arrive at a phaser.

The deadlocked state:

$$\langle\{h\colon\{l_1\colon\langle 0;\mathbf{un}\rangle, l_2\colon\langle 0;\mathbf{ar}\rangle\}\};\ \{l_1\colon\langle\{h\colon 0\};\mathbf{let}\ w=\mathbf{awaitAll}\ \mathbf{in}\ w\rangle,\ l_2\colon\langle\{h\colon 0\};\mathbf{let}\ z=\mathbf{awaitAll}\ \mathbf{in}\ z\rangle\}\rangle$$

4 Type System and Results

This section introduces our type system and its main results, namely type preservation and progress for typable states.

We rely on a set of type variables, ranged over by α. The syntax of types is defined by the grammar in Figure 6, and include those for the unit constant, **unit**, for the natural numbers, **nat**, and for phasers, α. We assign a different (singleton) type α to each phaser, in order to track how phasers are used throughout the program. The type system for our programming language is also defined in Figure 6. *Typings* Γ are maps from variables and phaser names to types. *Arrival maps* Φ map type variables (singleton phaser types) into arrival status. The relation for well-formed types $\Phi \vdash \tau$ ensures that activities only make use of the phasers they are registered with.

The typing rules for bounds $\Gamma; \Phi_1 \vdash B\colon \Phi_2$ assign an arrival map Φ_2 to bounds B, under a context consisting of a typing Γ and an arrival map Φ_1.

The syntax of types:

$$\tau \ ::= \ \textbf{unit} \mid \textbf{nat} \mid \alpha$$

Well-formed types, $\Phi \vdash \tau$:

$$\Phi \vdash \textbf{unit} \qquad \Phi \vdash \textbf{nat} \qquad \Phi, \alpha\colon s \vdash \alpha \qquad \text{(T-WF-U, T-WF-N, T-WF-P)}$$

Typing rules for bounds, $\Gamma; \Phi \vdash B\colon \Phi$:

$$\frac{\Gamma; \Phi_1 \vdash B\colon \Phi_2 \quad \Gamma; \Phi_1 \vdash v_1\colon \alpha \quad \Gamma; \Phi_1 \vdash v_2\colon \textbf{nat}}{\Gamma; \Phi_1 \vdash (B \uplus \{v_1\colon v_2\})\colon (\Phi_2 \uplus \{\alpha\colon \Phi_1(\alpha)\})} \qquad \Gamma; \Phi \vdash \emptyset\colon \emptyset$$

(T-BOUND-CONS,T-BOUND-NIL)

Typing rules for values, $\Gamma; \Phi \vdash v\colon \tau$:

$$\Gamma; \Phi \vdash ()\colon \textbf{unit} \qquad \Gamma; \Phi \vdash n\colon \textbf{nat} \qquad \frac{\Gamma(v) = \tau \qquad \Phi \vdash \tau}{\Gamma; \Phi \vdash v\colon \tau} \quad \text{(T-UNIT, T-NAT, T-VAL)}$$

Typing rules for terms and for expressions, $\Gamma; \Phi \vdash t\colon (\tau, \Phi)$ and $\Gamma; \Phi \vdash e\colon (\tau, \Phi)$:

$$\frac{\Gamma; \Phi \vdash v\colon \textbf{nat} \qquad \alpha \notin \operatorname{dom} \Phi}{\Gamma; \Phi \vdash (\textbf{newPhaser}\ v)\colon (\alpha, \Phi \uplus \{\alpha\colon \textbf{un}\})} \qquad \text{(T-PHASER)}$$

$$\frac{\Gamma; \Phi \uplus \{\alpha\colon s\} \vdash v\colon \alpha}{\Gamma; \Phi \uplus \{\alpha\colon s\} \vdash (\textbf{drop}\ v)\colon (\textbf{unit}, \Phi)} \qquad \frac{\Gamma; \Phi \uplus \{\alpha\colon \textbf{un}\} \vdash v\colon \alpha}{\Gamma; \Phi \uplus \{\alpha\colon \textbf{un}\} \vdash (\textbf{arrive}\ v)\colon (\textbf{unit}, \Phi \uplus \{\alpha\colon \textbf{ar}\})}$$

(T-DROP,T-ARRIVE)

$$\Gamma; \{\alpha_1\colon \textbf{ar}, \ldots, \alpha_n\colon \textbf{ar}\} \vdash \textbf{awaitAll}\colon (\textbf{unit}, \{\alpha_1\colon \textbf{un}, \ldots, \alpha_n\colon \textbf{un}\}) \qquad \text{(T-AWAIT)}$$

$$\Gamma; \{\alpha_1\colon \textbf{ar}, \ldots, \alpha_n\colon \textbf{ar}\} \vdash \textbf{skipAll}\colon (\textbf{unit}, \{\alpha_1\colon \textbf{un}, \ldots, \alpha_n\colon \textbf{un}\}) \qquad \text{(T-SKIP)}$$

$$\frac{\Gamma; \Phi_1 \vdash B\colon \Phi_2 \qquad \Gamma; \Phi_2 \vdash e\colon (\_, \emptyset)}{\Gamma; \Phi_1 \vdash (\textbf{async}\ B\ e)\colon (\textbf{unit}, \Phi_1)} \qquad \frac{\Gamma; \emptyset \vdash e\colon (\tau, \emptyset)}{\Gamma; \Phi \vdash (\textbf{finish}\ e)\colon (\textbf{unit}, \Phi)}$$

(T-ASYNC,T-FINISH)

$$\frac{\Gamma; \Phi \vdash v\colon \tau}{\Gamma; \Phi \vdash v\colon (\tau, \Phi)} \qquad \frac{\Gamma; \Phi_1 \vdash e_1\colon (\tau_1, \Phi_2) \qquad \Gamma \uplus \{x\colon \tau_1\}; \Phi_2 \vdash e_2\colon (\tau_2, \Phi_3)}{\Gamma; \Phi_1 \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2)\colon (\tau_2, \Phi_3)}$$

(T-VALUE,T-LET)

Fig. 6. The syntax of types and the typing rules

Rule T-BOUND-CONS ensures that B maps phasers into natural numbers, and also that it holds distinct phasers. The fact that phasers are associated to singleton types enables us to track aliasing in bounds.

The typing rules for values $\Gamma; \Phi \vdash v\colon \tau$ are straightforward. Rule T-VAL asserts that the value, either a variable or a phaser name, must be in typing Γ and that its type well formed.

For expressions we define a *type and effect* system $\Gamma; \Phi_1 \vdash e\colon (\tau, \Phi_2)$ stating that expression e is of type τ and effect Φ_2. The effects are important to track the changes in the arrival map, forced by the evaluation of an expression. The type of a **newPhaser** term is a new singleton type α that is also introduced in the effect. All remaining terms are of type **unit**. The effect a **drop** v term is the incoming arrival map from which α was removed, so that value v cannot be

further used (*cf.* rule T-VAL). Rule T-ARRIVE ensures that activities can **arrive** at phaser α only once, by requiring that the phaser's arrival status is **un** and by changing it into **ar**.

Terms **awaitAll** and **skipAll** mark the end of a phase: the rules check that all phasers have arrived and then reset the phasers to **un**. For example, in line 6 of Listing 1.2, activity l1 does not **arrive** at x, so we must type the term **awaitAll** under a typing $\{x\colon \alpha\}$ and an arrival map $\{\alpha\colon \mathbf{un}\}$ which does not succeed according to rule T-AWAIT.

In rule T-ASYNC, the spawned activity e is checked against an arrival map Φ_2 for the phasers in B. Furthermore, e must deregister from all its phasers before terminating (hence the empty effect for e). The effect of the **finish** itself is the incoming phaser map, so that the spawning activity inherits the arrival status of the phasers. For example, we reject the program in Listing 1.1, since the spawned activity does not drop all its phasers before terminating. In fact, the empty task in line 3 must be typed under context $\{\mathsf{x}\colon \alpha\}$ and phaser map $\{\alpha\colon \mathbf{un}\}$, but the arrival map is not empty for the unit term ().

Rule T-FINISH also forces e to deregister from all the phasers it has created, and therefore **finish** e has no effect on the arrival map Φ. In order to avoid deadlocks, we prevent e from accessing any existing phaser, thus eliminating (nested) dependencies between phasers and **finish**. The typing rule for **let** is standard.

The typing rules for states are introduced in Figure 7. We rely on a *set of activity names* Λ, a *phase difference* map Δ mapping pairs of activity names to integer values (not necessarily natural numbers), and a *phase difference tree* map Σ mapping activity names to pairs composed of a phase difference map (for the root activity) and a phase difference tree map (for the children activities, if any). A state $\langle Q; A\rangle$ can be seen as a set of activity trees, the trees in A. Regular activities $\langle B; e\rangle$ are leaf nodes, whereas finish activities $\langle S; B; e\rangle$ are internal nodes whose children are the activities in S. When type checking a state, the topology of the phase difference tree Σ matches that of the activity tree.

We use $\Delta \vdash P$ to check that the difference of phases between activities l_i and l_j are recorded in $\Delta(l_i, l_j)$, for any pair l_i, l_j registered with P. Judgement $\Gamma \vdash_l Q\colon \Phi$ collects the arrival statuses of every phaser activity l is registered with. Judgement $\Delta; \Lambda \vdash Q\colon \Gamma$ checks the phase difference and the registered activity names for each phaser in Q, while building a context Γ for Q.

We have two rules for activities. For regular activities, rule T-ACT ensures that the bounds B and the arrival map Φ mentions the same phasers, while ensuring that expression e deregisters from all phasers when before terminating (the effect of typing the expression is the empty arrival map). For finish activities, rule T-F-ACT ensures that both the state S and the finish continuation $\langle B; e\rangle$ are well typed. To type check a state $\Delta; \Sigma \vdash \langle Q; A\rangle$, rule T-STATE uses the activity names in A and the phase difference Δ to type check the phaser map Q; it also checks that the activity map A is well typed according to the phase difference tree Σ.

Phase difference for phasers, $\Delta \vdash P$:

$$\frac{\Delta(l_i, l_j) = n_i - n_j \qquad \forall 1 \leq i, j \leq k}{\Delta \vdash \{l_1 \colon \langle n_1; \_\rangle, \ldots l_k \colon \langle n_k; \_\rangle\}} \quad \text{(T-DIF)}$$

Arrival map of a phaser map, $\Gamma \vdash_l Q \colon \Phi$:

$$\frac{\Gamma \vdash_l Q \colon \Phi \qquad \Gamma(h) = \alpha \qquad P(l) = \langle \_; s \rangle}{\Gamma \vdash_l (Q \uplus \{h \colon P\}) \colon (\Phi \uplus \{\alpha \colon s\})} \qquad \frac{\Gamma \vdash_l Q \colon \Phi \qquad l \notin \operatorname{dom} P}{\Gamma \vdash_l (Q \uplus \{\_ \colon P\}) \colon \Phi} \qquad \Gamma \vdash_l \emptyset \colon \emptyset$$

(T-AR-CONS,T-AR-SKIP,T-AR-NIL)

Typing context of a phaser map, $\Delta; \Lambda \vdash Q \colon \Gamma$:

$$\frac{\alpha \notin \operatorname{range} \Gamma \qquad \operatorname{dom} P \subseteq \Lambda \qquad \Delta \vdash P \qquad \Delta; \Lambda \vdash Q \colon \Gamma}{\Delta; \Lambda \vdash (Q \uplus \{h \colon P\}) \colon (\Gamma \uplus \{h \colon \alpha\})} \qquad \Delta; \Lambda \vdash \emptyset \colon \emptyset$$

(T-PM-CONS,T-PM-NIL)

Typing rules for activities, $\Delta; \Sigma; \Gamma; \Phi \vdash a$:

$$\frac{\Gamma; \Phi \vdash B \colon \Phi \qquad \Gamma; \Phi \vdash e \colon (\_, \emptyset)}{\emptyset; \emptyset; \Gamma; \Phi \vdash \langle B; e \rangle} \qquad \frac{\Delta; \Sigma \vdash S \qquad \emptyset; \emptyset; \Gamma; \Phi \vdash \langle B; e \rangle}{\Delta; \Sigma; \Gamma; \Phi \vdash \langle S; B; e \rangle}$$

(T-ACT,T-F-ACT)

Typing rules for activity maps, $\Sigma; \Gamma \vdash_Q A$:

$$\frac{\Gamma \vdash_l Q \colon \Phi \qquad \Delta_1; \Sigma_1; \Gamma; \Phi \vdash a \qquad \Sigma; \Gamma \vdash_Q A}{\Sigma \uplus \{l \colon \langle \Delta_1; \Sigma_1 \rangle\}; \Gamma \vdash_Q A \uplus \{l \colon a\}} \qquad \emptyset; \Gamma \vdash_Q \emptyset$$

(T-AM-CONS,T-AM-NIL)

Typing rule for states, $\Delta; \Sigma \vdash S$: $$\frac{\Delta; \operatorname{dom} A \vdash Q \colon \Gamma \qquad \Sigma; \Gamma \vdash_Q A}{\Delta; \Sigma \vdash \langle Q; A \rangle} \quad \text{(T-STATE)}$$

Fig. 7. Typing rules for states

We complete this section by presenting the main results of the paper.

Lemma 1. *If $\Sigma \uplus \{l \colon \langle \Delta_1; \Sigma_1 \rangle\}; \Gamma \vdash_Q A \uplus \{l \colon a\}$, then there exists Φ such that $\Gamma \vdash_l Q \colon \Phi$, $\Delta_1; \Sigma_1; \Gamma; \Phi \vdash a$, and $\Sigma; \Gamma \vdash_Q A$.*

Theorem 1 (Subject reduction). *If $\Delta_1; \Sigma_1 \vdash S_1$ and $S_1 \to S_2$, then there exists Δ_2 and Σ_2 such that $\Delta_2; \Sigma_2 \vdash S_2$.*

Proof (Sketch). The proof follows by induction on the derivation of the reduction step. In each case we have to exhibit Δ_2 and Σ_2. For R-PHASER we make use of a weakening lemma. Cases R-DROP, R-ARRIVE, R-AWAIT, R-SKIP, and R-ASYNC are similar. We prove that changes made in the phaser map after reduction have no effect on any activity besides the one under reduction. When the derivation of the reduction step ends with rule R-ACTIVITY, we know that $\Sigma_1 = \Sigma \uplus \{l \colon \langle \Delta_1'; \Sigma_1' \rangle\}$; by induction it follows that $\Delta_2'; \Sigma_2' \vdash S_2$. We take $\Delta_2 = \Delta_1$ and $\Sigma_2 = \Sigma \uplus \{l \colon \langle \Delta_2'; \Sigma_2' \rangle\}$. We apply Lemma 1 to the hypothesis to, and the induction hypothesis to complete the proof. Case R-JOIN, and R-UNFOLD are similar to R-ACTIVITY. For R-FINISH, we know that $\Sigma_1 = \Sigma \uplus \{l_1 \colon \langle \emptyset; \emptyset \rangle\}$; we

take $\Delta_2 = \Delta_1$ and $\Sigma_2 = \Sigma \uplus \{l_1 \colon \langle \emptyset; \{l_2 \colon \langle \emptyset; \emptyset \rangle\} \rangle\}$. We use a strengthening lemma: $\Gamma; \emptyset \vdash e \colon (\tau, \emptyset)$ and Γ only contains phaser names in its domain implies $\emptyset; \emptyset \vdash e \colon (\tau, \emptyset)$. Strengthening is applied to the newly created state. For R-LET we need a substitution lemma, strengthening and weakening.

For progress, we start by extracting a total order for the activities in a typable state.

Lemma 2. *If* $\Delta; \Sigma \vdash \langle Q; \_ \rangle$ *then the relation* $\{(l_1, l_2) \mid P \in \operatorname{range} Q, P(l_1) = \langle n_1; \_ \rangle, P(l_2) = \langle n_2; \_ \rangle, n_1 \leq n_2\}$ *is a total order.*

Theorem 2 (Progress). *If* $\Delta; \Sigma \vdash S_1$ *then* S_1 *is either* $\operatorname{halted}(S_1)$ *or there is a state* S_2 *such that* $S_1 \to S_2$.

Proof (Sketch). Activities can block for a number of reasons, easily deduced from the reduction rules in Figure 4. The cases for **drop**, **arrive**, **skipAll**, **async** and **let** are easily dismissed by a simple analysis of the typing derivation rules. For example, when the reduction step ends with rule R-ASYNC, we must show that $\Delta; \Sigma \vdash \langle Q; A \uplus \{l_1 \colon \langle B_1; \mathsf{let}\ x = \mathsf{async}\ B_2\ e_2\ \mathsf{in}\ e_1 \rangle\} \rangle$ and $l_2 \notin \operatorname{dom}(A, B_1, B_2)$ implies that $\operatorname{copy}(l_1, l_2, \operatorname{dom} B_2, Q)$ is defined. We proceed by induction on the structure of Q. The interesting case is when Q is $Q' \uplus \{h \colon P\}$, where we must show that $l_1 \in \operatorname{dom} P$. From $\Delta; \Sigma \vdash S$ we know that $\Gamma \vdash_{l_1} Q \colon \Phi$. By showing that $\Gamma \vdash_l Q \colon \Phi$ implies $\operatorname{dom} Q = \operatorname{dom} \Gamma$, we obtain $h \in \operatorname{dom} \Gamma$. Then we show that $h \in \operatorname{dom} \Gamma$ and $\Gamma \vdash_l Q \colon \Phi$ implies $l \in \operatorname{dom} Q(h)$.

Otherwise suppose that all activities in S_1 are blocked at **awaitAll**. Lemma 2 ensures that there is a total order on activity names. Since the order is total, there is one activity name that is smaller than all others; let it be l. From $\Delta; \Sigma \vdash \langle Q; A \uplus \{l \colon \langle B; \mathsf{let}\ x = \mathsf{awaitAll}\ \mathsf{in}\ e_1 \rangle\} \rangle$, we know that $P(l) = \langle m; \mathsf{ar} \rangle$, for some m. We also know that $\operatorname{phase}(P) = \min\{n + 1 \mid \langle n; \mathsf{ar} \rangle \in \operatorname{range} P\} = 1 + \min\{n \mid \langle n; \mathsf{ar} \rangle \in \operatorname{range} P\} = 1 + m$. Hence l is unblocked and we have attained a contradiction.

In absence of infinite computations, it follows from the above theorem that all typable states eventually reach a halted state.

5 Conclusion and Further Work

We presented a calculus and a type system for a fork/join programming model with a flexible phaser mechanism. We favour explicit operations, yielding phaser operators which are simpler than those of Habanero Java [17]. Our proposal unifies the semantics of clocks [7], regular phasers [17], and phaser beams [16], but goes further by allowing tasks to be ahead of others by a bounded number of phases.

HJ permits writing applications where the same task is registered with a phaser as a regular barrier and with another phaser that disregards all forms of synchronisation. For such cases, our **skipAll** operation is not enough. We can however introduce *unbounded local views* for tasks registered at phasers with an

infinite bound (tasks that do not want to synchronise, only to influence others). An expression **advance** v, available only for activities registered with an infinite bound, would advance a single phaser. We believe that such an extension can be easily accommodated in our system.

To focus on the intricacies of synchronisation, we kept our language very simple. Extensions required for real world programming include provision for unbounded computations (in the form of recursion or loops) and for a mutable store (in the form of imperative variables). Loops can be introduced in our calculus while causing little interference with the model and results, as long as they preserve an invariant on the registered phasers at every iteration. In order to build circular buffers of phasers, HJ applications may create phasers within a loop. A possible workaround to accommodate such a feature in our language is to introduce a primitive that allocates an array of phasers.

Acknowledgements. This work was partially supported by project PTDC/EIA-CCO/122547/2010. The first author would like to thank Vivek Sarkar for welcoming him at the Habanero group at Rice University, during the year of 2012. We are grateful to Jun Shirako and anonymous referees for their feedback on this paper and to Vivek Sarkar for discussions related to phasers.

References

1. Aditya, S., Stoy, J.E., Arvind: Semantics of barriers in a non-strict, implicitly-parallel language. In: Proceedings of FPCA 1995, pp. 204–215. ACM (1995)
2. Agarwal, S., Barik, R., Sarkar, V., Shyamasundar, R.K.: May-happen-in-parallel analysis of X10 programs. In: Proceedings of PPoPP 2010, pp. 183–193. ACM (2007)
3. Albrecht, J., Tuttle, C., Snoeren, A.C., Vahdat, A.: Loose synchronization for large-scale networked systems. In: Proceedings of ATEC 2006, p. 28. USENIX Association (2006)
4. Arvind, Maessen, J.-W., Nikhil, R.S., Stoy, J.E.: λ_s: an implicitly parallel λ-calculus with letrec, synchronization and side-effects. Electronic Notes Theoretical Computer Science 16(3), 265–290 (1998)
5. Barnes, F.R., Welch, P.H., Sampson, A.T.: Barrier Synchronisation for occam-pi. In: Proceedings of PDPTA 2005, pp. 173–179. CSREA Press (2005)
6. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-Java: the new adventures of old X10. In: Proceedings of PPPJ 2011, pp. 51–61. ACM (2011)
7. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of OOPSLA 2005, pp. 519–538. ACM (2005)
8. Cole, C., Williams, R.: Photoshop scalability: Keeping it simple. Queue 8, 20–28 (2010)
9. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. Computing in Science and Engineering 5(1), 46–55 (1998)
10. Lee, J.K., Palsberg, J.: Featherweight X10: a core calculus for async-finish parallelism. In: Proceedings of PPoPP 2010, pp. 25–36. ACM (2010)

11. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: Proceeding of OOPSLA 2009, pp. 227–242. ACM (2009)
12. Martins, F., Vasconcelos, V.T., Cogumbreiro, T.: Types for X10 Clocks. In: Proceedings of PLACES 2010. EPTCS, vol. 69, pp. 111–129 (2011)
13. Oracle. Java Specification Request JSR-166 (2002)
14. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media (2007)
15. Saraswat, V., Jagadeesan, R.: Concurrent clustered programming. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 353–367. Springer, Heidelberg (2005)
16. Shirako, J., Peixotto, D., Sbirlea, D., Sarkar, V.: Phaser beams: Integrating stream parallelism with task parallelism. In: X10 Workshop (2011)
17. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In: Proceedings of ICS 2008, pp. 277–288. ACM (2008)
18. Shirako, J., Sharma, K., Sarkar, V.: Unifying barrier and point-to-point synchronization in OpenMP with phasers. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 122–137. Springer, Heidelberg (2011)
19. Süß, M., Leopold, C.: Implementing irregular parallel algorithms with OpenMP. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 635–644. Springer, Heidelberg (2006)
20. Xiong, W., Park, S., Zhang, J., Zhou, Y., Ma, Z.: Ad hoc synchronization considered harmful. In: Proceedings of OSDI 2010, pp. 1–8. USENIX Association (2010)

Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions

Mario Coppo[1], Mariangiola Dezani-Ciancaglini[1], Luca Padovani[1], and Nobuko Yoshida[2]

[1] Università di Torino, Dipartimento di Informatica
[2] Imperial College London, Department of Computing

Abstract. Conventional session type systems guarantee progress within single sessions, but do not usually take into account the dependencies arising from the interleaving of simultaneously active sessions and from session delegations. As a consequence, a well-typed system may fail to have progress, even assuming that helper processes can join the system after its execution has started. In this paper we develop a static analysis technique, specified as a set of syntax-directed inference rules, that is capable of verifying whether a system of processes engaged in simultaneously active multiparty sessions has the progress property.

1 Introduction

A system of multiparty sessions has the *global progress property* if all processes in the system that are involved in ongoing sessions do not get stuck waiting for a message that is never sent and if every message sent is eventually consumed. On the one hand, this notion of progress is stronger than requiring that a non-terminated system can always reduce. For example, a system containing two processes engaged in an "infinite chatter" (like two non terminating threads which communicate with each other) does not have the progress property if some other process involved in an open session is stuck and unable to complete its own task. On the other hand, this notion of progress is weaker than requiring that all processes in the system must be able to reduce. For example, a system with an incomplete session, i.e. a session that has not been initiated and for which some participants are missing, does have the progress property if it can be completed with the missing participants to a system that has the progress property.

Communication type systems such as those introduced in [12,6] can check that processes behave correctly with respect to the protocols associated with the single sessions. The same type systems can also assure a local progress property *within the single sessions*, but they fall short in assuring the global progress property when several multiparty sessions are interleaved with each other or the communication topology of the system changes as a consequence of *delegations* across these sessions.

In previous work [6] we have defined an *interaction type system* that, when used in conjunction with the communication type system, can assure the global progress property for processes in a calculus of asynchronous multiparty sessions. The interaction type system pivots around three different typing rules for service initiations. To build the type deduction for a process, provided that one exists, it is crucial, for each service

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 45–59, 2013.

occurring in the process, to choose the right typing rule. In practice, this means that the interaction type system can be efficiently used only for *verifying* whether a given process has a given type. A naive type inference algorithm based directly on the rules of the type system would require backtracking, resulting in an exponential explosion of the search space. The contribution of the present paper is the definition of a deterministic, compositional inference algorithm which is proved to be sound and complete with respect to the interaction type system of [6]. The algorithm is presented in a "natural deduction" style, as a set of inference rules that can be evaluated in a single-pass analysis according to the structure of processes. The complexity is quadratic in the size of processes, since the application of the rules requires evaluations of linear functions. The basic idea is to devise a suitable data structure that stores the information about all the possible ways a service initiation can be typed in the interaction type system, postponing the commitment to a specific typing rule as long as possible. The inference algorithm refines the information in this data structure discarding the typing rules of service initiations that are found to be incompatible with the structure of the processes being analyzed.

In §2 we define syntax and reduction semantics of the calculus of multiparty sessions. In §3 we illustrate, through a number of smaller examples, various behavioral patterns that we want to consider and how and when these may cause deadlocks. This tutorial informally hints at the information available to the inference algorithm that helps preventing deadlocks and how such information can be inferred from the structure of processes. The inference algorithm and the data structures it uses are described in §4, which ends by showing the algorithm at work on a few examples. Related work is discussed in §5, while §6 concludes with a summary of the results and an account of ongoing and future work.

2 The Calculus of Multiparty Sessions

Syntax. We begin by fixing some notation for the following sets: *service names* are ranged over by a, b, ...; *value variables* are ranged over by x, x', ...; *identifiers*, i.e., service names and variables, are ranged over by u, w, ...; *channel variables* are ranged over by y, z, t, ...; *labels*, functioning like method selectors, are ranged over by l, l', ...; we write $\mathscr{S}$ for the set of all service names and $\mathscr{V}$ for the set of all channel variables. *Processes*, ranged over by P, Q, ..., and *expressions*, ranged over by e, e', ..., are given by the grammar in Table 1, where the syntax occurring only at runtime appears shaded.

The process $\overline{u}[\mathtt{p}](y).P$ initiates a new session through an identifier u with the other participants, each of the form $u[\mathtt{q}](y).Q_{\mathtt{q}}$ where $1 \leq \mathtt{q} \leq \mathtt{p}-1$. The (bound) variable y is the channel used for the private communications inside the session. We call $\mathtt{p}$, $\mathtt{q}$, ... (ranging over natural numbers) the *participants* of the session and we use Π, Π' to denote finite, non-empty sets of participants.

Communications that take place inside an established session are represented using the next three pairs of primitives: the sending and receiving of a value; the sending and receiving of a session channel (where the sender delegates the receiver to participate in a session by passing a channel associated with the session); selection and branching

Table 1. Calculus of multiparty sessions (syntax)

$P ::= \mathbf{0}$	Inaction	$v ::= a \mid \mathsf{true} \mid \mathsf{false}$	Value
$\mid \overline{u}[\mathtt{p}](y).P$	Service request		
$\mid u[\mathtt{p}](y).P$	Service accept	$e ::= x \mid v \mid \cdots$	Expression
$\mid c!\langle \Pi, e\rangle.P$	Send value		
$\mid c?(\mathtt{p}, x).P$	Receive value	$c ::= y \mid s[\mathtt{p}]$	Channel
$\mid c!\langle\langle \mathtt{p}, c'\rangle\rangle.P$	Send channel		
$\mid c?((\mathtt{q}, y)).P$	Receive channel	$m ::= (\mathtt{q}, \Pi, v)$	Value in transit
$\mid c \oplus \langle \Pi, l\rangle.P$	Select	$\mid (\mathtt{q}, \mathtt{p}, s[\mathtt{p}'])$	Session in transit
$\mid c\&(\mathtt{p}, \{l_i : P_i\}_{i\in I})$	Branch	$\mid (\mathtt{q}, \Pi, l)$	Label in transit
$\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Conditional		
$\mid P \mid Q$	Parallel	$h ::= \varnothing \mid h \cdot m$	Queue
$\mid (\nu a : G)P$	Restricted service		
$\mid (\nu s)P$	Restricted session		
$\mid s : h$	Named queue		

(where the former chooses one of the branches offered by the latter). All these operations specify the channel and the index of the sender or the receiver. Thus, $c!\langle \Pi, e\rangle$ sends a value on channel c to all the participants in Π, while $c?(\mathtt{p}, x)$ denotes the intention of receiving a value on channel c from the participant $\mathtt{p}$. The same holds for delegation/reception (but the receiver is only one) and for selection/branching. We write $c!\langle \mathtt{p}, e\rangle.P$ and $c \oplus \langle \mathtt{p}, l\rangle.P$ in place of $c!\langle \{\mathtt{p}\}, e\rangle.P$ and $c \oplus \langle \{\mathtt{p}\}, l\rangle.P$. An *output action* is a value sending, session sending or label selection. An *input action* is a value reception, session reception or label branching; an *input process* is a process prefixed by an input action. The service restrictions are decorated with the global types of the services. Global types describe the communication protocol followed by the session participants; we omit their syntax and refer the interested reader to [6] for the details. Conditional processes and parallel composition are standard.

Queues and channels with role are generated by the operational semantics (see Table 2). A *channel with role* is a pair $s[\mathtt{p}]$ representing the runtime endpoint of session s used by participant $\mathtt{p}$. As in [12], we model TCP-like asynchronous communications (where the message order is preserved and send actions are non-blocking) with unbounded queues of messages in a session, denoted by h. A message in a queue can be a value message $(\mathtt{q}, \Pi, v)$, indicating that the value v was sent by participant $\mathtt{q}$ to the recipients in Π; a channel message (delegation) $(\mathtt{q}, \mathtt{p}, s[\mathtt{p}'])$, indicating that $\mathtt{q}$ delegates to $\mathtt{p}$ the role of $\mathtt{p}'$ on the session s (represented by the channel with role $s[\mathtt{p}']$); and a label message $(\mathtt{q}, \Pi, l)$ (similar to a value message). By $\varnothing$ and $h \cdot m$ we respectively denote the empty queue and the queue obtained by concatenating m to the queue h. With some abuse of notation we will also write $m \cdot h$ to denote the queue with head element m. By $s : h$ we denote the queue h of the session s. In $(\nu s)P$ all occurrences of $s[\mathtt{p}]$ and the queue name s are bound.

We write $\mathsf{fs}(P)$, $\mathsf{fc}(P)$ respectively for the sets of service names and channel names occurring free in P. We define $\mathsf{fn}(P) = \mathsf{fs}(P) \cup \mathsf{fc}(P)$. A *user process* is a process which does not contain runtime syntax.

Table 2. Reduction (selected rules)

$\prod_{i=1}^{n} a[i](y).P_i \mid \overline{a}[n+1](y).P_{n+1} \to (\nu s)(\prod_{i=1}^{n} P_i\{s[i]/y\} \mid P_{n+1}\{s[n+1]/y\} \mid s : \varnothing)$	[INIT]
$s[\mathtt{p}]!\langle \Pi, e\rangle.P \mid s : h \to P \mid s : h \cdot (\mathtt{p}, \Pi, v) \;\; (e \downarrow v)$	[SEND]
$s[\mathtt{p}]!\langle\langle \mathtt{q}, s'[\mathtt{p}']\rangle\rangle.P \mid s : h \to P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}'])$	[DELEG]
$s[\mathtt{p}]\oplus\langle \Pi, l\rangle.P \mid s : h \to P \mid s : h \cdot (\mathtt{p}, \Pi, l)$	[SEL]
$s[\mathtt{p}]?(\mathtt{q}, x).P \mid s : (\mathtt{q}, \mathtt{p}, v) \cdot h \to P\{v/x\} \mid s : h$	[RCV]
$s[\mathtt{p}]?((\mathtt{q}, y)).P \mid s : (\mathtt{q}, \mathtt{p}, s'[\mathtt{p}']) \cdot h \to P\{s'[\mathtt{p}']/y\} \mid s : h$	[SRCV]
$s[\mathtt{p}]\&(\mathtt{q}, \{l_i : P_i\}_{i\in I}) \mid s : (\mathtt{q}, \mathtt{p}, l_k) \cdot h \to P_k \mid s : h \;\; (k \in I)$	[BRANCH]

Operational Semantics. The operational semantics is defined as the combination of reduction rules expressing actual computation steps and structural equivalence rules that rearrange terms so as to enable reductions. Structural equivalence is almost standard (and therefore omitted). The only peculiar rules allow rearranging the order of messages in a queue when the senders or the receivers are not the same and for splitting a message targeted to multiple recipients. Table 2 shows a selection of the relevant rules for the process reduction relation $P \to P'$. We briefly comment the rules in what follows.

Rule [INIT] describes the initiation of a new session involving $n+1$ participants that synchronize over the service name a. Here we use $\prod_{i=1}^{n} P_i$ to denote $P_1 \mid \cdots \mid P_n$. The last participant $\overline{a}[n+1](y).P_{n+1}$, distinguished by the overbar on the service name, determines the number $n+1$ of participants. After the initiation, the participants share a private session name s and the queue associated with s, which is initially empty. The variable y in each participant $\mathtt{p}$ is replaced by the corresponding channel with role $s[\mathtt{p}]$. The output rules [SEND], [DELEG], and [SEL] respectively push values, channels and labels into the queue of the session s (in rule [SEND], the side condition $e \downarrow v$ denotes the evaluation of the expression e to the value v). The input rules [RCV], [SRCV] and [BRANCH] perform the corresponding complementary operations. Note that these operations check that the sender of the message matches the expected one so that the message is actually meant for the receiver. Reduction is closed under evaluation contexts, which are special terms with holes $[\,]$ generated by the grammar below:

$$\mathscr{E} \;::=\; [\,] \;\mid\; P \;\mid\; (\nu a : G)\mathscr{E} \;\mid\; (\nu s)\mathscr{E} \;\mid\; (\mathscr{E} \mid \mathscr{E})$$

We write $\mathscr{E}[P_1, \ldots, P_n]$ for $\mathscr{E}$ where the i-th (left-to-right) hole has been filled with P_i.

The Communication Type System. The communication type system checks that processes use service names and channels according to the global types associated with them. It ensures that messages are exchanged in the right order and have the right types within sessions. The communication type system also guarantees progress within a single session, if this session is not interleaved with other sessions, but it cannot guarantee progress when multiple sessions are interleaved. We omit the specification of the communication type system because it is well understood (see [12,6] for details). In fact all processes in this paper are (assumed to be) well typed with respect to the communication type system.

Progress. Informally, we intend that a process has the progress property if each session, once started, is guaranteed to satisfy all the requested interactions. A formal definition of the progress property is not straightforward and the definition in [1] is unsatisfactory in presence of infinite computations. We explain the key ideas and problems separately.

A natural requirement for progress in the case of communication protocols is that an input process can always read a message in the expected queue and vice versa a message in a queue is always read by an input process. Hence, we must assure that any request of interaction on a session channel will always be satisfied. For instance, take the processes:

$$P_1 = a[1](y).b[1](z).y?(2,x).z!\langle 2,x\rangle \quad Q_1 = \overline{a}[2](y).\overline{b}[2](z).z?(1,x').y!\langle 1,x'\rangle$$

The problem of $P_1 \mid Q_1$ is that it reduces to a process in which the output actions of both sessions are prefixed by input actions of the other session. Indeed, $P_1 \mid Q_1$ reduces to

$$(\nu s)(\nu s')(s[1]?(2,x).s'[1]!\langle 2,x\rangle \mid s'[2]?(1,x').s[2]!\langle 1,x'\rangle)$$

where the private sessions s and s' respectively established for the a and b services have replaced the channel variables y and z in P_1 and Q_1. This configuration is stuck because the two processes are blocked mutually waiting for a message from restricted channels. Instead, the process $P_1 \mid Q_1'$ where:

$$Q_1' = \overline{a}[2](y).\overline{b}[2](z).y!\langle 1,\mathsf{true}\rangle.z?(1,x).\mathbf{0}$$

has progress and reduces to $\mathbf{0}$.

Building on Kobayashi's definition of lock-freedom [13] and on the definition of communication safety of [8] we require that each input process will always be able to receive an appropriate message along some computation and that each message in a queue will always be received by an appropriate input process along some computation. However, we must also consider that an incomplete session (i.e., without all the required participants) on service a occurring in a process P can always be allowed to start by composing P with a process containing the missing participants for a. For this reason, we use *catalyser processes* to provide the missing participants to sessions and to make sure that rule [INIT] can always be applied, so that session accept and session request prefixes are never blocking. We omit here the precise definition of catalysers which requires a number of auxiliary definitions (see [6] for the details). Intuitively, a catalyser is a parallel composition of processes where each process implements the behavior of a *single* participant. In particular, in a catalyser it is never the case that actions pertaining to different sessions are interleaved with each other in the same sequential thread. Therefore, catalysers cannot generate deadlocks.

The last notion we need before defining progress is a natural duality between input processes and message queues, which only takes into account top inputs in processes and leftmost messages in queues.

Definition 2.1 (Duality). *The* duality *between input processes and message queues is the least symmetric relation defined by:*

$$\begin{gathered} s[\mathtt{p}]?(\mathtt{q},x).P \bowtie s : (\mathtt{q},\mathtt{p},v)\cdot h \\ s[\mathtt{p}]?((\mathtt{q},y)).P \bowtie s : (\mathtt{q},\mathtt{p},s'[\mathtt{p}'])\cdot h \\ s[\mathtt{p}]\&(\mathtt{q},\{l_i : P_i\}_{i\in I}) \bowtie s : (\mathtt{q},\mathtt{p},l_k)\cdot h \qquad (k\in I) \end{gathered}$$

We are now able to define progress as follows:

Definition 2.2 (Progress). *A process P* has the progress property *if for all catalysers Q such that $P \mid Q$ is well typed in the communication system, if $P \mid Q \rightarrow^* \mathscr{E}[R]$, where R is an input process or a non-empty message queue, then there are a catalyser Q', and $\mathscr{E}', R'$ such that $\mathscr{E}[R] \mid Q' \rightarrow^* \mathscr{E}'[R, R']$ and $R \bowtie R'$.*

3 A Tutorial to Progress Inference

Service dependencies. The basic idea for preventing deadlocks is to forbid mutual dependencies between services. A dependency between two services originates when an input action pertaining to one of the services guards (hence potentially blocks) any action of the other service. A paradigmatic example of process without progress is $P_1 \mid Q_1$ that we have already examined in §2. Observe that in process P_1 we have an input action on service a that guards an output action on service b. This dependency can be recorded as the relation $a \prec b$ associated with process P_1. In process Q_1 the situation is reversed, determining $b \prec a$. If we take P_1 and Q_1 in isolation, then no circular dependency is detected. However, when considering $P_1 \mid Q_1$, the relations associated with this composition result into the circular dependency $a \prec b \prec a$.

The idea of avoiding circular dependencies between services breaks apart as soon as service names are first-class entities that can be sent as messages. When this happens, the actual dependencies between services may dynamically change as the system evolves and it might happen that a system without circular dependencies *turns into* one with circular dependencies. To illustrate the issue, consider the processes

$$P_2 = c[1](t).t?(2,x).x[1](y).b[1](z).y?(2,x').z!\langle 2,x'\rangle \qquad Q_2 = \overline{c}[2](z).z!\langle 1,a\rangle$$

and observe that Q_2 sends to P_2 the name of service a. The analysis of process P_2 may determine the relation $x \prec b$, because there is an action pertaining to service x that blocks another action pertaining to service b. However, since x is a *bound variable* in P_2, there is no obvious way to associate this dependency with P_2. On the other hand, the analysis of process Q_2 yields no apparent dependencies for a. Overall, no dependency is inferred for $P_2 \mid Q_2$, even though at runtime the system will reduce to a configuration that yields the relation $a \prec b$. Then, if $P_2 \mid Q_2$ is composed with a process that yields the inverse dependency $b \prec a$, a deadlock can occur. Indeed $P_2 \mid Q_2 \mid Q_1$ reduces to $P_1 \mid Q_1$ which leads to a deadlock, as we have seen in §2.

The idea then is to identify a class of services which do not cause deadlocks even when they are involved into circular dependencies, and to allow a service name to be sent as a message only if it refers to a service in this class. A practically relevant class of services with this property is that of *nested* ones, which are characterized by the fact that they can only be blocked by actions pertaining to nested invocations of services that are themselves nested. As an example, consider the processes

$$\begin{aligned} P_3 &= \overline{a}[2](y).y?(1,x).\overline{a}[2](z).z?(1,x').z!\langle 1,\mathsf{true}\rangle.y!\langle 1,\mathsf{false}\rangle \\ Q_3 &= a[1](y).y!\langle 2,\mathsf{false}\rangle.a[1](z).z!\langle 2,\mathsf{true}\rangle.z?(2,x').y?(2,x) \\ R_3 &= a[1](y).y!\langle 2,\mathsf{false}\rangle.a[1](z).y?(2,x).z!\langle 2,\mathsf{true}\rangle.z?(2,x') \end{aligned}$$

and observe that P_3 represents the request of two nested invocations of service a. Observe also that in P_3 there is an input action on channel z that guards an output action on channel y and that both actions pertain to the service a. As a consequence, these dependencies result in the relation $a \prec a$ that denotes a circular dependency. However, P_3 has a peculiar structure in that all the actions related to the innermost invocation of a are completely nested within the ones related to the outermost invocation of a. More generally, there is no blocking action of the outermost invocation of a that is interleaved with actions of the innermost invocation of a. In fact, this interaction structure closely resembles an ordinary function call of a sequential programming language, where a caller function is suspended until the callee has terminated. The point is that if all request and accept operations concerning service a follow this pattern (i.e., they are not interleaved with blocking actions from other sessions), then the process P_3 cannot deadlock even if its structural analysis establishes the circular dependency $a \prec a$. For example, also Q_3 gives rise to the same circular dependency, but it follows the same structure as P_3 and the composition $P_3 \mid Q_3$ is deadlock free. By contrast, in R_3 we notice that, after the innermost invocation of a has been accepted, there is an input action on y, which pertains to the outermost invocation, blocking the actions pertaining to the innermost one. Indeed, the composition $P_3 \mid R_3$ yields a deadlock.

Relative and Nested services. To promote P_3 (and Q_3) among the safe processes, we associate services with different *features* and we impose different constraints on the structure of services depending on the features they have. We say that a service that is never involved in circular dependencies with other services has the R (for Relative) feature. A service a where no action from other sessions can block the sessions initiated on a has the N (for Nested) feature. This is precisely the case of the innermost invocation of a in P_3 and Q_3. But there is more: if the innermost session cannot deadlock, it becomes "unobservable" as far as the dependency analysis is concerned so we can say that also the outermost invocation of a in P_3 and Q_3 is not blocked by actions of other sessions. As a consequence, the outermost service a has the N feature as well.

The N feature may also be used for dealing with circular dependencies between *different* services. As an example, consider the processes

$$P_4 = \overline{a}[2](y).\overline{b}[2](z).z?(1,x).y?(1,x') \qquad Q_4 = \overline{b}[2](z).\overline{a}[2](y).y?(1,x).z?(1,x')$$

representing two clients which, for unspecified reasons, request the two services a and b in different orders. If P_4 and Q_4 run within the same system, then they immediately yield the circular dependencies $a \prec b \prec a$. Still, if the processes implementing a and b (not shown here) are independent, in the sense that they do not rely on each other, then there is no danger of deadlock. The fundamental observation here is that neither service seems to have the N feature if considered in isolation: each service request is blocked by an action from the other service. However, if b is *assumed* to have the N feature, then a has the N feature also, and vice versa. In other words, the circular dependency $a \prec b \prec a$ identifies a clique of services that is safe (i.e., deadlock-free) if *every* service in the clique has the N feature under the hypothesis that all the others do as well.

In general, the same service may have both the R and the N features at the same time. This is the case of a and b in P_4 and Q_4 above *when each process is considered*

in isolation. However, note that the b service in P_1 has the R feature but not the N one, while neither a nor b has the R feature in $P_4 \mid Q_4$. This observation is crucial for the inference algorithm because the fact that a service *a does not have* a particular feature may affect other services related to a by the dependency relation. In particular, if $a \prec b$ and b does not have the R feature (hence it has the N one), then a cannot have the R feature (and it must have the N one). Dually, if a does not have the N feature, then b cannot have the N feature.

Bounded services. The next usage pattern that we wish to consider concerns private services. Take for example the process

$$b[1](y).(\nu a : 1 \to 2 : \langle \mathsf{bool} \rangle)(\overline{a}[2](z).z?(1,x).y!\langle 2, \mathsf{false} \rangle)$$

where the a service has been restricted and is therefore inaccessible from the outside. Even if the a service has both the R and N features, the fact that it is restricted makes it observably equivalent to the idle process. This has severe consequences on the outer service b, because the output action on channel y cannot be executed. In essence, we devise a third feature B (for Bounded) associated with services that can be restricted and that prevents them to be followed by *any* communication action on free channels.

Wrap up. To summarize, when we analyze a system of interleaved multiparty sessions we associate services in the system with (a combination of) three features R, N, and B:

- A service has the R feature if it never generates circular dependencies with other services it is interleaved with.
- A service has the N feature if it is never interleaved with blocking actions from other services not having the N feature.
- Finally, a service has the B feature if it has the N feature and it is never followed by any action on free communication channels.

Overall there are eight feature combinations. One of these corresponds to the fact that a service has none of the features outlined above. In this case, the service will be rejected by our system as being ill typed. Furthermore, having the B feature implies having the N feature. Therefore, each well-typed service may be classified into one of five feature combinations. Note that, in the informal definitions above, "never" means both "for no occurrence of the service in the system" and "at any time during the evolution of the system". The inference algorithm has to find a trade off between flexibility (the number of systems for which progress can be guaranteed) and feasibility (the analysis is solely based on the initial state of the system). In fact, when discussing first-class service names we have already seen a case in which the algorithm is forced to act conservatively due to the lack of precise information about the runtime evolution of a system.

The inference of the progress property performs an analysis on the structure of processes, keeping track of the dependencies between services and incrementally refining the features associated with services, making sure that each service has at least one of the features described above. Initially, each service has every feature. As the analysis proceeds bottom up on the structure of processes, features are removed from services that are found to be incompatible with them. In a nutshell, the most relevant refinement steps taken by the algorithm occur at the following events:

- As soon as a circular dependency is detected, all processes involved in the circularity (and those preceding them in the dependency relation) lose the R feature.
- When a process of shape $\tilde{a}[\mathtt{p}](y).P$ is encountered, where $\tilde{a}$ is either an accept action a or a request action $\overline{a}$, a loses the N feature if it is not minimal in the dependency relation (meaning that it may be blocked by another session of a service not having the N feature). Also, a loses the B feature if P has free channels other than y.
- When a process of shape $y?(\mathtt{p},x).P$ is encountered, the dependencies are enriched with relations $y \prec z$ for every channel z that occurs free in P. The same happens for session receives and branching processes, since these are all blocking actions.
- When a process of shape $P \mid Q$ is encountered, the dependencies computed for P and those computed for Q are merged together, while the features for every service in the overall process are those in common between P and Q. Similar operations are performed when analyzing branching and conditional processes, where multiple processes come together.
- When a process of shape $y!\langle \mathtt{p}, a\rangle.P$ is encountered, the service a loses the R feature.
- Special measures must be taken when channels are communicated. These will be detailed shortly.

The next section is devoted to formalizing all the concepts and procedures outlined in this tutorial.

4 Progress Inference

In this section we introduce a deterministic, compositional type inference algorithm, defined via a set natural semantics rules, assuring that a given *user* process has the progress property. As we have anticipated in §3, the basic idea of the inference algorithm is to keep track of dependencies between services.

A *service qualifier* is either a service name a or a channel variable y; we write $\Lambda = \mathscr{S} \cup \mathscr{V}$ for the set of all service qualifiers; we let λ range over elements of Λ and $\mathscr{L}$ over subsets of Λ.

A *dependency relation* is a transitive relation $\mathsf{D} \subseteq \Lambda \times \Lambda$. We denote with $\lambda \prec \lambda'$ the elements of $\Lambda \times \Lambda$. The meaning of $\lambda \prec \lambda'$ is, roughly, that an input action on the channel (or on the channel bound by service) λ can block a communication action on the channel (or on the channel bound by service) λ'.

The inference algorithm makes use of some auxiliary operators for D that are introduced below:

- $\mathsf{D} \downarrow \lambda \stackrel{\text{def}}{=} \{\lambda\} \cup \{\lambda' \mid \lambda' \prec \lambda \in \mathsf{D}\}$ is the set of elements that are smaller than or equal to λ in D, namely the set of service qualifiers having an input action that can block a communication action on λ, plus λ itself.
- $\mathsf{D} \uparrow \lambda \stackrel{\text{def}}{=} \{\lambda\} \cup \{\lambda' \mid \lambda \prec \lambda' \in \mathsf{D}\}$ is the symmetric operation that determines the set of service qualifiers that may be blocked by an input action on λ, plus λ itself.
- $\mathsf{D} \setminus \mathscr{L} \stackrel{\text{def}}{=} \{\lambda \prec \lambda' \in \mathsf{D} \mid \lambda \notin \mathscr{L} \wedge \lambda' \notin \mathscr{L}\}$ is the subset of D pertaining to all the service qualifiers not occurring in $\mathscr{L}$.
- $\mathsf{D}^{\infty} \stackrel{\text{def}}{=} \{\lambda \mid \lambda \prec \lambda \in \mathsf{D}\}$ is the set of service qualifiers involved in circular dependencies in D.

Table 3. Inference algorithm for the interaction type system

$$\{\textsc{Inact-I}\}\quad \mathbf{0} \Mapsto \emptyset;\mathscr{S};\mathscr{S};\mathscr{S}$$

$$\{\textsc{Init*-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}{\tilde{a}[\mathfrak{p}](y).P \Mapsto \mathfrak{F}(\mathsf{D}\{a/y\}^{+},\mathsf{R},\mathsf{N},\mathsf{B}\setminus\{a \mid \mathsf{fc}(P) \not\subseteq \{y\}\})}$$

$$\{\textsc{InitV-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B} \qquad \mathsf{fc}(P) \subseteq \{y\}}{\tilde{x}[\mathfrak{p}](y).P \Mapsto \mathfrak{F}(\mathsf{D}\setminus\{y\},\mathsf{R}\setminus(\mathsf{D}\downarrow y),\mathsf{N},\mathsf{B})}$$

$$\{\textsc{NRes-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B} \qquad a \in \mathsf{B}}{(\nu a : G)P \Mapsto \mathsf{D}\setminus\{a\};\mathsf{R}\setminus\{a\};\mathsf{N}\setminus\{a\};\mathsf{B}\setminus\{a\}}$$

$$\{\textsc{Send-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}{y!\langle\Pi,e\rangle.P \Mapsto \mathfrak{F}(\mathsf{D},\mathsf{R}\setminus\{e\},\mathsf{N},\mathsf{B})}$$

$$\{\textsc{Rcv-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}{y?(\mathfrak{q},x).P \Mapsto (\mathsf{pre}(y,\mathsf{fc}(P))\cup\mathsf{D})^{+};\mathsf{R};\mathsf{N};\mathsf{B}}$$

$$\{\textsc{Deleg-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}{y!\langle\langle\mathfrak{p},z\rangle\rangle.P \Mapsto (\{y \prec z\}\cup\mathsf{D})^{+};\mathsf{R};\mathsf{N};\mathsf{B}}$$

$$\{\textsc{SRcv-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B} \qquad \mathsf{D}\setminus\mathscr{S} \subseteq \{y \prec z\}}{y?((\mathfrak{q},z)).P \Mapsto \mathsf{D}\setminus\{z\};\mathsf{R};\mathsf{N};\mathsf{B}}$$

$$\{\textsc{Sel-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}{y\oplus\langle\Pi,l\rangle.P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}$$

$$\{\textsc{Branch-I}\}\quad \frac{P_i \Mapsto \mathsf{D}_i;\mathsf{R}_i;\mathsf{N}_i;\mathsf{B}_i \ {}^{(i\in I)} \qquad \mathsf{D} = (\mathsf{pre}(y,\bigcup_{i\in I}\mathsf{fc}(P_i))\cup\bigcup_{i\in I}\mathsf{D}_i)^{+}}{y\&(\mathfrak{p},\{l_i : P_i\}_{i\in I}) \Mapsto \mathfrak{F}(\mathsf{D},\bigcap_{i\in I}\mathsf{R}_i,\bigcap_{i\in I}\mathsf{N}_i,\bigcap_{i\in I}\mathsf{B}_i)}$$

$$\{\textsc{Par-I}\}\quad \frac{P_i \Mapsto \mathsf{D}_i;\mathsf{R}_i;\mathsf{N}_i;\mathsf{B}_i \ {}^{(i=1,2)} \qquad \mathsf{D} = (\mathsf{D}_1\cup\mathsf{D}_2)^{+}}{P_1 \mid P_2 \Mapsto \mathfrak{F}(\mathsf{D},\mathsf{R}_1\cap\mathsf{R}_2,\mathsf{N}_1\cap\mathsf{N}_2,\mathsf{B}_1\cap\mathsf{B}_2)}$$

$$\{\textsc{If-I}\}\quad \frac{P_i \Mapsto \mathsf{D}_i;\mathsf{R}_i;\mathsf{N}_i;\mathsf{B}_i \ {}^{(i=1,2)} \qquad \mathsf{D} = (\mathsf{D}_1\cup\mathsf{D}_2)^{+}}{\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \Mapsto \mathfrak{F}(\mathsf{D},\mathsf{R}_1\cap\mathsf{R}_2,\mathsf{N}_1\cap\mathsf{N}_2,\mathsf{B}_1\cap\mathsf{B}_2)}$$

We extend $\downarrow$ and $\uparrow$ to sets $\mathscr{L}$ of service qualifiers in the natural way. We also write $\mathsf{D}\{a/y\}$ for the relation obtained from D where every occurrence of y has been replaced by a and $\mathfrak{R}^{+}$ for the transitive closure of a generic relation $\mathfrak{R}$.

The inference rules prove judgments of the form $P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}$, where D is a dependency relation and R, N, and B are sets of service names. As a first approximation, we can think of the services in these sets as those that respectively have the R, N, and B feature. However, for services that are communicated in messages it is not easy to *statically* guarantee that they will not be involved in a circular dependency *at runtime*. Therefore, we conservatively remove communicated services from the R set even if they are not explicitly involved in circular dependencies.

A judgment $P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}$ is *well formed* if:

1. If a service a has the R feature, then all the services following a in D have R feature. Also, no service involved in a circular dependency can have the R feature. This is formally expressed as $\mathsf{D}\uparrow\mathsf{R}\subseteq\mathsf{R}\setminus\mathsf{D}^{\infty}\cup\mathscr{V}$.

2. If a service a has the N feature, then all service qualifiers preceding a in D must be services with the N feature. That is, $\mathsf{D} \downarrow \mathsf{N} \subseteq \mathsf{N}$.
3. The set of services having the B feature is included in those having the N feature. That is, $\mathsf{B} \subseteq \mathsf{N}$.
4. All services occurring free in P have at least the R or the N feature. If some service in P has neither the R nor the N feature, then our inference algorithm does not guarantee the progress property for P. That is, $\mathsf{fs}(P) \subseteq \mathsf{R} \cup \mathsf{N}$.

In general, the inference rules add dependencies to the D relation and remove service names from the R, N, B sets when these services lose features. To be sure that the quadruple resulting from the application of an inference rule still satisfies the conditions (1–3) above, we define a function $\mathfrak{F}$ that, given a quadruple $\mathsf{D}, \mathsf{R}, \mathsf{N}, \mathsf{B}$, computes a new one where services are removed from the sets R, N, B whenever they are found to be incompatible with the corresponding feature:

$$\mathfrak{F}(\mathsf{D}, \mathsf{R}, \mathsf{N}, \mathsf{B}) \stackrel{\text{def}}{=} \mathsf{D}; \mathsf{R}'; \mathsf{N}'; \mathsf{B} \cap \mathsf{N}'$$

where $\mathsf{R}' = \{a \in \mathsf{R} \mid \mathsf{D} \uparrow a \subseteq \mathsf{R} \setminus \mathsf{D}^{\infty} \cup \mathscr{V}\}$ and $\mathsf{N}' = \{a \in \mathsf{N} \mid \mathsf{D} \downarrow a \subseteq \mathsf{N}\}$.

Table 3 defines the inference for the interaction type system. We implicitly assume that an inference rule can be applied only if the judgment in the conclusion is well formed. In the next paragraphs we describe each inference rule in detail.

{INACT-I} is by far the simplest inference rule, which yields no dependencies and poses no constraints on the features of services. In particular, D is $\emptyset$ and the R, N, and B components are the full set $\mathscr{S}$ of service names.

{INIT*-I} is used for typing accept and request operations on a known service name a (recall that we use $\tilde{a}$ for either a or $\overline{a}$). The rule computes a new quadruple $\mathfrak{F}(\mathsf{D}\{a/y\}^{+}, \mathsf{R}, \mathsf{N}, \mathsf{B} \setminus \{a \mid \mathsf{fc}(P) \setminus \{y\} \neq \emptyset\})$ from the one obtained by typing the continuation process P, where $\mathsf{D}\{a/y\}^{+}$ replaces the channel variable y with a in D so that all the dependencies already established for a are enriched with those computed for y. Also, a loses the B feature if P contains free channels other than y.

{INITV-I} is analogous to {INIT*-I}, but considers the case in which the session is initiated on an unknown service x. Because nothing is known on the service a that will replace x at runtime, the rule acts conservatively assuming that a has both the N and the B features. In particular, the continuation process P is required to have no free channel other than y (this is necessary if a has the B feature) and all services preceding y in D lose the R feature (this is necessary if a has the N feature but not the R one). Note that it is not possible to keep track, in D, of all the dependencies related to y as we did in {INIT*-I}. In fact, any dependency related to y in D is removed. This may prevent the inference algorithm from statically detecting circular dependencies for services that are communicated in messages. For this reason, we will require that all service names communicated by rule {SEND-I} must have the N feature (Example 4.2 shows that this is necessary for communicated services to prevent deadlocks).

When a service name a is restricted in a process P, rule {NRES-I} checks that a has the B feature. Then, all dependencies related to a and a itself are removed from all the components of the quadruple in the conclusion of the rule.

Rules {SEND-I} and {SEL-I} do not change the dependency relation because send operations are non-blocking. In the case of {SEND-I}, however, we must check that if

the message sent e is a service name, then it cannot have the R feature. The application of the $\mathfrak{F}$ function makes sure that all the components of the quadruple remain consistent after this removal.

Rule {RCV-I} is used for typing value receptions. In this case, only the dependency relation is changed to record the fact that the input action on channel y may block subsequent actions on the free channels occurring in P. The function $\mathsf{pre}(y, \mathsf{fc}(P))$ creates the dependency relation that contains the pairs $y \prec z$ for all $z \in \mathsf{fc}(P)$. Note that no dependency is recorded between y and the free service names possibly occurring in P. This is because these services can always be unblocked by adding suitable catalysers (see Definition 2.2) provided that the communication occurring on y does not reach a deadlock.

Rule {BRANCH-I} is a natural generalization of rule {RCV-I} to a process with multiple branches. In this case, the dependencies inferred for each branch are merged together and services lose those features that are not present in every branch.

Rule {DELEG-I} is similar to {SEND-I} and {SEL-I} in that it deals with a non-blocking send operation. However, in this case the process is sending a channel variable z over channel y, meaning that an action blocking a communication on y may also block a communication on z, because z cannot be used by the receiver process until delegation happens. Consequently, the dependency relation is enriched with the $y \prec z$ dependency.

Rule {SRCV-I} is similar to {RCV-I}, except that it is used for typing the reception of a session channel. The rule is particularly restrictive because it is meant to prevent a dangerous phenomenon called self-delegation, which happens when one process ends up owning two (or more) endpoints of the same session. An example of this phenomenon is shown in the processes

$$P_5 = b[1](z).a[1](y).y!\langle\langle 2, z\rangle\rangle \qquad Q_5 = \overline{b}[2](z).\overline{a}[2](y).y?((1,x)).x?(2,w).z!\langle 1, \mathsf{false}\rangle$$

which, when executed in parallel, open two sessions on services a and b. Then, P_5 sends the channel z related to the session on b over the channel y, which is related to the session on a. At this point, Q_5 owns both endpoints of the session on b and tries to use them in an order that causes a deadlock. Indeed, $P_5 \mid Q_5$ reduces to

$$(\nu s)(s[1]?(2,w).s[2]!\langle 1, \mathsf{false}\rangle)$$

which is stuck. Remarkably, the process $P_5 \mid Q_5$ is typable in the communication type system hence it is the interaction type system that must detect the problem in this case. The premise $\mathsf{D} \setminus \mathscr{S} \subseteq \{y \prec z\}$ requires that the continuation process P_5 cannot perform any potentially blocking action on any channel other than y, and that if a potentially blocking action is performed on y then it must necessarily block a communication action on z. This restriction prevents self-delegation and, in general, suffices to guarantee progress. Note that P_5 is still allowed to open new sessions on other services.

{PAR-I} and {IF-I} conclude the inference system by suitably combining dependencies and features, similarly to what we have already seen for the {BRANCH-I} rule.

The algorithm is quadratic in the size of processes, being defined on their structure, if we use appropriate data structures to represent the dependency relation and the service sets, getting linear complexity for the evaluation of the required functions.

The algorithm is sound, namely:

Theorem 4.1. *If* $P \Rightarrow \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}$*, then* P *has the progress property.*

This theorem can be proved by showing that the inference algorithm is sound and complete with respect to the interaction type system defined in [6].

We end with the application of the inference algorithm on two examples used earlier.

Example 4.1. Below are two executions of the inference algorithm on P_1 and Q_1 of §2. For the sake of readability, we develop the inference bottom up assuming $\mathscr{S} = \{a,b\}$.

P_1	D	R	N	B	
$z!\langle 2,x\rangle$	$\emptyset$	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	{SEND-I}
$y?(2,x)$	$\{y \prec z\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	{RCV-I}
$b[1](z)$	$\{y \prec b\}$	$\{a,b\}$	$\{a\}$	$\{a\}$	{INIT*-I}
$a[1](y)$	$\{a \prec b\}$	$\{a,b\}$	$\{a\}$	$\{a\}$	{INIT*-I}

Q_1	D	R	N	B	
$y!\langle 1,x'\rangle$	$\emptyset$	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	{SEND-I}
$z?(1,x')$	$\{z \prec y\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	{RCV-I}
$\overline{b}[2](z)$	$\{b \prec y\}$	$\{a,b\}$	$\{a,b\}$	$\{a\}$	{INIT*-I}
$\overline{a}[2](y)$	$\{b \prec a\}$	$\{a,b\}$	$\{a,b\}$	$\{a\}$	{INIT*-I}

From the above table it turns out that both P_1 and Q_1 are well typed in isolation, in particular we have $P_1 \Vdash \{a \prec b\};\{a,b\};\{a\};\{a\}$ and $Q_1 \Vdash \{b \prec a\};\{a,b\};\{a,b\};\{a\}$ but the application of rule {PAR-I} fails since $\mathfrak{F}(\mathsf{D},\{a,b\},\{a\},\{a\}) = (\mathsf{D},\emptyset,\emptyset,\emptyset)$ where $\mathsf{D} = \{a \prec b, b \prec a\}^+$, and the resulting judgment would not satisfy condition 4 of the definition of well formedness. In particular the circular dependency removes the R feature from both a and b and the N feature is removed from b in P_1 and then also from a in the composition $P_1 \mid Q_1$ because of $b \prec a$ (see the definition of $\mathfrak{F}$). ■

Example 4.2. The inference algorithm is not always able to statically determine a violation of the R feature, therefore it is unsafe to leave service names that are sent as messages in the R set. Below is the result of the inference algorithm on the processes P_2 and Q_2 of §3 assuming $\mathscr{S} = \{a,b,c\}$:

P_2	D	R	N	B	
$z!\langle 2,x'\rangle$	$\emptyset$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	{SEND-I}
$y?(2,x')$	$\{y \prec z\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	{RCV-I}
$b[1](z)$	$\{y \prec b\}$	$\{a,b,c\}$	$\{a,c\}$	$\{a,c\}$	{INIT*-I}
$x[1](y)$	$\emptyset$	$\{a,b,c\}$	$\{a,c\}$	$\{a,c\}$	{INITV-I}
$t?(2,x)$	$\emptyset$	$\{a,b,c\}$	$\{a,c\}$	$\{a,c\}$	{RCV-I}
$c[1](t)$	$\emptyset$	$\{a,b,c\}$	$\{a,c\}$	$\{a,c\}$	{INIT*-I}

Q_2	D	R	N	B	
$t!\langle 1,a\rangle$	$\emptyset$	$\{b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	{SEND-I}
$\overline{c}[2](t)$	$\emptyset$	$\{b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	{INIT*-I}

Note that the dependency $y \prec b$ in P_2 is erased because it concerns an unknown service x that is bound in P_2. This means that b is actually involved in dependencies $a \prec b$ for every service a that is sent to P_2, which is precisely what Q_2 does. Indeed we have $P_2 \mid Q_2 \Vdash \emptyset;\{b,c\};\{a,c\};\{a,c\}$ but $P_2 \mid Q_2 \mid Q_1$, where Q_1 is defined in Example 4.1, cannot be typed. In fact, adding c to the set of services we get immediately $Q_1 \Vdash \{b \prec a\};\{a,b,c\};\{a,b,c\};\{a,c\}$ but rule {PAR-I} cannot be applied since $\mathfrak{F}(\{b \prec a\},\{b,c\},\{a,c\},\{a,c\}) = (\{b \prec a\},\{b,c\},\{c\},\{c\})$ does not satisfy the condition 4 of the definition of well-formedness for service a. Indeed we have the reduction $P_2 \mid Q_2 \mid Q_1 \rightarrow^* P_1 \mid Q_1$ which leads to a deadlock, as we have seen in §3. ■

5 Related Work

Our notion of progress is strongly related to, and partly inspired from, the notion of *lock-freedom* in [13], where Kobayashi develops a type system to enforce it. Intuitively,

a process is lock-free if, no matter how it reduces, every top-level prefix can be eventually consumed. In our case this roughly corresponds to the property that no process gets stuck on an input action and that every message in a queue can be received. Kobayashi's type system seems capable of a much more fine-grained analysis than our type system. However, despite the similarities between progress and lock-freedom, the two type systems are difficult to compare, because of several major differences in both processes and types. In addition to the fact that we consider progress modulo the availability of catalysers, our type system is given for an asynchronous language with a native notion of (multiparty) session, while Kobayashi's type system is defined for a basic variant of the synchronous, pure π-calculus. A natural way for comparing these analysis techniques would require compiling a session-based process into the pure π-calculus [7], and then using Kobayashi's type system for reasoning on progress of the original process in terms of lock-freedom of the one resulting from the compilation.

A strategy that is alternative to compiling/encoding session-based processes is to lift the technique underlying Kobayashi's type system to a session type system for reasoning directly on the progress properties of processes. Some preliminary experiments in this sense are reported in [14].

Most papers on service-oriented calculi only assure that clients are never stuck inside a *single* session [12,9,8]. The first papers considering progress for interleaved sessions required the nesting of sessions in Java [11,5].

The papers more related to the present one are [10] and [3]. In both these papers there are constructions of processes providing missing participants, which are simpler than our catalysers since sessions are dyadic.

[2] proposes a sophisticated proof system which builds a well-founded ordering on events to enforce progress for processes of the Conversation Calculus [15], also in presence of dynamic join and leave of participants. Their progress is guaranteed under the assumption that all communications are matched with sufficient joiners.

Formal theories of contracts using multiparty interaction structures are studied in [4]. Contracts record the overall behaviour of a process, and typable processes themselves may not always satisfy properties such as progress: it is proved *later* by checking whether a whole contract satisfies a certain form. Proving properties with contracts requires an exploration of all possible interleaved or non-deterministic paths of a protocol.

6 Conclusions and Future Work

We have presented a sound and complete inference algorithm for the interaction type system defined in [6] restricted to finite processes. This system guarantees progress of interleaved multiparty sessions with session delegation and service communication.

There is a number of extensions stemming from this work, we focus on two of them. First of all, it appears that the algorithm can be easily adapted to deal with recursive processes, although soundness and completeness of such extension remain to be formally established. Second, we plan to investigate how the approach can be applied to concrete programming languages. The point is that the inference algorithm (and the interaction type system as well) makes the fundamental assumption that a process can be examined in terms of the complete sequence of input/output operations it performs. In practice,

programs are made of opaque structures (higher-order functions, methods, modules, etc.) and it is currently unclear whether such structures can be faithfully encoded as processes in our calculus, or if instead it is necessary to devise richer type constructs to describe them and to reason on global progress of systems in a modular way.

Acknowledgments. The authors are grateful to the reviewers for their useful comments and to Naoki Kobayashi for discussions on the notion of lock-freedom. This work was partially supported by EPSRC EP/G015635/1 and EP/K011715/1, NSF Ocean Observatories Initiative, MIUR Project CINA and Ateneo/CSP Project SALT.

References

1. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global Progress in Dynamically Interleaved Multiparty Sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
2. Caires, L., Vieira, H.T.: Conversation types. Theoretical Computer Science 411(51-52), 4399–4440 (2010)
3. Carbone, M., Debois, S.: A graphical approach to progress for structured communication in web services. In: Bliudze, S., Bruni, R., Grohmann, D., Silva, A. (eds.) ICE 2010. EPTCS, vol. 38, pp. 13–27 (2010)
4. Castagna, G., Padovani, L.: Contracts for Mobile Processes. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 211–228. Springer, Heidelberg (2009)
5. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N.: Asynchronous Session Types and Progress for Object Oriented Languages. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 1–31. Springer, Heidelberg (2007)
6. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global Progress for Dynamically Interleaved Multiparty Sessions. Mathematical Structures in Computer Science (to appear)
7. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. In: De Schreye, D., Janssens, G., King, A. (eds.) PPDP 2012, pp. 139–150. ACM Press (2012)
8. Deniélou, P.-M., Yoshida, N.: Dynamic Multirole Session Types. In: Ball, T., Sagiv, M. (eds.) POPL 2011, pp. 435–446. ACM Press (2011)
9. Dezani-Ciancaglini, M., de'Liguoro, U.: Sessions and Session Types: an Overview. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 1–28. Springer, Heidelberg (2010)
10. Dezani-Ciancaglini, M., de'Liguoro, U., Yoshida, N.: On Progress for Structured Communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 257–275. Springer, Heidelberg (2008)
11. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session Types for Object-Oriented Languages. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 328–352. Springer, Heidelberg (2006)
12. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: Necula, G.C., Wadler, P. (eds.) POPL 2008, pp. 273–284. ACM Press (2008)
13. Kobayashi, N.: A Type System for Lock-Free Processes. Information and Computation 177, 122–159 (2002)
14. Padovani, L.: From Lock Freedom to Progress Using Session Types. In: Yoshida, N., Vanderbauwhede, W. (eds.) PLACES (to appear, 2013)
15. Vieira, H.T., Caires, L., Seco, J.C.: The Conversation Calculus: A Model of Service-Oriented Computation. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 269–283. Springer, Heidelberg (2008)

Pattern Matching and Bisimulation

Thomas Given-Wilson[1] and Daniele Gorla[2]

[1] NICTA, Sydney, Australia*
[2] Dip. di Informatica, "Sapienza" Università di Roma

Abstract. Concurrent Pattern Calculus (CPC) is a minimal calculus whose communication mechanism is based on a powerful form of symmetric pattern unification. However, the richness of patterns and their unification entails some flexibility in the challenge-reply game that underpins bisimulation. This leads to an ordering upon patterns that is used to define the valid replies to a given challenge. Such a theory can be smoothly adapted to accomplish other, less symmetric, forms of pattern matching (e.g. those of Linda, polyadic π-calculus, and π-calculus with polyadic synchronization) without compromising the coincidence of the two equivalences.

1 Introduction

Concurrent Pattern Calculus [20] is a minimal process calculus that uses symmetric pattern unification as the basis of communication. CPC's expressive power is obtained by extending the messages sent during *interaction* from traditional *names* to a class of *patterns* that are *unified* in an *intensional* manner (i.e., inspecting their internal structure). This unification supports equality testing and bi-directional communication in an atomic step.

The exploration of intensionality in the concurrent setting is inspired by the increased expressive power that the intensional SF-calculus has over λ-calculus [23]. Since intensionality, as captured by pattern matching, is more expressive in sequential computation, it is natural to explore the expressiveness of intensionality, as captured by pattern unification, in concurrent computation. Indeed, CPC formally generalises both the sequential intensional computation of SF-calculus and the traditional (non-intensional) concurrent computation of π-calculus [18]. The expressive power of CPC is also testified to by the possibility of encoding some well-known process languages [20,18]: π-calculus [26], Linda [16] and Spi-calculus [2]. CPC's symmetric form of communication has similarities to Fusion [28]; however, the two calculi are unrelated (neither one can be encoded in the other) [20,18]. Finally, CPC has been implemented in [17].

The main features of CPC are illustrated in the following sample trade interaction:

$$\begin{array}{rl} & (\nu\ sharesID)\ulcorner ABCShares\urcorner \bullet sharesID \bullet \lambda x \rightarrow \langle charge\ x\ for\ sale\rangle \\ | & (\nu\ bankAcc)\ulcorner ABCShares\urcorner \bullet \lambda y \bullet bankAcc \rightarrow \langle save\ y\ as\ proof\rangle \end{array}$$

$$\longmapsto (\nu\ sharesID)(\nu\ bankAcc)(\langle charge\ bankAcc\ for\ sale\rangle \mid \langle save\ sharesID\ as\ proof\rangle)$$

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 60–74, 2013.

The first line models a seller that will synchronise with a buyer, using the protected information *ABCShares*, and exchange its shares (*sharesID*) for bank account information to charge (bound to x). The second line models a buyer. Notice that the information exchange is bidirectional and simultaneous: *sharesID* replaces y in the (continuation of the) buyer and *bankAcc* replaces x in the (continuation of the) seller. Moreover, the two patterns $\ulcorner ABCShares \urcorner \bullet sharesID \bullet \lambda x$ and $\ulcorner ABCShares \urcorner \bullet \lambda y \bullet bankAcc$ also specify the details of the shares being traded, that must be matched for equality in the pattern matching as indicated by the syntax $\ulcorner \cdot \urcorner$.

Pattern unification in CPC is even richer than indicated in this example, as unification may bind a compound pattern to a single name; that is, patterns do not need to be fully decomposed in unification. For example, the bank account information could be specified, and matched upon, in much more detail. The buyer could provide the account name and number such as in the following pattern: $(\nu\ accName)(\nu\ accNum)\ulcorner ABCShares \urcorner \bullet \lambda y \bullet (name \bullet accName \bullet number \bullet accNum)$. This more detailed buyer would still match against the seller, now yielding $\langle charge\ name \bullet accName \bullet number \bullet accNum\ for\ sale \rangle$. Indeed, the seller could also specify a desire to only accept bank account information that includes a name and number with the following pattern: $\ulcorner ABCShares \urcorner \bullet sharesID \bullet (\ulcorner name \urcorner \bullet \lambda a \bullet \ulcorner number \urcorner \bullet \lambda b)$ and continuation $\langle charge\ a\ b\ for\ sale \rangle$. This would also match with the detailed buyer information by unifying *name* with $\ulcorner name \urcorner$, *number* with $\ulcorner number \urcorner$, and binding *accName* and *accNum* to a and b respectively. The second seller exploits the intensionality of CPC to only interact with a buyer whose pattern is of the right structure (four sub-patterns) and contains the right information (the protected names *name* and *number*, and shared information in the other two positions). CPC is built up around this rich form of pattern unification by using three standard operators taken from the π-calculus: name restriction, parallel composition and replication (not used in this simple example).

The focus of this paper is the investigation of the behavioural theory for CPC. As usual in concurrency theory, this is done by first defining a notion of barbed congruence and then capturing this via a labelled bisimulation-based equivalence. The main difficulty relies in the richness of the pattern unification mechanism adopted, that entails some flexibility in the challenge-reply game underlying the definition of the bisimulation. For example, the challenge $\lambda x \bullet \lambda y$ can be replied to by λz, because of the non-fully decomposing form of pattern matching. (Such as the seller who accepts anything as bank account information.) Indeed, every pattern matching the challenge has the form $p \bullet q$, where p and q are communicable (i.e., they do not contain protected names $\ulcorner n \urcorner$ nor binding names λw), yielding the substitution $\{p/x, q/y\}$. The same pattern also matches λz, now yielding the substitution $\{p \bullet q/z\}$. Of course, for $P \xrightarrow{\lambda x \bullet \lambda y} P'$ to be simulated by $Q \xrightarrow{\lambda z} Q'$, it must be that $\{p/x, q/y\}P'$ is bisimilar to $\{p \bullet q/z\}Q'$. Another subtlety is in the unification of shared information n with protected information $\ulcorner n \urcorner$. Since the latter is a request for the communicating party to also know this information, the two patterns unify. (Such as the more careful seller checking that the buyer provides *name* and *number* for a bank account.) These ideas are formalised via an ordering on patterns that characterises the valid replies to a given challenge: every pattern 'greater than' the challenge is a valid reply, provided that, by applying the resulting substitutions to the respective continuations, bisimilar processes are obtained.

The form of pattern unification adopted in CPC generalises other forms of pattern matching already presented in the literature. It is then desirable that CPC's theory and results can be adapted to such simpler forms. Section 4 shows that this job is rather straightforward for the form of pattern matching underlying Linda, for two simple extensions of Linda, for the polyadic π-calculus, and for the π-calculus with polyadic synchronization. This provides a complete behavioural theory for the languages adopting such forms of pattern matching. Moreover, for the π-calculus, the result coincides with the usual notions of barbed congruence and early bisimulation congruence; this can be seen as a confirmation of the validity of the theory presented here.

2 Concurrent Pattern Calculus

Suppose a countable set of *names* $\mathcal{N}$ (meta-variables $n, m, x, y, z, \ldots$ – even if in the examples symbolic names will be used). The *patterns* (meta-variables $p, p', p_1, q, q', q_1, \ldots$) are built using names and have the following forms:

$$p \ ::= \ \lambda x \mid x \mid \ulcorner x \urcorner \mid p \bullet p$$

Binding names λx denote information sought by a trader; variable names x represent such information. Protected names $\ulcorner x \urcorner$ represent recognised information that cannot be traded. A compound $p \bullet q$ combines the two patterns p and q; compounds are left associative.

Given a pattern p the sets of: *variables names*, denoted $\mathsf{vn}(p)$; *protected names*, denoted $\mathsf{pn}(p)$; and *binding names*, denoted $\mathsf{bn}(p)$, are as expected with the union being taken for compounds. The *free names* of a pattern p, written $\mathsf{fn}(p)$, is the union of the variable names and protected names of p. A pattern is *well formed* if its binding names are pairwise distinct and different from the free ones. All patterns appearing in the rest of this paper are assumed to be well formed.

As protected names are limited to recognition and binding names are being sought, neither should be communicable to another process. Thus, a pattern is *communicable*, able to be traded to another process, if it contains no protected or binding names. Protection of a name can be extended to a communicable pattern p by defining $\ulcorner p \bullet q \urcorner = \ulcorner p \urcorner \bullet \ulcorner q \urcorner$.

A *substitution* σ is defined as a partial function from names to communicable patterns. The *domain* of σ is denoted $\mathsf{dom}(\sigma)$; the free names of σ, written $\mathsf{fn}(\sigma)$, is given by the union of the sets $\mathsf{fn}(\sigma x)$ where $x \in \mathsf{dom}(\sigma)$. The *names* of σ, written $\mathsf{names}(\sigma)$, are $\mathsf{dom}(\sigma) \cup \mathsf{fn}(\sigma)$. Notationally, given two substitutions σ and θ, denote with $\theta[\sigma]$ the composition of σ and θ, with domain limited to the domain of σ, i.e. the substitution mapping every $x \in \mathsf{dom}(\sigma)$ to $\theta(\sigma(x))$. For later convenience, define the identity substitution on a set of names X, written id_X: it maps every name in X to itself.

Substitutions are applied to patterns as follows:

$$\sigma x = \begin{cases} \sigma(x) & \text{if } x \in \mathsf{dom}(\sigma) \\ x & \text{otherwise} \end{cases} \qquad \sigma \ulcorner x \urcorner = \begin{cases} \ulcorner \sigma(x) \urcorner & \text{if } x \in \mathsf{dom}(\sigma) \\ \ulcorner x \urcorner & \text{otherwise} \end{cases}$$

$$\sigma(\lambda x) = \lambda x \qquad \sigma(p \bullet q) = (\sigma p) \bullet (\sigma q)$$

The *symmetric matching* (or *unification*) of two patterns p and q, written $\{p \parallel q\}$, attempts to unify p and q by generating substitutions for their binding names. When defined, the result is a pair of substitutions whose domains are the binding names of p and of q, respectively. The rules to generate the substitutions are:

$$\begin{aligned}
\{x \parallel x\} = \{x \parallel \ulcorner x \urcorner\} = \{\ulcorner x \urcorner \parallel x\} = \{\ulcorner x \urcorner \parallel \ulcorner x \urcorner\} &\stackrel{\text{def}}{=} (\{\}, \{\}) \\
\{\lambda x \parallel q\} &\stackrel{\text{def}}{=} (\{q/x\}, \{\}) && \text{if } q \text{ is communicable} \\
\{p \parallel \lambda x\} &\stackrel{\text{def}}{=} (\{\}, \{p/x\}) && \text{if } p \text{ is communicable} \\
\{p_1 \bullet p_2 \parallel q_1 \bullet q_2\} &\stackrel{\text{def}}{=} (\sigma_1 \cup \sigma_2\,,\, \rho_1 \cup \rho_2) && \text{if } \{p_i \parallel q_i\} = (\sigma_i, \rho_i) \text{ for } i \in \{1, 2\}
\end{aligned}$$

Variable and protected names unify if they are the same name. A binding name unifies with any communicable pattern to produce a binding for its bound name. Two compounds unify if their corresponding components do; the resulting substitutions are given by taking the union of those produced by unifying the components (necessarily disjoint, as patterns are well-formed). Otherwise the patterns cannot be unified and the matching is undefined. Notice that pattern matching is deterministic because of left-associativity of compounds.

The processes of CPC are given by:

$$P \;::=\; \mathbf{0} \;\mid\; P \mid P \;\mid\; !P \;\mid\; (\nu x)P \;\mid\; p \to P$$

The null process $\mathbf{0}$ is the inactive process; $P \mid Q$ is the parallel composition of processes P and Q, allowing the two processes to evolve independently or by interacting; the replication $!P$ provides as many parallel copies of P as desired; $(\nu x)P$ declares a new name x, visible only within P and distinct from any other name. The traditional input and output primitives of process calculi are replaced by the *case*, viz. $p \to P$, that has a pattern p and a body P. A case with the null process as the body may also be written by only specifying the pattern. For later convenience, $\widetilde{n}$ denotes a collection of names $n_1, \ldots, n_i$; for example, $(\nu n_1)(\ldots((\nu n_i)P))$ will be written $(\nu\widetilde{n})P$.

The free names of processes, denoted $\mathsf{fn}(P)$, are defined as usual for all the traditional primitives and $\mathsf{fn}(p \to P) = \mathsf{fn}(p) \cup (\mathsf{fn}(P) \backslash \mathsf{bn}(p))$ for the case, where the binding names of the pattern bind their free occurrences in the body.

The *structural equivalence relation* $\equiv$ is defined just as in π-calculus [25]: it includes α-conversion and its defining axioms are:

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad !P \equiv P \mid !P$$

$$(\nu n)\mathbf{0} \equiv \mathbf{0} \qquad (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \qquad P \mid (\nu n)Q \equiv (\nu n)(P \mid Q) \;\text{ if } n \notin \mathsf{fn}(P)$$

The operational semantics of CPC is formulated via a *reduction relation* between pairs of processes. Its defining rules are:

$$(p \to P) \mid (q \to Q) \;\longmapsto\; (\sigma P) \mid (\rho Q) \quad \text{if } \{p \parallel q\} = (\sigma, \rho)$$

$$\frac{P \longmapsto P'}{P \mid Q \longmapsto P' \mid Q} \qquad \frac{P \longmapsto P'}{(\nu n)P \longmapsto (\nu n)P'} \qquad \frac{P \equiv Q \quad Q \longmapsto Q' \quad Q' \equiv P'}{P \longmapsto P'}$$

CPC has one interaction axiom, stating that, if the unification of two patterns p and q is defined and generates (σ, ρ), the substitutions σ and ρ are applied to the bodies P and Q, respectively. If the matching of p and q is undefined then no interaction occurs. The interaction rule is then closed under parallel composition, restriction and structural equivalence in the usual manner.

3 Behavioural Theory

This section follows a standard approach in concurrency to defining behavioural equivalences, beginning with a barbed congruence and following with a labelled transition system (LTS) and a definition of bisimulation for CPC. Some properties of patterns will be explored as a basis for showing coincidence of the semantics.

3.1 Barbed Congruence

The first crucial step is to characterise the interactions a process can participate in via *barbs*. Since a barb is an opportunity for interaction, a simplistic definition could be the following:

$$P \downarrow \text{ iff } P \equiv p \to P' \mid P'', \text{ for some } p, P' \text{ and } P'' \tag{1}$$

However, this definition is too strong: for example, $(\nu n)(n \to P)$ does not exhibit a barb according to (1), but it can interact with an external process, e.g. $\lambda x \to \mathbf{0}$. Thus, an improvement to (1) is as follows:

$$P \downarrow \text{ iff } P \equiv (\nu \widetilde{m})(p \to P' \mid P''), \text{ for some } \widetilde{n}, p, P' \text{ and } P'' \tag{2}$$

Now, this definition is too weak. Consider $(\nu n)(\ulcorner n \urcorner \to P)$: it exhibits a barb according to (2), but cannot interact with any external process. A further refinement on (2) could be:

$$P \downarrow \text{ iff } P \equiv (\nu \widetilde{m})(p \to P' \mid P''), \text{ for some } \widetilde{n}, p, P', P'' \text{ s.t. } \mathsf{pn}(p) \cap \widetilde{n} = \emptyset \tag{3}$$

This definition is not yet the final one, as it is not sufficiently discriminating to have only a single kind of barb (the contexts in Definition 9 use two kinds of barbs, to define success and failure). Thus, like in CCS and π-calculus [27], barbs must be indexed, e.g. on some names that give an abstract account of the matching capabilities of the process. Because of the rich form of interactions, CPC barbs also include the set of names that *may* be tested for equality in an interaction, not just those that *must* be equal.

Definition 1 (Barb). *Let $P \downarrow_{\widetilde{m}}$ mean that $P \equiv (\nu \widetilde{n})(p \to P' \mid P'')$ for some $\widetilde{n}, p, P'$ and P'' such that $\mathsf{pn}(p) \cap \widetilde{n} = \emptyset$ and $\widetilde{m} = \mathsf{fn}(p) \backslash \widetilde{n}$.*

Using this definition, a barbed congruence can be defined in the standard way [21], by requiring three properties. Let $\mathfrak{R}$ denote a binary relation on processes and let a *context* $C(\cdot)$ be a process with the hole '$\cdot$' replacing one instance of the null process.

Definition 2 (Barb preservation). *$\mathfrak{R}$ is barb preserving iff, for every $(P, Q) \in \mathfrak{R}$, it holds that $P \downarrow_{\widetilde{m}}$ implies $Q \downarrow_{\widetilde{m}}$.*

$$\textsf{case}: \quad (p \to P) \xrightarrow{p} P \qquad\qquad \textsf{resnon}: \quad \frac{P \xrightarrow{\mu} P'}{(\nu n)P \xrightarrow{\mu} (\nu n)P'} \quad n \notin \mathsf{names}(\mu)$$

$$\textsf{resin}: \quad \frac{P \xrightarrow{(\nu \widetilde{n})p} P'}{(\nu m)P \xrightarrow{(\nu \widetilde{n}, m)p} P'} \quad m \in \mathsf{vn}(p) \setminus (\widetilde{n} \cup \mathsf{pn}(p) \cup \mathsf{bn}(p)) \qquad\qquad \textsf{rep}: \quad \frac{!P \mid P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'}$$

$$\textsf{match}: \quad \frac{P \xrightarrow{(\nu \widetilde{m})p} P' \quad Q \xrightarrow{(\nu \widetilde{n})q} Q'}{P \mid Q \xrightarrow{\tau} (\nu \widetilde{m}, \widetilde{n})(\sigma P' \mid \rho Q')} \quad \begin{array}{l} \{p \parallel q\} = (\sigma, \rho) \\ \widetilde{m} \cap \mathsf{fn}(Q) = \widetilde{n} \cap \mathsf{fn}(P) = \emptyset \\ \widetilde{m} \cap \widetilde{n} = \emptyset \end{array}$$

$$\textsf{parext}: \quad \frac{P \xrightarrow{(\nu \widetilde{n})p} P'}{P \mid Q \xrightarrow{(\nu \widetilde{n})p} P' \mid Q} \quad (\widetilde{n} \cup \mathsf{bn}(p)) \cap \mathsf{fn}(Q) = \emptyset \qquad\qquad \textsf{parint}: \quad \frac{P \xrightarrow{\tau} P'}{P \mid Q \xrightarrow{\tau} P' \mid Q}$$

Fig. 1. LTS (the symmetric version of parint and parext have been omitted)

Definition 3 (Reduction closure). $\mathfrak{R}$ *is reduction closed iff, for every* $(P, Q) \in \mathfrak{R}$*, it holds that* $P \longmapsto P'$ *implies* $Q \longmapsto Q'$*, for some* Q' *such that* $(P', Q') \in \mathfrak{R}$*.*

Definition 4 (Context closure). $\mathfrak{R}$ *is context closed iff, for every* $(P, Q) \in \mathfrak{R}$ *and for every context* $C(\cdot)$*, it holds that* $(C(P), C(Q)) \in \mathfrak{R}$*.*

Definition 5 (Barbed congruence). *Barbed congruence,* $\simeq$*, is the largest symmetric, barb preserving, reduction and context closed binary relation on processes.*

Barbed congruence equates processes with the same behaviour, as captured by barbs: two equivalent processes must exhibit the same behaviours, and this property should hold along every sequence of reductions and in every execution context. This defines the *strong* version of barbed congruence; its *weak* counterpart can be obtained in the usual manner [26,27], with more complex contexts for proving the completeness theorem.

The problem in proving (strong/weak) barbed congruence is its closure under any context. As is typical we solve this by giving an easier to reason about coinductive (bisimulation-based) characterization using an alternate operation semantics; an LTS.

3.2 Labelled Transition System

The following is an adaption of the standard late LTS for the π-calculus [26]. *Labels* are defined as follows:

$$\mu \ ::= \ \tau \quad | \quad (\nu \widetilde{n})p$$

Labels are used in *transitions* $P \xrightarrow{\mu} P'$ between processes, whose defining rules are given in Figure 1. Rule case states that a case's pattern can be used to interact with external processes. Rule resnon is used when a restricted name does not appear in the names of the label: it simply maintains the restriction on the process after the transition. By contrast, rule resin is used when a restricted name occurs in the label: as the restricted name is going to be shared with other processes, the restriction is moved from the process to

the label (this is called *extrusion*, by using a π-calculus terminology). Of course an extruded name cannot already be restricted, cannot be protected (as this would prevent interaction), and cannot be a binding name. Rule match defines when two processes can interact to perform an internal action: this can occur whenever the processes exhibit labels with unifiable patterns and with no possibility of clash or capture due to restricted names. Rule rep unfolds the replicated process to infer the action. Rule parint states that, if either process in a parallel composition can transition by an internal action, then the whole process can transition by an internal action. Rule parext is similar, but is used when the label is visible: when one of the processes in parallel exhibits an external action, then the whole composition exhibits the same external action, as long as the restricted or binding names of the label do not appear free in the parallel component that does not generate the label.

Note that α-conversion is always assumed to satisfy the side conditions whenever needed and the symmetric rules have been omitted for brevity.

The presentation of the LTS is concluded with the following two results. First, the LTS is structurally image finite, i.e. for every P and μ, there are finitely many $\equiv$-equivalence classes of μ-reducts of P (Proposition 1). Second, the τ's in the LTS induce the same operational semantics as the reductions (Proposition 2).

Proposition 1. *The LTS defined in Figure 1 is structurally image finite.*

Proposition 2. *If $P \xrightarrow{\tau} P'$ then $P \longmapsto P'$. Conversely, if $P \longmapsto P'$ then there exists P'' such that $P \xrightarrow{\tau} P'' \equiv P'$.*

3.3 Bisimulation

The next step is to develop a *bisimulation* relation that equates processes with the same interactional behaviour as captured by the labels of the LTS. The complexity is that the labels for external actions contain patterns, and some patterns are 'more general' than others, in terms of their matching capabilities. Two examples can clarify the point.

Example 1. Consider the processes $P = \lambda x \bullet \lambda y \to x \bullet y$ and $Q = \lambda z \to z$. Every process that can interact with P (by exhibiting a pattern matching against $\lambda x \bullet \lambda y$) can interact with Q, but not vice versa: e.g., $n \to \mathbf{0}$ can interact with Q but not with P. In this sense, the pattern λz is considered 'more general' than $\lambda x \bullet \lambda y$.

Example 2. Consider the processes $P = \ulcorner n \urcorner \to \mathbf{0}$ and $Q = n \to \mathbf{0}$. Every process that can interact with P can interact with Q, but not vice versa: consider, e.g., $\lambda x \to \mathbf{0}$. Thus, the pattern n is considered 'more general' than $\ulcorner n \urcorner$.

Now define an order relation on patterns that can be used to develop the bisimulation. In most process calculi, a challenge is replied to with an identical action [26]. However, there are situations in which an exact reply would make the bisimulation equivalence too fine for characterising barbed congruence [3,12]. This is due to the impossibility for the language contexts to force barbed congruent processes to execute the same action; in such calculi more liberal replies must be allowed, as here for CPC. To this aim, define

$\hat{\sigma}$ as a normal substitution, except that it operates on binding names rather than on free ones. Formally:

$$\hat{\sigma}x = x \qquad \hat{\sigma}\ulcorner x \urcorner = \ulcorner x \urcorner \qquad \hat{\sigma}(\lambda x) = \begin{cases} \sigma(x) & \text{if } x \in \mathsf{dom}(\sigma) \\ \lambda x & \text{otherwise} \end{cases} \qquad \hat{\sigma}(p \bullet q) = (\hat{\sigma}p) \bullet (\hat{\sigma}q)$$

Definition 6. *Let p, q, σ and ρ be such that $\mathsf{bn}(p) = \mathsf{dom}(\sigma)$ and $\mathsf{bn}(q) = \mathsf{dom}(\rho)$. Define inductively that p is* compatible *with q by σ and ρ, denoted $p, \sigma \ll q, \rho$, whenever:*

$$p, \sigma \ll \lambda y, \{\hat{\sigma}p/y\} \text{ if } \mathsf{fn}(p) = \emptyset \qquad n, \{\} \ll n, \{\}$$

$$\ulcorner n \urcorner, \{\} \ll \ulcorner n \urcorner, \{\} \qquad \ulcorner n \urcorner, \{\} \ll n, \{\}$$

$$p_1 \bullet p_2, \sigma_1 \cup \sigma_2 \ \ll \ q_1 \bullet q_2, \rho_1 \cup \rho_2 \ \text{ if } p_i, \sigma_i \ll q_i, \rho_i, \text{ for } i \in \{1, 2\} .$$

The next result captures the idea behind the definition of compatibility: the patterns matched by p are a subset of the patterns matched by q.

Lemma 1. *$p, \sigma \ll q, \rho$ and $\{p \parallel r\} = (\sigma, \theta)$ implies $\{q \parallel r\} = (\rho, \theta)$.*

Moreover, compatibility preserves information used for barbs, is stable under substitution composition, is reflexive and transitive.

Proposition 3. *If $p, \sigma \ll q, \rho$ then $\mathsf{fn}(p) = \mathsf{fn}(q)$ and $\mathsf{fn}(\sigma) = \mathsf{fn}(\rho)$. Moreover, $\mathsf{vn}(p) \subseteq \mathsf{vn}(q)$ and $\mathsf{pn}(q) \subseteq \mathsf{pn}(p)$.*

Lemma 2. *If $p, \sigma \ll q, \rho$ then $p, \theta[\sigma] \ll q, \theta[\rho]$, for every θ.*

Proposition 4. *Given p and σ such that $\mathsf{dom}(\sigma) = \mathsf{bn}(p)$, then $p, \sigma \ll p, \sigma$.*

Proposition 5. *$p, \sigma \ll q, \rho$ and $q, \rho \ll r, \theta$ imply $p, \sigma \ll r, \theta$.*

Definition 7 (Bisimulation). *A symmetric binary relation on processes $\mathfrak{R}$ is a bisimulation if, for every $(P, Q) \in \mathfrak{R}$ and $P \xrightarrow{\mu} P'$, it holds that:*

- *if $\mu = \tau$, then $Q \xrightarrow{\tau} Q'$, for some Q' such that $(P', Q') \in \mathfrak{R}$;*
- *if $\mu = (\nu \widetilde{n})p$, for $(\mathsf{bn}(p) \cup \widetilde{n}) \cap \mathsf{fn}(Q) = \emptyset$, then for all σ with $\mathsf{dom}(\sigma) = \mathsf{bn}(p)$ and $\mathsf{fn}(\sigma) \cap \widetilde{n} = \emptyset$ there exist q, Q' and ρ such that $Q \xrightarrow{(\nu \widetilde{n})q} Q'$ and $p, \sigma \ll q, \rho$ and $(\sigma P', \rho Q') \in \mathfrak{R}$.*

Denote with $\sim$ the largest bisimulation closed under any substitution.

The definition is inspired by the early bisimulation congruence for the π-calculus [26]: first of all, to be a congruence, we need to consider its closure under all possible substitutions (otherwise, it would not be closed under prefixes). Then, for every possible instantiation σ of the binding names, there exists a proper reply from Q. Of course, σ cannot be chosen arbitrarily: it cannot use in its range names that were restricted in P. Also the action μ cannot be arbitrary, as in the π-calculus: its restricted and binding names cannot occur free in Q.

Differently from the π-calculus, however, the reply from Q can be different from the challenge from P: this is due to the fact that CPC contexts are not powerful enough to enforce an identical reply (as highligthed in Examples 1 and 2). Indeed, this notion of bisimulation allows a challenge p to be replied to by any compatible q, provided that σ is properly adapted (yielding ρ, as described by the compatibility relation) before being applied to Q'. This feature somehow resembles the symbolic characterization of open bisimilarity given in [29,6]. There, labels are pairs made up of an action and a set of equality constraints. A challenge can be replied to by a smaller (i.e. less constraining) set. However, the action in the reply must be the same (in [29]) or becomes the same once we apply the name identifications induced by the equality constraints (in [6]).

3.4 Soundness and Completeness of Bisimulation

Soundness is proved by showing that the bisimilarity relation is included in barbed congruence; this is done by showing that $\sim$ is an equivalence, it is barb preserving, reduction closed and context closed. All the details can be found in [19].

Theorem 1 (Soundness of bisimilarity). $\sim \subseteq \simeq$.

Completeness is proved by showing that barbed congruence is a bisimulation. First, is to show that barbed congruence is closed under substitutions.

Lemma 3. *If $P \simeq Q$ then $\sigma P \simeq \sigma Q$, for every σ.*

Second, is to show that, for any challenge, a proper reply can be yielded via closure under an appropriate context. When the challenge is an internal action, the reply is also an internal action; thus, the empty context suffices, as barbed congruence is reduction closed. The complex scenario is when the challenge is a pattern together with a set of restricted names, i.e., a label of the form $(\nu\widetilde{n})p$. Observe that in the bisimulation such challenges also fix a substitution σ whose domain is the binding names of p.

First of all, define a notion of success and failure that can be reported. A fresh name w is used for reporting success, with a barb $\downarrow_w$ indicating success, and $\Downarrow_w$ indicating a reduction sequence that eventually reports success. Failure is handled similarly using the fresh name f. A process P *succeeds* if $P \Downarrow_w$ and $P \not\Downarrow_f$; P is *successful* if $P \equiv (\nu\widetilde{n})(\ulcorner w \urcorner \bullet p \mid P')$, for some $\widetilde{n}$, p and P' such that $w \notin \widetilde{n}$ and $P' \not\Downarrow_f$. *P becomes successful* if it can reduce to a successful process.

Now develop a reply for a challenge of the form $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$; the general setting (with an arbitrary σ) will be recovered by relying on Lemma 2. The context for forcing a proper reply is developed in three steps. The first step presents the *specification* of a pattern and a set of names N (to be thought of as the free names of the processes being compared for bisimilarity); this is the information required to build a reply context. The second step develops auxiliary processes to test specific components of a pattern, based on information from the specification. The third step combines these into a reply context that becomes successful if and only if it interacts with a process that exhibits a proper reply to the challenge. In what follows, we use the *first projection* $\mathsf{fst}(-)$ and *second projection* $\mathsf{snd}(-)$ of a set of pairs.

Definition 8. *The* specification $\mathsf{spec}^N(p)$ *of a pattern* p *with respect to a finite set of names* N *is defined as follows:*

$$\mathsf{spec}^N(\lambda x) = x, \{\}, \{\} \qquad \mathsf{spec}^N(\ulcorner n \urcorner) = \ulcorner n \urcorner, \{\}, \{\}$$

$$\mathsf{spec}^N(n) = \begin{cases} \lambda x, \{(x,n)\}, \{\} & \text{if } n \in N \text{ and } x \notin N \cup \{n\} \\ \lambda x, \{\}, \{(x,n)\} & \text{if } n \notin N \text{ and } x \notin N \cup \{n\} \end{cases}$$

$$\mathsf{spec}^N(p \bullet q) = p' \bullet q', F_p \uplus F_q, R_p \uplus R_q \text{ if } \begin{cases} \mathsf{spec}^N(p) = p', F_p, R_p \\ \mathsf{spec}^N(q) = q', F_q, R_q \end{cases}$$

where $F_p \uplus F_q$ *denotes* $F_p \cup F_q$*, provided that* $\mathsf{fst}(F_p) \cap \mathsf{fst}(F_q) = \emptyset$ *(a similar meaning holds for* $R_p \uplus R_q$*).*

Given a pattern p, the specification $\mathsf{spec}^N(p) = p', F, R$ of p with respect to a set of names N has three components: (1) p', called the *complementary pattern*, is a pattern used to ensure that the context interacts with a process that exhibits a pattern compatible with p; (2) F is a collection of pairs (x, n) made up by a binding name in p' and the expected (free) name it will be bound to; finally, (3) R is a collection of pairs (x, n) made up by a binding name in p' and the expected (restricted) name it will be bound to. Observe that can be assumed p' well formed as all binding names can be taken as (pairwise) different.

From now on, adopt the following notation: if $\widetilde{n} = n_1, \ldots, n_i$, then $\ulcorner w \urcorner \bullet \widetilde{n}$ denotes $\ulcorner w \urcorner \bullet n_1 \bullet \ldots \bullet n_i$. Moreover, $\theta(\widetilde{n})$ denotes $\theta(n_1), \ldots, \theta(n_i)$; hence, $\ulcorner w \urcorner \bullet \theta(\widetilde{n})$ denotes $\ulcorner w \urcorner \bullet \theta(n_1) \bullet \ldots \bullet \theta(n_i)$.

Definition 9. *The* characteristic process $\mathsf{char}^N(p)$ *of a pattern* p *with respect to a finite set of names* N *is* $\mathsf{char}^N(p) = p' \rightarrow \mathsf{tests}^N_{F,R}$ *where* $\mathsf{spec}^N(p) = p', F, R$ *and*

$$\begin{aligned} \mathsf{tests}^N_{F,R} \stackrel{\text{def}}{=} (\nu \widetilde{w_x})(\nu \widetilde{w_y})(& \\ & \ulcorner w_{x_1} \urcorner \rightarrow \ldots \rightarrow \ulcorner w_{x_i} \urcorner \rightarrow \ulcorner w_{y_1} \urcorner \rightarrow \ldots \rightarrow \ulcorner w_{y_j} \urcorner \rightarrow \ulcorner w \urcorner \bullet \widetilde{x} \\ & \mid \textstyle\prod_{(x,n) \in R} \mathsf{equality}^R(x, n, w_x) \\ & \mid \textstyle\prod_{(y,n) \in F} \mathsf{free}(y, n, w_y) \\ & \mid \textstyle\prod_{(y,n) \in R} \mathsf{rest}^N(y, w_y) \ ) \end{aligned}$$

where $\widetilde{x} = \{x_1, \ldots, x_i\} = \mathsf{fst}(R)$ *and* $\widetilde{y} = \{y_1, \ldots, y_j\} = \mathsf{fst}(F) \cup \mathsf{fst}(R)$.

Although the details of the tests are omitted here (see [19] for details), their behaviour is described by the following Lemmas.

Lemma 4. *Let* θ *be such that* $\{n, w\} \cap \mathsf{dom}(\theta) = \emptyset$*; then,* $\theta(\mathsf{free}(x, n, w))$ *succeeds if and only if* $\theta(x) = n$*.*

Lemma 5. *Let* θ *be such that* $(N \cup \{w, f\}) \cap \mathsf{dom}(\theta) = \emptyset$*; then,* $\theta(\mathsf{rest}^N(x, w))$ *succeeds if and only if* $\theta(x) \in \mathcal{N} \setminus N$*.*

Lemma 6. *Let* θ *be such that* $(\mathsf{snd}(R) \cup \{w, f, m\}) \cap \mathsf{dom}(\theta) = \emptyset$*; then,* $\theta(\mathsf{equality}^R(x, m, w))$ *succeeds if and only if, for every* $(y, n) \in R$*,* $m = n$ *if and only if* $\theta(x) = \theta(y)$*.*

Definition 10. *A* reply context $C_p^N(\cdot)$ *for the challenge* $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$ *with a finite set of names* N *such that* $\widetilde{n}$ *is disjoint from* N *is defined as follows:*

$$C_p^N(\cdot) \stackrel{\text{def}}{=} \mathsf{char}^N(p) \mid \cdot$$

It can be proved (see [19]) that the minimum number of reductions required for $C_p^N(Q)$ to become successful (for any Q) is the number of reduction steps for $\theta(\mathsf{tests}_{F,R}^N)$ to become successful plus 1; this number only depends on N and p, i.e. not on θ. Denote this number as $\textsc{Lb}(N, p)$. The main feature of $C_p^N(\cdot)$ is described by the following key Lemma.

Lemma 7. *Suppose given a challenge* $((\nu\widetilde{n})p, \mathsf{id}_{\mathsf{bn}(p)})$*, a finite set of names* N*, a process* Q *and fresh names* w *and* f *such that* $(\widetilde{n} \cup \{w, f\}) \cap N = \emptyset$ *and* $(\mathsf{fn}((\nu\widetilde{n})p) \cup \mathsf{fn}(Q)) \subseteq N$*. If* $Q \xrightarrow{(\nu\widetilde{m})q} Q'$ *and there exists* ρ *such that* $p, \mathsf{id}_{\mathsf{bn}(p)} \ll q, \rho$*, then* $C_p^N(Q) \longmapsto^k (\nu\widetilde{n})(\rho Q' \mid \ulcorner w \urcorner \bullet \widetilde{n} \mid Z)$*, where* $k = \textsc{Lb}(N, p)$ *and* $Z \simeq \mathbf{0}$*. Conversely, if* $C_p^N(Q)$ *becomes successful in* $\textsc{Lb}(N, p)$ *reduction steps, then there exist* q*,* Q' *and* ρ *such that* $Q \xrightarrow{(\nu\widetilde{n})q} Q'$ *and* $p, \mathsf{id}_{\mathsf{bn}(p)} \ll q, \rho$*.*

The last result needed for proving Theorem 2 is an auxiliary Lemma that allows us to remove success and dead processes from both sides of a barbed congruence, while also opening the scope of the names exported by the success barb.

Lemma 8. *Let* $(\nu\widetilde{m})(P \mid \ulcorner w \urcorner \bullet \widetilde{m} \mid Z) \simeq (\nu\widetilde{m})(Q \mid \ulcorner w \urcorner \bullet \widetilde{m} \mid Z)$*, for* $w \notin \mathsf{fn}(P, Q, \widetilde{m})$ *and* $Z \simeq \mathbf{0}$*; then* $P \simeq Q$*.*

Theorem 2 (Completeness of the bisimulation). $\simeq \subseteq \sim$.

4 On Variations of Pattern Matching

The form of pattern unification used so far in CPC is very rich. More limited forms of pattern matching have been used in the literature; as shown below, they can all be adopted in our language without compromising the coincidence of barbed congruence and bisimilarity.

The first variant is the form of pattern matching used in Linda [16]. Differently from CPC, Linda distinguishes between input and output patterns (the latter are usually called *tuples* in a *tuplespace*):

$$p ::= \pi \mid \varpi \qquad \pi ::= \lambda x \mid \ulcorner x \urcorner \mid \pi \bullet \pi \qquad \varpi ::= x \mid \varpi \bullet \varpi$$

Thus, communication is asymmetric; consequently, the pattern matching function is defined only between an input and an output pattern and yields a single substitution. It is defined as:

$$\{\ulcorner x \urcorner \parallel x\} \stackrel{\text{def}}{=} \{\} \qquad \{\lambda x \parallel n\} \stackrel{\text{def}}{=} \{n/x\} \qquad \{\pi \bullet \pi' \parallel \varpi \bullet \varpi'\} \stackrel{\text{def}}{=} \{\pi \parallel \varpi\} \cup \{\pi' \parallel \varpi'\} \quad (4)$$

From the second rule, it is apparent that communicable patterns in Linda are single variable names. The operational rules for matching in the reductions and in the LTS are the following:

$$(\pi \to P) \mid (\varpi \to Q) \longmapsto \sigma P \mid Q \quad \text{if } \{\pi \parallel \varpi\} = \sigma \qquad \frac{P \xrightarrow{\pi} P' \quad Q \xrightarrow{(\nu \widetilde{n})\varpi} Q' \quad \{\pi \parallel \varpi\} = \sigma}{P \mid Q \xrightarrow{\tau} (\nu \widetilde{n})(\sigma P' \mid Q')} \; \widetilde{n} \cap \mathsf{fn}(P) = \emptyset$$

The theory of bisimulation is simplified in this setting, as $\ll$ is the identity. Barbed congruence can be defined as in Section 3.1 and the two equivalences do coincide.

Two interesting extensions of Linda's pattern matching (intermediate between Linda's and CPC's ones) are:

1. Accept a "non-fully decomposing" form of pattern matching; e.g., λx can match $n \bullet m$. In this case, it suffices to modify the definition of pattern matching by generalizing the second axiom in (4) to

$$\{\lambda x \parallel n_1 \bullet \ldots \bullet n_k\} \stackrel{\text{def}}{=} \{n_1 \bullet \ldots \bullet n_k / x\}$$

   (i.e., by rolling back to the original definition of communicable patterns as sequences of variable names) and by defining $\ll$ as in Definition 6, except for the fourth axiom (that must be ignored).
2. Allow the output process to specify which names can be passed and which ones can only be used for testing equality; e.g., $n \bullet \ulcorner m \urcorner$ can be matched by $\lambda x \bullet \ulcorner m \urcorner$, but not by $\lambda x \bullet \lambda y$. In this case, output patterns are defined as

$$\varpi ::= x \mid \ulcorner x \urcorner \mid \varpi \bullet \varpi$$

   This is resolved by adding to (4) the axiom $\{\ulcorner x \urcorner \parallel \ulcorner x \urcorner\} \stackrel{\text{def}}{=} \{\}$ and by defining $\ll$ as in Definition 6, except for the first axiom (that must be ignored).

In both cases, reductions and LTS are like Linda's ones; barbed congruence and bisimulation are defined as in Section 3 and, again, they do coincide.

Another well-known form of pattern matching is the one underlying the *polyadic π-calculus* [25]. In this case, (input and output) patterns have the form

$$\pi ::= \ulcorner a \urcorner \bullet \lambda x_1 \bullet \ldots \bullet \lambda x_k \qquad\qquad \varpi ::= \ulcorner a \urcorner \bullet n_1 \bullet \ldots \bullet n_k$$

for any $k > 0$ (these are usually written as $a(x_1, \ldots, x_k)$ and $\bar{a}(n_1, \ldots, n_k)$). Now pattern matching is defined as in (4), but with $\{\ulcorner x \urcorner \parallel \ulcorner x \urcorner\} \stackrel{\text{def}}{=} \{\}$ in place of the first axiom. Reductions, LTS and compatibility are like in Linda. Notice that the first two relations are the usual ones for the polyadic π-calculus; similarly, the bisimulation arising in this framework is the same as the standard early bisimulation congruence defined for the calculus. It is worth noticing that the barbs we exploit are different from the traditional ones for the π-calculus [27], where only the channel and the kind of action (either input or output) are observed. In our formulation of the polyadic π-calculus, input and output barbs can be usually distinguished: $\ulcorner a \urcorner \bullet \lambda x$ generates $\downarrow_{\{a\}}$ whereas $\ulcorner a \urcorner \bullet n$ generates $\downarrow_{\{a,n\}}$ (the two are indistinguishable only if $n = a$). In general, our barbs are more informative

than π-calculus' ones, since they also observe the argument of the output. However, since this barbed congruence coincides with the early bisimulation (that, in turn, coincides with the barbed congruence relying on the "standard" π-calculus' barbs), by transitivity we obtain that the two kinds of barbs yield the same congruence.

Similarly, also the form of pattern matching underlying the π-calculus with polyadic synchronization [8] can be easily rendered. It suffices to take

$$\pi ::= \ulcorner a_1 \urcorner \bullet \ldots \bullet \ulcorner a_k \urcorner \bullet \lambda x \qquad\qquad \varpi ::= \ulcorner a_1 \urcorner \bullet \ldots \bullet \ulcorner a_k \urcorner \bullet n$$

(usually written $a_1 \cdot \ldots \cdot a_k(x)$ and $\overline{a_1 \cdot \ldots \cdot a_k}(n)$). Pattern matching, reductions, LTS and compatibility are then the same as in polyadic π-calculus.

5 Conclusions and Future Work

CPC demonstrates the expressive power possible with a minimal process calculus whose interaction is defined by symmetric pattern unification. The behavioural theory required to capture CPC turns out to have some interesting properties based on patterns and pattern matching. Perhaps, the most curious one is that a symmetric relation (viz., bisimilarity) is defined by an (asymmetric) ordering upon patterns. Indeed, the resulting bisimulation can be smoothly and modularly adapted to cope with other forms of pattern matching and other process calculi.

Related Work. To the best of our knowledge, there are very few notions of behavioural equivalences for process calculi that rely on pattern matching. We start with a few calculi based on a Linda-like pattern matching. A first example is [13], where the authors develop a testing framework; however, no coinductive and label-based equivalence is provided. Another paper where a Linda-like pattern matching is explored for bisimulation is [12]; however, there the focus is on the distribution and connectivity of processes and, consequently, the pattern matching is simplified by relying on patterns of length 1. A similar choice is taken in other works, e.g. [7,10,11]. Of course, this choice radically simplifies the theory.

Recently, Psi [4] has emerged as a rich framework that can encode different process calculi, including calculi with sophisticated forms of pattern matching. However, CPC and Psi are uncomparable: CPC cannot encode formulae (e.g. the indirect computation of channel equality), while Psi cannot encode self-matching processes (same as π, see [20]). The same holds for the applied π-calculus [1], because of the presence of active substitutions.

A more complex notion of bisimulation is the one for the Join calculus [14] given in [15]. The difficult part lays in the definition of the LTS, since some names can be marked as visible from outside their definition and, consequently, interact with the execution context. The definition of bisimilarity is then standard and, hence, the interplay with pattern matching is totally hidden within the LTS. We prefer to make it explicit in the bisimulation, both to keep the LTS as standard as possible and for showing the exact impact that pattern matching has on the semantics of processes. By the way, the form of pattern matching used in Join cannot be rendered in CPC. Indeed, in a process like **def** $a(x) \mid b(y) \triangleright P$ **in** R, process R can independently produce the outputs on a and b

needed to activate P. This would correspond to some form of "unordered and multiparty pattern matching" that is far from the design choices of CPC.

Other complex notions of bisimulation equivalences for process calculi are [5,30]. However, these exploit environmental knowledge, whereas in our work we do not have such knowledge and need only satisfy compatibility.

Future Work. One interesting path of further development is to introduce types into CPC and extend the pattern unification mechanism by taking types into account, as done e.g. in [9]. The study of *typed* equivalences would then be the most natural path to follow, by combing the theory in this paper with the assumed types. Another intriguing direction is the introduction of richer forms of pattern matching, based, e.g., on regular expressions [22]; in this case, it would be very challenging to devise the ordering on patterns that defines the 'right' bisimulation. A natural way to follow is Kozen's axiomatization for inclusion of regular language [24]. Indeed, in this proof system, a regular expression e_1 is smaller than e_2 if and only if every string belonging to the language generated by e_1 also belongs to the language generated by e_2. This corresponds to the same intuition as our ordering on patterns (Lemma 1), once we consider the language generated by a pattern as the set of patterns that it matches, together with the associated substitutions.

References

1. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proc. of POPL, pp. 104–115. ACM (2001)
2. Abadi, M., Gordon, A.: A calculus for cryptographic protocols: The spi calculus. Information and Computation 148(1), 1–70 (1999)
3. Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous π-calculus. Theoretical Computer Science 195(2), 291–324 (1998)
4. Bengtson, J., Johansson, M., Parrow, J., Victor, B.: Psi-calculi: a framework for mobile processes with nominal data and logic. Logical Methods in Computer Science 7(1) (2011)
5. Boreale, M., De Nicola, R., Pugliese, R.: Proof techniques for cryptographic processes. SIAM J. Comput. 31(3), 947–986 (2001)
6. Buscemi, M.G., Montanari, U.: Open bisimulation for the concurrent constraint pi-calculus. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 254–268. Springer, Heidelberg (2008)
7. Busi, N., Gorrieri, R., Zavattaro, G.: A process algebraic view of linda coordination primitives. Theoretical Computer Science 192(2), 167–199 (1998)
8. Carbone, M., Maffeis, S.: On the expressive power of polyadic synchronisation in pi-calculus. Nordic Journal on Computing 10(2), 70–98 (2003)
9. Castagna, G.: Patterns and types for querying xml documents. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 1–26. Springer, Heidelberg (2005)
10. Ciancarini, P., Gorrieri, R., Zavattaro, G.: Towards a calculus for generative communication. In: Proc. of FMOODS, pp. 283–297. Chapman & Hall (1996)
11. de Boer, F.S., Klop, J.W., Palamidessi, C.: Asynchronous communication in process algebra. In: Proc. of LICS, pp. 137–147. IEEE (1992)
12. De Nicola, R., Gorla, D., Pugliese, R.: Basic observables for a calculus for global computing. Information and Computation 205(10), 1491–1525 (2007)

13. De Nicola, R., Pugliese, R.: A process algebra based on linda. In: Hankin, C., Ciancarini, P. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 160–178. Springer, Heidelberg (1996)
14. Fournet, C., Gonthier, G.: The reflexive cham and the join-calculus. In: Proc. of POPL, pp. 372–385. ACM Press (1996)
15. Fournet, C., Laneve, C.: Bisimulations in the join-calculus. Theoretical Computer Science 266(1-2), 569–603 (2001)
16. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems 7(1), 80–112 (1985)
17. Given-Wilson, T., Gorla, D., Jay, B.: Concurrent pattern calculus. In: Calude, C.S., Sassone, V. (eds.) TCS 2010. IFIP AICT, vol. 323, pp. 244–258. Springer, Heidelberg (2010)
18. Given-Wilson, T.: Concurrent pattern unification (2012), http://www.progsoc.org/~sanguinev/files/GivenWilson-PhD-simple.pdf
19. Given-Wilson, T., Gorla, D., Jay, B.: Concurrent pattern calculus (extended version), http://wwwusers.di.uniroma1.it/~gorla/papers/cpc-full.pdf
20. Given-Wilson, T., Gorla, D., Jay, B.: Concurrent pattern calculus. In: Calude, C.S., Sassone, V. (eds.) TCS 2010. IFIP AICT, vol. 323, pp. 244–258. Springer, Heidelberg (2010)
21. Honda, K., Yoshida, N.: On reduction-based process semantics. Theoretical Computer Science 151(2), 437–486 (1995)
22. Hosoya, H., Pierce, B.: Regular expression pattern matching for XML. Journal of Functional Programming 13(6), 961–1004 (2003)
23. Jay, B., Given-Wilson, T.: A combinatory account of internal structure. Journal of Symbolic Logic 76(3), 807–826 (2011)
24. Kozen, D.: A completeness theorem for kleene algebras and the algebra of regular events. Information and Computation 110(2), 366–390 (1994)
25. Milner, R.: The polyadic π-calculus: A tutorial. In: *Logic and Algebra of Specification*. Series F, NATO ASI, vol. 94. Springer (1993)
26. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I/II. Information and Computation 100, 1–77 (1992)
27. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
28. Parrow, J., Victor, B.: The fusion calculus: Expressiveness and symmetry in mobile processes. In: Proc. of LICS, pp. 176–185. IEEE Computer Society (1998)
29. Sangiorgi, D.: A theory of bisimulation for the pi-calculus. Acta Informatica 33(1), 69–97 (1996)
30. Sangiorgi, D., Kobayashi, N., Sumii, E.: Environmental bisimulations for higher-order languages. ACM Trans. Program. Lang. Syst. 33(1), 5 (2011)

Component-Based Autonomic Managers for Coordination Control*

Soguy Mak Karé Gueye[1], Noël de Palma[1], and Eric Rutten[2]

[1] LIG / UJF, Grenoble, France
{soguy-makkare.gueye,noel.depalma}@imag.fr
[2] LIG / INRIA, Grenoble, France
eric.rutten@inria.fr

Abstract. The increasing complexity of computing systems has motivated the automation of their administration functions in the form of autonomic managers. The state of the art is that many autonomic managers have been designed to address specific concerns, but the problem remains of coordinating them for a proper and effective global administration. In this paper, we define controllable autonomic managers encapsulated into components, and we approach coordination as their synchronization and logical control. We show that the component-based approach supports building such systems with introspection, adaptivity and reconfiguration. We investigate the use of reactive models and discrete control techniques, and build a hierarchical controller, enforcing coherency properties on the autonomic managers at runtime. One specificity and novelty of our approach is that discrete controller synthesis performs the automatic generation of the control logic, from the specification of an objective, and automata-based descriptions of possible behaviors. Experimental validation is given by a case-study where we coordinate two self-optimization autonomic managers and self-repair in a replicated web-server system.

Keywords: Adaptive and autonomic systems, Coordination models and paradigms, Software management and engineering, Discrete control.

1 Coordinating Autonomic Loops

1.1 The Need for Coordination Control

The administration of distributed systems is automated in order to avoid human manual management because of cost, duration and slowliness, and error-proneness. Autonomic administration loops provide for management of dynamically reconfigurable and adaptive systems. The architecture of an autonomic system is a loop defining basic notions of Managed Element (ME) and Autonomic Manager (AM). The ME, system or resource is monitored through sensors. An analysis of this information is used, in combination with knowledge about the system, to plan and decide upon actions. These reconfiguration operations

* This work is supported by the ANR INFRA project Ctrl-Green.

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 75–89, 2013.

are executed, using as actuators the administration functions offered by the system API. Self-management issues include self-configuration, self-optimization, self-healing (fault tolerance and repair), and self-protection. In this work we consider AMs for self-optimization and self-repair.

Complex and complete autonomic systems feature numerous different loops for different purposes, managing different dimensions. This is causing a problem of co-existence, because it is hard to avoid inconsistencies and possible interferences. Therefore, coordination is recognized as an important challenge, not completely solved by present research, and requiring special attention [11]. Coordinating AMs can be seen as the problem of synchronization and logical control of administration operations that can be applied by AMs on the MEs in response to observed events. Such an additional layer, above the individual administration loops, constitutes a coordination controller. AMs are considered as being MEs themselves, with an upper-level AM for their coordination. The state of the art in designing these hierarchical coordination controllers is to develop them, e.g., using metrics dedicated to the aspect around which coordination is defined, like energy and performance [6]. As a hand-made methodology, this remains complex and error-prone, and hard to re-use.

1.2 Coordination as Discrete Control of Components

Our contribution is an automated methodology for the coordination control of autonomic managers. A novelty is that it provides for concrete design assistance, in the form of the automatic generation, from a Domain-Specific Language (DSL), of the control logic, for a class of coordination problems expressed as invariance properties on the states of the AMs. The benefit is manifold: the designer's work is eased by the automatic generation, the latter is done based on formal techniques insuring correctness of the result, and the re-use of designs is facilitated by the automated generation of the control logic.

We identify the needs for the coordination of AMs, which have to be instrumented in order to be observable (e.g., current state or execution mode) and controllable (e.g., suspend activity), so that they can be coordinated. For this, component-based approaches provide us with a well-structured framework. Each component is defined separately, independently of the way it can be used in assemblies. They also have to be equipped with a model of their behavior, for which we use Finite state Machines (FSM), also called automata.

We specify the coordination policy as a property between the states of the AMs, like for example mutual exclusion, which must be kept invariant by control. Given this control objective, and the behavioral model FSMs, we use techniques stemming from Control Theory to obtain the controller. Such techniques have recently been applied to computing systems, especially using classical continuous control models, typically for quantitative aspects [10]. The advantage is that they ensure important properties on the resulting behavior of the controlled system e.g., stability, convergence, reachability or avoidance of some evolutions. We use discrete control techniques, especially Discrete Controller Synthesis (DCS) [5], well adapted for logical or synchronization purposes. In this work, we focus

on software engineering and methodology; formal aspects available elsewhere [8] are not in the scope of this paper. Our work in this paper builds upon previous preliminary results [9] where we had considered a specific experiment in coordinating two administration loops (self-sizing and DVFS), implemented and tested as a simple prototype. Here, our new contributions are:

1. a generalized method, leveraged to the level of a component-based approach, to design controllable AMs, with a DSL for describing administration behavior as automata, with controllable points at the interfaces;
2. a method for the specification and generation of coordination controllers enforcing invariance properties, based on the formal DCS technique;
3. a fully implemented experimental validation of a case-study coordinating self-optimization autonomic managers (self-sizing and DVFS), completed with a new one for self-repair; we thus demonstrate reusability by combining the same sizing loop in a new context, with a new coordination policy.

In the remainder of the paper we will make recalls in Section 2 on component-based approaches, and the reactive language BZR. The kernel of our contribution is first in Section 3 where we describe the method for instrumenting a basic AM component in order to become an ME itself; and then in Section 4 where we propose construction methods on these bases for composite components, where coordination of the AMs is managed. An additional contribution is the experimental validation by the case study in Section 5, with self-optimization and self-repair autonomic managers in a replicated web-server system.

2 Background: Components and Reactive Control

2.1 Component Model

In classical component models, a component is a run-time entity that is encapsulated, and that has a distinct identity. A component has one or more interfaces. An interface is an access point to a component, that supports a finite set of service. Interfaces can be of two kinds: (i) server interfaces, which correspond to access points accepting incoming service calls, and (ii) client interfaces, which correspond to access points supporting outgoing service calls. Communication between components is only possible if their interfaces are connected through an explicit binding. A component can be composite, i.e. defined as an assembly of several subcomponents, or primitive, i.e. encapsulating an executable program. The above features (hierarchical components, bindings between components, strict separation between component interfaces and component implementation) are representative of a classical component model.

Reflexive component models extend these features to allow well scoped introspection and dynamic reconfiguration capabilities over component's structure. They perfectly match our needs since they are endowed with controllers, which provide access to component internals, allowing for component introspection and control of their behavior. These general reflexive component-based concepts

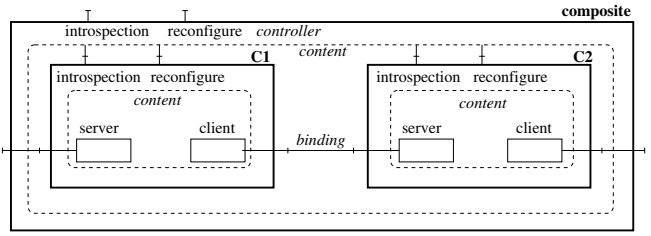


Fig. 1. A composite component

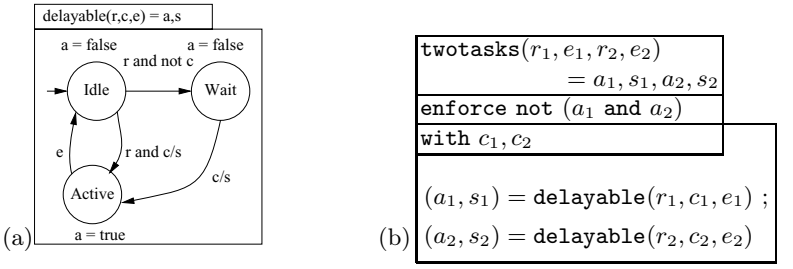


Fig. 2. Delayable task: (a) graphical syntax ; (b) exclusion contract

have a number of particular instances, amongst which Fractal [4]. This reflexive component model comes with several useful forms of controllers, which can be combined and extended to yield components with different control interfaces. This includes controllers for: (i) attributes, which are configurable through getter and setter methods, (ii) bindings, to (dis)connect client interfaces to server interfaces, (iii) contents, to list, add and remove subcomponents and (iv) life-cycle, to give an external control over component execution, including starting and stopping execution.

Figure 1 shows an example of a composite component with: functional and control interfaces, a content built from two subcomponents (C1 and C2) that implement the functional interfaces and, a controller that implements the control interfaces. The controller in the composite can react e.g., to a reconfiguration access by removing subcomponent C2: i.e., stopping it, unbinding it from C1 and from the composite's client interface, re-binding C1's client to the composite, and uninstalling C2's code. The experimental validation in Section 5 relies on the Java-based reference implementation of Fractal (Julia): in this framework, a component is represented by a set of Java objects.

2.2 Reactive Languages and BZR

Automata and Data-Flow Nodes. We briefly introduce, with examples, the basics of the Heptagon language; due to space limitations, formal definitions are not recalled [8]. It supports programming of nodes, with mixed synchronous data-flow equations and automata, with parallel and hierarchical composition.

The basic behavior is that at each reaction step, values in the input flows are used, as well as local and memory values, in order to compute the values in the output flows for that step. Inside the nodes, this is expressed as a set of equations defining, for each output and local, the value of the flow, in terms of an expression on other flows, possibly using local flows and state values from past steps. Figure 2(a) shows a small program in this language. It programs the control of a `delayable` task, which can either be idle, waiting or active. When it is in the initial `Idle` state, the occurrence of the `true` value on input `r` *requests* the starting of the task. Another input `c` can either allow the activation, or temporarily block the request and make the automaton go to a `waiting` state. Input `e` notifies termination. The outputs represent, resp., `a`: activity of the task, and `s`: triggering starting operation in the system's API.

Such automata and data-flow reactive nodes can be reused by instantiation, and composed in parallel (noted `";"`) and in a hierarchical way, as illustrated in the body of the node in Figure 2(b), with two instances of the `delayable` node. They run in parallel: one global step corresponds to one local step for every node, with possible communication through share flows. The compilation produces executable code in target languages such as C or Java, in the form of an initialisation function *reset*, and a *step* function implementing the transition function of the resulting automaton, which takes incoming values of input flows gathered in the environment, computes the next state on internal variables, and returns values for the output flows. This function is called at relevant instants from the infrastructure where the controller is used.

Contracts and Control. BZR (`http://bzr.inria.fr`) extends Heptagon with a new behavioral contract [8]. Its compilation involves *discrete controller synthesis* (DCS), a formal operation [5] on a FSM representing possible behaviors of a system, its variables partitioned into controllable ones and uncontrollable ones. For a given control objective (e.g., staying invariantly inside a subset of states, considered "good"), the DCS algorithm automatically computes, by exploration of the state graph, the constraint on controllable variables, depending on current state, for any value of the uncontrollables, so that remaining behaviors satisfy the objective. This constraint is inhibiting the minimum possible behaviors, therefore it is called *maximally permissive*. Algorithms are related to model checking techniques for state space exploration; they are exponential in the states, but can manage our models built at coarse grain abstraction level.

Concretely, the BZR language allows for the declaration, using the `with` statement, of controllable variables, the value of which are not defined by the programmer. These free variables can be used in the program to describe choices between several transitions. They are then defined, in the final executable program, by the controller computed by DCS during compilation, according to the expression given in the `enforce` statement. BZR compilation invokes a DCS tool, and inserts the synthesized controller in the generated executable code, which has the same structure as above: *reset* and *step* functions.

Figure 2(b) shows an example of contract coordinating two instances of the `delayable` node of Figure 2. The `twotasks` node has a `with` part declaring

controllable variables c_1 and c_2, and the `enforce` part asserts the property to be enforced by DCS. Here, we want to ensure that the two tasks running in parallel will not be both active at the same time: `not` (A1 `and` A2). Thus, c_1 and c_2 will be used by the computed controller to block some requests, leading automata of tasks to the waiting state whenever the other task is active. The constraint produced by DCS can have several solutions: the BZR compiler generates deterministic executable code by favoring, for each controllable variable, value `true` over `false`, in the order of declaration in the `with` statement.

3 Designing Controllable AMs

3.1 Design of an AM

Controllable AM components have to share features of an AM, as well as of an ME, as we recall here. Basic features required for a system to be an ME managed in an autonomic fashion have been identified in previous work e.g., in the context of component-based autonomic management [13]. It must be observable and controllable, and we need to know its possible behaviors, for which we use automata models. *Observability* goes through a capacity of introspection, exhibiting internals to the outside world, particularly to an AM in charge of management, in a context with other MEs. In the autonomic framework, it corresponds to the sensors. In components as in Figure 1, it is in introspection interfaces that can be called from outside. At implementation level, it can be for example `get` functions accessing internal variables. *Controllability* is defined by the capacity to change features inside the ME from the outside. In the autonomic framework, it corresponds to the actuators. In components, it corresponds to reconfiguration actions e.g., mode switching, or attribute updating, as mentioned in Section 2.1. At implementation level, it can be done in different ways, from interceptors on the interfaces in Java, to actions modifying the structure as with Fscript [7]. In short, the system is equipped with a library of sensors/monitors and actions, with an API given by the system, and programmed by the designer.

The AM is a reactive node, transforming flows of sensor observations into flows of reconfiguration actions. Internally, it features decision-making mechanisms, which can range from simple threshold triggers to elaborate MAPE-K. They can involve quantitative measures and continuous control, or logical aspects modeled as FSMs. Therefore a reactive language such as BZR can be used as a DSL for the decision part on the AMs: it offers high-level programming, as well as formal tools to bring safe design and guaranteed behaviors. In components, the AM is the controller in Figure 1, or it can be an additional component in the assembly of MEs, bound with the appropriate control interface of other subcomponents. At implementation level, it reacts to notifications by treating them, depending on applications, in FIFO order or considering most recent values. It calls the action functions of the controlled MEs as above, or modifies the composite by ME additions or removals.

We aim at combining the two, to have AMs which can be manipulated as MEs, which involves the same features: making AMs observable and controllable.

3.2 Controllable AMs

To make an AM *observable*, relevant states of the component are exhibited to the outside. They convey information necessary for coordination decisions. Automata represent states of components, w.r.t their activity and/or ability to be coordinated, but independently of the coordination policy. The most basic case, for an AM as for any ME, is to distinguish only idle state (where the AM is not performed) and activity (with actual effect on the ME). Beyond this, one can have a more refined, *grey box* model, distinguishing states corresponding to phases in a sequence, where the AM can be stopped or suspended without loss of consistency. States support monitoring e.g., to explicitly represent that some bound is reached, such as minimum number of a resource, or maximum capacity.

To make an AM *controllable*, we use the transitions between states, which are guarded by conditions, and can fire reconfiguration actions. They represent possible choices offered by the AM between different reactions, in order for an external coordinator to choose between them. They offer control points to be used according to a given policy. Hence they give the controllability of the AM.

Figure 3(a) shows how an FSM can be an instantiation of the general autonomic loop, with knowledge on possible behaviors represented as states, and analysis and planning as the automaton transition function. In the autonomic framework observability comes through additional outputs, as shown by dashed arrows in Figure 3(a) for an FSM AM, exhibiting (some) of the knowledge and sensor information. Controllability corresponds to having the AM accept additional input for control. Its values can be used in the guards to guide choices between different transitions. Such automata are specified in BZR, benefiting from its modular structures and compiler. In BZR, a transition with a condition `e and c`, where `e` is a Boolean expression, will be taken only if `c` is `true`: it can inhibit or enable the transition.

In components, automata are associated with the component controller in the membrane as in Figure 3(b), and are updated at runtime to reflect the current state of component. Automata outputs trigger actions in reconfiguration control interfaces, such as `"stop AM"`, or convey information to upper layers.

At implementation level, the modular compilation of BZR generates for each node two methods (in C or Java): *reset* and *step*. *reset* initializes internal variables and is called only once. *step* makes the internal variable reflect continuously

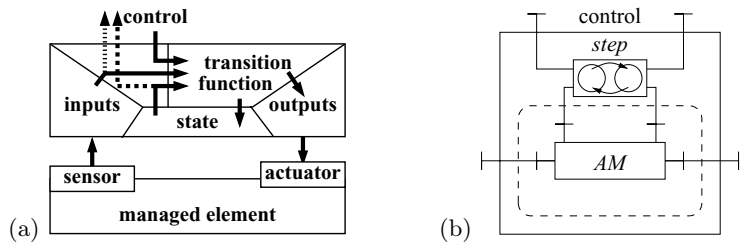


Fig. 3. Controllable AM: (a) case of a FSM manager; (b) wrapped component

the state of the component. Each call to *step* takes as inputs relevant information from inside, through the bindings to sub-components, or from outside, through control interfaces of the composite, and control interfaces feature functions accessing returned values. Returned values trigger reconfiguration actions.

4 Coordinated Assembly of Controllable AMs

4.1 Coordination Behaviors, Objective, and Controller

We now present how such controllable AMs can be assembled in composites, where the coordination is performed in a hierarchical framework. The AMs and other components involved in the coordination (which can be composites themselves, recursively) are grouped into a composite. It orchestrates their execution, using the possibilities offered by each of them, through control interfaces, in order to enforce a coordination policy or strategy. We base our approach on the hierarchical structure in Figure 4: the top-level AM coordinates lower-level ones. We describe the problem with one level of hierarchy, but it can be recursive.

The first thing is to construct a model of the global behavior of the assembly of components. This is done using the local automata from each of the concerned subcomponents: they are composed in parallel in a new BZR node: we then have a model of all the possible behaviors in the absence of control. Controllable variables must be identified and designated i.e., the possible choice points, which will be the actuators offered to the discrete controller to enforce the coordination.

Specification of the coordination policy is done by associating a BZR contract to this global behavior. It can make reference to the information explicitly exhibited by subcomponents, and to inputs of the coordinator. The control objective is to restrict possible behaviors to states where a Boolean expression will remain invariantly true, whatever the sequences of uncontrollables.

Once we have a global automaton model of the behavior, a list of controllables, and a control objective, we manually encode the control problem into the BZR language as a contract. The BZR language compiler, and its associated

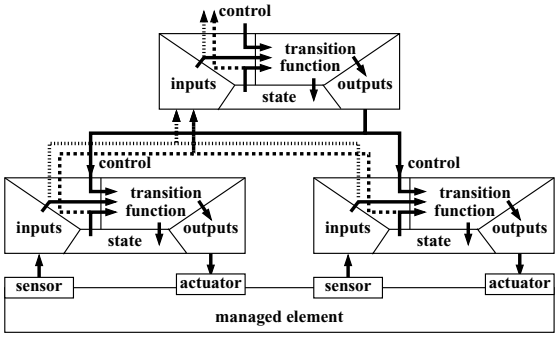


Fig. 4. Autonomic coordination for multiple administration loops

DCS tool, solves the control problem by automatically computing the controller ensuring, by automated formal computation, a correct behavior respecting the contract. It must be noted that as in any control or constraint problem, it may happen that the system does not offer enough controllability for a solution to exist. Either it lacks actuators, or sensors, or explicit states to base its decisions on. In these cases, the DCS fails, which constitutes a diagnostic of non-existence of a solution for the current coordination problem. Then the system has to be redesigned, either by relaxing the contract, or by augmenting controllability. Another meaningful point is that the BZR language features hierarchical contracts [8], where the synthesis of a controller can use knowledge of contracts enforced by sub-nodes, without needing to go into the details of these nodes. This favors scalability by decomposing the synthesis computation. It also corresponds very closely to the hierarchical structure proposed here, and can be exploited fully.

4.2 Hierarchical Architecture

In the autonomic framework our approach defines a hierarchical structure as shown in Figure 4. Given that AMs have additional outputs exhibiting their internals, and additional inputs representing their controllability, an upper-level AM can perform their coordination using their additional control input to enforce a policy. Considering the case of FSM managers makes it possible to encode the coordination problem as a DCS problem. The transition function of this upper-level AM is the controller synthesized by DCS.

In components, this translates to a hierarchical structure where a composite features a controller, bound with the outside and with subcomponents, through control interfaces. This way it can use knowledge on life-cycles of subcomponents in order to decide on reconfiguration actions to execute for their coordination.

At implementation level, there is an off-line phase, where the BZR compilation is used. It involves extracting the relevant automata from AMs descriptions : to each of them is associated a BZR node. These nodes are used to compose a new BZR program, with a contract, declaring controllable variables in the `with` statement. The policy has to be specified in the `enforce` statement. The complete BZR program is compiled, and code is generated as previously described. The resulting *step* function is integrated at the coordinator component level, calling steps from sub-components. At run-time, in our implementation, the controller in the composite component is responsible of executing the *step* method call: it gathers the necessary information from the sub-components, then executes the *step*, which interacts with the local *step* functions in subcomponents. It then triggers and propagates reconfiguration commands to sub-components based on the returned result, as discussed before.

4.3 Change Coordination Policy

This separation of concerns, between description of possible behaviors, and coordination policy, favors the possibility to change policy without changing the individual components. For example, if we wanted to change priorities between

AMs, this would impact the contract but not the individual components. On the other hand, if for the same overall policy we wanted to use other components with a different implementation of the same management, we can switch an AM by another one and recompute the coordination controller. The fact that our approach is modular, and that it uses automated DCS techniques, facilitates reuse of components, and also the modification of systems, by re-compilation.

5 Case Study: Coordinating Administration Loops

5.1 Description of the Case Study

We consider the coordination of two administration loops for resource optimization (self-sizing) and server recovery (self-repair) for the management of a replicated servers system based on the load balancing scheme.

Self-sizing. It addresses the resource optimization of a replication-based system. It dynamically adapts the degree of replication depending on the CPU load of the machines hosting the active servers. Its management decisions rely on thresholds, `Minimum` and `Maximum`, delimiting the optimal CPU load range. It computes a moving average of the collected CPU load (`Avg_CPU`) and performs operations if the average load is out of the optimal range: Adding operations (`Avg_CPU` $>=$ `Maximum`), removal operations (`Avg_CPU` $<=$ `Minimum`).

Self-repair. It addresses fail-stop failure of a machine hosting a single or replicated server. Its management decisions rely on Heartbeat. When the machine hosting the server becomes non-responsive, it redeploys the server towards another machine selected from an unused machine pool. Then it updates the new server configuration from the configuration of the failed server and starts it.

Co-existence and Coordination Problem. Failures can trigger incoherent management decisions by self-sizing. The failure of the load balancer can cause an underload of the replicated servers since the latter do not receive requests until the load balancer is repaired. The failure of a replicated server can cause an overload of the remaining servers because they receive more requests due to the load balancing. A strategy to achieve an efficient resource optimization could be to (1)*avoid removing a replicated server when the load balancer fails*, and (2) *avoid adding a replicated server when one fails.*

5.2 Modelling and Control for Coordinating the AMs

We pose the coordination problem as a discrete control problem, first modeling the behaviors of AMs with automata, then defining controllables and a control objective, in order to finally apply DCS.

Modelling AMs. Figure 5 shows the automata modelling both the behaviors and the control of the self-sizing manager. The two external ones model the control of the adding (resp. removal) operations. This is done with the local flows `disU` (resp. `disD`), which, when `true`, prevent transitions where output `add`

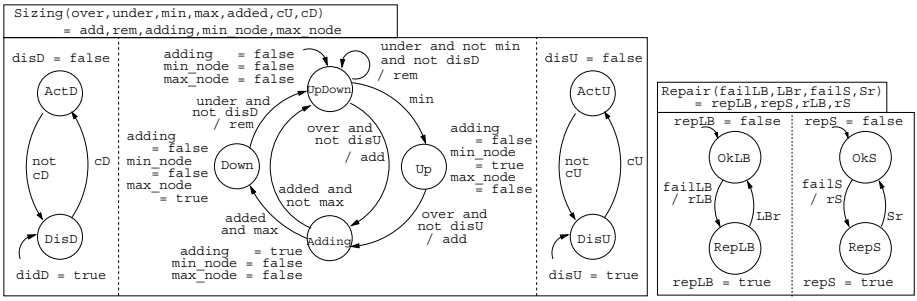


Fig. 5. Managers models : self-sizing (left) and self-repair (right)

(resp. `rem`) triggers operations. The center one models the behaviors. Initially in the `UpDown` state, when an Overload occurs and adding operations are allowed, the manager requests a new server, and goes to the `Adding` state and can no longer perform operations. It awaits until the server becomes active (`node_added` is `true`), and returns back to the `UpDown` state or goes to the `Down` state if the degree of replication is maximal. The `Down` state is left once one server is removed upon an `Underload` event. The `Up` state is the state in which the degree of replication is minimal. The manager cannot remove server but can add server upon `Overload` event if allowed.

Figure 5 shows the automata modelling the behaviors of the self-repair managers for the load balancer (LB) and the replicated servers (S). The right automaton concerns servers, and is initially in `OkS`. When `failS` is `true`, it emits repair order `rS` and goes to the `RepS` state, where `repS` is `true`. It returns back to `OkS` after repair termination (`Sr` is `true`). Repair of the LB is similar.

Coordination Controllers. The automata are composed in order to have the global behavior model, and a contract specifies the coordination policy. Automata in Figure 5 are composed. The policies (1) and (2) in Section 5.1 give a contract as follows:

```
enforce ((repLB implies disD) and (repS implies disU)) with Cu,Cd
```

With implications, DCS keeps solutions where `disU, disD` are always `true`, correct but not progressing; but BZR favors `true` over `false` (Section 2.2) for `cU, cD`, hence enabling Sizing when possible.

Compilation. The BZR compiler generates the corresponding Java code of the BZR program. Two main methods allow to interact with the program: `reset` for initializing it and `step`. The latter takes as parameters all the automata inputs and returns all the automata outputs.

5.3 Experiments

We apply our approach for coordinating one instance of self-sizing and two instances of self-repair managers for the management of a replicated-based system.

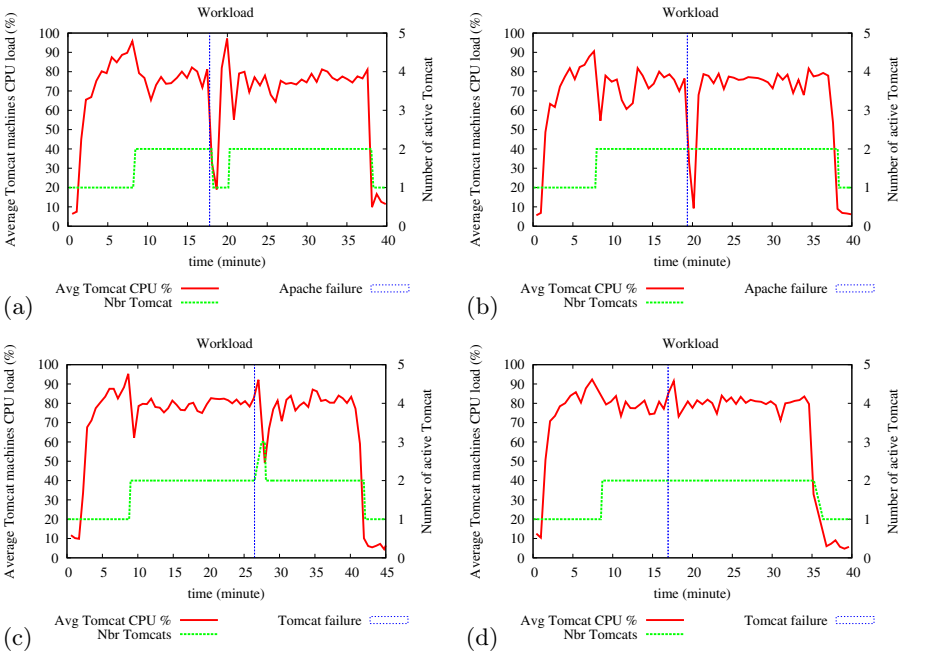


Fig. 6. Apache (a,b) and Tomcat (c,d) server failure ; uncoordinated and coordinated

The system is composed of one Apache server (self-repair) and replicated Tomcat servers (self-sizing and self-repair). Homogeneous machines are used to host the Tomcat servers. Initially during each execution, the system is started with one Apache server and one Tomcat server. We inject a workload in two phases, a ramp-up workload followed by a constant workload. We wait until there are two active Tomcat servers before injecting a failure.

Figure 6(a,b) shows executions in which the Apache server fails, which causes a decrease of the CPU utilization of the machines hosting the Tomcat servers. In the uncoordinated execution (Figure 6(a)), this leads to the removal of a Tomcat but once the Apache is repaired another Tomcat is added. In the coordinated execution (Figure 6(b)) the decrease does not lead to the removal of a Tomcat since the coordination controller inhibits removal operations because of the failure. Figure 6(c,d) shows executions in which a Tomcat server fails, which causes an increase of the CPU utilization of the remaining Tomcat servers. While in the uncoordinated execution (Figure 6(c)) this increase leads to adding a new tomcat (which is removed after the repair), in the coordinated execution (Figure 6(d)) no adding operation is performed. The adding operations are inhibitted by the coordination controller which is aware of the Tomcat failure.

5.4 Reusability of Models: Integrating the Dvfs Manager

We consider the management of the CPU frequency for more resource optimization. Each machine hosting a replicated server is equipped with a Dvfs manager.

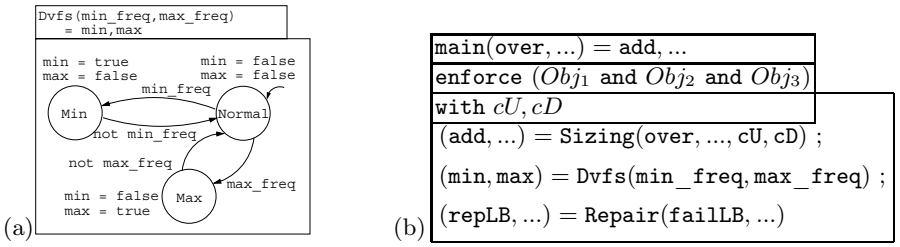


Fig. 7. DVFS model, and its Coordination with sizing and repair

This latter dynamically increases or decreases the CPU-frequency of the machine depending on the CPU load. Its management decisions, like self-sizing, rely on thresholds, `Minimum` and `Maximum`, delimiting the optimal CPU load range. In a context where Dvfs are activated, a strategy to improve resource optimization could be to delay as long as possible adding a new server when the machines hosting active servers are not in their maximum CPU frequency: (3)*avoid adding unless all machines are at maximum speed.*

We model the global states of the set of Dvfs in Figure 7(a). Initially in the `Normal` state, when the input `min_freq` (resp. `max_freq`) is `true`, it goes to the `Min` (resp. `Max`) state when the CPU frequency of all machines hosting replicated servers is minimal (resp. maximal) and the output `min` (resp. `max`) is `true`. It returns back the `Normal` state if the CPU frequency of at least, one of the machines is neither maximal nor minimal. Coordinating the three AMs consists in adding the automaton for the Dvfs in the composition with the following contract to be enforced: ($not\ max$ **`implies`** $disU$) representing the policy (3), as shown in Figure 7(b), with Obj_i representing policy (i) $(i = 1, 2, 3)$.

6 Related Work and Discussion

The general question of coordinating autonomic managers remains an important challenge in Autonomic Computing [11] although it is made necessary in complete systems with multiple loops, combining dimensions and criteria. Some works propose extensions of the MAPE-K framework in order to allow for synchronization [15], which can be e.g., through the access to a common knowledge [2]. A distinctive aspect of our approach is to rely on FSM-based behavioral models, amenable to formal techniques like verification or DCS. Coordination of multiple energy management loops is done in various ways, e.g., by defining power vs. performance tradeoffs based on a multi-criteria utility function in a non-virtualized environment [6]. These approaches seem to require modifying AMs for their interaction, and to define the resulting behavior by quantitative integration of the measure and utilities, which relies on intuitive tuning values, not handling logical synchronization aspects. We coordinate AMs by controling their logical state, rather than modifying them.

Component-based frameworks are classically associated with implementations offering APIs for sensing and actuating. For example FScript [7] is a middleware

layer offering relatively high level support for programming complex reconfiguration actions. However, there is no support in expressing explicitly and directly the set of configurations or modes, and the switches between them: they have to be managed by tedious manual programming with side effects. Our work proposes higher level programming of control aspects, with first-class language constructs explicitly representing control states and events. The formally based semantics of our language support enables concrete use of verification and synthesis techniques. Relating component-based frameworks and formal models is widely done, e.g. relying on process calculi for abstract reasoning on composition constructs. Closer to the concrete modeling of behavioral aspects, some modeling approaches offer access to verification as model checking [3]. We specifically concentrate on reconfiguration control, not on the general component approach, and it relates this aspect of Fractal with the synchronous approach to reactive systems, and its support for correct program design, particularly DCS. [1] proposes a framework allowing multiple autonomic managers for the management of a Behavioral Skeleton. The coordination of the managers is ensured through consensus which seems to be manually implemented, which can become complex. [12] proposes a component-based programming framework to build an autonomic application as the composition of autonomic components, with agents based on rules with priority level for conflicting decisions resolution.

Concerning decision and control, some approaches rely upon Artificial Intelligence and planning [14] which has the advantage of managing situation where configurations are not all known in advance, but the corresponding drawback of costly run-time exploration of possible behaviors, and lack of insured safety of resulting behaviors. Our work adheres to the methodology of control theory, and in particular DES, applied to computing systems [10]. Compared to traditional error-prone programming followed by verification and debugging, such methods bring correctness by design of the control. Particularly, DCS offers automated generation of the coordination controller, facilitating design effort compared to hand-writing, and modification and re-use (see Section 4.3). Also, maximal permissivity of synthesized controllers is an advantage compared to over-constrained manual control, impairing performance even if correct. Applications of DCS to computing systems have been rare until now e.g. to address deadlock avoidance [16]. Compared to this, we consider more user-defined objectives.

7 Conclusion

We address the problem of coordinating multiple AMs using a component-based approach. We define a method for the componentization of AMs as MEs, equipped with automata-based behavioral models. The control of coordinated AMs is based on a formal control technique : DCS, which performs automatic generation. It support addition of new AMs, without re-designing the whole system, and is encapsulated in a design process for non-expert users. A fully implemented case-study, for sizing and repair AMs, gives experimental validation.

In perspective, we are busy evaluating our approach for coordination of a variety of autonomic administration loops in the context of an industrial

data-center. We have ongoing work on managing coordination in a multi-tier system, involving intra-tier and inter-tier coordination, hierarchically. We consider using more elaborate control techniques, with DCS for optimal control, or the distribution of the controllers. Finally, we have ongoing work on programming language support for our methodology, integrating the whole approach in the compilation process of a component-based language.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Xhagjika, V.: LIBERO: A framework for autonomic management of multiple non-functional concerns. In: Guarracino, M.R., Vivien, F., Träff, J.L., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Di Martino, B., Alexander, M. (eds.) Euro-Par-Workshop 2010. LNCS, vol. 6586, pp. 237–245. Springer, Heidelberg (2011)
2. Alvares de Oliveira Jr., F., Sharrock, R., Ledoux, T.: Synchronization of multiple autonomic control loops: Application to cloud computing. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 29–43. Springer, Heidelberg (2012)
3. Barros, T., Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural Models for Distributed Fractal Components. Res. Report RR-6491, INRIA (2008)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The fractal component model and its support in java. Software – Practice and Experience (SP&E) 36(11-12) (September 2006)
5. Cassandras, C., Lafortune, S.: Introduction to Discrete Event Systems. Springer-Verlag New York, Inc., Secaucus (2006)
6. Das, R., Kephart, J.O., Lefurgy, C., Tesauro, G., Levine, D.W., Chan, H.: Autonomic multi-agent management of power and performance in data centers. In: Proc. Conf. AAMAS (2008)
7. David, P.-C., Ledoux, T., Léger, M., Coupaye, T.: FPath and FScript: Language support for navigation and reliable reconfiguration of fractal architectures. Annals of Telecommunications 64(1), 45–63 (2009)
8. Delaval, G., Marchand, H., Rutten, E.: Contracts for modular discrete controller synthesis. In: Proc. Conf. LCTES (2010)
9. Gueye, S.M.K., de Palma, N., Rutten, E.: Coordinating energy-aware administration loops using discrete control. In: Proc. Conf. ICAS (2012)
10. Hellerstein, J., Diao, Y., Parekh, S., Tilbury, D.: Feedback Control of Computing Systems. Wiley-IEEE (2004)
11. Kephart, J.: Autonomic computing: The first decade. In: Proc. Conf. ICAC (2011)
12. Liu, H., Parashar, M., Hariri, S.: A component based programming framework for autonomic applications. In: Proc. ICAC (2004)
13. Sicard, S., Boyer, F., De Palma, N.: Using components for architecture-based management: the self-repair case. In: Proc. Conf. ICSE (2008)
14. Sykes, D., Heaven, W., Magee, J., Kramer, J.: Plan-directed architectural change for autonomous systems. In: Proc. WS SAVCBS (2007)
15. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On interacting control loops in self-adaptive systems. In: Proc. Conf. SEAMS (2011)
16. Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., Mahlke, S.: The theory of deadlock avoidance via discrete control. In: Proc. ACM POPL (2009)

Multi-threaded Active Objects

Ludovic Henrio[1], Fabrice Huet[1], and Zsolt István[2]

[1] INRIA-I3S-CNRS, University of Nice Sophia Antipolis, France
[2] Department of Computer Science, ETH Zurich, Switzerland
ludovic.henrio@cnrs.fr, fabrice.huet@inria.fr,
zsolt.istvan@inf.ethz.ch

Abstract. Active objects offer a paradigm which simplifies writing distributed applications. Since each active object has a single thread of control, data races are prevented. However, this programming model has its limitations: it is deadlock-prone, and it is not efficient on multicore machines. To overcome these limitations, we present an extension of the active object model, called multi-active objects, that allows each activity to be multi-threaded. The new model is implemented as a Java library; it relies on method annotations to decide which requests can be run in parallel. It provides implicit parallelism, sparing the programmer from low-level concurrency mechanisms. We define the operational semantics of the multi-active objects and study the basic properties of this model. Finally, we show with two applications that our approach is easy to program and efficient.

Keywords: Concurrency and distribution, active-objects, multicore architectures.

1 Introduction

Writing distributed applications is a difficult task because the programmer has to face both concurrency and location-related issues. The active object [1–3] paradigm provides a solution by abstracting away the notions of concurrency and of object location. An object is said to be active if it has its own thread of control. As a consequence, every call to such an object will be some form of remote method invocation – a *request* – that is handled by the object's thread of control. Active objects are partly inspired by Actors [4, 5]: they share the same asynchronous treatment of messages, and ensure the absence of data race-conditions. They do differ however in how the internal state of the object is represented. Active objects are mono-threaded, which prevents data races without the use of synchronized blocks. Distributed computations rely on the absence of sharing between processes, allowing them to be placed on different machines.

In classical remote method invocation, the invoker is blocked waiting for the result of the remote call. Active objects, on the other hand, return futures [6] as placeholders for the result, allowing the invoker to continue its execution. Futures can be created and accessed either explicitly, like in Creol [7] and JCoBox [8], or implicitly as in ASP [3] (Asynchronous Sequential Processes) and AmbientTalk [9]. A key benefit of the implicit creation is that no distinction is made between synchronous (i.e., local) and asynchronous (i.e., remote) operations in the program. Hence, when the accessed object is remote, a future is immediately obtained. Similarly to their creation, the access

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 90–104, 2013.

to futures can happen either explicitly using operations like *claim* and *get*, or implicitly, in which case operations that need the real value of an object (*blocking* operations) automatically trigger synchronisation with the future update operation. In most active object languages, future references can be transmitted between remote entities without requiring the future to be resolved.

There are different models related to active objects. The ASP calculus [3] is a distributed active object calculus with futures; the ProActive library is its reference implementation. ASP has strong properties and is easy to program, but the strict sequential service of requests creates deadlocks. A different approach is provided by active objects languages with cooperative multithreading (Creol [7, 10], Jcobox [8]). In this model a thread serving a request can stop in order to let the service of another request progress. While no data race condition is possible, interleaving of the different request services triggered by the different release points makes the behaviour more difficult to predict; in particular Creol does not feature the determinism properties of ASP. Explicit future access, explicit release points, and explicit asynchronous calls make JCobox and Creol richer, but also more difficult to program.

This paper provides a new extension of the active object model with a *different trade-off* between expressiveness and ease of programming compared to the existing approaches. Our approach replaces the strict mono-threading that avoids data-races in active-objects, by an expressive and simple mechanism of request annotation that controls parallelism. Our approach also achieves better performance than classical active-objects on multi-core machines. Our contribution can be summarised as follows:

- A new programming model called *multi-active objects* is proposed (Section 2). It extends active objects with *local multi-threading*; this both enhances efficiency on multicore machines, and *prevents* most of the *deadlocks* of active objects.
- We rely on declarative *annotations* for expressing potential concurrency between requests, allowing easy and high-level expression of concurrency. However, the expert programmer can still use lower level concurrency constructs, like locks.
- We define the *operational semantics* of multi-active objects in Section 3. This semantics allows us to prove properties of the programming model.
- We experimentally show that *multi-active objects make writing of parallel and distributed applications easier*, but also that the *execution of programs based on multi-active objects is efficient* (Section 4). The programming model has been implemented as an extensions to a Java middleware[1].

Additionally, Section 5 compares our contribution with the closest languages, and finally Section 6 concludes the paper. Details on the semantics, the implementation, and the experiments are available in a research report [11].

2 Multi-active Object Programming Model

Illustrating example We will illustrate our proposal with an example inspired by a simple peer-to-peer network based on the CAN [12] routing protocol. In a CAN, data are stored in peers as key-value pairs inside a local datastore. Keys are mapped through a bijective function to the coordinates of a N-dimensional space, called the key-space.

[1] Available at: `www-sop.inria.fr/oasis/Ludovic.Henrio/java/PA_ma.zip`

The key-space is partitioned so that each key is owned by a single peer. A peer knows its immediate neighbours, and when an action concerns a key that does not belong to this particular peer, it routes the request towards the responsible peer according to the coordinates of the key. Our CAN example provides three operations: *join*, *add*, and *lookup*. When a new peer *joins* the network, it always joins an existing peer. This joined peer splits its partition of the key-space in two; it keeps one half and gives the other to the new peer. The *add* operation stores a key-value pair, and *lookup* retrieves it.

Active objects are a natural choice to implement this application by representing each peer with an active object. Locally, requests will be served one-by-one. This limits the performance of the application, and also possibly leads to deadlocks if re-entrant requests are issued. With multi-active objects, a peer will be able to handle several operations in parallel. We will illustrate multi-active objects by showing how to simply program a safe parallelisation of the CAN peers.

2.1 Assumptions and Design Choices

To allow active objects to serve several requests in parallel, we introduce multi-active objects that enable local parallelism inside active objects in a safe manner. For this, the programmer can annotate the code with information about concurrency by defining a compatibility relationship between *concerns*. This notion of annotation for concurrency share some common ideas with JAC[13]. For instance, in our CAN example we distinguish two non-overlapping concerns: one is network management (*join*) and another is routing (*add*, *lookup*). For two concerns that deal with completely disjoint resources it is possible to execute them in parallel, but for others that could conflict on resources (e.g. joining nodes and routing at the same time in the same peer) this must not happen. Some of the concerns enable parallel execution of their operations (looking up values in parallel in the system will not lead to conflicts), and others do not (a peer can split its zone only with one other peer at a time).

In the RMI style of programming, every remote invocation to an object will be run in parallel, as a result, data-races can happen on concurrently accessed data. A classic approach to solve this problem in Java is to protect concurrent executions by making all methods *synchronized*. This coarse-grain approach is inefficient when some of the methods could be run in parallel. Alternatively, the programmer can protect the data accesses by using low-level locking mechanisms, but this approach is too fine-grained and possibly error-prone.

By nature, active objects materialise a much safer model where no inner concurrency is possible. In this work, we look for a concurrency model that is more flexible than the single threaded active-objects, but more constrained and less error-prone than Java concurrency. We extend active-objects by assigning methods to *group*s (concerns). Then, methods belonging to compatible groups can be executed in parallel, and methods belonging to conflicting groups will be guaranteed not to be run concurrently. This way the application logic does not need to be mixed with low-level synchronisations. The idea is that *two groups should be made compatible if their methods do not access the same data, or if concurrent accesses are protected by the programmer,* and *the two methods can be executed in any order*. The programmer should declare as compatible both the non-conflicting groups, and the groups where the conflicting code has been protected by

means of locks or synchronised blocks. We chose annotations to express compatibility rules because we think that this notion is strongly dependent on the application logic, and should be attached to the source code.

We start from active objects *à la* ASP, featuring transparent creation, synchronisation, and transmission of futures. We think that the transparency featured by ASP and ProActive helps writing simple programs, and is not an issue when writing complex distributed applications. We also think that the non-uniform active objects of ASP, and JCoBox, reflect better the way efficient distributed applications are designed: some objects are co-allocated and only some of them are remotely accessible. In our model, only one object is active in a given activity but this work can easily be extended to multiple active-object per activity (i.e. cobox). An active object can be transformed into a multi-active object by applying the following design methodology:

- Without annotations, a multi-active object behaves identically to an active object, no race condition is possible, but no local parallelism is possible either.
- If some parallelism is desired, e.g. for efficiency reasons or because dead-locks appeared, each remotely invocable method can be assigned to a group. Then compatibility between the groups can be defined based on, for example, the variables accessed by each method. Methods belonging to compatible groups can be executed in parallel and out of the original order. This implies that two groups should only be declared compatible if the order of execution of methods of one group relatively to the other is not significant.
- If even more parallelism is required, the programmer has two non-exclusive options: either he protects the access to some of the variables by a locking mechanism which will allow him to declare more groups as compatible, or he realises that, depending on runtime conditions, such as invocation parameters or the object's state, some groups might become compatible and he defines a compatibility function allowing him to decide at runtime which request executions are compatible.

We assume that the programmer defines groups and their compatibility relations inside a class correctly. Dynamic checks or static analysis should be added to ensure, for example, that no race condition appear at runtime. Verifying that annotations written by the programmer are correct or even inferring them, e.g. [14], is out of scope here.

2.2 Defining Groups

The programmer can use an annotation, `Group`, to define a group and can specify whether the group is `selfCompatible`, i.e., two requests on methods of the group can run in parallel. The syntax for defining groups in the class header is shown on lines 1–5 of Figure 1.

Compatibilities between groups can be expressed as `Compatible` annotations. Each such annotation receives a set of groups that are pairwise compatible, as illustrated on lines 6–9 of Figure 1. A method's membership to a group is expressed by annotating the method's declaration with `MemberOf`. Each method belongs to only one group. In case no membership annotation is specified, the method belongs to an anonymous group that is neither compatible with other groups, nor self-compatible. This way, if no method of a class is annotated, the multi-active object behaves like an ordinary active object. The `MemberOf` annotation is shown on lines 11, 13, 15, and 17 of Figure 1.

```
@DefineGroups({
    @Group(name="join", selfCompatible=false)
    @Group(name="routing", selfCompatible=true)
    @Group(name="monitoring", selfCompatible=true)
})
@DefineRules({
    @Compatible({"join", "monitoring"})
    @Compatible({"routing", "monitoring"})
})
public class Peer {
@MemberOf("join")
    public JoinResponse join(Peer other) { ... }
@MemberOf("routing")
    public void add(Key k, Serializable value) { ... }
@MemberOf("routing")
    public Serializable lookup(Key k) { ... }
@MemberOf("monitoring")
    public void monitor() { ... }
}
```

Fig. 1. The CAN Peer annotated for parallelism

Figure 1 illustrates the annotations in the context of a CAN peer active object in which *add*s and *lookup*s can be performed in parallel – they belong to the same self-compatible group `routing`. Since there is no compatibility rule defined between them, methods of `join` and `routing` will not be served in parallel. To fully illustrate our annotations, we added *monitoring* as a third concern independent from the others.

2.3 Dynamic Compatibility

Sometimes the compatibility of requests can be more precisely decided at run-time. For example, two methods writing in the same array can be compatible if they don't access the same cells. For this reason, we first introduce an optional *group-parameter* which indicates the type of a a parameter which will be used to decide compatibility. This parameter must appear in all methods of the group and in case a method has several parameters of this type, the leftmost one is chosen. In Figure 2, we add `parameter="can.Key"` to the *routing* group to indicate that the parameter of type *Key* will be used. Overall, at runtime, compatibility between two requests can be decided as a function depending on three parameters: the group-parameter of the two requests, and the status of the active-object. We describe below how we integrated this idea into our framework and allowed the programmer to define compatibility functions inside his/her objects.

To actually decide the compatibility, we add a condition in the form of a *compatibility function* which takes as input the common parameters of the two compared groups and returns *true* if the methods are compatible. The general syntax for this rule is:

```
@compatible{value={"group1","group1"}, condition="SomeCondition"}
```

The compatibility function can be defined as follows:

– when `SomeCondition` is in the form `someFunc`, the compatibility will be decided by executing `param1.someFunc(param2)` where `param1` is the parameter of one request and `param2` is the parameter of the other.

```
@DefineGroups({
@Group(name="routing",selfCompatible=true,parameter="can.Key",condition="!equals")
    @Group(name="join", selfCompatible=false) })
@DefineRules({@Compatible(value={"routing", "join"},condition="!this.isLocal") })
public class Peer {
    private boolean isLocal(Key k){
        synchronized (lock) { return myZone.containsKey(k); }       }
}
```

Fig. 2. The CAN Peer annotated for parallelism with dynamic compatibility

– when `SomeCondition` is in the form `[REF].someFunc`, the compatibility will depend on the results of `someFunc(param1, param2)` with the group parameters as arguments. `[REF]` can be either **`this`** if the method belongs to the multi-active object itself, or a class name if it is a static method.

Additionally the result of the comparator function can be negated using "!", e.g. `condition="!this.isLocal"`. Since the compatibility method can run concurrently with executing threads, the programmer should ensure mutual exclusion, if necessary. One can define dynamic compatibility even when only one of the two groups has a parameter, in that case the compatibility function should accept one less input parameter. It is even possible to dynamically decide compatibility when none of the two groups has a parameter (e.g. based on the state of the active object); in that case the compatibility function should be a static method or a method of the active object, with no parameter.

As an example, we show how to better parallelise the execution of joins and routing operations in our CAN. During a *join* operation, the peer which is already in the network splits its key-space and transfers some of the key-value pairs to the peer which is joining the network. During this operation, ownership is hard to define. Thus a *lookup* (or *add*) of a key belonging to one of the two peers cannot be answered during the transition period. Operations that target "external" keys, on the other hand, could be safely executed in parallel with a join. Figure 2 shows a modified version of the `Peer` class which supports dynamic compatibility checks. For the sake of clarity, we omit unmodified code. The modifications are as follows. 1) a group parameter, `can.Key` has been added to the `routing` group; 2) a compatibility rule has been defined for groups `routing` and `join` with condition `"!this.isLocal"`; 3) a method **`boolean`** `isLocal(Key k)` has been created, which checks whether a key falls in the zone of a peer At runtime, the method `isLocal(Key k)` will be executed when checking the compatibility of groups `routing` and `join`. This method is selected because it is the only one matching the name of the condition and the group parameter type. We also defined a condition for self-compatibility in the `routing` group: to guarantee that there is no overtaking between requests on the same key, we configure the group to be `selfCompatible` only when the key parameter of the two invocations is not equal, see Figure 2, line 2.

3 A Calculus of Multi-active Objects

This section describes MultiASP, the multi-active object calculus. We present its small step operational semantics and its properties. In MultiASP, there is no explicit notion of place of execution, but the calculus is particularly adapted to distribution because, first,

inter-activity communication behaves like remote method invocation, and second, each object belongs to a single activity. Overall, each active object can be considered as a unit of distribution. The operational semantics is parametrised by a function deciding whether a request should be served concurrently on a new thread or sequentially by the thread that triggered the service.

3.1 Syntax and Runtime Structures

While x, y range over variable names, we let l_i range over field names, and m_i over method names (m_0 is a reserved method name; it is called upon the activation of an object and encodes the service policy). :: denotes the concatenation of lists; we also use it to append an element to a list. $\varnothing$ denotes an empty list or an empty set. The syntax of MultiASP is identical to ASP [3] (except for the clone operator that is of no interest here). Active objects and futures can be created at runtime. New objects can be allocated in a local store (there is one local store for each active object). Thus, we let ι_i range over references to the local store (ι_0 is a reserved location for the active object), α, β, γ range over active object identifiers, and f_i range over future identifiers. Among the terms above, static terms are the ones that contain no references to futures, active objects, or store location. Note that every term is an object and $[]$ is an empty object:

$a ::= x$	variable (y also ranges over variables),
$\mid [l_i = a_i; m_j = \varsigma(x_j, y_j)a'_j]^{i\in 1..n}_{j\in 1..m}$	object definition (x_j binds self; y_j is the parameter),
$\mid a.l_i$	field access,
$\mid a.l_i := a'$	field update,
$\mid a.m_j(a')$	method call,
$\mid Active(a)$	creates an active object from a,
$\mid Serve(M)$	serves a request among M, a set of method labels.$M = \{m_i\}^{i\in 1..p}$
$\mid \iota$	location (only at runtime)
$\mid f$	future reference (only at runtime)
$\mid \alpha$	active object reference (only at runtime)

A *reduced object* is either a future, an activity reference, or an object with all fields reduced to a location. A store maps locations to reduced objects; it stores the *local state* of the active object:

$$o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} \mid f \mid \alpha \qquad\qquad \sigma ::= \{\iota_i \mapsto o_i\}^{i\in 1..k}$$

To ensure absence of sharing, the operation *Copy&Merge*$(\sigma, \iota\,;\, \sigma', \iota')$ performs a *deep copy* of the object at location ι in σ into the location ι' of σ', and ensures that communicated values are "self-contained". This is achieved by copying the entry for ι at location ι' of σ' (if this location already contains an object it will be erased). All locations ι'' referenced (recursively) by the object $\sigma(\iota)$ are also copied in σ' at fresh locations (See [3, 11] for the formal definition).

F ranges over *future value association lists*; such a list stores computed results where ι_i is the location of the value associated with the future f_i: $F ::= \{f_i \mapsto \iota_i\}^{i\in 1..k}$. The list of *pending requests* is denoted by $R ::= [m_i; \iota_i; f_i]^{i\in 1..N}$, where each request consists of: the name of the *target method* m_i, the location of the *argument* passed to the request ι_i, the *future* identifier which will be associated to the result f_i. $(f \mapsto \iota) \in F$

means $(f \mapsto \iota)$ is one of the entries of the list F and similarly $[m; \iota; f] \in R$ means $[m; \iota; f]$ is one of the requests of the queue R.

There are two parallel composition operators: $\|$ expresses *local parallelism*, it separates threads residing in the same activity, and $\|\|$ expresses *distributed parallelism*, it separates different activities. A request being evaluated is a term together with the future to which it corresponds $(a_i \mapsto f_i)$. An activity has several parallel threads each consisting of a list of requests being treated: the leftmost request of each thread is in fact currently being treated, the others are in a waiting state. C is a *current request structure*: it is a parallel composition of threads where each thread is a list of requests. By nature, $\|$ is symmetric, and current requests are identified modulo reordering of threads:

$$C ::= \varnothing \mid [a_i \mapsto f_i]^{i \in 1..n} \| C$$

Finally, an activity is composed of a name α, a store σ, a list of pending requests R, a set of computed futures F, and a current request structure C. A configuration Q is made of activities. Configurations are identified modulo the reordering of activities.

$$Q ::= \varnothing \mid \alpha[F; C; R; \sigma] \,\|\|\, Q$$

An *initial configuration* consists of a single activity treating a request that evaluates a static term a_0: $b_0 = \alpha_0[\varnothing; a_0 \mapsto f_\varnothing; \varnothing; \varnothing]$.

Contexts. Reduction contexts are terms with a hole indicating where the reduction should happen. For each context $\mathcal{R}$, the operation $(\mathcal{R}[c])$ replaces the hole by a given term c. Contrarily to substitution, filling a hole is not capture avoiding: the term filling the hole is substituted as it is.

Sequential reduction contexts indicate where reduction occurs in a current request:

$$\begin{aligned}\mathcal{R} ::= {} & \bullet \mid \mathcal{R}.l_i \mid \mathcal{R}.m_j(b) \mid \iota.m_j(\mathcal{R}) \mid \mathcal{R}.l_i := b \mid \iota.l_i := \mathcal{R} \\ & \mid [l_i = \iota_i, l_k = \mathcal{R}, l_{k'} = b_{k'}; m_j = \varsigma(x_j, y_j) a_j]_{j \in 1..m}^{i \in 1..k-1, k' \in k+1..n} \mid Active(\mathcal{R})\end{aligned}$$

A *parallel reduction context* extracts one thread of the current request structure (remember that current requests are identified modulo thread reordering):

$$\mathcal{R}_c ::= \quad [\mathcal{R} \mapsto f_1] :: [a_j \mapsto f_j]^{j \in 2..n} \quad \| C$$

3.2 Operational Semantics

Our semantics is built in two layers, a local reduction $\rightarrow_{\text{loc}}$ defined in [3] and in [11] that corresponds to a classical object calculus, and a parallel semantics that encodes distribution and communications. Activities communicate by remote method invocations and handle several local threads; each thread can evolve and modify the local store according to the local reduction rules. Thanks to the parallel reduction contexts $\mathcal{R}_c$, multiple threads are handled almost transparently in the semantics. The main novelty in MultiASP is the request service that can either serve the new request in the current thread or in a concurrent one; for this we rely on two functions: *SeqSchedule* and *ParSchedule*. Given a set of method names to be served (the parameter of the *Serve* primitive), the set of futures calculated by the current thread, and the set of futures calculated by the other threads of the activity, these functions decide whether it is possible to serve a request sequentially or in parallel. The last parameter is the request queue that will be

Table 1. Parallel reduction (used or modified values are non-gray)

LOCAL

$$\frac{(a,\sigma) \rightarrow_{\text{loc}} (a',\sigma')}{\alpha\big[F;\mathcal{R}_c[a];R,\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F;\mathcal{R}_c[a'];R;\sigma'\big] \,|||\, Q}$$

ACTIVE

$$\frac{\gamma \text{ fresh activity name} \qquad f_\varnothing \text{ fresh future} \qquad \sigma_\gamma = \mathit{Copy\&Merge}(\sigma,\iota\,;\,\varnothing,\iota_0)}{\begin{array}{c}\alpha\big[F;\mathcal{R}_c[\mathit{Active}(\iota)];R;\sigma\big] \,|||\, Q \longrightarrow \\ \alpha\big[F;\mathcal{R}_c[\gamma];R;\sigma\big] \,|||\, \gamma\big[\varnothing;[\iota_0.m_0([]) \mapsto f_\varnothing];\varnothing;\sigma_\gamma\big] \,|||\, Q\end{array}}$$

REQUEST

$$\frac{\sigma_\alpha(\iota) = \beta \qquad \iota'' \notin \mathit{dom}(\sigma_\beta) \qquad f \text{ fresh future} \qquad \sigma'_\beta = \mathit{Copy\&Merge}(\sigma_\alpha,\iota'\,;\,\sigma_\beta,\iota'')}{\begin{array}{c}\alpha\big[F;\mathcal{R}_c[\iota.m_j(\iota')];R;\sigma_\alpha\big] \,|||\, \beta\big[F';C';R';\sigma_\beta\big] \,|||\, Q \longrightarrow \\ \alpha\big[F;\mathcal{R}_c[f];R;\sigma_\alpha\big] \,|||\, \beta\big[F';C';R'::[m_j;\iota'';f];\sigma'_\beta\big] \,|||\, Q\end{array}}$$

ENDSERVICE

$$\frac{\iota' \notin \mathit{dom}(\sigma) \qquad \sigma' = \mathit{Copy\&Merge}(\sigma,\iota\,;\,\sigma,\iota')}{\alpha\big[F;\iota \mapsto f::[a_i \mapsto f_i]^{i\in 1..n} \| C;R;\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F::f \mapsto \iota';[a_i \mapsto f_i]^{i\in 1..n} \| C;R;\sigma'\big] \,|||\, Q}$$

REPLY

$$\frac{\sigma_\alpha(\iota) = f \qquad \sigma'_\alpha = \mathit{Copy\&Merge}(\sigma_\beta,\iota_f\,;\,\sigma_\alpha,\iota) \qquad (f \mapsto \iota_f) \in F'}{\alpha\big[F;C;R;\sigma_\alpha\big] \,|||\, \beta\big[F';C';R';\sigma_\beta\big] \,|||\, Q \longrightarrow \alpha\big[F;C;R;\sigma'_\alpha\big] \,|||\, \beta\big[F';C';R';\sigma_\beta\big] \,|||\, Q}$$

SERVE

$$\frac{\begin{array}{c}C = [\mathcal{R}[\mathit{Serve}(M)] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \| C' \\ \mathit{SeqSchedule}(M,\{f_i\}^{i\in 0..n},\mathit{Futures}(C'),R) = ([m,f,\iota],R')\end{array}}{\alpha\big[F;C;R;\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F;[\iota_0.m(\iota) \mapsto f]::[\mathcal{R}[[]] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \| C';R';\sigma\big] \,|||\, Q}$$

PARSERVE

$$\frac{\begin{array}{c}C = [\mathcal{R}[\mathit{Serve}(M)] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \| C' \\ \mathit{ParSchedule}(M,\{f_i\}^{i\in 0..n},\mathit{Futures}(C'),R) = ([m,f,\iota],R')\end{array}}{\alpha\big[F;C;R;\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F;[\iota_0.m(\iota) \mapsto f] \| \mathcal{R}[[]] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \| C';R';\sigma\big] \,|||\, Q}$$

split into a request to be served and the remaining of the request queue. If no request can be served neither sequentially nor in parallel, both functions are undefined. We define $Futures(C)$ as the set of futures being computed by the current requests C. Then, parallel reduction $\longrightarrow$ is described in Table 1. We will denote by $\longrightarrow^*$ the reflexive transitive closure of $\longrightarrow$. Table 1 consists of seven rules:

LOCAL triggers a local reduction $\rightarrow_{\text{loc}}$ described in [3] and in [11]. **ACTIVE** creates a new activity: from an object and all its dependencies, this rule creates a new activity at a fresh location γ. The method m_0 is called at creation, the initial request is associated with $f_\varnothing$, a future that is never referenced and never used. **REQUEST** invokes a request on a remote active object: when an activity α performs an invocation on another activity β this creates a fresh future f, and enqueues a new request in β. The parameter is deep copied to the destination's store. **ENDSERVICE** finishes the service of a request: it adds

an entry corresponding to the newly calculated result in the future value association list. The result object is copied to prevent further mutations. **REPLY** sends a future value: if an activity α has a reference to a future f, and another activity β has a value associated to this future, the reference is replaced by the calculated value[1]. **SERVE** serves a new request sequentially: it relies on a call to *SeqSchedule* that returns a request $[m, f, \iota]$ and the remaining of the request queue R'. The request $[m, f, \iota]$ is served *by the current thread*. The *Serve* instruction is replaced by an empty object that will be stored, so that execution of the request can continue with the next instruction. *SeqSchedule* receives, the set of method names M, the set of futures of the current thread, the set of futures of the other threads, and the request queue. **PARSERVE** serves a new request in parallel: it is similar to the preceding rule except that it relies on a call to *ParSchedule*, and that a new thread is created that will handle the new request to be served $[m, f, \iota]$. The particular case where the source and destination are the same require an adaptation of the rules REQUEST and REPLY [11].

We can show that $\longrightarrow$ does not create references to futures or activities that do not exist, and thus the parallel reduction is well-formed. Quite often an active-object will serve all the requests in a FIFO order: m_0 consists of a loop: *while* (*true*) *Serve*(M_A) where M_A is the set of all method names[2] and the other methods never perform a *Serve*. m_0 is compatible with all the other methods and thus all services can be done in parallel, but the service of a request might have to wait until another request finishes. We call this particular case *FIFO request service*.

3.3 Scheduling Requests

Several strategies could be designed for scheduling parallel or sequential services. Future identifiers can be used to identify requests uniquely; also it is easy to associate some meta-information with them (e.g. the name or the parameters of the invoked method). Consequently, we rely on a compatibility relation between future identifiers: *compatible*(f, f') is true if requests corresponding to f and f' are compatible. We suppose this relation is symmetric.

Table 2 shows a suggested definition of functions *SeqSchedule* and *ParSchedule* which *maximises parallelism while ensuring that no two incompatible methods can be run in parallel*. The following of this section explains in what sense this definition is correct and optimal. The principle of the compatibility relation is that two requests served by two different threads should be compatible:

Property 1 (Compatibility). If two requests are served by two different threads then they are compatible: suppose $Q_0 \longrightarrow^* \alpha\big[F; C; R; \sigma\big] \,|||\, Q$ then:

$$C = \big[a_i \mapsto f_i\big]^{i \in 1..n} \| \big[a'_j \mapsto f'_j\big]^{j \in 1..m} \| C' \Rightarrow \forall i \in 1..n.\, \forall j \in 1..m.\, \mathit{compatible}(f_i, f'_j)$$

We consider that parallelism is "maximised" if a new request is served whenever possible, and those services are performed by as many threads as possible. Thus, to maximise parallelism, a request should be served by the thread that performs the *Serve* operation only if there is an incompatible request served in that thread. The "maximal parallelism" property can then be formalised as an invariant:

[2] *while* and *true* can be defined in pure ASP [3].

Table 2. A possible definition of *SeqSchedule* and *ParSchedule*

$$\frac{\begin{array}{c}\forall f \in F'.\, \mathit{compatible}(f_j, f) \qquad \exists f \in F.\, \neg \mathit{compatible}(f_j, f) \\ m_j \in M \qquad \forall k < j.\, m_k \in M \Rightarrow \big(\mathit{compatible}(f_j, f_k) \ \wedge\ \exists f \in F'.\, \neg \mathit{compatible}(f_k, f)\big)\end{array}}{\mathit{SeqSchedule}(M, F, F', [m_i, f_i, \iota_i]^{i \in 1..N}) = ([m_j, f_j, \iota_j], [m_i, f_i, \iota_i]^{i \in 1..j-1} :: [m_i, f_i, \iota_i]^{i \in j+1..N})}$$

$$\frac{\begin{array}{c}\forall f \in F'.\, \mathit{compatible}(f_j, f) \qquad \forall f \in F.\, \mathit{compatible}(f_j, f) \\ m_j \in M \qquad \forall k < j.\, m_k \in M \Rightarrow \big(\mathit{compatible}(f_j, f_k) \ \wedge\ \exists f \in F'.\, \neg \mathit{compatible}(f_k, f)\big)\end{array}}{\mathit{ParSchedule}(M, F, F', [m_i, f_i, \iota_i]^{i \in 1..N}) = ([m_j, f_j, \iota_j], [m_i, f_i, \iota_i]^{i \in 1..j-1} :: [m_i, f_i, \iota_i]^{i \in j+1..N})}$$

Property 2 (Maximum parallelism). Except the leftmost request of a thread, each request is incompatible with one of the other requests served by the same thread (and that precede it). More formally, if $Q_0 \longrightarrow^* \alpha\big[F; C; R; \sigma\big] \,|||\, Q$ then:

$$\big(C = [a_i \mapsto f_i]^{i \in 1..n} \| C' \ \wedge\ k > 1\big) \Rightarrow \exists i' < k.\, \neg \mathit{compatible}(f_k, f_{i'})$$

The properties above justify the two first premises of the rules in Table 2. The last premise decides which request to serve. For this, we filter the request queue by the set M of method labels. Then we serve the first request that is compatible with all requests served by the other threads. These definitions ensure that any two requests served inside two different threads are always compatible, and also that requests are served in the order of the request queue filtered by the set of methods M, except that *a request can always overtake requests with which it is compatible*. We say that parallelism is maximised because we eagerly serve new requests on as many parallel threads as possible.

We conclude this section by explaining why it is reasonable to let a request overtake compatible ones. Indeed, whenever two compatible requests are served in parallel compatibility implies that the operations of these requests can be freely interleaved. Overtaking is just a special case of interleaving, namely when all operations of one request are executed before the operations of the other request. Thus, even if requests were served in the exact incoming order, a request may actually overtake a compatible one. More generally, if all requests are necessarily served at some point, then allowing a request to be overtaken by compatible ones is a safe decision.

4 Evaluation

In this section we show that our proposal provides an effective compromise between programming simplicity and execution efficiency of parallel and distributed applications: multi-active objects achieve the same performance as classical concurrent approaches while simplifying the programming of distributed applications. A detailed analysis of the results and the description of the multi-active API is provided in [11]

Our proposal is implemented on top of ProActive, the reference implementation of ASP. No preprocessing or modified Java compiler are required, all decisions are made at runtime by reifying constructors and method invocation. The flexibility and portability we obtain using these techniques induce a slight overhead, but experience shows it

has no significant impact at the application level. The implementation of multi-active objects comes with an API for customising the request service policy, the possibility to limit the number of threads inside a multi-active object, and an inheritance mechanism for compatibility annotations [11].

NAS Parallel Benchmarks. We first compare multi-active objects with Java threads based on a well-known parallel benchmark suite: the NAS Parallel Benchmarks. To achieve a multi-active implementation, we modify the Java-based version of the benchmark [15] and create a multi-active object version from each kernel. A single multi-active object replaces all worker threads, and futures and wait-by-necessity replace *wait()* and *notify()*. This makes the code easier to understand and to maintain. Code related to parallelism in the multi-active version is much shorter than the original (between 35% and 70%), depending on the benchmark. Also, our annotations make synchronisation much more natural, and on a higher level of abstraction than the original program. The performance of the modified application is very close to the original.

CAN Experimental Results. We implemented the CAN illustrating example to show the efficiency brought by multi-active objects compared to active-objects in a distributed multicore environment. The purpose of these experiments is to evaluate the benefits of multi-active objects, not the performance of our CAN implementation. Therefore, our experiments were designed to provide an interesting workload for the active objects themselves, not necessarily for the CAN network. We created and populated a CAN using `join` and `add` operations, then we measured the benefits of `lookup` request parallelisation in the following situations:

- *All from two*: In this scenario, two corners send `lookup` requests to all other nodes, and then wait until all the results are returned. This experiment gives an insight about the overall throughput of the overlay.
- *Centre from all*: In this test case, all the peers lookup concurrently a key located in a peer at the centre of the CAN. This experiment highlights the scalability of a peer under heavy load.

We repeated each scenario 50 times and measured the overall execution time (the difference between successive runs was found to be negligible). The creation and population of the network was not measured as part of the execution time. We used up to 128 machines located at the Sophia-Antipolis site of the Grid5000[3] platform. All hosts are interconnected through Gigabit Ethernet, and are equipped with quad-core CPUs (Intel Xeon E5520, AMD Opteron 275 and AMD Opteron 2218), running Java 7 Hotspot 64-Bit Server VMs. Each requested value was 24KB in size.

Figure 3 shows the execution times and speedup (when turning active objects into multi-active ones) for several sizes for the two scenarios. Both scenarios achieve significant speedup thanks to the communication and request handling performed in parallel, however, the gain in the first scenario is smaller because the lookups are issued from the two corners in sequence; the sequential sending of the initial lookups limits the number of lookups present at the same time in the network. In the second case, the active object version has a bottleneck because the centre peer can only reply to one request at a time whereas those requests can be highly parallelised with our model. As shown before,

[3] `http://www.grid5000.fr/`

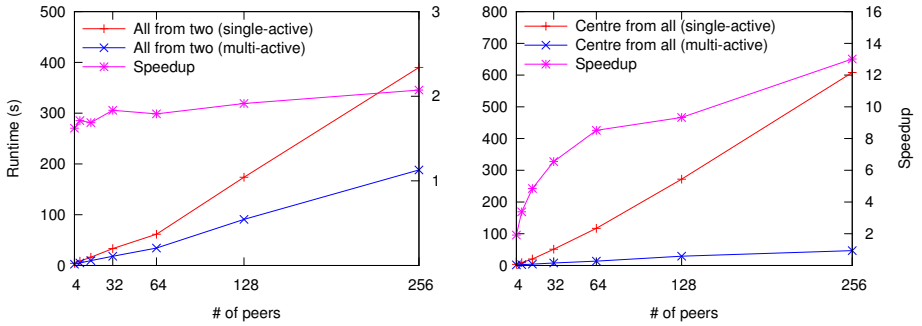


Fig. 3. CAN experimental results

these speedups are simply achieved by adding a few annotations to the class declaration without changing the rest of the code.

5 Comparison with Related Work

Parallel actor monitors [16] (PAM) provide multi-threading capacities to actors based on explicit scheduling functions. However, we believe that the compatibility annotations of multi-active objects provide a higher level of abstraction than PAM, and that this high-level of abstraction is what makes active-objects and actors easy to program.

The main difference between our approach and active-objects with cooperative multi-threading like JCoBox [8] and Creol [7] is twofold. On one hand, Creol-like languages are not really multi-threaded (only one thread is active at a time [7]), thus they do not necessarily address the issue of efficiency on multicore architectures; JCoBox proposes a shared immutable state that can be used efficiently on multicore architectures but as the distributed implementation is still a prototype, it is difficult to study how an application mixing local concurrency and distribution like our CAN example would behave. On the other hand, concerning synchronisation, in cooperative multi-threaded solutions between explicit release points (`await`s) the programs are executed sequentially. Adequately placing those release points is the main challenge in programming in Creol or JCoBox: too many release points leads to a complex interleaving between sequential code portions, whereas not enough of them will probably lead to a deadlock.

Multi-active objects provide an alternative approach by allowing local concurrency in active objects: with annotations, the programmer can reason on high-level compatibility rules, and parallelism can be expressed in a simple manner. Compatible methods are run concurrently, with potential race-conditions but also local multi-threading. Overall, in ASP and MultiASP distribution and concurrency are much more transparent than in JCoBox and Creol; this difference in point of view explains most of the differences between the two models: transparent vs. explicit futures and compatibility annotations vs. explicit thread release. However, the principles of multi-active objects could be applied to an active object language with explicit futures and explicit release points, but in this case thread activation (after an await statement) must take into account compatibility information.

Cunha and Sobral [17] use Java annotations to parallelise sequential objects in an OpenMP fashion. A method can be called asynchronously if it is flagged with the `Future` annotation, but the programmer must follow the flow of futures carefully and declare which methods can access them. There is also an `ActiveObject` annotation that creates a proxy and a scheduler, but its exact semantics is not well-defined in [17]. In our opinion, JAC's and our compatibility rules offer a greater control and a higher abstraction level than OpenMP style fork-join blocks, they are also better adapted to active-objects.

Our annotation system looks like JAC's proposal, and this paper could also be seen as an adaptation of a concurrency model *à la* JAC to the active object model. The inheritance model of JAC annotations is well designed [13] and resembles the way our annotations are inherited from class to class. However, multi-active objects offer a simpler annotation system and a higher synchronisation logic encapsulation. Moreover, the dynamic compatibility rules of multi-active objects are not directly translatable into JAC annotations: JAC provides a precondition mechanism that can be used to express dynamic compatibility but it does not guarantee safe access to shared variables. Compared to JAC, we think multi-active objects are simpler to program, have stronger properties, and are better suited to distribution. In particular, the transparent inclusion of annotations leads in our opinion to a powerful and interesting programming model.

Now that our active objects are equipped with multiple threads, strategies for optimizing the number and utilization of threads will have to be considered (e.g. [11, 18]).

6 Conclusion

In active object languages, programming efficiently on multicore architectures is not possible. Indeed, to have multiple threads on the same machine, all the other languages require to create multiple active objects (or coboxes). Then, the communication between those objects is either costly (it relies on a request invocation and heavy parameter passing), or restricted (JCoBoxes can share a state but it must be immutable). In our case, several threads inside an active objects can co-exist and we provide an annotation system to control their concurrent execution. The annotations can be written from a high-level point of view by declaring compatibility relations between the different concerns an active object manages. The operational semantics defined in Section 3 allowed us to prove that request services can be scheduled such that *parallelism is somehow maximised while preventing two incompatible requests from being served in parallel.*

The originality of our contribution lies in the interplay between the formal and precise study of the MultiASP language, and a middleware implementation efficient enough to compete with classical multi-threading benchmarks. The use of dynamic constraints for compatibility allows a fine-grain control over local concurrency and improves expressiveness. We implemented the proposed model in Java and ran experiments to ensure that our approach is efficient. The experiments showed that the performance of multi-active objects is similar to the manually multi-threaded version while code dedicated to parallelism is much simpler when using multi-active objects. We also illustrated the performance gain brought by multi-active objects compared to a classical active object version. Overall, *multi-active objects outperform simple active objects, and are easier to program than classical multi-threading*. Interleaved execution and

race-conditions can of course appear due to multithreading, but, on one side, annotations provide a good way to control this non-determinism, and on the other side, we defined an operational semantics that will allow us to study the possible executions of a multi-active objects and extend properties that the authors proved in the past for ASP.

References

1. Lavender, R.G., Schmidt, D.C.: Active object: an object behavioral pattern for concurrent programming. In: Pattern Languages of Program Design 2, pp. 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston (1996)
2. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming ABCL/1. In: Conference Proceedings of OOPLSA 1986. ACM, NY (1986)
3. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous sequential processes. Information and Computation 207(4), 459–495 (2009)
4. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge (1986)
5. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming 7(1), 1–72 (1997)
6. Halstead Jr., R.H.: Multilisp: A language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems (TOPLAS) 7(4), 501–538 (1985)
7. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science 365(1-2), 23–66 (2006)
8. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
9. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented programming in ambientTalk. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 230–254. Springer, Heidelberg (2006)
10. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
11. Henrio, L., Huet, F., István, Z.: A language for multi-threaded active objects. Research Report RR-8021, INRIA (July 2012)
12. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 161–172. ACM (2001)
13. Haustein, M., Lohr, K.: Jac: declarative Java concurrency. Concurrency and Computation: Practice and Experience 18(5), 519–546 (2006)
14. Shanneb, A., Potter, J., Noble, J.: Exclusion requirements and potential concurrency for composite objects. Science of Computer Programming 58(3), 344–365 (2005)
15. Frumkin, M., Schultz, M., Jin, H., Yan, J.: Implementation of NAS parallel benchmarks in Java. In: A Poster Session at ACM 2000 Java Grande Conference (2000)
16. Scholliers, C., Tanter, É., De Meuter, W.: Parallel actor monitors. In: 14th Brazilian Symposium on Programming Languages (2010)
17. Cunha, C., Sobral, J.: An annotation-based framework for parallel computing. In: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 113–120. IEEE Computer Society (2007)
18. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)

Scheduling Open-Nested Transactions in Distributed Transactional Memory

Junwhan Kim, Roberto Palmieri, and Binoy Ravindran

ECE Department, Virginia Tech, Blacksburg, VA, 24061
{junwhan,robertop,binoy}@vt.edu

Abstract. Distributed transactional memory (DTM) is a powerful concurrency control model for distributed systems sparing the programmer from the complexity of manual implementation of lock-based distributed synchronization. We consider Herlihy and Sun's dataflow DTM model, where objects are migrated to invoking transactions, and the *open nesting* model of managing inner (distributed) transactions. In this paper we present DATS, a dependency-aware transactional scheduler, that is able to boost the throughput of open-nested transactions reducing the overhead of running expensive compensating actions and abstract locks in the case of outer transaction aborts. The contribution of the paper is twofold: *(A)* DATS allows the *commutable* outer transactions to be validated concurrently and (B) allows the *non-commutable* outer transactions, depending on their inner transactions, to commit be committed before others without dependencies.

1 Introduction

Transactional Memory (TM) is an emerging innovative programming paradigm for transactional systems. The main benefit of TM is synchronization transparency in concurrent applications. In fact, leveraging the proven concept of atomic and isolated transactions, TM spares programmers from the pitfalls of conventional manual lock-based synchronization, significantly simplifying the development of parallel and concurrent applications. Moreover lock-based concurrency control suffers from programmability, scalability, and composability challenges [13] and TM promises to alleviate these difficulties. In TM, the developer simply organizes read and write operations on shared objects as transactions and leaves the responsibility of executing those transactions to the TM, ensuring atomicity, consistency and isolation. Two transactions conflict if they access to the same object and at least one access is a write. The contention manager, the component in TM responsible for resolving conflicts among concurrent transactions, typically aborts one and allows the other to commit, yielding (the illusion of) atomicity. Aborted transactions are typically re-started after rollingback the changes in memory.

The Transaction Scheduler (TS) is the component that supports the contention manager in making a decision on how to resolve conflicts (which transaction to abort). The goal of the TS is to order concurrent transactions as to avoid or minimize conflicts (and thereby aborts).

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 105–120, 2013.

The hazards of manual implementation of lock-based concurrency control increases in distributed settings due to an additional synchronization level among nodes in the system. Distributed STM (DTM) has been motivated as an alternative to distributed lock-based concurrency control. DTM can be classified based on the system architecture: cache-coherent DTM (cc DTM) [14,23], in which a set of nodes communicate by message-passing links over a communication network, and a cluster model (cluster DTM) [6,21], in which a group of linked computers work closely together to form a single computer. cc DTM uses a cache-coherence protocol [8,14] to locate and move objects in the network.

Support for nesting (distributed) transactions is essential for DTM, for the same reasons that they are so for multiprocessor TM – i.e., code composability, performance, and fault-management [20,24,25]. Three types of nesting have been studied for multiprocessor TM: *flat*, *closed*, and *open*. If an inner transaction I is *flat-nested* inside its outer transaction A, A executes as if the code for I is inlined inside A. Thus, if I aborts, it causes A to abort. If I is *closed-nested* inside A [19], the operations of I only become part of A when I commits. Thus, an abort of I does not abort A, but I aborts when A aborts. Finally, if I is *open-nested* inside A, then the operations of I are not considered as part of A. Thus, an abort of I does not abort A, and vice versa.

The differences between the nesting models are shown in Figure 1, in which there are two transactions containing a nested-transaction. With flat nesting, transaction T_2 cannot execute until transaction T_1 commits. T_2 incurs full aborts, and thus has to restart from the beginning. Under closed nesting, only T_2's inner-transaction needs to abort and be restarted while T_1 is still executing. The portion of work T_2 executes before the data-structure access does not need to be retried, and T_2 can thus finish earlier. Under open nesting, T_1's inner-transaction commits independently of its outer, releasing memory isolation over the shared data-structure. T_2's inner-transaction can therefore proceed immediately, thus enabling T_2 to commit earlier than in both closed and flat nesting.

The flat and closed nested models have a clear negative impact on large monolithic transactions in terms of concurrency. In fact, when a large transaction is aborted all its flat/closed-nested transactions are also aborted and rolled-back, even if they do not conflict with any other transaction. Closed nesting potentially offers better performance than flat nesting because the aborts of closed-nested inner transactions do not affect their outer transactions. However the open-nesting approach outperforms both in terms of concurrency allowed. When an open-nested transaction commits, its modifications on objects become immediately visible to other transactions, allowing those transactions to start using those objects without a conflict, increasing concurrency [20]. In contrast, if the inner transactions are closed- or flat-nested, then those object changes are not made visible until the outer transaction commits, potentially causing conflicts with other transactions that may want to use those objects.

To achieve high concurrency in open nesting, inner transactions have to implement *abstract serializability* [27]. If concurrent executions of transactions result in the consistency of shared objects at an "abstract level", then the executions are said to be abstractly serializable. If an inner transaction I commits, I's

modifications are immediately committed in memory and I's read and write sets are discarded. At this time, I's outer transaction A does not have any conflict with I due to memory accessed by I. Thus, programmers consider the internal memory operations of I to be at a "lower level" than A. A does not consider the memory accessed by I when it checks for conflicts, but I must acquire an *abstract lock* and propagates this lock for A. When two operations try to acquire the same abstract lock, the open nesting concurrency control is responsible for managing this conflict (so this is defined "abstract level").

If an outer transaction (with open-nested inner transactions) aborts, all of its (now committed) open-nested inner transactions must rollback and their actions must be undone to ensure transaction serializability. Thus, with the open nesting model, programmers must provide a *compensating* action for each open-nested transaction [2]. In scenarios in which outer transactions increasingly encounter conflicts after that a large number of their open-nested transactions have committed (e.g., long running transactions), the overall performance could collapse and all the benefits of the open-nesting approach vanish due to the processing of compensating actions to undo the modifications provided by the committed open-nested transactions. In closed nesting, since closed-nested transactions are not committed in memory until the outer transaction commits (nested transactions' changes are visible only to the outer), no undo (e.g., compensation) is required. Moreover, in scenarios in which the load of the system is high, the probability of having concurrent conflicting transactions grows and with it the probability to abort outer transactions with a number of open-nested transactions (already committed in memory). Aborting outer transactions with dependencies, instead of alleviating the load of the system, will increase it due to the execution of compensating actions, possibly bringing the system toward dangerous states.

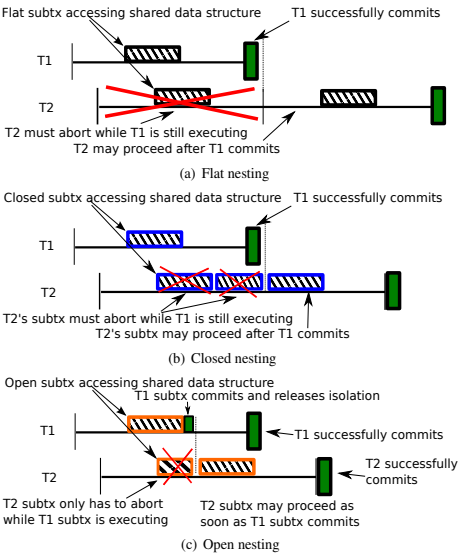


Fig. 1. Two transactions under flat, closed and open nesting (From [24])

We focus on these problems: the overhead of compensating actions and abstract locks in the open-nested model. Our goal is to boost performance of nested transactions in DTM by increasing concurrency and reducing the aforementioned overhead through transactional scheduling. For these reasons, we designed a scheduler, called the *Dependency-Aware Transactional Scheduler* (or DATS). DATS is responsible for helping the concurrency control minimizing the number of outer-transactions aborted. In order to do that, DATS relies on the notions of *commutable transactions* and *transaction dependencies*.

Commutable Transactions. Two transactions are defined as commutable if they conflict and they leave the state of the shared data-set consistent even if validated and committed concurrently. A very intuitive example of commutativity is when two operations, $call1(X)$ and $call2(X)$, both access the same object X but different fields of X (See Section 3.2 for discussion about commutativity).

Transaction Dependencies. An outer transaction materializes dependencies with its inner transactions if *(i)* the inner transactions accesses the outer write-set for performing local computation or *(ii)* the results of outer processing are used to decide whether or not to invoke an inner transaction.

DATS is able to detect commutable transactions and validate/commit them, avoiding useless aborts. In the case of non-commutable transactions, DATS identifies how much each outer transaction depends on its inner transactions and schedules the outer transaction with the highest dependency to commit before other outer transactions with lower or no dependencies. Committing this outer transaction prevents its dependent inner transactions from aborting and reduces the number of compensating actions. Moreover, even though the other outer transactions abort, their independent inner transactions will be preserved, resulting in a reduced number of compensating actions and abstract locks without violating the correctness of the object.

We implemented DATS in a Java DTM framework, called HyFlow [22], and conducted an extensive experimental study involving both micro-benchmarks (e.g., Hash Table, Skip-, Linked-List) and a real application benchmark (TPC-C). Our study reveals that throughput is improved by up to 1.7× in micro-benchmarks and up to 2.2× in TPC-C over open-nested DTM without DATS. To the best of our knowledge, DATS is the first ever scheduler that boosts throughput with open-nested transactions in DTM.

The rest of the paper is organized as follows. We present preliminaries of the DTM model and state our assumptions in Section 2. We describe DATS and analyze its properties in Section 3. Section 4 reports our evaluation. We overview past and related efforts in Section 5, and Section 6 concludes the paper.

2 Preliminaries and System Model

We consider a distributed system which consists of a set of nodes $N = \{n_1, n_2, \cdots\}$ that communicate with each other by message-passing links over a network. Similar to [14], we assume that the nodes are scattered in a metric space.

Transaction Model. A set of shared objects $O = \{o_1, o_2, \ldots\}$ are distributed in the network among nodes. A transaction is defined as a sequence of requests, each of which is a read or write operation request to an a single object in O. An execution of a transaction is a sequence of timed operations that ends by either a commit (success) or an abort (failure). A transaction is in three possible states: *live*, *aborted*, or *committed*. Each transaction has a unique identifier and is invoked by a node in the system. We consider the data flow DTM model [14]. In this model, transactions are immobile and objects move from node to node to invoking transactions. Each node has a *TM proxy* that provides interfaces

allowing the local application to interact with the other proxies located on other nodes. When a transaction T_i at node n_i requests object o_j, the TM proxy of n_i first checks whether o_j is in its local cache. If the object is not present, the proxy invokes a distributed cache-coherence protocol (CC) to fetch o_j in the network.

Atomicity, Consistency, and Isolation. We use the *Transactional Forwarding Algorithm with Open Nesting* (TFA-ON) [24], which extends the TFA algorithm [22] (which originally does not provide any transaction nesting support), to manage flat, closed and open-nested transactions. TFA provides *early validation* of remote objects, guarantees a consistent view of shared objects between distributed transactions, and ensures atomicity for object operations in presence of asynchronous clocks. The early validation of remote objects means that a transaction validated first commits its objects successfully. Validation in distributed systems includes global registration of object ownership. TFA is responsible for caching local copies of remote objects and changing the ownership.

TFA-ON changes the scope of object validations. The behavior of open-nested transactions under TFA-ON is similar to the behavior of regular transactions under TFA. In addition, TFA-ON manages the abstract locks and the execution of commit and compensating actions [24]. To provide conflict detection at the abstract level, an abstract locking mechanism has been integrated into TFA-ON. Abstract locks are acquired only at commit time, once the inner transaction is verified to be conflict free at the low level. The commit protocol requests the abstract lock of an object from the object owner and the lock is released when its outer transaction commits. To abort an outer transaction properly, a programmer provides an abstract compensating action for each of its inner transaction to revert the data-structure to its original semantic state.

TFA-ON is the first ever implementation of a DTM system with support for open-nested transactions [24]. DATS has been integrated in TFA-ON.

3 The DATS Scheduler

3.1 Motivations

Figure 2 shows an example of open-nested transactions with compensating actions and abstract locks. Listings 1.1 and 1.2 in Figure 2 illustrate two outer transactions, T_1 and T_2, and an inner transaction in Listing 1.3. The inner transaction INSERT includes an *insert* operation in a Linked List. T_1 has a *delete* operation with a value. If the operation of T_1 executes successfully, its inner transaction INSERT executes. Conversely, regardless of the success of T_2's *delete* operation, its inner transaction INSERT will execute. *OnCommit* and *OnAbort*, which include a compensating action, are registered when the inner transaction commits. If the outer transaction (i.e., T_1 or T_2) commits, *OnCommit* executes. When the inner transaction commits, its modification becomes immediately visible for other transactions. Thus, if the inner transaction commits, and its outer transaction T_1 or T_2 aborts, a *delete* operation as a compensating action (described in *OnAbort*) executes. Let us assume that T_2 aborts, and *OnAbort* executes. Even though T_2's inner transaction (INSERT) does not depend on its

Listing 1.1. Transaction T_1

```
new Atomic<Boolean>(){
  @Override boolean atomically(Txn t){
    List ll = (List)t.open(tree-2);
    deleted = ll.delete(7,t);
    if(deleted) INSERT(t,10);//inner tx
      return deleted;
  }
}
```

Listing 1.2. Transaction T_2

```
new Atomic<Boolean>(){
  @Override boolean atomically(Txn t){
    List ll = (List)t.open(tree-2);
    deleted = ll.delete(9,t);
    INSERT(t,10);//inner tx
    return deleted;
  }
}
```

Listing 1.3. Inner Transaction INSERT

```
public boolean INSERT(Txn t,int value){
  private boolean inserted = false;
  @Override boolean atomically(t){
    List ll = (List)t.open(tree-1);
    inserted = ll.insert(value,t);
    t.acquireAbstractLock (ll,value);
    return inserted;
  }
  @Override onAbort(t){
    List ll = (List)t.open(tree-1);
    //compensation
    if(inserted)ll.delete(value,t);
    t.releaseAbstractLock(ll,7);
  }
  @Override onCommit(t){
    List ll = (List)t.open(tree-1);
    t.releaseAbsractLock(ll,value);
  }
}
```

Fig. 2. Two open-nested transactions with abstract locks and compensating actions

delete operation, unlike T_1, *OnAbort* will execute. Thus, the conflict of object "tree-2" in T_2 causes the execution of compensating action on object "tree-1" in INSERT. The INSERT operation acquires the abstract lock again when it restarts. Finally, whenever an outer transaction aborts, its inner transaction must execute a compensating action, regardless of the operation's dependencies.

This drawback is particularly evident in distributed settings. In fact, distributed transactions typically have an execution time several orders of magnitude bigger than in a centralized STM, due to communication delays that are incurred in requesting and acquiring objects [16]. If an outer transaction aborts, clearly the impact of the time needed for running compensating actions and for acquiring abstract locks for distributed open-nested transactions is exacerbated due to the communication overhead. Moreover it increases the likelihood of conflicts, drastically reducing concurrency and degrading performance.

Motivated by these observations, we propose the DATS scheduler for open-nested DTM. DATS, for each outer transaction T_a, identifies the number of inner transactions depending from T_a and schedules the outer transactions with the greatest number of dependencies to validate first and (hopefully) commit. This behavior permits the transactions with high compensation overhead to commit; the remaining few outer transactions that are invalidated will be restarted excluding their independent inner transactions to avoid useless compensating actions and acquisition of abstract locks. In the next subsection the meaning of dependent transactions for DATS will be described.

3.2 Abstract and Object Level Dependencies

We consider two types of dependencies among transactions.

Abstract Level Dependency. The first is called *abstract level dependency* (ALD) and it indicates the dependency between an outer transaction and its inner

Algorithm 1. Algorithms for checking AOL and OLD

```
1  Procedure Commit
   Input: txid, objects
   Output: commit, abort
2  foreach objects do
3      if txid is open nesting then
4          ▷ Extract <operations,values,DL>
5          Send
           <operations,values,DL,object.id>
6                o object.owner
7          Wait until receive status from
           object.owner
8          if status=noncommute then
9              noncommutativity.put(object);
10 if noncommutativity=∅ then
11     ▷ All objects commute or no conflicts
       detected
12     Retrieve the dependency queue from
       object.owner;
13     Validate objects; ▷ Change the object
       ownership
14     find highest DL from
       dependency.get(object.id);
15     Send object to the node with the highest
       DL;
16     return commit;
17 foreach object ∈ noncommutativity do
18     ▷ Checking abstract level dependency
       (ADL)
19     nestedTxId = CheckALD(object);
20     ▷ Enqueue dependent nested transactions
21     NestedTxs.put(object.id,nestedTxId);
22 Abort(txid, DependentObjects);
23 return abort;
24
25
26 Procedure Retrieve_Object
   Input: operation, value, DL, oid
27 object = findObject(oid);
28 if object=null then
29     ▷ Object just validated, checking object
       level dependency (OLD)
30     if CheckOLD(operation, value) then
31         commutativity.put(object.id, new
           request(operation, values));
32         return commute;
33     ▷ Dependency queue to track updates.
34     dependency.put(oid, DL);
35     return non − commiute;
36 return no − conflict;
37
38 Procedure Abort
   Input: txid, objects
39 if txid is outer-transaction then
40     foreach objects do
41         nestedIds = NestedTxs.get(object.id);
42         if nestedIds ≠ null then
43             foreach nestedIds do
44                 ▷ Execute onAbort() for
                   nestedId
45                 AbortNestedTx(nestedId);
46 AbortOuterTx(txid);
```

transactions at an abstract level. We define the *dependency level* (DL) as the number of inner transactions that will execute *OnAbort* when the outer transactions abort. For example, T_1 illustrated in Figure 2 depends on its INSERT due to the *deleted* variable. Thus, DATS detects a dependency between T_1 and its INSERT (its inner transaction) because the *delete* operations in T_1 shares the variable *deleted* with the conditional if statement declared for executing INSERT. In this case, the DL=1 for T_1. Conversely, T_2 executes INSERT without checking any pre-condition so its DL=0 because T_2 does not have dependencies with its inner transactions. The purpose of the abstract level dependency is to avoid unnecessary compensating actions and abstract locks. Even though T_2 aborts, *OnAbort* in INSERT will not be executed because its DL=0, and the compensating action will not be processed. Meanwhile, executing *OnAbort* implies running INSERT and acquiring the abstract lock again when T_2 restarts.

Summarizing, aborting outer transactions with smaller DLs leads to a reduced number of compensating actions and abstract lock acquisitions. Such identification can be done automatically at run-time by DATS using byte-code analysis or relying on explicit indication by the programmer. The first scenario is completely transparent from the application point of view but in some cases could add additional overhead. The second approach, although it requires the collaboration

of the developer, is more flexible because it allows the programmer to bias the behavior of the scheduler. In fact, even though the logic of an outer transaction reveals a certain number of dependencies, the programmer may want to force running compensations in case of an abort. This can be done by simply changing the value of DL associated to the outer transaction.

Object Level Dependency. The second is called *object level dependency* (OLD) and it indicates the dependency among two or more concurrent transactions accessing the same shared object. For example, in Figure 2, T_1 depends on T_2 because they share the same object "tree-2". If T_1 and T_2 work concurrently, a conflict between them occurs. However, *delete*(7) of T_1 and *delete*(9) of T_2 commute because they are two operations executing on the same object ("tree-2") but accessing different items (or fields when applicable) of the object (item "7" and item "9"). We recall that, two operations commute if applying them in either order they leave the object in the same state and return the same responses [12]. DATS detects object level dependency at transaction commit phase, splitting the validation phase into two. Say T_a is the transaction that is validating. In the first phase, T_a checks the consistency of the objects requested during the execution. If a concurrent transaction T_b has requested and already committed a new version of some object requested by T_a, then T_a aborts in order to avoid isolation corruption. After the successful completion of the first phase of T_a's validation, DATS detects the object level dependencies among concurrent transactions that are validating with T_a in the second phase. To do that, DATS relies on the notion of commutativity already introduced at the end of Section 1. Suppose T_a and T_b are conflicting transactions but simultaneously validating. If all of T_a's operations commute with all of T_b's operations, they can proceed to commit together avoiding a useless abort. Otherwise one of T_a or T_b must be aborted. This scheduler is in charge of the decision (see next sub-section).

In order to compute commutativity, DATS joins two supports. In the first, the programmer annotates each transaction class with the fields accessed. The second is a field-based timestamping mechanism, used for checking the field-level invalidation. The goal is to reduce the granularity of the timestamp from object to field. With a single object timestamp, it is impossible to detect commutativity because of fields modifications. In fact, writes to different fields of the same object are all reflected with the increment of the same object timestamp. In order to do that efficiently, DATS exploits the annotations provided by the developer on the fields accessed by the transaction to directly point only to the interested fields (instead of iterating on all the object fields, looking for the ones modified). On such fields, it uses field-based timestamping to detect object invalidation.

The purpose of the object level dependency is to enhance concurrency of outer transactions. Even though inner transactions terminate successfully, aborting their outer transactions affects these inner transactions (due to compensation). Thus, DATS checks for the commutativity of conflicting transactions and permits them to be validated, reducing the aborts.

3.3 Scheduler Design

We designed DATS using abstract level dependencies and object level dependencies. In Figure 1 is presented the pseudo-code with the procedures used by DATS for detecting ALD and OLD at validation/commit time. When outer transactions are invoked, the DL with their inner transactions is checked. When the outer transactions request an object from its owner, the requests with their DLs will be sent to the owner and moved into its scheduling queue. The object owner maintains the scheduling queue holding all the ongoing transactions that have requested the object with their DLs. When T_1 (one of the outer transactions) validates an object, we consider two possible scenarios. First, if another transaction T_2 tries to validate the same object, a conflict between T_1 and T_2 is detected on the object. Thus, DATS checks for the object level dependency. If T_1 and T_2 are independent (according to the object level dependency rules), DATS allows T_1 and T_2 to proceed with the validation. Otherwise, the transaction with lower DL will be aborted. In this way, dependent transactions with the minimal cost of abort and compensating actions are aborted and restarted, permitting transactions with a costly abort operation to commit.

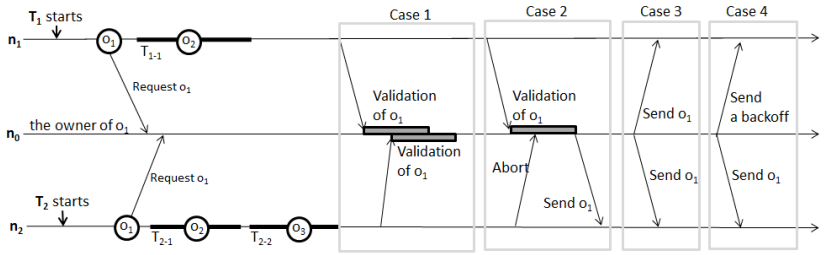


Fig. 3. Four Different Cases for Two Transactions T_1 and T_2 in DATS

Figure 3 illustrates an example of DATS with two transactions T_1 and T_2 invoked on nodes n_1 and n_2, respectively. The transaction T_1 has a single inner-transaction and T_2 has two nested transactions. Let us assume that T_1's DL=1 and T_2's DL=2. The circles indicate written objects. The horizontal line corresponds to the status of each transaction described in the time domain. Figure 3 shows four different cases when T_1 and T_2 terminate. When T_1 and T_2 are invoked, DATS analyzes their DLs, operations, and values. When T_1 requests o_1 from n_0, the meta-data for DLs, operations and values of o_1 will be sent to n_0. These are moved to the scheduling queue of n_0. We consider four different cases regarding the termination of T_1 and T_2.

Case 1. T_1 and T_2 validate concurrently o_1. DATS checks for the object level dependency. If T_1 and T_2 are not dependent at the object level (i.e., the operations of T_1 and T_2 over o_1 commute), T_1 and T_2 commit concurrently.
Case 2. T_1 starts to validate and detects it is dependent with T_2 (that is still executing) at the object level on the object o_1. In this case T_2 will abort due to early validation. When T_1 commits, the updated o_1 is sent to n_2.

Case 3. Another transaction committed o_1 before T_1 and T_2 validate. If T_1 and T_2 are not dependent at the object level, o_1 is sent to n_1 and n_2 simultaneously as soon as the transaction commits.
Case 4. Another transaction committed o_1 before T_1 and T_2 validate. If T_1 and T_2 are dependent at the object level, DATS checks for the abstract level dependency, and o_1 is sent to n_2 because T_2's DL is larger than that of T_1. Aborting T_1, the scheduler is forced to run a single compensation (for T_{1-1}) instead of two compensations (T_{2-1} and T_{2-2}) in case of T_2's abort. Further, considering the case in which the DL of T_1 is 0, the abort of T_1 does not affect T_{1-1}. In fact, its execution will be preserved and only the operations of T_1 will be re-executed.

4 Implementation and Experimental Evaluation

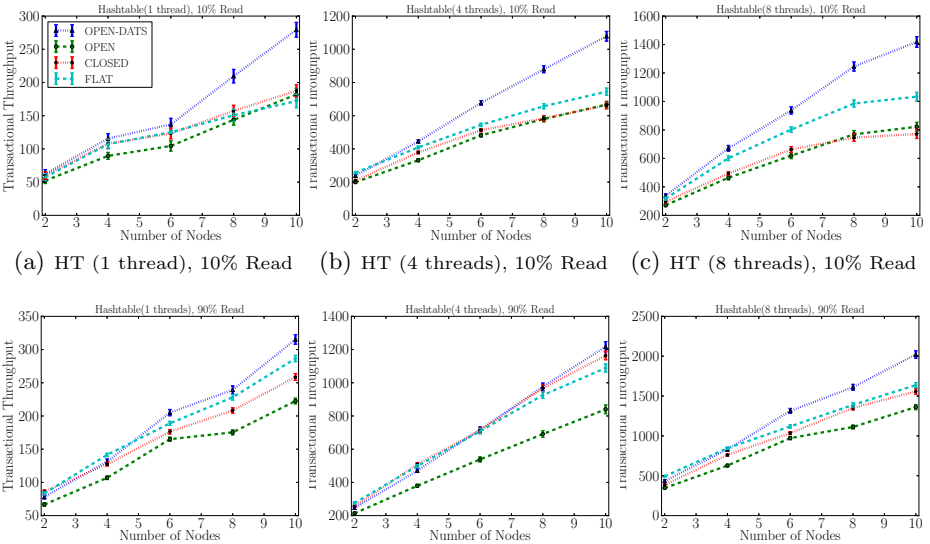
Experimental Setup. We implemented DATS in the HyFlow DTM framework [22,25]. We cannot compare our results with any competitor, as none of the DTMs that we are aware of support open nesting and scheduling. Thus, we compared DATS under TFA-ON (DATS) with only TFA-ON (OPEN) [24], closed nested transaction (CLOSED) [25], and flat nested transaction (FLAT). We contrast with CLOSED and FLAT to show that OPEN does not always perform better than them, while DATS consistently outperforms OPEN.

We assess the performance of DATS using Hash Table, Skip List and Linked List as micro-benchmarks, TPC-C [7] as a real-application benchmark. Our testbed is comprised of 10 nodes, each one is an Intel Xeon 1.9GHz processor with 8 CPU cores. We varied the number of application threads performing operations for each node from 1 to 8, considering a spectrum between 2 and 80 concurrent threads in the system. We measured the *throughput* (number of committed transactions per second). All data-points reported are the result of multiple executions, so plots present for each data-point the mean value and the error-bar. In order to assess the goodness of DATS we also present the percentage of aborted transactions and the scheduler overhead.

Benchmarks. The Skip List and Linked List benchmarks are data structures maintaining sorted and unsorted, lists of items, respectively, whereas Hash Table is an associative array mapping keys to values. We configured the benchmarks with the small number of objects and a large number of inner transactions – eight inner transactions per transaction and ten objects, incurring high contention.

Regarding TPC-C, the write transactions consist of update, insert, and/or delete operations accessing a database of nine tables maintained in memory, where each row has a unique key. Multiple operations commute if they access a row (or object) with the same key and modify different columns. We configured the benchmark with a limited number of warehouses (#3) in order to generate high conflicts. We recall that, in the data flow model, objects are not bound on fixed nodes but move, increasing likelihood of conflicts.

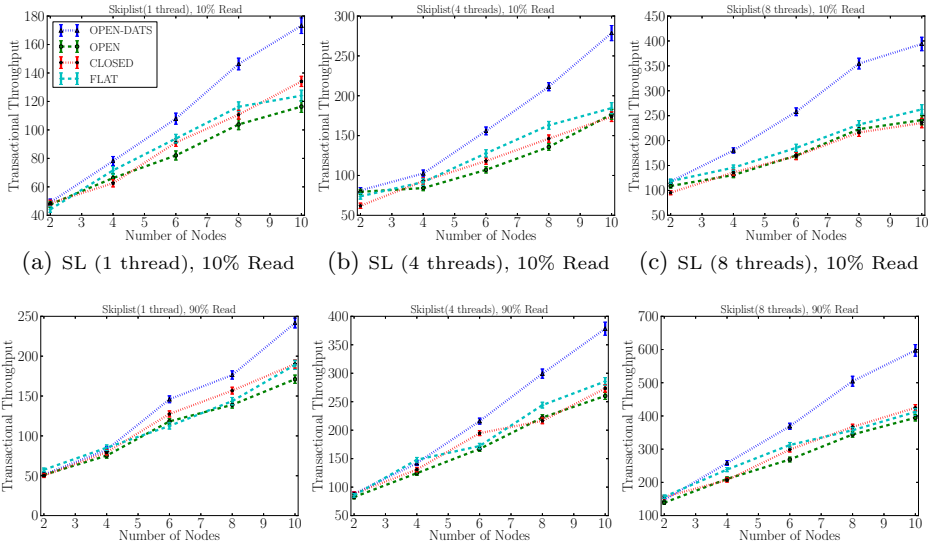
Evaluation. Figures 4, 5 and 6(*a-f*) show the throughput of micro-benchmarks under 10% and 90% of read transactions. The purpose of DATS is to reduce the



(a) HT (1 thread), 10% Read (b) HT (4 threads), 10% Read (c) HT (8 threads), 10% Read

(d) HT (1 thread), 90% Read (e) HT (4 threads), 90% Read (f) HT (8 threads), 90% Read

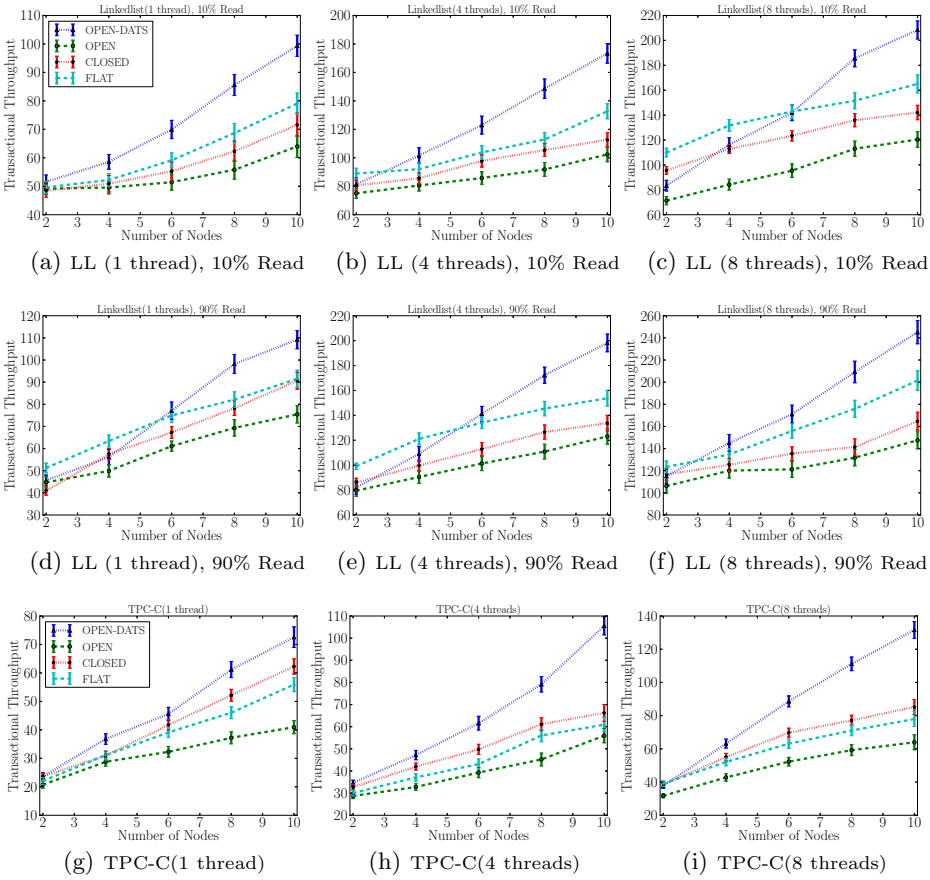
Fig. 4. Performance of DATS Using Hash Table (HT)



(a) SL (1 thread), 10% Read (b) SL (4 threads), 10% Read (c) SL (8 threads), 10% Read

(d) SL (1 thread), 90% Read (e) SL (4 threads), 90% Read (f) SL (8 threads), 90% Read

Fig. 5. Performance of DATS Using Skip List (SL)

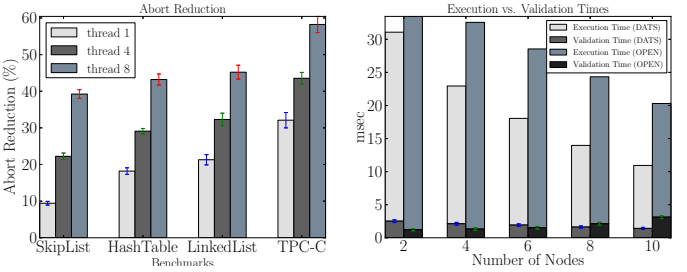


(a) LL (1 thread), 10% Read (b) LL (4 threads), 10% Read (c) LL (8 threads), 10% Read

(d) LL (1 thread), 90% Read (e) LL (4 threads), 90% Read (f) LL (8 threads), 90% Read

(g) TPC-C(1 thread) (h) TPC-C(4 threads) (i) TPC-C(8 threads)

Fig. 6. Performance of DATS Using Linked List (LL) and TPC-C



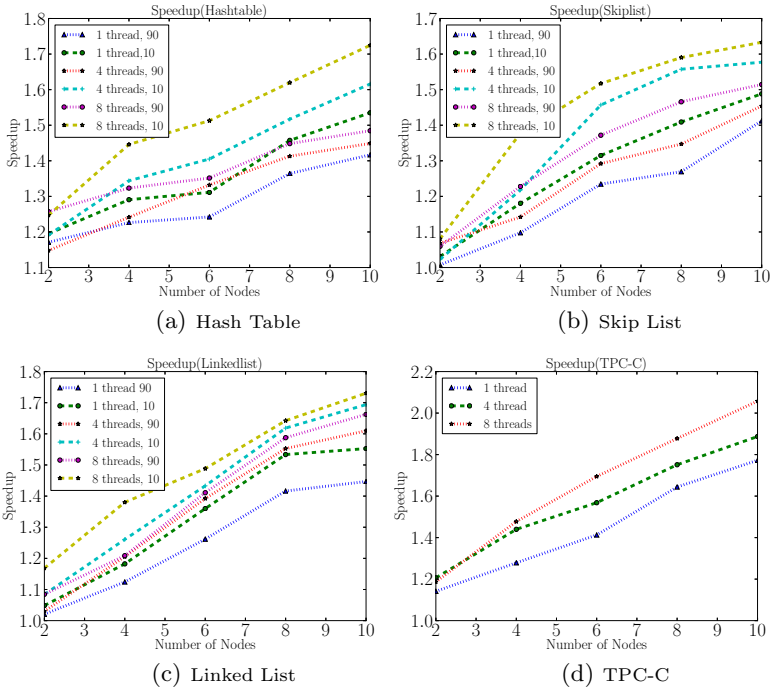
(a) % Aborted transactions (b) Execution vs. Validation Times

Fig. 7. Analysis of Scheduling Overhead and Abort Reduction

overheads of compensating actions and abstract locks. In 10% read transactions, the number of aborts increases due to high contention. Outer transactions frequently abort, and corresponding compensating actions are executed; so DATS outperforms OPEN in throughput because it mitigates the abort of outer transactions and the corresponding compensating actions.

For the experiments with TPC-C in Figure 6(g),6(h),6(i), we used the amount of read and write transactions that its specification recommends. TPC-C benchmark accesses large tables to read and write values. Due to the non-negligible transaction execution time, the number of compensating actions and abstract locks in TPC-C significantly degrades the overall performance. Thus, DATS increases the performance in high contention (a large number of threads and nodes). By these results, it is evident how much unnecessary aborts of inner transactions affects performance and how much performance is improved through minimizing aborts. Even if DATS reduces the number of compensating actions and acquisition of abstract locks, the performance of OPEN is degraded because of the commit overheads of inner transactions [24]; so the throughput of DATS is slightly better than CLOSED and FLAT, but significantly better than OPEN.

Figure 8 shows throughput speed-up relative to OPEN using Hash Table, Skip List, Linked List and TPC-C. Our results show that DATS performs up to 1.7× and 2.2× better than OPEN in micro-benchmarks and TPC-C, respectively.



(a) Hash Table (b) Skip List

(c) Linked List (d) TPC-C

Fig. 8. Speed-up (Throughput Relative to OPEN) in Hash Table, Skip List, Linked List, TPC-C

Figure 7 shows the analysis of scheduling overhead and abort reduction. Checking dependencies occurs when a transaction validates, so we measure the average execution time and the average validation time of committed transactions as illustrated in Figure 7(b). The gap between the two validation times of DATS and OPEN proves the scheduling overhead. Even though the validation time of DATS is up to two times more than OPEN's, a large number of transactions validated simultaneously according to the increment of nodes, results in a shorten transaction response time, reducing the average validation time and aborts. Figure 7(a) the comparison between the percentage of aborted transactions of OPEN and DATS. As long as the number of threads increases, the number of aborts in DATS and OPEN increases too. However, the increasing abort ratio in DATS is less than in OPEN, proving how much DATS reduces the abort rate.

5 Related Work

Nested transactions (using closed nesting) originated in the database community and were thoroughly described in [18]. This work focused on the popular two-phase locking protocol and extended it to support nesting.

Open nesting also originates in the database community [11], and was extensively analyzed in the context of undo-log transactions [26]. In these works, open nesting is used to decompose transactions into multiple levels of abstraction, and maintain serializability on a level-by-level basis.

One of the early works introducing nesting to Transactional Memory has been presented in [20]. They describe the semantics of transactional operations in terms of system states, which are tuples that group together a transaction ID, a memory location, a read/write flag, and the value read or written. They also provide sketches for several possible HTM implementations, which work by extending existing cache coherence protocols. They further focus on open nested transactions in [19], explaining how using multiple levels of abstractions can help in differentiating between fundamental and false conflicts and therefore improve concurrency. The authors of [17] implemented closed and open nesting in LogTM HTM. They implement nesting models by maintaining a stack of log frames, similar to the run-time activation stack, with one frame for each nesting level. In [1] the authors combined closed and open nesting by introducing the concept of transaction ownership. They propose the separation of TM systems into transactional modules (or Xmodules), which own data. Thus, a sub-transaction commits data owned by its own Xmodule directly to memory using an open-nested model. However, for data owned by foreign Xmodules, it employs the closed nesting model and does not directly write to the memory. The past closed nesting models [20,17,1] have been studied for multiprocessor STM. N-TFA [25] and TFA-ON [24] are the first ever DTM implementation with support for closed and open-nesting, respectively, but do not consider transactional scheduling.

Transactional scheduling has been explored in a number of multiprocessor STM efforts [10,3,28,9,4]. In [10], the authors describe an approach that schedules transactions based on their predicted read/write access sets. In [3], they discuss the Steal-On-Abort transaction scheduler, which queues an aborted transaction

behind the non-aborted transaction, and thereby prevents the two transactions from conflicting again. The Adaptive Transaction Scheduler (ATS) is present in [28], that adaptively controls the number of concurrent transactions based on the contention intensity: when the intensity is below a threshold, the transaction begins normally; otherwise, the transaction stalls and does not begin until dispatched by the scheduler. The CAR-STM scheduling approach is presented in [9], which uses per-core transaction queues and serializes conflicting transactions by aborting one and en-queuing it on another queue, preventing future conflicts. CAR-STM pre-assigns transactions with high collision probability (application-described) to the same core, thereby minimizing conflicts. In [5] they propose the Proactive Transactional Scheduler (PTS). Their scheme detects "hot spots" of contention that can degrade performance, and proactively schedules affected transactions around the hot spots. Attiya and Milani present the BIMODAL scheduler [4], which targets read-dominated and bimodal (i.e., those with only early-write and read-only) workloads. Kim and Ravindran extend the BIMODAL scheduler for DTM in [15]. Their scheduler, called Bi-interval, groups concurrent requests into read and write intervals, and exploits the tradeoff between object moving times (incurred in dataflow DTM) and concurrency of reading transactions, yielding high throughput. None of the past transactional schedulers for STM and DTM consider open-nested transactions.

6 Conclusions

When transactions with committed open-nested transactions conflict later and are re-issued, compensating actions for the open-nested transactions can reduce throughput. DATS avoids this by reducing unnecessary compensating actions, and minimizing inner transactions' remote abstract lock acquisitions through object dependency analysis. DATS shows the important of scheduling open-nested transactions in order to reduce the number of compensating actions and abstract locks in case of abort. Our implementation and experimental evaluation shows that DATS enhances transactional throughput for open-nested transactions over no DATS by as much as 1.7× and 2.2× with micro-benchmarks and real-application benchmark, respectively.

Acknowledgments. This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385.

References

1. Agrawal, K., Lee, I.-T.A., Sukha, J.: Safe open-nested transactions through ownership. In: SPAA (2008)
2. Agrawal, K., Leiserson, C.E., Sukha, J.: Memory models for open-nested transactions. In: MSPC (2006)
3. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C., Watson, I.: Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 4–18. Springer, Heidelberg (2009)

4. Attiya, H., Milani, A.: Transactional scheduling for read-dominated workloads. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) OPODIS 2009. LNCS, vol. 5923, pp. 3–17. Springer, Heidelberg (2009)
5. Blake, G., Dreslinski, R.G., Mudge, T.: Proactive transaction scheduling for contention management. In: Microarchitecture, pp. 156–167 (December 2009)
6. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D2STM: Dependable distributed software transactional memory. In: PRDC (November 2009)
7. TPC Council. Tpc-c benchmark, revision 5.11 (February 2010)
8. Demmer, M.J., Herlihy, M.P.: The arrow distributed directory protocol. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 119–133. Springer, Heidelberg (1998)
9. Dolev, S., Hendler, D., Suissa, A.: CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In: PODC (2008)
10. Dragojević, A., Guerraoui, R., et al.: Preventing versus curing: avoiding conflicts in transactional memories. In: PODC 2009, pp. 7–16 (2009)
11. Garcia-Molina, H.: Using semantic knowledge for transaction processing in a distributed database. ACM Trans. Database Syst. 8(2), 186–213 (1983)
12. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: PPoPP 2008, pp. 207–216. ACM (2008)
13. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. In: OOPSLA, pp. 253–262 (2006)
14. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing 20(3), 195–208 (2007)
15. Kim, J., Ravindran, B.: On transactional scheduling in distributed transactional memory systems. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 347–361. Springer, Heidelberg (2010)
16. Kim, J., Ravindran, B.: Scheduling closed-nested transactions in distributed transactional memory. IPDPS, 1–10 (2012)
17. Moravan, Bobba, Moore, Yen, Hill, Liblit, Swift, Wood: Supporting nested transactional memory in logTM. SIGPLAN Not. 41(11) (2006)
18. Moss, E.B.: Nested transactions: An approach to reliable distributed computing. Technical report, Cambridge, MA, USA (1981)
19. Moss, J.E.B.: Open-nested transactions: Semantics and support. In: Workshop of Memory Performance Issues (2006)
20. Moss, J.E.B., Hosking, A.L.: Nested transactional memory: model and architecture sketches. Sci. Comput. Program. 63, 186–201 (2006)
21. Palmieri, R., Quaglia, F., Romano, P.: Osare: Opportunistic speculation in actively replicated transactional systems. In: SRDS (2011)
22. Saad, M., Binoy, R.: Supporting STM in distributed systems: Mechanisms and a Java framework. In: ACM SIGPLAN Workshop on Transactional Computing 2011 (2011)
23. Saad, M.M., Ravindran, B.: HyFlow: a high performance distributed software transactional memory framework. In: HPDC 2011 (2011)
24. Turcu, R.: On open nesting in distributed transactional memory. In: SYSTOR (2012)
25. Turcu, A., Ravindran, B.: On closed nesting in distributed transactional memory. In: Seventh ACM SIGPLAN workshop on Transactional Computing (2012)
26. Weikum, G.: Principles and realization strategies of multilevel transaction management. ACM Trans. Database Syst. 16(1), 132–180 (1991)
27. Yang, Menon, Ali-Reza, Antony, Hudson, Moss, Saha, Shpeisman: Open nesting in software transactional memory. In: PPoPP. ACM, New York (2007)
28. Yoo, R.M., Lee, H.-H.S.: Adaptive transaction scheduling for transactional memory systems. In: SPAA, pp. 169–178 (2008)

Peer-Based Programming Model for Coordination Patterns

Eva Kühn, Stefan Craß, Gerson Joskowicz,
Alexander Marek, and Thomas Scheller

Vienna University of Technology, Institute of Computer Languages
Argentinierstr. 8, 1040 Vienna, Austria
{eva,sc,josko,amarek,ts}@complang.tuwien.ac.at

Abstract. Modern distributed software systems must integrate in near-time parallel processes and heterogeneous information sources provided by active, autonomous software systems. Such lively information sources are e.g. sensory data, weather data, traffic data, or booking data, operated by independent distributed sites. The complex integration requires the coordination of these data flows to guarantee consistent global semantics. Design, implementation, analysis and control of distributed concurrent systems are notoriously complex tasks. Petri Nets are widely used to model concurrent activities. However, a higher-level programming abstraction is needed. We propose a new programming model for modeling concurrent coordination patterns, which is based on the idea of "peer workers" that represent re-usable coordination and application components. These components encapsulate behavior, structure distributed data and control flow, and allow integration of pre-existing service functions. A domain-specific language is presented. The usability of the peer-based programming model is evaluated with the Split/Join pattern.

1 Introduction

A common problem in software projects is the tightrope walk between design and implementation. While the designer's focus is to keep the system clean and verifiable under all possible operating conditions, the developer's main intent is getting things done while dealing with exceptional conditions in later refinements. The *Peer Model* that we introduce in this paper tries to bridge this gap, at least within a specified problem domain, by providing a domain-specific modeling language to design, analyze, and implement system integration patterns in distributed environments using predefined and application-specific components.

The need for methodologies that connect verifiable designs and implementations occurs especially in systems where a multitude of autonomous systems are cooperating to fulfill a specific task. One example that illustrates these problems is in the realm of distributed firewall configuration: firewalls belong to different organizations and the rollout of a new configuration must guarantee that no inconsistencies take place. Errors occur during the rollout process when sites are down, the network is unavailable, local configuration constraints are violated,

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 121–135, 2013.

access rights are not granted etc. A modeling tool is required to specify the configuration process. It must allow modeling when a valid state is reached, and at which time and under which conditions new configurations may become active in the entire network or in parts of it. This will be the case if e.g., a certain number or a majority of firewall sites reported back that they have implemented the updates. Network configuration is a dynamic and quite complex task with many different error situations. Failure impact analysis shall be possible for all different configuration possibilities. It is therefore desirable that this process can be modeled, verified, and executed. A service-oriented security concept for the coordination of such firewalls has been proposed in [6], whereas in this paper, we investigate a new model for specification and implementation of complex coordination and integration patterns.

A further example that motivated the design of the Peer Model is from the rail traffic management domain. Here, measurement systems – like wheel sensors or RFID sensors – along the tracks generate information about approaching trains. This data is transported over mesh networks. A use case is for example a traffic control center that queries distributed sensors, collects answers, and aggregates them to a gain a sufficiently consistent view about the technical status of a train. As sensors are low cost components, wireless communication is error prone, and field conditions are harsh, many failure situations must be coped with.

Both examples have a basic pattern in common that consists of the following steps: 1) splitting of a task into individual subtasks to be executed by different nodes (firewalls resp. sensors), and 2) joining of received answers according to complex rules and under the assumption of failures, in order to derive a global decision. We term this recurring situation the *Split/Join pattern* and will use it as an example throughout this paper.

Many modeling methodologies for the purpose of designing concurrent software systems have been proposed, such as Petri Nets [1], Reo [3] and UML MARTE [2]. These methodologies are general and not domain-specific, as they are used to solve almost any software problem. On the one hand, that is what makes them powerful as the designer *can model everything*, but on the other hand, the designer also *does have to model everything* and cannot assume any functionality to be present.

As the Peer Model is targeted at modeling coordination patterns in distributed environments, it can make several assumptions on the target system. For example, it assumes a tuple space-based communication middleware to be present and can therefore resort to transactional transmissions of data between communicating components and to coordination mechanisms for accessing distributed data structures. Additionally, it follows a component-based approach and assumes several predefined, reusable components to be present. Designers can also structure their scenario-specific coordination logic into new reusable components, rearranging the resulting models more clearly and providing better scalability when additional logic is added. New components can also be created by combining existing components and using them as sub-components, thus leading to a nested component structure. It is important to note that only coordination logic

is modeled, not business logic. One weakness in Petri Net models of concurrent systems is that the resulting models mix business logic and coordination logic, thus reducing their readability and maintainability. For that reason our model strictly forbids defining arbitrary business logic within the model. Rather, it integrates and connects external services, thereby separating clearly concurrency, parallelism and distribution mechanisms of the collaborating components from the business logic.

The contribution of this paper is a higher-level programming and design model termed "Peer Model", or "PM" for short, aiming to ease the design and implementation of complex integration patterns in distributed environments. We introduce its Domain Specific Language (DSL), meta-model and graphical notation, but not its implementation itself and prove our claims by using its DSL to model a well-known pattern and comparing the resulting model to equivalent ones using Reo [3] and Colored Petri Net [10] notation.

In Section 2 we introduce the Peer Model along with its graphical notation, Section 3 introduces the pattern used for evaluation purposes, Section 4 gives an overview of the Peer Model's meta-model and DSL. In Section 5 we compare our resulting model of the pattern with those of other modeling languages and Section 6 finally points out our conclusion and future work.

2 Peer Model

The Peer Model is a novel programming model targeted at developers of highly concurrent applications. Its design is inspired by asynchronous message queue or tuple space communication, staged event-driven architecture, and data-driven workflow. In the Peer Model, *space containers* realize stages; a construct termed *peer* is introduced that represents a structured, re-usable, addressable component. It is modeled by means of two stages for input and output. Between them the internal logic of the peer takes place. Inter-peer collaboration occurs between output stage and a foreign peer's input stage. Stages thus mark end-points of distribution. *Wirings* model the flow between stages within and between peers. Data belonging to the same flow is marked with a unique *flow identifier*.

The conception of the Peer Model assumes a tuple-space [8] to bootstrap the model. Space-based middleware lets autonomous components coordinate themselves in a highly decoupled way. We use a space as described in [7], which provides shared *containers* that support configurable coordination mechanisms [11], and a flexible and extensible API for accessing them, although other spaces with similar functionality may also be used to bootstrap the Peer Model.

Data and requests are modeled as *entries* maintained in containers. An entry has an *application*- and a *coordination*-specific data part, termed *app-data* resp. *co-data*. The latter holds meta-information for system-internal mechanisms like entry selection and transactions, and is identified by labeled properties. The entry type is explicitly modeled by a *type property*, which is also queryable co-data. Note that the entry type differs from the type of the app-data transported by this entry. The entry type serves for coordinating the flow in the system.

Fig. 1. (a) Peer. (b) Space Peer.

A container is referenced by a URI in the network. It provides (1) *put* operations that *write* new entries into a container, and (2) *get* operations that *read* or *take* entries. The coordination principle (e.g. first-in-first-out order, key, template matching, SQL-like query etc.) by which entries are administrated and queried in the container is controlled by the entries' meta information. Read and take operations select entries according to the chosen coordination principle. They wait within the given timeout for the query to complete. In addition, take also removes the entries from the container. Put and get operations are carried out in transactions; both support bulk data processing. The write operation puts entries into a container in a single step. Read and take retrieve a certain amount of entries which is controlled by a counter specification as explained below.

The container is a basic building block for the specification of the Peer Model. Special capabilities of the container implementation like persistency and authorization [5,6] influence the quality of the Peer Model with respect to reliability, consistency and security. However, these issues are out of the scope of this paper. For the following, we only assume two very basic coordination laws: (i) selection of entries by means of their type property, and (ii) specification of the amount of entries required to fulfill a query, otherwise the operation will block. The amount of required entries is expressed as a relation that represents the exact number ("="), a minimum (">", ">=") or a maximum ("<", "<=") of required entries. The semantics is to always take as much entries as possible.

2.1 Peer

A *peer* P is the main resource of the Peer Model. The graphical notation for the basic form of a peer is shown in Figure 1.a. It controls the execution of services. A peer possesses a name (URI), an input space termed *peer-in-container* (*PIC*), and an output space termed *peer-out-container* (*POC*). Via the PIC it receives request entries, takes them out of the PIC, processes them, and finally puts replies into the POC. The PIC restricts the entries it accepts to pre-defined types. The behavior of a peer is modeled by means of nested *sub-peers*, *wirings* and *services* (see Section 2.2). A sub-peer is a peer that is created in the scope of a peer. A wiring allows incorporating zero or more services into a peer. It waits until a defined selection of entries is available, invokes services, and moves or copies entries between the peer's PIC, its POC, and/or PICs and POCs of direct sub-peers. Therefore, wirings are the only active part of the system.

Major peer derivatives are: A *space peer (SPA)* as shown in Figure 1.b is a specialized peer that models the functionality of a space container. Its PIC and POC are melted into one place where all incoming entries are stored using the

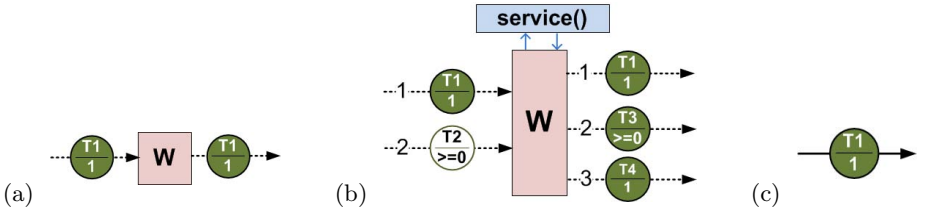


Fig. 2. Wirings

write operation, and from which they can be read or taken. A SPA is used to store data that are shared between concurrent threads and/or processes, i.e. it serves as medium for communication and synchronization. In the Split/Join pattern example (see Section 3, Figure 5.b) it will be used to collect the workers' answers to be merged by a join peer. A *coordination peer (COP)* is a predefined system peer encapsulating re-usable coordination logic; examples are lookup, routing and filtering peers. An *application peer (APP)* includes application-specific logic provided by developers in form of service methods. Together, all peers of a site form a "Peer Space", whose runtime environment is bootstrapped via a *runtime peer (RTP)*, the name of which refers to the URI of the local site. It enables the dynamic creation and deletion of peers and their composition through wirings and services at the local site.

2.2 Wiring

Wirings are the active part of the system: They are the transport system within the Peer Model, controlling the movement of entries between PIC and POC containers of peers with the help of get and put space operations. A wiring has a name and consists of three sections: a *guard*, which defines the conditions under which the wiring activates, optionally followed by *service calls*, and an *action* that defines how to dispose the resulting entries. A guard resp. action section consists of 0, 1, or more *link operations*. As an extra condition, a guard must contain at least one consuming get operation (i.e. take).

The graphical notation of the control part of a wiring named "W" is represented in Figure 2.a, which illustrates a wiring with one input and one output link that are represented by dotted arrows. Figure 2.b shows a wiring with multiple input and output links that also calls a service. The notation for entries consists of a type query in the upper part and a counter query in the lower part of the circle. Filled circles on an input link denote the take operation, whereas unfilled ones – like the entry with query "$type = T2 \mid count >= 0$" on the second input link in Figure 2.c – model the read operation. This query selects all available entries of type $T2$. Details on the query mechanism are explained in [7]. The operator "|" behaves like the pipe operator in the UNIX shell in that it streams the results of the query at its left side to the query at its right side. Entry specifications on output links involve the write operation, i.e. these entries are written to the specified destination container (neither source nor destination

containers are shown in Figure 2). Therefore, in Figure 2.a, exactly one entry of type T1 is removed from some container and written into another one. This simple "move" behavior is the default behavior of a wiring and Figure 2.c depicts the graphical short cut for it, which omits the wiring control symbol and uses a filled line. By using bulk operations on containers, get and put operations can transport more than one entry across one link in a single step.

For simplicity we assume that all sections of W are committed in one atomic step at the end of the wiring, but may optionally already commit after the guard or the service phase. Early commits are beneficial for unblocking other wirings that wait for the same resource. Considering a long running service, the wiring could remove read locks on get links before the service completes, so that other wirings requiring the same entry are not blocked for the complete execution time of the service. A wiring implicitly is associated with a transaction, dubbed *wiring transaction*. Each wiring transaction possesses a configured timeout, after which the transaction automatically aborts. In addition, the RTP configuration specifies how often and in which interval a failed wiring transaction retries to process entries of the same flow.

The functioning of a wiring is: 1) Fulfill all input links sequentially in the specified order, leading to an *entry collection (EC)* which can be understood as an internal space container only visible to this wiring[1]. All input links must succeed. If one cannot be fulfilled, the entire guard blocks. 2) Call all services in the given order and pass them the current entry collection from which the services may read and take entries and into which they may write other entries which represent the results of the service. These operations resemble the get and put operations of a space with the difference that blocking mode is forbidden on EC. After service execution has completed, EC has been filled with all results of the services. 3) Finally execute the output links. The task of the output links is to distribute the entries of EC to PIC and/or POC containers of the own peer or to PIC containers of sub-peers. In contrast to the input links, not all output links must succeed. The semantics is to execute one after the other in the given order: if it is satisfiable then perform it, otherwise skip it and proceed with the next one. E.g. in Figure 2.b the wiring collects one entry of type T1 and all available entries of type T2, which it passes to the service that in turn may change the wiring's internal EC. Let us assume that the service consumes all entries of type T2, and adds two of type T3. So at this point in time, EC is a multi-set $EC = \{E^{T1}, E^{T3}, E^{T3}\}$ of three entries E^{type} (note that the app-data of the two entries E^{T3} might differ). Output link 1 gets the entry of type T1 from EC and writes it to some container (not shown in the picture). Output link 2 delivers both entries of type T3. Output link 3 is tried, but its query cannot be fulfilled and thus it is skipped. This link only works if an entry of type T4 is emitted by the service. After the action section, the effects of the wiring are automatically committed, and remaining entries are removed from EC.

[1] EC access need not be modeled explicitly by Peer Model users, which provides an abstraction of this mechanism. However, it is modeled explicitly in the formal model.

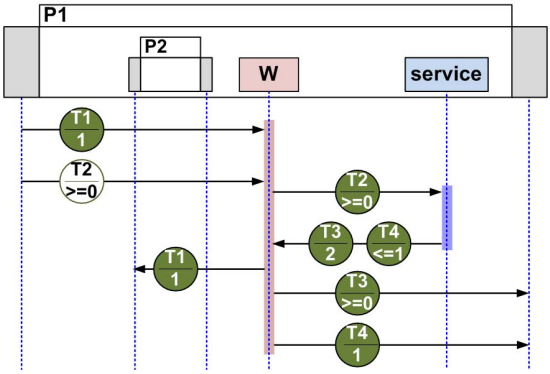


Fig. 3. Example: Wiring modeling with UML-like sequence diagram

The functionality of wirings is modeled via UML-like sequence diagrams. The objects between which messages are exchanged are: PICs and POCs of peers, the wiring box itself, and service methods. These objects are quoted by means of their respective graphical notation plus a life line. Messages between objects represent link operations. Figure 3 shows a sequence diagram for the example wiring in Figure 2.b. It assumes a sub-peer P2, and lets W get entries E^{T1} and E^{T2} from its own peer's PIC, i.e. from P1/PIC, call the service, put E^{T1} to P2/PIC, and finally all entries E^{T3} and E^{T4} to P1/POC.

Wirings are the necessary mechanism to integrate pre-existing service functionality. The entry collection models a flexible interface for the invocation of arbitrary services. By using the features of the underlying space-based middleware, wirings dynamically connect peers and services in a data-driven way that provides high decoupling. Conceptually, all wirings whose guards are fulfilled run concurrently. The model allows for an arbitrary number of instances of the same wiring in parallel. This means that within a peer, the same service could run many times in concurrent threads. The coordination law of the containers can also be used to determine the concurrency. By default, arbitrarily many instances of W may be active at the same time, but it could also be beneficial to enforce a strict sequential execution. However, this requires slightly more complex coordination mechanisms than simple type queries. To control the concurrency level, our Java implementation of the Peer Model – termed "Peer Space" – provides system configurations for maximum thread pool sizes.

2.3 Flow Identifier

The entirety of all wirings that constitute an integration pattern is called a *flow*. In terms of enterprise systems it refers to a workflow. A flow involves a number of wiring executions that together solve a global task. A flow is identified by a unique *flow identifier (FID)* that is generated at its creation time by the Peer Space. A flow is started by emitting a first entry into the Peer Space. The initial status of a flow is "active". A flow can end under different circumstances:

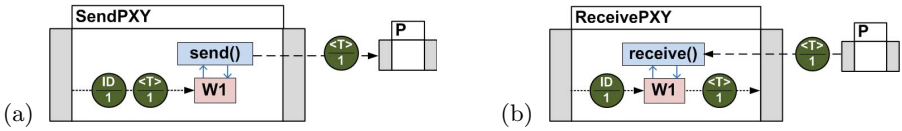


Fig. 4. (a) Send Proxy Peer. (b) Receive Proxy Peer.

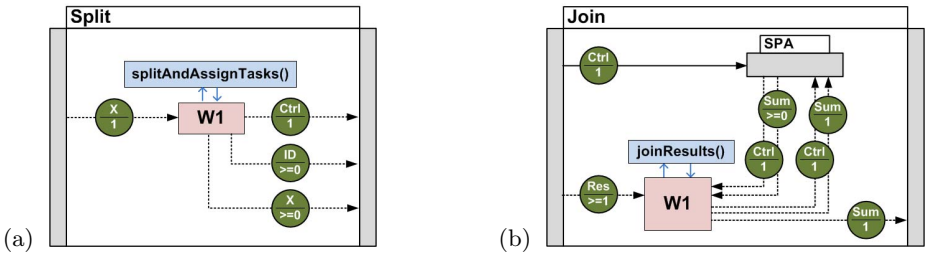


Fig. 5. (a) Split Peer. (b) Join Peer.

through an explicit success or failure result, or after the flow has reached a maximum time-to-live (TTL). Eventually all entries belonging to this flow will be recognized by wirings and automatically removed. If the developer creates an entry, s/he indicates to which flow this entry belongs. The entry automatically carries the TTL with it. A wiring will not treat an entry when its TTL has expired, but will wrap it into an error entry and put this entry into its own POC. For error entries, automatic wirings exist that let the error "roll back" up in the system until it ends at the client who originated the flow. Each wiring must only fetch entries in its guard that belong to the same flow.

3 Split/Join Integration Pattern

As an example use case, the Split/Join integration pattern [15] is chosen. The pattern's objective is to split a task of type X into several sub-tasks, distribute the execution of the sub-tasks to an arbitrary number of workers, wait for all executions, and aggregate and return the results. Developers must provide the logic how to split the original task, which worker to assign and how to join the results. All other parts are pure coordination logic. The challenge is to clearly separate application from coordination logic.

The Split/Join problem is decomposed into several patterns: Two of them are predefined by the system, termed *send* and *receive proxy* patterns. They are pure coordination patterns and modeled as two coordination peers (COPs) termed SendPXY (see Figure 4.a) and ReceivePXY (see Figure 4.b). SendPXY assumes a built-in service called send() that takes as input the ID of a peer, and an entry E^T of a configured type T. Let P be a local or remote peer to which the ID resolves. SendPXY writes E^T into P/PIC. ReceivePXY is the analogous

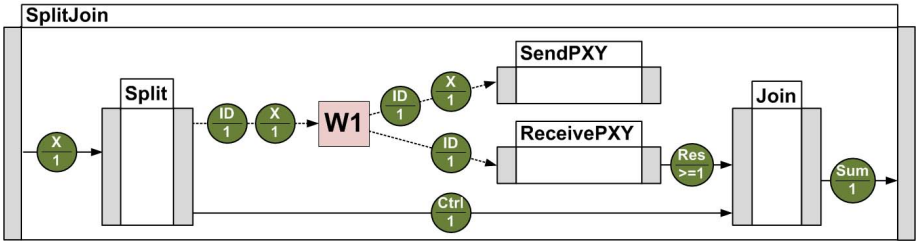


Fig. 6. Split/Join Peer

counterpart: It uses the built-in system service receive() to take one entry from a remote peer's P/POC, where P's ID is the input entry of the pattern.

Two further patterns are Split and Join in Figure 5, both modeled as application peers (APPs) that require application specific logic. The Split peer has a wiring W1 with a guard that expects one task entry of type X as input, and passes it to the splitAndAssignTasks() method which is a user-programmed service that produces an arbitrary amount of sub-tasks and adds them to the internal entry collection of this wiring instance together with a multi-set of peer worker IDs that shall execute sub-tasks. If the same ID is generated twice the same worker must process two sub-tasks; however, which ones is irrelevant. Sub-task entries are also added to the entry collection. Finally, in the action section, the wiring puts a control entry, all IDs, and all sub-tasks into Split/POC. The control entry holds in its application data part specific control information needed for the join, e.g. how many results are needed, which quality is required etc. The Join peer's first wiring is a default wiring that moves one control entry of type Ctrl into a local space peer termed SPA, which serves as local memory for the joinResults() service. Its wiring W1 waits until at least one intermediary result of a worker peer has arrived. It then takes all already available result entries of type Res from worker peers as input, takes the control entry from SPA and also an entry of type Sum (if here) and passes all these entries to the joinResults() service method provided by the developer. The method checks control and result entries, and merges them into the sum entry (or generates one after the first invocation of W1 for this flow). If the service decides that enough results are here, it writes Sum to Join/POC. Otherwise, it writes Sum and Ctrl back into its local SPA, commits and continues its next iteration.

In addition, the user must provide the implementation of worker peers which are not detailed in the example. A worker peer takes one task of type X as input, then performs its specific application logic, and finally generates one output of type Res that it puts into its POC.

All these peer components are composed towards the Split/Join pattern depicted in Figure 6. It starts with applying the Split pattern on input entry X, which outputs many peer IDs, one control entry and many sub-tasks of type X. Wiring W1 is an inter-peer wiring and is executed for pairs of ID and X entries. Its guard requires one ID and one X entry and uses the ID entry to initialize

the SendPXY peer as well as the ReceivePXY peer. In addition, the sub-task entry is written into SendPXY/PIC. SendPXY writes the entry of type X to the corresponding worker peer. In parallel, ReceivePXY waits for the result of this worker peer, takes the results from the worker peer's POC and returns it in ReceivePXY/POC. From there a default move wiring transports it to the Join peer, which also gets the control entry from Split/POC into its Join/PIC as input. The Join peer now waits until enough results are received, merges them and returns the final output which is the aggregated sum of all obtained results. This sum entry is transported to the SplitJoin/POC and from there it is delivered to the originator of the flow. Many flows may run concurrently. Their entries are automatically correlated by the unique identifier (FID) of the flow.

The design of the pattern is resistant against adding more or different, local or remote worker peers. The pattern flexibly supports new requirements and challenges like changed logic for how to join the results: e.g., use only the best results, or stop after a certain amount of results, or stop if a certain condition is met. Only the corresponding joinResults() service must be rewritten. If the FID is invalidated, all still running processes of the flow will eventually stop. All other parts and components of the Split/Join pattern remain unaffected.

4 Meta Model and Domain Specific Language

The intention of the Peer Model is to provide a sound mechanism for modeling integration patterns. We have implemented the system with Promela [9] to achieve a runnable specification and lay the basis for future work on verifying certain properties by means of the SPIN model checker. In addition, a Java version has been implemented to get first experiences with an object-oriented Peer Model API. A presentation of this API is beyond the scope of this paper. We will instead outline the Domain Specific Language for the Peer Model (PM-DSL) that was implemented with Promela. First a meta-meta-model is built, consisting of data structures that model peer types. Then the meta-model that describes the functionality of all peers in a specific Peer Space is derived from the meta-meta-model. The space container operations are implemented by means of (blocking) channel operations. The PM-DSL provides the following operations:

`defPeerType(`*PeerTypeName*`,` *PeerTypeID*`);`
`addSubPeer(`*PeerTypeID*`,` *PeerTypeName*`,` *PeerName*`);`
`addWiring(`*PeerTypeID*`,` *WiringName*`,` *WiringID*`);`
`addGuardLink(`*WiringID*`,` *FromPeerName*`,` *FromContainerName*`,`
ReadOrTake`,` *TypeQuery*`,` *Count*`,` *CountDetails*`);`
`addService(`*WiringID*`,` *ServiceName*`,` *NArgs*`,` *Arg*1`,` *Arg*2`, ...);`
`addActionLink(`*WiringID*`,` *ToPeerName*`,` *ToContainerName*`,`
TypeQuery`,` *Count*`,` *CountDetails*`);`
`createPeer(`*PeerTypeName*`,` *PeerName*`,` *PeerID*`);`
`startPeer(`*PeerID*`);`
`containerWrite(`*PeerID*`,` *ContainerName*`,` *EntryType*`,` *AppData*`,`
CoData`);`

First the meta-meta-model must by initialized. For this, PM-DSL provides the following operations: defPeerType() specifies a new peer type of given type name and in the second argument returns the id by which this type is referenced in the following; addSubPeer() adds a peer given by a peer type to another peer type (given by id) and names it; addWiring() adds a named wiring to a peer type and returns its id; addGuardLink() adds an input link to a wiring by saying from which peer's PIC or POC with which query entries shall be read or taken; addService() adds a service to a wiring – there exist some built-in services with defined names resp. the developer may program own services and refer to them by their name – and here one can configure static arguments (e.g. the number of sub-tasks to be produced by the Split peer) of the service; addActionLink() adds one output link to a wiring and says to which peer's PIC or POC which entries shall be put. "THIS_PEER" refers to the own peer.

As an example, the SplitJoin peer pattern of Figure 6 is modeled with the PM-DSL. We show only parts of the specification of the meta-model for the SplitJoin peer; and we assume that the other peer types already exist.

```
/* SplitJoin peer type: */
defPeerType(SPLIT_JOIN_PEER_TYPE, ptID);
/* Split sub peer: */
addSubPeer(ptID, SPLIT_PEER_TYPE, SPLIT_PEER);
/* SendPXY sub peer: */
addSubPeer(ptID, SENDPXY_PEER_TYPE, SENDPXY_PEER);
/* ReceivePXY sub peer: */
addSubPeer(ptID, RECEIVEPXY_PEER_TYPE, RECEIVEPXY_PEER);
/* Join sub peer: */
addSubPeer(ptID, JOIN_PEER_TYPE, JOIN_PEER);
/* Wiring W1: */
addWiring(ptID, W1, wID1);
addGuardLink(wID1, SPLIT_PEER, POC, TAKE, X_TYPE, 1, EQUAL);
addGuardLink(wID1, SPLIT_PEER, POC, TAKE, ID_TYPE, 1, EQUAL);
addActionLink(wID1, SENDPXY_PEER, PIC, ID_TYPE, 1, EQUAL);
addActionLink(wID1, SENDPXY_PEER, PIC, X_TYPE, 1, EQUAL);
addActionLink(wID1, RECEIVEPXY_PEER, PIC, ID_TYPE, 1, EQUAL);
/* Wiring W2 (default wiring): */
addWiring(ptID, W2, wID2);
addGuardLink(wID2, THIS_PEER, PIC, X_TYPE, 1, EQUAL);
addActionLink(wID2, SPLIT_PEER, PIC, X_TYPE, 1, EQUAL);
```

For creation of the meta-model the PM-DSL offers the operation createPeer() which creates a new peer with given name and initializes its meta-model from the specified peer type, including all resolved wirings. It recursively also creates all sub-peers. Their names must be resolved in the right scope, i.e. the name of a sub-peer is only visible to its super and sibling peers. Only through lookup mechanisms these names become visible to other peers. Description of lookup peers is straight forward, as existing peer-to-peer algorithms can be applied. The Peer Model can now be started with Promela's run operation calling startPeer()

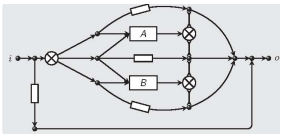


Fig. 7. Reo with 2 services: Synchronous Merge Connector, Fig. 1 from [4]

which starts the wirings of all peers as concurrently running processes. For each wiring multiple instances are assumed to exist concurrently (cf. "bang" operator in [12]). Creation of flows takes place through injection of entries into the system with containerWrite() whereby the FID is contained in the co-data part of the entry. Coming back to our use case, the final steps to start the pattern are:

```
createPeer(SPLIT_JOIN_PEER_TYPE, SPLIT_JOIN_PEER, peerID);
run startPeer(peerID);
containerWrite(peerID, PIC, X_TYPE, /* task */, /* FID etc. */);
```

5 Related Work

An approach related to Split/Join has been undertaken with Reo [4] termed Synchronous Merge Connector (see Figure 7). Flows are modeled by means of different connector types. The example starts with an exclusive router that non-deterministically selects one output which can be either service A or service B. From there three walkthroughs become possible: service A, service B, or both. The exclusive router is also used to control the joining of the execution. Reo supports composition of software components using a channel-based coordination model. In contrast to the Peer Model, Reo is oblivious to any coordination and concurrency inside the component instances; also the used communication mechanisms are of no interest [3], whereas the Peer Model builds upon tuple space-based middleware and allows for nesting of components.

Colored Petri Nets (CPN) [10,1] are a graphical language for modeling concurrent and often also distributed systems. However, they can be applied in almost any thinkable domain[2]. Typically it is used for testing and performance analysis [10]. As the modeling is quite low-level and does not make any assumptions on the domain, it is very powerful; but on the other hand it does not provide any higher-level abstractions. We used CPN-Tools [14] to model the Split/Join use case of Section 3 in two variants: Figure 8.a without requiring the underlying Petri Net to allow for complex inscriptions on input and output arcs, and Figure 8.b making use of advanced features of the ML language used by CPN. For Figure 8.a the pattern was modeled with two services A and B, and every way through the network is controlled by an explicit transition. The other variant uses ML lambda expression to specify more complex arc conditions (Figure 8.b).

[2] Examples of Industrial Use of CP-nets: `http://cs.au.dk/cpnets/industrial-use/`

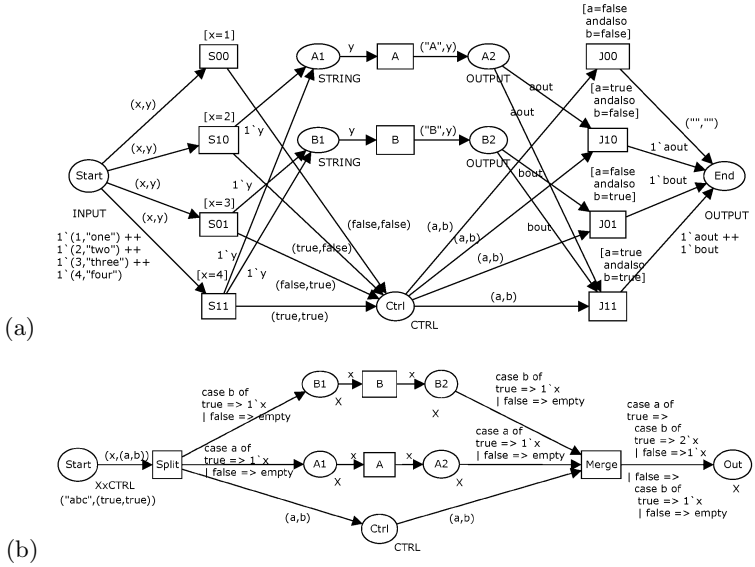


Fig. 8. CPN for 2 services with (a) weak, and (b) complex arc logic assumptions

Next, we increased the design up to 4 services. Figure 9.a shows the basic variant, and Figure 9.b the one employing ML's expressiveness. Both figures demonstrate how complex these designs become. The latter needs less places, but the condition on the output link of the Merge transition becomes more complicated if more services are assumed. Moreover, the control logic is spread among the network. If services are added, the type definitions for the control token must be adapted, as well as for the Start and CTRL place. For each service two places and one transition, plus four arcs must be added explicitly, because the network is static. Two of these arcs must have a special condition. In summary, the comparison shows that Petri Nets do not scale because they are static. Every new component must be modeled explicitly.

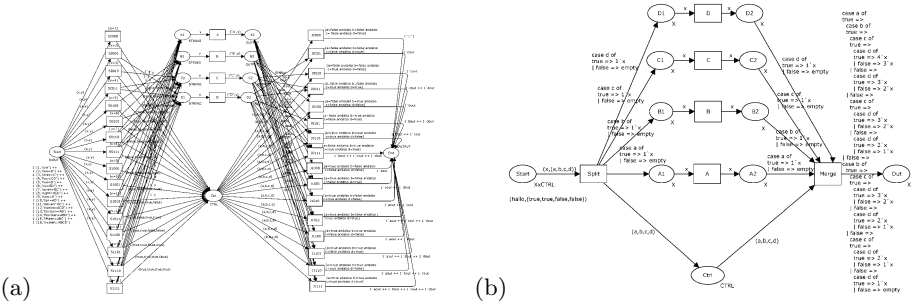


Fig. 9. CPN for 4 services with (a) weak, and (b) complex arc logic assumptions

The main advantages of the Peer Model approach in contrast to more general and static tools like CPNs and REO are:

Design Scalability. The Peer Model allows modeling of large patterns and scales naturally to any number of services, without exponential complexity. In the example this is accomplished by a send and a receive proxy peer, which (de-)multiplex sub-tasks to/from arbitrary worker peers addressed via their IDs. Dynamic addition of workers and increasing the number of sub-tasks is possible.

Dynamics. During runtime, new peers can be added on-the-fly. Also, peers can be changed in that, e.g., wirings are added dynamically.

Composability. The Peer Model design allows to break down a problem into smaller, re-usable patterns that can be composed to larger ones. CPNs also support hierarchically nested modules, but the composition requires global names for the "fusion" places.

Transparent Remoting. The physical distribution of processes to other sites does not cause a difference in design compared to local processes.

Architecture Agility. The model is robust against changes due to new requirements. We have termed this "architecture agility" in [13] and shown that the space-based architectural style helps to avoid costly architecture limiters or even breakers. The Peer Model extends this idea by not only considering passive data, but also modeling the active part, i.e. processes that access the data.

6 Conclusion

The Peer Model is a design tool for the specific domain of programming parallel and distributed applications. In contrast to other widespread modeling tools like Colored Petri Nets it is less general as it takes specific assumptions on its domain. More precisely, it requires a coordination middleware whose features have an impact on the quality of the Peer Model. At least, access to shared data by means of blocking operations, local transactions, and type-based queries with cardinality specification are required. The process space is structured into so-called peers which are framework components. They separate coordination from application logic to achieve software architecture agility. Application logic is integrated in form of service methods. Peers are reusable components that implement coordination patterns. Wirings control the entry flow between and within peers. Flow-identifiers automatically correlate entries of the same flow, thus allowing many instances of different flows to be processed concurrently.

We have presented the basic concepts of the Peer Model, its graphical notation, and domain specific language which is based on a formal model. Its conception is intended as foundation for real implementations. The usability of the Peer Model was evaluated with respect to design scalability, composability, remoting, architecture agility, and support of dynamics. Our future work will concern implementation of a designer tool, mappings to the Colored Petri Net formalism, and extending the mechanisms of how to model concurrency.

Acknowledgements. The work is partially funded by the Austrian Federal Ministry for Transport, Innovation and Technology, FFG-BRIDGE project no. 834162 "LOPONODE" (Coordination Middleware for Wireless Networks of Low Power Nodes).

References

1. Final Draft International Standard ISO/IEC 15909. High-level petri nets - concepts, definitions and graphical notation. Technical report, V. 4.7.1 (October 2000)
2. André, C., Mallet, F., de Simone, R.: Modeling time(s). In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 559–573. Springer, Heidelberg (2007)
3. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
4. Clarke, D., Proença, J.: Partial connector colouring. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 59–73. Springer, Heidelberg (2012)
5. Craß, S., Dönz, T., Joskowicz, G., Kühn, E.: A coordination-driven authorization framework for space containers. In: 7th Int'l Conf. on Availability, Reliability and Security (ARES), pp. 133–142. IEEE (2012)
6. Craß, S., Dönz, T., Joskowicz, G., Kühn, E., Marek, A.: Securing a space-based service architecture with coordination-driven access control. Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA) 4(1), 76–97 (2013)
7. Craß, S., Kühn, E., Salzer, G.: Algebraic foundation of a data model for an extensible space-based collaboration protocol. In: Int. Database Engineering and Applications Symposium (IDEAS), pp. 301–306. ACM (2009)
8. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. 7(1), 80–112 (1985)
9. Holzmann, G., Najm, E., Serhrouchni, A.: Spin model checking: an introduction. International Journal on Software Tools for Technology Transfer (STTT) 2(4), 321–327 (2000)
10. Jensen, K., Kristensen, L., Wells, L.: Coloured petri nets and CPN tools for modelling and validation of concurrent systems. Int. Journal on Software Tools for Technology Transfer (STTT) 9, 213–254 (2007)
11. Kühn, E., Mordinyi, R., Keszthelyi, L., Schreiber, C.: Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In: 8th Int'l Conf. on Autonomous Agents and Multiagent Systems (AAMAS), pp. 625–632. IFAAMAS (2009)
12. Milner, R.: The polyadic pi-calculus: a tutorial. Technical report, Logic and Algebra of Specification (1991)
13. Mordinyi, R., Kühn, E., Schatten, A.: Towards an architectural framework for agile software development. In: 17th IEEE Int. Conf. and Workshops on the Engineering of Computer-Based Systems (ECBS), pp. 276–280. IEEE (2010)
14. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: CPN tools for editing, simulating, and analysing coloured petri nets. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 450–462. Springer, Heidelberg (2003)
15. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and parallel databases 14(1), 5–51 (2003)

Decidability Results for Dynamic Installation of Compensation Handlers

Ivan Lanese and Gianluigi Zavattaro

Focus Team, University of Bologna & INRIA, Italy

Abstract. Dynamic compensation installation allows for easier specification of fault handling in complex interactive systems since it enables to update the compensation policies according to run-time information. In this paper we show that in a simple π-like calculus with static compensations the termination of a process is decidable, but it is undecidable in one with dynamic compensations. We then consider three commonly used patterns for dynamic compensations, showing that process termination is decidable for parallel and replacing compensations while it remains undecidable for nested compensations.

1 Introduction

Nowadays, applications are composed of different interacting entities, living in environments such as the Internet or the cloud. Programming applications in this setting is challenging, due to their own complexity, and on the unpredictability of the environment. For instance, a communication partner may disappear during an interaction, or a message may be lost due to an unreliable network. Nevertheless, the users expect their applications to always provide reliable services. To build reliable services in an unreliable environment coping with unexpected events is certainly one of the main challenges.

In the setting of service-oriented computing, *long running transactions* have been put forward to solve this problem. A long running transaction is a computation that either *succeeds*, or it *aborts*. In the second case, a *compensation* is executed to undo unwanted side effects of the aborted computation. Many languages provide nowadays support for long running transactions [25, 26], and different proposals exist in the literature [2, 3, 8–12, 16, 22]. Originally, the compensation of a long running transaction was statically fixed [26]. Recent proposals show however that being able to dynamically update the compensation as far as the computation progresses allows the programmer to write more easily the compensation code for complex interactions [16].

From a language design point of view, the question of whether dynamic compensations are just syntactic sugar, and thus need not be implemented in the core language, or not, is relevant. Strangely, while many papers in the literature put forward proposals of transaction constructs, very little has been done on comparing them. A main work in this direction is [20]. In [20] it is shown that the ability to add new compensation items to be executed in parallel with the

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 136–150, 2013.

static compensation does not increase the expressive power, while more general patterns do. The study is carried out relying on proofs of encodability and/or non-encodability between the different formalisms. However, there is no clear agreement in the community on which conditions such encodings should satisfy, and the results in [20] strongly depend both on the chosen conditions and on the availability of suitable mechanisms in the compared languages.

We want here to tackle the same problem, but with a completely different approach. In fact, we compare π-like core calculi featuring the basic mechanisms for static and dynamic compensations according to the (un)decidability of *process termination*, that is of the absence of an infinite computation starting from a given process. Clearly, calculi where such a property is undecidable cannot be encoded in calculi where the same property is decidable, and this non-encodability result is valid for all the encodings preserving the property.

We show that process termination is decidable in a π-calculus with static compensations, while it is not in one with dynamic compensations. To better understand where this difference stems from, we limit the expressive power of the dynamic compensation mechanism in different directions. We show that if compensations can only be replaced, then decidability is recovered. If instead compensations can be nested using linear patterns, we are still in an undecidable setting. To further constrain linear patterns aiming at decidability we force the patterns to only add new compensation items in parallel, obtaining again a decidability result.

2 Primitives for Compensations

2.1 Syntax

We base our studies on a π-calculus extended with transactions and primitives for compensation installation. We then consider different fragments, corresponding to various compensation installation patterns. Our calculus is similar to the calculus in [20]. A main difference is that we do not consider restriction. This is forced since, if we add restriction, then process termination (and similar properties) become undecidable even in CCS (without transactions).

The syntax of our calculus relies on a countable set of names N, ranged over by lower case letters. We use $\boldsymbol{x}$ to denote a tuple $x_1, \cdots, x_n$ of names, and $\{\boldsymbol{x}\}$ denotes the set of elements in the tuple. We use $\{\boldsymbol{v}/\boldsymbol{x}\}$ for denoting the substitution of names in $\boldsymbol{v}$ for names in $\boldsymbol{x}$, and we use a similar notation for substitutions of processes for process variables (introduced later).

We start by presenting the syntax of the π-calculus, reported in Fig. 1. Prefixes can be either outputs $\overline{a}\langle\boldsymbol{v}\rangle$ of a tuple of names $\boldsymbol{v}$ on channel a, or corresponding inputs $a(\boldsymbol{x})$, receiving a tuple of names $\boldsymbol{v}$ on channel a and applying substitution $\{\boldsymbol{v}/\boldsymbol{x}\}$ to the continuation. The π-calculus syntax includes the inactive process $\mathbf{0}$, guarded choice $\sum_{i\in I}\pi_i.P_i$, guarded replication $!\pi.P$ and parallel composition $P \mid Q$. We write $\overline{a}$ for $\overline{a}\langle\boldsymbol{v}\rangle$ when $\boldsymbol{v}$ is empty, and a for $a(\boldsymbol{x})$ when $\boldsymbol{x}$ is empty. When I is a singleton, $\sum_{i\in I}\pi_i.P_i$ is shortened into $\pi_i.P_i$. We may also drop trailing $\mathbf{0}$s.

$$
\begin{array}{rcll}
\pi & ::= & \pi\text{-calculus prefixes} & \\
 & \overline{a}\langle \boldsymbol{v} \rangle & \text{(Output prefix)} & \mid \; a(\boldsymbol{x}) \quad \text{(Input prefix)} \\
P, Q & ::= & \pi\text{-calculus processes} & \\
 & \mathbf{0} & \text{(Inaction)} & \mid \; \sum_{i \in I} \pi_i.P_i \quad \text{(Guarded choice)} \\
 & \mid \; !\pi.P & \text{(Guarded replication)} & \mid \; P \mid Q \quad \text{(Parallel composition)}
\end{array}
$$

Fig. 1. π-calculus processes

$$
\begin{array}{rcl}
P, Q ::= & & \textit{Static compensation processes} \\
 & \ldots & (\pi\text{-calculus processes}) \\
 \mid & t[P, Q] & \text{(Transaction scope)} \\
 \mid & \langle P \rangle & \text{(Protected block)}
\end{array}
$$

Fig. 2. Static compensation processes

$$
\begin{array}{rcl}
P, Q ::= & & \textit{Dynamic compensation processes} \\
 & \ldots & \text{(Static compensation processes)} \\
 \mid & X & \text{(Process variable)} \\
 \mid & \mathsf{inst}\lfloor \lambda X.Q \rfloor.P & \text{(Compensation update)}
\end{array}
$$

Fig. 3. Dynamic compensation processes

We now extend the π-calculus with transactions and *static compensations*. The syntax of the extended calculus is in Fig. 2. Static compensations can be programmed by adding just two constructs to π-calculus: *transaction scope* and *protected block*. A transaction scope $t[P, Q]$ behaves as process P until an error is notified to it by an output $\overline{t}$ on the name t of the transaction scope. When such a notification is received the transaction atomically *aborts*: the body P of the transaction scope is killed and compensation Q is executed. Q is executed inside a protected block. In this way it will not be influenced by successive external errors. Error notifications may be generated both from the body P of the transaction scope and from external processes. Error notifications are simply output messages (without parameters). Protected block $\langle P \rangle$ behaves as process P, but it is not killed in case of failure of a transaction scope enclosing it.

The calculus with *dynamic compensations* extends the one with static compensations. The main difference is that with dynamic compensations the body P of transaction scope $t[P, Q]$ can update the compensation Q. *Compensation update* is performed by an additional operator $\mathsf{inst}\lfloor \lambda X.Q' \rfloor.P'$, where function $\lambda X.Q'$ is the compensation update (X can occur inside Q'). Applying such a compensation update to compensation Q produces a new compensation $Q'\{^{Q}/_{X}\}$. Note that Q may not occur at all in the resulting compensation, and it may also occur more than once. For instance, $\lambda X.\mathbf{0}$ deletes the current compensation. The syntax of processes with dynamic compensations extends the one of processes

with static compensations with the compensation update operator and process variables (see Fig. 3). We use X to range over process variables.

We define for processes with dynamic compensations the usual notions of free and bound names. Names in $\boldsymbol{x}$ are bound in $a(\boldsymbol{x}).P$. Other names are free. Also, variable X is bound in $\lambda X.Q$. Bound names and variables inside processes can be α-converted as usual. We consider only processes with no free variables.

Processes with static compensations are processes with dynamic compensations where the compensation update operator is never used. We will show that dynamic compensations are very expressive, making relevant properties undecidable. Thus we consider different subcalculi, constraining the allowed patterns for compensation installation. As a first observation, note that in a compensation update of the form $\lambda X.Q$ there are no constraints on how many times X may occur in Q. Having more than one occurrence of X, allowing to replicate the previous compensation, is rarely used in practice. Thus a meaningful restriction is considering just linear compensations, where X occurs exactly once in Q. We call them *nested compensations*, since the old compensation becomes nested inside the new one, which acts as a context. Another relevant case is when X does not occur at all in Q. We call compensations of this form *replacing compensations*, since the new compensation completely replaces the old one, which is discarded. Finally, a relevant subcase of nested compensations are *parallel compensations*, where Q has the form $Q' \mid X$ and X does not occur in Q'. In this case new and old compensation items are in parallel in the final term.

2.2 Operational Semantics

In this section we define the operational semantics of processes with dynamic compensations. We need however an auxiliary definition. When a transaction scope $t[P, Q]$ is killed, part of its body P may be preserved, in particular the protected blocks inside it.

The definition of function $\mathsf{extr}(P)$ computing the part of process P to be preserved depends on the meaning of nesting of transaction scopes. In the literature, three main approaches are considered. When the enclosing transaction scope is killed, its subtransactions may be *aborted*, *preserved* or *discarded*. The aborting semantics is used by SAGAs calculi [9], WS-BPEL [26], and others. The preserving semantics is, for instance, the approach of Webπ [22]. Finally, the discarding semantics has been proposed by ATc [3] and TransCCS [12]. We consider all the three possibilities, since they just differ in the definition of function $\mathsf{extr}(\bullet)$.

Definition 1 (Extraction function). *We denote the functions corresponding to aborting, preserving, and discarding semantics for transaction nesting respectively as* $\mathsf{extr_a}(\bullet)$, $\mathsf{extr_p}(\bullet)$ *and* $\mathsf{extr_d}(\bullet)$. *The function* $\mathsf{extr_a}(\bullet)$ *is defined in Fig. 4. The definition of function* $\mathsf{extr_p}(\bullet)$ *is the same but for the clause for transaction scope, which is replaced by the clause* $\mathsf{extr_p}(t\,[P, Q]) = t\,[P, Q]$. *The definition of function* $\mathsf{extr_d}(\bullet)$ *instead is obtained by replacing the clause for transaction scope by the clause* $\mathsf{extr_d}(t\,[P, Q]) = \mathbf{0}$.

$$\begin{aligned}
\mathsf{extr_a}(\mathbf{0}) &= \mathbf{0} & \mathsf{extr_a}(\langle P\rangle) &= \langle P\rangle\\
\mathsf{extr_a}(\textstyle\sum_{i\in I}\pi_i.P_i) &= \mathbf{0} & \mathsf{extr_a}(t\,[P,Q]) &= \mathsf{extr_a}(P)\mid\langle Q\rangle\\
\mathsf{extr_a}(!\pi.P) &= \mathbf{0} & \mathsf{extr_a}(P\mid Q) &= \mathsf{extr_a}(P)\mid\mathsf{extr_a}(Q)\\
\mathsf{extr_a}(\mathsf{inst}\lfloor\lambda X.Q\rfloor.P) &= \mathbf{0}
\end{aligned}$$

Fig. 4. Extraction function for aborting semantics

$$\text{(P-Out)}\ \frac{}{\overline{a}\langle\boldsymbol{v}\rangle.P \xrightarrow{\overline{a}\langle\boldsymbol{v}\rangle} P} \qquad \text{(P-In)}\ \frac{}{a(\boldsymbol{x}).P \xrightarrow{a(\boldsymbol{v})} P\{\boldsymbol{v}/\boldsymbol{x}\}} \qquad \text{(L-Choice)}\ \frac{\pi_j.P_j \xrightarrow{\alpha} P'_j \quad j\in I}{\sum_{i\in I}\pi_i.P_i \xrightarrow{\alpha} P'_j}$$

$$\text{(L-Rep)}\ \frac{\pi.P \xrightarrow{\alpha} P'}{!\pi.P \xrightarrow{\alpha} P'|!\pi.P} \qquad \text{(L-Par)}\ \frac{P \xrightarrow{\alpha} P'}{P\mid Q \xrightarrow{\alpha} P'\mid Q} \qquad \text{(L-Synch)}\ \frac{P \xrightarrow{x(\boldsymbol{v})} P' \quad Q \xrightarrow{\overline{x}\langle\boldsymbol{v}\rangle} Q'}{P\mid Q \xrightarrow{\tau} P'\mid Q'}$$

$$\text{(L-Scope-out)}\ \frac{P \xrightarrow{\alpha} P' \quad \alpha \neq \lambda X.Q}{t[P,Q] \xrightarrow{\alpha} t[P',Q]} \qquad \text{(L-Recover-out)}\ \frac{}{t[P,Q] \xrightarrow{t} \mathsf{extr_a}(P)\mid\langle Q\rangle} \qquad \text{(L-Recover-in)}\ \frac{P \xrightarrow{\overline{t}} P'}{t[P,Q] \xrightarrow{\tau} \mathsf{extr_a}(P')\mid\langle Q\rangle}$$

$$\text{(L-Inst)}\ \frac{}{\mathsf{inst}\lfloor\lambda X.Q\rfloor.P \xrightarrow{\lambda X.Q} P} \qquad \text{(L-Scope-inst)}\ \frac{P \xrightarrow{\lambda X.R} P'}{t[P,Q] \xrightarrow{\tau} t[P',R\{Q/X\}]} \qquad \text{(L-Block)}\ \frac{P \xrightarrow{\alpha} P'}{\langle P\rangle \xrightarrow{\alpha} \langle P'\rangle}$$

Fig. 5. LTS for dynamic compensation processes

The operational semantics of dynamic compensations and, implicitly, of static, replacing, parallel and nested compensation processes, is defined below.

We use $a(\boldsymbol{v})$, $\overline{a}\langle\boldsymbol{v}\rangle$, τ and $\lambda X.Q$ as labels, and we use α to range over labels. The first three forms of labels are as in π-calculus: $a(\boldsymbol{v})$ is the input of a tuple of values $\boldsymbol{v}$ on channel a, $\overline{a}\langle\boldsymbol{v}\rangle$ is the corresponding output, and τ is an internal action. However, an output label without parameters can also be used for error notification, and an input without parameters for receiving the notification. The last label, $\lambda X.Q$, is specific of dynamic compensation processes and corresponds to compensation update. We write a for $a(\boldsymbol{v})$ and $\overline{a}$ for $\overline{a}\langle\boldsymbol{v}\rangle$ if $\boldsymbol{v}$ is empty. We may use t instead of a to emphasize that the name is used for error notification.

Definition 2 (Operational semantics). *The operational semantics of dynamic compensation processes with aborting semantics for transaction nesting is the minimum LTS closed under the rules in Fig. 5 (symmetric rules are considered for (*L-Par*) and (*L-Synch*)). The preserving semantics (resp. discarding semantics) is obtained by replacing function* $\mathsf{extr_a}(\bullet)$ *with* $\mathsf{extr_p}(\bullet)$ *(resp.* $\mathsf{extr_d}(\bullet)$*).*

The first six rules are standard π-calculus rules [23], the others define the behavior of transactions, compensations and protected blocks.

Auxiliary rules (P-Out) and (P-In) execute output and input prefixes, respectively. The input rule guesses the received values $\boldsymbol{v}$ in the early style. Rules (L-Choice) and (L-Rep) deal with guarded choice and replication, respectively. Rule (L-Par) allows one of the components of parallel composition to progress while the other one stays idle. Rule (L-Synch) performs communication, synchronizing an input $x(\boldsymbol{v})$ and a corresponding output $\overline{x}\langle\boldsymbol{v}\rangle$.

Rule (L-Scope-out) allows the body P of a transaction scope to progress, provided that the performed action is not a compensation update. Rule (L-Recover-out) allows external processes to abort a transaction scope via an output $\overline{t}$. The resulting process is composed of two parts: the first one extracted from the body P of the transaction scope, and the second one corresponding to compensation Q, which will be executed inside a protected block. Rule (L-Recover-in) is similar to (L-Recover-out), but now the error notification comes from the body P of the transaction scope. Rule (L-Inst) requires to perform a compensation update. Rule (L-Scope-inst) updates the compensation of a transaction scope. Finally, rule (L-Block) defines the behavior of protected blocks. The property of protected blocks of being unaffected by external aborts is enforced by the definition of function $\mathsf{extr}(\bullet)$.

In the following we consider a structural congruence $\equiv$ to rearrange the order of parallel processes and to garbage collect process $\mathbf{0}$. Formally, $\equiv$ is the least congruence such that $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$ and $P \mid \mathbf{0} \equiv P$.

As discussed in the Introduction, we will consider the (un)decidability of process termination: a process P terminates if there exists no infinite sequence of processes $P_1, P_2, \ldots, P_i, \ldots$ such that $P \xrightarrow{\tau} P_1 \xrightarrow{\tau} P_2 \xrightarrow{\tau} \ldots \xrightarrow{\tau} P_i \xrightarrow{\tau} \ldots$.

Example 1. We give here a few examples of transitions.

- Transaction scopes can compute:
  $\overline{a}\langle b\rangle \mid t[a(x).\overline{x}.0, Q] \xrightarrow{\tau} t[\overline{b}.0, Q]$
- Transaction scopes can be killed:
  $\overline{t} \mid t[\overline{a}.0, Q] \xrightarrow{\tau} \langle Q\rangle$
- Transaction scopes can commit suicide:
  $t[\,\overline{t}.0 \mid \overline{a}.0, Q] \xrightarrow{\tau} \langle Q\rangle$
- Protected blocks survive after kill:
  $t[\,\overline{t}.0 \mid \langle\overline{a}.0\rangle, Q] \xrightarrow{\tau} \langle\overline{a}.0\rangle \mid \langle Q\rangle$
- New compensation items can be added in parallel:
  $t[\mathsf{inst}\lfloor\lambda X.P|X\rfloor.\overline{a}.0, Q] \xrightarrow{\tau} t[\overline{a}.0, P|Q]$
- New compensation items can be added at the beginning:
  $t[\mathsf{inst}\lfloor\lambda X.\overline{b}.X\rfloor.\overline{a}.0, Q] \xrightarrow{\tau} t[\overline{a}.0, \overline{b}.Q]$
- Compensations can be deleted:
  $t[\mathsf{inst}\lfloor\lambda X.0\rfloor.\overline{a}.0, Q] \xrightarrow{\tau} t[\overline{a}.0, 0]$

3 Termination Undecidability for Nested Compensations

We now move to the proof of undecidability of termination in the calculus with nested compensations. This contrasts with the decidability of termination for static compensations (the proof of this result is deferred to Corollary 2).

The undecidability proof is by reduction from the termination problem in Random Access Machines (RAMs) [24], a well-known Turing powerful formalism based on registers containing non-negative natural numbers. The registers are used by a program, that is a set of indexed instructions I_i of two possible kinds:

- $i : Inc(r_j)$ that increments the register r_j and then moves to the execution of the instruction with index $i + 1$ and
- $i : DecJump(r_j, s)$ that attempts to decrement the register r_j; if the register does not hold 0 then the register is actually decremented and the next instruction is the one with index $i + 1$, otherwise registers are unchanged and the next instruction is the one with index s.

We assume that given a program $I_1, \cdots, I_n$, it starts by executing I_1. It terminates when an undefined program instruction is reached. Since the computational model is Turing complete, the termination of a RAM program is undecidable.

We encode RAMs as follows. Each register r_j containing the value n is encoded as a transaction $r_j[R_j, Q_j]$ where Q_j is a process $\overline{u}.\overline{u}.\cdots.\overline{u}.\overline{z}$ with exactly n prefixes $\overline{u}$. The process R_j is responsible for updating its compensation Q_j by performing $\mathsf{inst}\lfloor\lambda X.\overline{u}.X\rfloor$ every time the register must be incremented. Each instruction I_i will be encoded as a process $!p_i.P_i$: the instruction will be activated by $\overline{p_i}$ and then P_i will be performed. If $i : Inc(r_j)$ is an increment instruction on r_j, P_i will interact with R_j in order to activate the update of its compensation Q_j. If $i : DecJump(r_j, s)$ is a decrement/jump instruction, on the other hand, P_i will terminate the transaction r_j so that the compensation Q_j becomes active. If Q_j is $\overline{z}$ then the value of the register is 0. In this case a new instance of the register $r_j[R_j, \overline{z}]$ is spawn and the jump is executed. If Q_j is $\overline{u}.\cdots.\overline{z}$ then the register is not 0. In this case, a new instance of the register $r_j[R_j, \overline{z}]$ is spawn and a protocol is started to initialize correctly this new register. The protocol is between the process R_j and the compensation $\overline{u}.\cdots.\overline{z}$ left by the previous instance of the register. The process R_j consumes the remaining prefixes $\overline{u}$, and for each of them performs an $\mathsf{inst}\lfloor\lambda X.\overline{u}.X\rfloor$ action in order to update its compensation accordingly. In this way, at the end of the protocol, the new register instance will have a compensation $\overline{u}.\cdots.\overline{z}$ with one prefix $\overline{u}$ less w.r.t. the previous register instance.

Formally, the translation of register j storing value n is as follows:

$$[\![r_j = n]\!] \triangleq r_j[!inc_j.\,\mathsf{inst}\lfloor\lambda X.\overline{u}.X\rfloor.\overline{ack} \mid !rec_j.\big(u.\,\mathsf{inst}\lfloor\lambda X.\overline{u}.X\rfloor.\overline{rec_j}{+}z.\overline{ack}\big),\ \overline{u}^n.\overline{z}]$$

where $\overline{u}^n$ is a sequence of n prefixes $\overline{u}$. The encoding of instructions is as follows:

$$[\![i : Inc(r_j)]\!] \triangleq !p_i.\overline{inc_j}.ack.\overline{p_{i+1}}$$
$$[\![i : DecJump(r_j, s)]\!] \triangleq !p_i.\overline{r_j}.\big(z.([\![r_j = 0]\!]|\overline{p_s}) + u.(\overline{rec_j}|[\![r_j = 0]\!]|ack.\overline{p_{i+1}})\big)$$

Hence, given a RAM program $I_1, \cdots, I_n$ with registers $r_1, \ldots, r_m$ with initial values $n_1, \ldots, n_m$ the corresponding encoding is:

$$\overline{p_1}|[\![I_1]\!]|\ldots|[\![I_n]\!]|[\![r_1 = n_1]\!]|\ldots|[\![r_m = n_m]\!]$$

In the proof of correctness of the encoding we use $P \rightarrow^k_{\equiv} Q$ to denote the existence of $Q_1, \cdots, Q_k$ such that $P \xrightarrow{\tau} Q_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} Q_k$ and $Q_k \equiv Q$.

Theorem 1.
Given $P \equiv \overline{p_l}|[\![I_1]\!]|\ldots|[\![I_n]\!]|[\![r_1 = n_1]\!]|\ldots|[\![r_j = n_j]\!]|\ldots|[\![r_m = n_m]\!]$ *we have:*

1. $I_l : Inc(r_j)$ *iff*
   $P \rightarrow^4_{\equiv} \overline{p_{l+1}}|[\![I_1]\!]|\ldots|[\![I_n]\!]|[\![r_1 = n_1]\!]|\ldots|[\![r_j = n_j + 1]\!]|\ldots|[\![r_m = n_m]\!]$;
2. $I_l : DecJump(r_j, s)$ *and* $n_j = 0$ *iff*
   $P \rightarrow^3_{\equiv} \overline{p_s}|[\![I_1]\!]|\ldots|[\![I_n]\!]|[\![r_1 = n_1]\!]|\ldots|[\![r_j = 0]\!]|\ldots|[\![r_m = n_m]\!]$;
3. $I_l : DecJump(r_j, s)$ *and* $n_j \neq 0$ *iff*
   $P \rightarrow^k_{\equiv} \overline{p_{l+1}}|[\![I_1]\!]|\ldots|[\![I_n]\!]|[\![r_1 = n_1]\!]|\ldots|[\![r_j = n_j - 1]\!]|\ldots|[\![r_m = n_m]\!]$ *with* $k = 3(n_j - 1) + 5$;
4. I_l *is undefined iff there exists no* P' *s.t.* $P \xrightarrow{\tau} P'$.

Proof. In each case there is just one possible computation, that we describe by listing the channels on which synchronizations happen or the installation of compensation performed:

1. p_l, inc_j, $\mathsf{inst}\lfloor \lambda X.\overline{u}.X \rfloor$, ack: 4 transitions;
2. p_l, r_j, z: 3 transitions;
3. p_l, r_j, u, rec_j, then the sequence u, $\mathsf{inst}\lfloor \lambda X.\overline{u}.X \rfloor$, rec_j repeated $n_j - 1$ times, and finally z, ack: $3(n_j - 1) + 5$ transitions;
4. no synchronization is possible. □

We finally conclude with the proof of the undecidability result.

Corollary 1. *Termination is undecidable in* π*-calculus with nested compensations.*

Proof. By Theorem 1 each step of a RAM precisely corresponds to a finite number of steps of its encoding, thus a RAM terminates iff its encoding terminates. Thus, termination of RAMs reduces to termination in π-calculus with nested compensations. Since termination in RAMs is undecidable then also termination in π-calculus with nested compensations is undecidable. □

4 Decidability for Parallel and Replacing Compensations

We now consider the cases in which all dynamic compensation installations follow the replace or the parallel patterns. In the first case, only finitely many different compensation processes can be considered. In the second case, infinitely many compensations can be reached, but all of them are parallel compositions of finitely many distinct processes (the processes Q occurring in the updates $\lambda X.Q \mid X$, and static compensations R in $t[P, R]$). This property of the calculus allows us to apply the theory of Well-Structured Transition Systems (WSTSs) to prove that termination is decidable.

We start by recalling some basic notions about WSTSs [1, 15]. A reflexive and transitive relation is called *quasi-ordering*. A *well-quasi-ordering* (wqo) is a quasi-ordering $(X, \leq)$ such that, for every infinite sequence $x_1, x_2, x_3, \cdots$, there exist $i < j$ with $x_i \leq x_j$. From this, it follows that there exists also an infinite increasing subsequence $x_{k_1}, x_{k_2}, x_{k_3}, \cdots$ such that $x_{k_l} \leq x_{k_m}$ for every

$l < m$. Given a wqo $(X, \leq)$, we denote its extension to k-tuples as $(X^k, \leq^k)$: $\langle x_1, \cdots, x_k \rangle \leq^k \langle y_1, \cdots, y_k \rangle$ iff $x_i \leq y_i$ for $1 \leq i \leq k$. Dickson's lemma [14] states that if $(X, \leq)$ is a wqo, then also $(X^k, \leq^k)$ is a wqo. Given a wqo $(X, \leq)$, we denote its extension to finite sequences as $(X^*, \leq^*)$: $\langle x_1, \cdots, x_n \rangle \leq^* \langle y_1, \cdots, y_m \rangle$ iff there exists a subsequence $\langle y_{l_1}, \cdots, y_{l_n} \rangle$ of the latter s.t. $x_i \leq y_{l_i}$ for $1 \leq i \leq n$. Higman's lemma [17] states that if $(X, \leq)$ is a wqo, then also $(X^*, \leq^*)$ is a wqo. We now report a definition of WSTS appropriate for our purposes.

Definition 3. *A WSTS is a transition system $(\mathcal{S}, \rightarrow, \preceq)$ where $\preceq$ is a wqo on $\mathcal{S}$ which is* compatible *with $\rightarrow$, i.e., for every $s_1 \preceq s'_1$ such that $s_1 \rightarrow s_2$, there exists $s'_1 \rightarrow s'_2$ such that $s_2 \preceq s'_2$. Moreover, the function Succ(s), returning the set $\{s' \in \mathcal{S} \mid s \rightarrow s'\}$ of immediate successors of s, is computable.*

A state s in a WSTS *terminates* if there exists no infinite computation $s \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$. The proposition below is a special case of Theorem 4.6 in [15].

Proposition 1. *Termination is decidable for WSTSs.*

Given a process P with replacing or parallel compensations, we prove that a transition system that includes all the derivatives of P is a WSTS. By derivatives, denoted with $\mathsf{der}(P)$, we mean the processes that can be reached from P via transitions labeled with τ, denoted simply with $\rightarrow$ in the following. We first observe that given a process Q, the set of its immediate successors according to $\rightarrow$ is finite (and computable). This follows from the limitation to τ-labeled transitions: the labeled transition system in Fig. 5 is not finitely branching because the rule (P-In) has an instantiation for each of the infinitely many possible vectors of values $\boldsymbol{v}$, but if we restrict to τ transitions, only finitely many names can be actually received because in our calculus no new names can be dynamically generated. Concerning names, we also make the nonrestrictive assumption that in process P the free names used in output actions are all distinct from the bound names used in input actions. In this way, it is not necessary to apply α-conversions to avoid name captures during substitutions. This guarantees that only the names initially present in P will occur in its derivatives.

We now move to the definition of our wqo. Intuitively, a process P is smaller than a process Q if Q can be obtained from P by adding some processes in parallel while preserving the same structure of transaction scopes and protected blocks.

Definition 4. *Let P, Q be two processes. We write $P \preceq Q$ iff there exist P', S, n, m, $t_1, \ldots, t_n$, $P_1, \ldots, P_n, P'_1, \ldots, P'_n$, $Q_1, \ldots, Q_n, Q'_1, \ldots, Q'_n$, $R_1, \ldots, R_m$ and $R'_1, \ldots, R'_m$ such that*

$$P \equiv P' \mid \prod_{i=1}^{n} t_i[P_i, Q_i] \mid \prod_{j=1}^{m} \langle R_j \rangle$$
$$Q \equiv P' \mid S \mid \prod_{i=1}^{n} t_i[P'_i, Q'_i] \mid \prod_{j=1}^{m} \langle R'_j \rangle$$

with $P_i \preceq P'_i$ and $Q_i \preceq Q'_i$, for $1 \leq i \leq n$, and $R_j \preceq R'_j$, for $1 \leq j \leq m$.

In order to prove that $\preceq$ is indeed a wqo over the derivatives of P we need some more notation and preliminary results. First we define the maximum nesting level $\mathsf{depth}(P)$ of nested transactions and protected blocks in a process P.

Definition 5. *Let P be a process. We define* $\mathsf{depth}(P)$ *inductively as follows:*

$$\begin{array}{l}\mathsf{depth}(\mathbf{0}) = \mathsf{depth}(X) = 0\\ \mathsf{depth}(\sum_{i\in I}\pi_i.P_i) = max_{i\in I}\,\mathsf{depth}(P_i)\\ \mathsf{depth}(!\pi.P) = \mathsf{depth}(P)\\ \mathsf{depth}(\mathsf{inst}\lfloor\lambda X.Q\rfloor.P) = max(\mathsf{depth}(P),\mathsf{depth}(Q))\\ \mathsf{depth}(P \mid Q) = max(\mathsf{depth}(P),\mathsf{depth}(Q))\\ \mathsf{depth}(t[P,Q]) = 1 + max(\mathsf{depth}(P),\mathsf{depth}(Q))\\ \mathsf{depth}(\langle P\rangle) = 1 + \mathsf{depth}(P)\end{array}$$

It is trivial to see that the extraction functions do not increase the maximum nesting levels in all the three considered cases. Formally, $\mathsf{depth}(\mathsf{extr}_\mathsf{a}(P)) \leq \mathsf{depth}(P)$, $\mathsf{depth}(\mathsf{extr}_\mathsf{p}(P)) \leq \mathsf{depth}(P)$ and $\mathsf{depth}(\mathsf{extr}_\mathsf{d}(P)) \leq \mathsf{depth}(P)$. We now prove that also the labeled transitions do not increase the nesting levels.

Proposition 2. *Let P be a process with replacing or parallel compensations. If $P \xrightarrow{\alpha} Q$ then* $\mathsf{depth}(Q) \leq \mathsf{depth}(P)$.

Proof. We first observe that for every transition $T \xrightarrow{\lambda X.S} T'$ we have that $\mathsf{depth}(S) \leq \mathsf{depth}(T)$. In the light of this preliminary result the thesis can be easily proved by induction on the depth of the proof of $P \xrightarrow{\alpha} Q$. The unique interesting case is when the rule (L-Scope-inst) is used. Consider the transition $t[P,Q] \xrightarrow{\lambda X.R} t[P', R\{^Q/_X\}]$ inferred by $P \xrightarrow{\lambda X.R} P'$. We have that $t[P', R\{^Q/_X\}]$ does not have a greater maximum nesting level because $\mathsf{depth}(R) \leq \mathsf{depth}(P)$, for the above observation, and $\mathsf{depth}(R\{^Q/_X\}) \leq max(\mathsf{depth}(Q),\mathsf{depth}(R))$ due to the specificity of the replace and parallel update patterns. □

As a trivial corollary we have that the maximum nesting level of the derivatives of P (i.e. processes in $\mathsf{der}(P)$) is smaller or equal to $\mathsf{depth}(P)$. This result will be used to define a superset of $\mathsf{der}(P)$ for which we will prove that $\preceq$ is indeed a wqo. In the definition of this superset we also need the notion of a sequential subprocess of P, that is a subterm of P in which the top operator is not a parallel composition, a transaction or a protection block.

Definition 6. *Let P be a process. The set* $\mathsf{seq}(P)$ *containing all the sequential subprocesses of P is defined inductively as follows:*

$$\begin{array}{l}\mathsf{seq}(\mathbf{0}) = \{\mathbf{0}\}\\ \mathsf{seq}(\sum_{i\in I}\pi_i.P_i) = \{\sum_{i\in I}\pi_i.P_i\} \cup \bigcup_{i\in I}\mathsf{seq}(P_i)\\ \mathsf{seq}(!\pi.P) = \{!\pi.P\} \cup \mathsf{seq}(P)\\ \mathsf{seq}(\mathsf{inst}\lfloor\lambda X.Q\rfloor.P) = \mathsf{inst}\lfloor\lambda X.Q\rfloor.P \cup \mathsf{seq}(P) \cup \mathsf{seq}(Q)\\ \mathsf{seq}(X) = \emptyset\\ \mathsf{seq}(P \mid Q) = \mathsf{seq}(t[P,Q]) = \mathsf{seq}(P) \cup \mathsf{seq}(Q)\\ \mathsf{seq}(\langle P\rangle) = \mathsf{seq}(P)\end{array}$$

The intuition is that no new sequential subprocesses can be generated by derivatives. To be more precise, after the execution of an input action, new subprocesses can be reached due to name substitution. But, as observed above, the names in

a derivative in $\mathsf{der}(P)$ already occur in P, thus they are finite. This allows us to characterize a superset of $\mathsf{der}(P)$ as follows.

Definition 7. *Given a process Q, we use $\mathsf{names}(Q)$ to denote the set of names occurring in Q. Let P be a process and n be a natural number; we denote with*

$$\mathsf{comb}_P(n) = \{Q \mid \mathsf{names}(Q) \subseteq \mathsf{names}(P), \mathsf{depth}(Q) \leq n, \\ \forall Q' \in \mathsf{seq}(Q).\exists P' \in \mathsf{seq}(P).Q' = P\{\boldsymbol{v}/\boldsymbol{x}\} \text{ for some } \boldsymbol{v} \text{ and } \boldsymbol{x}\}$$

the set of processes with names that already occur in P, with maximum nesting level smaller than n, and containing sequential subprocesses that already occur in P (up-to renaming).

We now prove that $\mathsf{comb}_P(\mathsf{depth}(P))$ is actually a superset of $\mathsf{der}(P)$.

Proposition 3. *Let P be a process with replacing or parallel compensations. Then $\mathsf{der}(P) \subseteq \mathsf{comb}_P(\mathsf{depth}(P))$.*

Proof. We first observe that $P \in \mathsf{comb}_P(\mathsf{depth}(P))$. Then we consider a process $Q \in \mathsf{comb}_P(\mathsf{depth}(P))$ such that $Q \to Q'$, and we show that also $Q' \in \mathsf{comb}_P(\mathsf{depth}(P))$. By Proposition 2 we have that $\mathsf{depth}(Q') \leq \mathsf{depth}(Q)$ hence also $\mathsf{depth}(Q') \leq \mathsf{depth}(P)$. Moreover, it is easy to see that Q' does not introduce new sequential subprocesses (it can at most apply a name substitution to sequential subprocesses of Q). Notice that in case the transition is a compensation update, no new sequential subprocesses can be obtained because either the replace or the parallel pattern is used. □

We are finally ready to prove that $(\mathsf{comb}_P(\mathsf{depth}(P)), \preceq)$ is indeed a wqo, by proving a slightly more general result.

Theorem 2. *Let P be a process and let n be a natural number. The relation $\preceq$ is a wqo over $\mathsf{comb}_P(n)$.*

Proof. Take an infinite sequence $P_1, P_2, \ldots, P_i, \ldots$, with $P_i \in \mathsf{comb}_P(n)$ for every $i > 0$. We prove, by induction on n, that there exist k and l such that $P_k \preceq P_l$.

Let $n = 0$. All the processes P_i do not contain neither transactions nor protected blocks because $\mathsf{depth}(P_i) \leq 0$. For this reason, we have that $P_i = \prod_{j=1}^{n_i} P_{i,j}$ with $P_{i,j}$ equal to some sequential subprocess of P (up-to renaming by using names already in P). This set is finite, then process equality $=$ is a wqo over this set. By Higman's lemma we have that also $=^*$ is a wqo over finite sequences of such processes. Hence there exists k and l such that $P_{k,1} \ldots P_{k,n_k}$ is a subsequence of $P_{l,1} \ldots P_{l,n_l}$, hence we have $P_k \preceq P_l$.

For the inductive step, let $n > 0$ and assume that the thesis holds for $\mathsf{comb}_P(n-1)$. We have that the following holds for every P_i:

$$P_i \equiv \prod_{j=1}^{n_i} P_{i,j} \mid \prod_{j=1}^{m_i} t_{i,j}[Q_{i,j}, R_{i,j}] \mid \prod_{j=1}^{o_i} \langle S_{i,j} \rangle$$

with $P_{i,j}$ equal to some sequential subprocess of P (up-to renaming by using names already in P), $t_{i,j} \in \mathsf{names}(P)$ and $Q_{i,j}$, $R_{i,j}$, $S_{i,j}$ have a maximum

nesting level strictly smaller than n, hence $Q_{i,j}, R_{i,j}, S_{i,j} \in \mathsf{comb}_P(n-1)$. We now consider every process P_i as composed of 3 finite sequences: $P_{i,1} \cdots P_{i,n_i}$, $\langle t_{i,1}, Q_{i,1}, R_{i,1}\rangle \cdots \langle t_{i,m_i}, Q_{i,m_i}, R_{i,m_i}\rangle$, and $S_{i,1} \cdots S_{i,o_i}$. As observed above, $=^*$ is a wqo over the sequences $P_{i,1} \cdots P_{i,n_i}$. For this reason we can extract an infinite subsequence of $P_1, P_2, \ldots$ making the finite sequences $P_{i,1} \cdots P_{i,n_i}$ increasing w.r.t. $=^*$. We now consider the triples $\langle t_{i,j}, Q_{i,j}, R_{i,j}\rangle$. Consider the ordering $(\mathsf{comb}_P(n-1) \cup \mathsf{names}(P), \sqsubseteq)$ such that $x \sqsubseteq y$ iff $x = y$, if $x, y \in \mathsf{names}(P)$, or $x \preceq y$, if $x, y \in \mathsf{comb}_P(n-1)$. As $\mathsf{names}(P)$ is finite and due to the inductive hypothesis according to which $(\mathsf{comb}_P(n-1), \preceq)$ is a wqo, we have that also $(\mathsf{comb}_P(n-1) \cup \mathsf{names}(P), \sqsubseteq)$ is a wqo. By Dickson's lemma we have that $\sqsubseteq^3$ is a wqo over the considered triples $\langle t_{i,j}, Q_{i,j}, R_{i,j}\rangle$. We can apply the Higman's lemma as above to prove that it is possible to extract, from the above infinite subsequence, an infinite subsequence making the finite sequences $\langle t_{i,1}, Q_{i,1}, R_{i,1}\rangle \cdots \langle t_{i,m_i}, Q_{i,m_i}, R_{i,m_i}\rangle$ increasing w.r.t. $(\sqsubseteq^k)^*$. Finally, as $S_{i,j} \in \mathsf{comb}_P(n-1)$ and by inductive hypothesis, we can finally apply again Higman's lemma to extract, from the last infinite sequence, an infinite subsequence making the finite sequences $S_{i,1} \cdots S_{i,o_i}$ increasing w.r.t. $\preceq^*$. It is now sufficient to take from this last subsequence two processes P_k and P_l, with $k < l$, and observe that $P_k \preceq P_l$. □

We now move to the proof of compatibility between the ordering $\preceq$ and the transition system $\rightarrow$.

Lemma 1. *If $P \preceq P'$ and $P \xrightarrow{\alpha} Q$ then there exists Q' such that $Q \preceq Q'$ and $P' \xrightarrow{\alpha} Q'$.*

Proof. By induction on the depth of the proof of $P \xrightarrow{\alpha} Q$. □

As the transitions $\rightarrow$ correspond to transitions labeled with τ, as a trivial corollary we have the compatibility of $\preceq$ with $\rightarrow$. Hence, we can conclude that given a process P with replacing or parallel (as well as static) compensations, $(\mathsf{comb}_P(\mathsf{depth}(P)), \rightarrow, \preceq)$ is a WSTS. As a consequence, we obtain our decidability result.

Corollary 2. *Let P be a process with replacing, parallel or static compensations. The termination of P is decidable.*

Proof. By definition, P terminates iff there exists no infinite computation $P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \ldots$. For replacing and parallel compensations, by Proposition 3, this holds iff P terminates in the transition system $(\mathsf{comb}_P(\mathsf{depth}(P)), \rightarrow)$. But this last problem is decidable, by Proposition 1, because $(\mathsf{comb}_P(\mathsf{depth}(P)), \rightarrow, \preceq)$ is a WSTS. The result for static compensations follows since they form a subcalculus of replacing/parallel compensations. □

5 Related Work and Conclusion

In this paper we studied decidability properties of π-calculus extended with primitives for specifying transactions and compensations. Fig. 6 shows all the considered

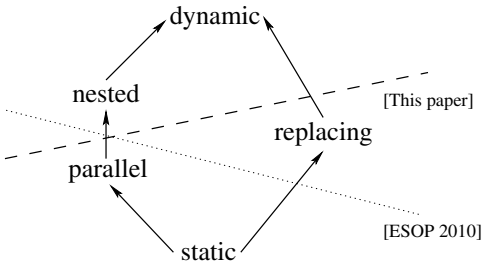


Fig. 6. Separation results for compensation mechanisms

calculi. Arrows denote the subcalculus relation. As already said, [20] is the closest paper to ours. There, relying on syntactic conditions restricting the allowed class of encodings and requiring some strong semantic properties to be preserved, the authors proved the separation result represented by the dotted line. The results in this paper instead, requiring only termination preservation, prove the separation result represented by the dashed line. Besides separation, [20] also showed an encoding proving the equivalence of static and parallel compensations. This result, compatible with our separation result, cannot be straightforwardly applied in our setting since it relies on the restriction operator. However, if one disallows transactions under a replication prefix, our decidability results still hold and the encoding in [20] can be applied. It would be interesting to look for termination-preserving encodings of dynamic into nested compensations and replacing into static compensations (such an encoding should violate some of the conditions in [20]).

The only other results comparing the expressive power of primitives for transactions and compensations are in the field of SAGAs [9]/cCSP [10], but their setting allows only for isolated activities, since it does not consider communication. There are two kinds of results: a few papers compare different variants of SAGAs [6, 7, 18], while others use SAGAs-like calculi as specifications for π-style processes [11, 21]. Both kinds of results cannot be easily compared with ours.

Interestingly, our results have been studied in the framework of π-calculus since it is the base of most proposals in the literature, but can similarly be stated in CCS. Sticking to π-calculus, adding priority of compensation installation to the calculus, as done by [16, 20, 27], does not alter the undecidability of termination for nested and dynamic compensations. For the decidability in parallel and replacing compensations instead, the proof cannot be applied. Note however that priority of compensation installation reduces the set of allowed traces, thus termination without priority ensures termination with priority (but the opposite is not true).

Decidability and undecidability results are a well-established tool to separate the expressive power of process calculi. We restrict our discussion to few recent papers. In [5] two operators for modeling the interruption of processes are considered: $P \lhd Q$ that behaves like P until Q starts and $\texttt{try}\, P\, \texttt{catch}\, Q$ that behaves like P until a `throw` action is executed by P to activate Q. Termination is proved

to be undecidable for $\texttt{try}\, P\, \texttt{catch}\, Q$ while it is decidable for $P \lhd Q$. The undecidability proof is different from the one in this paper since it exploits unbounded nesting of try-catch constructs. The decidability proof requires to use a weaker ordering (tree embedding) in order to deal with unbounded nesting of interrupt operators. Such ordering is not appropriate for the calculus in the present paper because compatibility is broken by the prefix $\mathsf{inst}\lfloor \lambda X.Q \rfloor$ that synchronizes with the nearest enclosing transaction and not with any of the outer transactions. In [13] higher-order π-calculus without restriction is considered. Despite higher-order communication is rather different w.r.t. dynamic compensations, a similar decidability result is proved: if the received processes cannot be modified when they are forwarded, termination becomes decidable, while this is not the case if they can [19]. The decidability proof is simpler w.r.t. the one in this paper because there is no operator, like $t[P, Q]$, that requires the exploitation of Dickson's lemma. Finally, we mention [4] where a calculus for adaptable processes is presented: running processes can be dynamically modified by executing update patterns similar to those used in this paper. A safety property is proved to be decidable if the update pattern does not add prefixes in front of the adapted process, while it becomes undecidable if a more permissive pattern is admitted. The undecidability proof in the present paper is more complex because update patterns can be executed only on inactive processes (the compensations). The decidability proof in [4] is similar to the one in [5]: the same comments above holds also in this case.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: Proc. of LICS 1996, pp. 313–321. IEEE (1996)
2. Bocchi, L., Laneve, C., Zavattaro, G.: A calculus for long-running transactions. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 124–138. Springer, Heidelberg (2003)
3. Bocchi, L., Tuosto, E.: A java inspired semantics for transactions in SOC. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) TGC 2010, LNCS, vol. 6084, pp. 120–134. Springer, Heidelberg (2010)
4. Bravetti, M., Di Giusto, C., Pérez, J.A., Zavattaro, G.: Adaptable processes. Logical Methods in Computer Science 8(4) (2012)
5. Bravetti, M., Zavattaro, G.: On the expressive power of process interruption and compensation. Math. Struct. Comp. Sci. 19(3), 565–599 (2009)
6. Bruni, R., Butler, M., Ferreira, C., Hoare, T., Melgratti, H., Montanari, U.: Comparing two approaches to compensable flow composition. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 383–397. Springer, Heidelberg (2005)
7. Bruni, R., Kersten, A., Lanese, I., Spagnolo, G.: A new strategy for distributed compensations with interruption in long-running transactions. In: Mossakowski, T., Kreowski, H.-J. (eds.) WADT 2010. LNCS, vol. 7137, pp. 42–60. Springer, Heidelberg (2012)
8. Bruni, R., Melgratti, H.C., Montanari, U.: Nested commits for mobile calculi: Extending join. In: Proc. of IFIP TCS 2004, pp. 563–576. Kluwer (2004)

9. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Proc. of POPL 2005, pp. 209–220. ACM Press (2005)
10. Butler, M., Hoare, T., Ferreira, C.: A trace semantics for long-running transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)
11. Caires, L., Ferreira, C., Vieira, H.: A process calculus analysis of compensations. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 87–103. Springer, Heidelberg (2009)
12. de Vries, E., Koutavas, V., Hennessy, M.: Communicating transactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 569–583. Springer, Heidelberg (2010)
13. Di Giusto, C., Pérez, J.A., Zavattaro, G.: On the expressiveness of forwarding in higher-order communication. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 155–169. Springer, Heidelberg (2009)
14. Dickson, L.E.: Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. Amer. J. Math. 35(4), 413–422 (1913)
15. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science 256, 63–92 (2001)
16. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundamenta Informaticae 95(1), 73–102 (2009)
17. Higman, G.: Ordering by divisibility in abstract algebras. Proc. London Math. Soc., 3rd Series 2, 326–336 (1952)
18. Lanese, I.: Static vs dynamic sagas. In: Proc. of ICE 2010. EPTCS, vol. 38, pp. 51–65 (2010)
19. Lanese, I., Pérez, J.A., Sangiorgi, D., Schmitt, A.: On the expressiveness and decidability of higher-order process calculi. In: Proc. of LICS 2008, pp. 145–155. IEEE Computer Society (2008)
20. Lanese, I., Vaz, C., Ferreira, C.: On the expressive power of primitives for compensation handling. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 366–386. Springer, Heidelberg (2010)
21. Lanese, I., Zavattaro, G.: Programming Sagas in SOCK. In: Proc. of SEFM 2009, pp. 189–198. IEEE Computer Society Press (2009)
22. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
23. Milner, R., Parrow, J., Walker, J.: A calculus of mobile processes, I and II. Inf. Comput. 100(1), 1–40,41–77 (1992)
24. Minsky, M.: Computation: finite and infinite machines. Prentice-Hall, Englewood Cliffs (1967)
25. Montesi, F., Guidi, C., Zavattaro, G.: Composing services with JOLIE. In: Proc. of ECOWS 2007, pp. 13–22. IEEE Computer Society (2007)
26. Oasis. Web Services Business Process Execution Language Version 2.0 (2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
27. Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 201–215. Springer, Heidelberg (2009)

Probabilistic Modular Embedding for Stochastic Coordinated Systems

Stefano Mariani and Andrea Omicini

DISI, Alma Mater Studiorum–Università di Bologna
{s.mariani,andrea.omicini}@unibo.it

Abstract. *Embedding* and *modular embedding* are two well-known techniques for measuring and comparing the expressiveness of languages—sequential and concurrent programming languages, respectively. The emergence of new classes of computational systems featuring stochastic behaviours – such as pervasive, adaptive, self-organising systems – requires new tools for probabilistic languages. In this paper, we recall and refine the notion of *probabilistic modular embedding* (PME) as an extension to modular embedding meant to capture the expressiveness of stochastic systems, and show its application to different coordination languages providing probabilistic mechanisms for stochastic systems.

Keywords: embedding, modular embedding, coordination languages, probabilistic languages, π-calculus.

1 Introduction

A core issue for computer science since the early days, expressiveness of computational languages is essential nowadays, when the focus is shifting from sequential to concurrent languages. This holds in particular for coordination languages, which, by focussing on interaction, deal with the most relevant source of complexity in computational systems [1]. Unsurprisingly, the area of coordination models and languages has produced a long stream of ideas and results on the subject, both adopting/adapting "traditional" approaches – such as Turing equivalence for coordination languages [2,3] – and inventing its own original techniques [4].

Comparing languages based on either their structural properties or the observable behaviour of the systems built upon them is seemingly a good way to classify their expressiveness. Among the many available approaches, the notion of *modular embedding* [5], refinement of Shapiro's *embedding* [6], is particularly effective in capturing the expressiveness of concurrent and coordination languages. However, the emergence of classes of systems featuring new sorts of behaviours – pervasive, adaptive, self-organising systems [7,8] – is pushing computational languages beyond their previous limits, and asks for new models and techniques to observe, model and, measure their expressiveness. In particular, modular embedding fails in telling probabilistic languages apart from non-probabilistic ones.

Accordingly, in this paper we recall, refine, and extend applicability of the notion of *probabilistic modular embedding* (PME) sketched in [9] as a formal

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 151–165, 2013.

framework to capture the expressiveness of probabilistic languages and stochastic systems. After recalling the basis of embedding and modular embedding (ME), along with some formal tools for dealing with probability (Section 2), in Section 3 we devise out the requirements for a probabilistic framework, and formalise them so as to define the full notion of PME. In order to demonstrate its ability to measure language expressiveness, in Section 4 we test PME against ME using a number of different case studies – probabilistic coordination languages and calculi –, and show how PME succeeds where ME simply fails. After discussing related works, such as *bisimulation* and *probabilistic bisimulation* (Section 5), we provide some final remarks and hints at possible future works (Section 6).

2 Background

2.1 Sequential and Modular Embedding

The informal definition of *embedding* assumes that a language could be *easily* and *equivalently* translated in another one. "Easily" is usually interpreted as "without the need for a global reorganisation of the program", whatever this means; whereas "equivalently" typically means "without affecting the program's *observable behaviour*", according to some well-defined *observation criteria*, usually to be specified for the application at hand.

Such an intuitive definition was formalised by Shapiro [6] for *sequential languages* as follows. Given two languages L, L', their program sets $Prog_L, Prog_{L'}$, and the powersets of their observable behaviours Obs, Obs', we assume that two *observation criteria* Ψ, Ψ' hold:

$$\Psi : Prog_L \to Obs \qquad \Psi' : Prog_{L'} \to Obs'$$

Then, L *embeds* L' (written $L \succeq L'$) *iff* there exist a *compiler* $C : Prog_{L'} \to Prog_L$ and a *decoder* $D : Obs \to Obs'$ such that for every program $W \in L'$

$$D(\Psi[C(W)]) = \Psi'[W]$$

Subsequently, De Boer and Palamidessi [5] argued such definition to be too weak to be applied proficiently, because any pair of Turing-complete languages would embed each other. Moreover, *concurrent languages* need at least *(i)* a novel notion of *termination* w.r.t. sequential ones, so as to handle deadlock and computation failure, and *(ii)* a different definition for the compiler, so as to consider also a priori unknown *run-time interactions* between concurrent processes.

Following their intuitions, De Boer and Palamidessi proposed a novel definition of embedding for which C and D should satisfy three properties:

Independent Observation. Elements $O \in Obs$ are sets representing all the possible outcomes of all the possible computations of a given system, hence they will be typically observed independently one from the other—since they are different systems. Thus, D can be defined to be *elementwise*, that is:

$$\forall O \in Obs : D(O) = \{d(o) \mid o \in O\} \text{ (for some } d)$$

Compositionality of C. In a concurrent setting, it is difficult to predict the behaviour of all the processes in the environment, due to run-time non-deterministic interactions. Therefore, it is reasonable to require compositionality of the compiler C both w.r.t. the parallel composition operator ($||$) and to the exclusive choice ($+$). Formally:

$$C(A \;||' \; B) = C(A) \;||\; C(B) \quad \text{and} \quad C(A \;+' \; B) = C(A) \;+\; C(B)$$

for every pair of programs $A, B \in L'$, where $'$ denotes symbols of L'.

Deadlock Invariance. Unlike sequential languages, where only successful computations do matter – basically because unsuccessful ones could be supposed to backtrack –, in a concurrent setting we need to consider at least deadlocks, interpreting failure as a special case of deadlock, which should then be preserved by the decoder D:

$$\forall O \in Obs, \forall o \in O : tm'(D_{d(o)}) = tm(o)$$

where tm and tm' refer to termination modes of L and L' respectively.

If an embedding satisfies all the three properties above, then it is called *modular*. In the following, we stick with symbol $\succeq$ since we assume the corresponding notion as our "default" reference embedding.

2.2 Expressiveness of Modular Embedding

A Case Study: `ProbLinCa` In [10], the authors define the `ProbLinCa` calculus, a probabilistic extension of the `LinCa` calculus therein defined, too. Whereas the latter is basically a process-algebraic formalisation of standard LINDA – accounting for `out`, `rd`, `in` primitives –, the former equips each tuple with a *weight*, resembling selection probability: the higher the weight of a tuple, the higher its probability to be selected for matching. Basically, when a tuple template is used in a LINDA primitive, the weights of the (possibly many, and different in kind) matching tuples are summed up, then each (j) is assigned a probability value $p \in [0, 1]$ obtained as follows: $p_j = \frac{w_j}{\sum_{i=1}^{n} w_i}$.

Suppose the following `ProbLinCa` process P and `LinCa` process Q are acting on tuple space S:

$$P = \mathtt{in}_p(T).\emptyset + \mathtt{in}_p(T).\mathtt{rd}_p(T').\emptyset \qquad Q = \mathtt{in}(T).\emptyset + \mathtt{in}(T).\mathtt{rd}(T').\emptyset$$
$$S = \langle \mathtt{t_l}[20], \mathtt{t_r}[10] \rangle$$

where T is a LINDA template matching both tuples $\mathtt{t_l}$ and $\mathtt{t_r}$, whereas T' matches $\mathtt{t_r}$ solely. Subscript p distinguishes a `ProbLinCa` primitive from a `LinCa` one; square brackets store the weight of each tuple. Let us suppose also that both processes have the following non-deterministic branching policy: branch left if consumption primitive ($\mathtt{in}_p$ and `in`) returns $\mathtt{t_l}$, branch right if consumption primitive returns $\mathtt{t_r}$—as subscript suggests.

From the *modular observable behaviour* viewpoint exploited in modular embedding (ME), P and Q are *not* distinguishable. In fact, according to any observation function Ψ defined based on [5], $\Psi[P] = \Psi[Q]$, that is, P and Q can reach the same final states:

$$\Psi[P] = (\texttt{success}, \langle \texttt{t}_\texttt{r}[10] \rangle) \text{ OR } (\texttt{deadlock}, \langle \texttt{t}_\texttt{l}[20] \rangle)$$
$$\Psi[Q] = (\texttt{success}, \langle \texttt{t}_\texttt{r}[10] \rangle) \text{ OR } (\texttt{deadlock}, \langle \texttt{t}_\texttt{l}[20] \rangle)$$

The main point here is that while P and Q are *qualitatively* equivalent, they are not *quantitatively* equivalent. Notwithstanding, by no means ME can distinguish between their behaviours: since ME cannot tell apart the probabilistic information conveyed by, e.g., a `ProbLinCa` primitive w.r.t. a `LinCa` one. For the don't know non-deterministic process Q, no *probability value* is available that could measure the chance to reach one state over the other, hence ME does not capture this property—quite obviously, since it was not meant to.

In fact, a "two-way" *modular encoding* can be trivially established between the two languages used above – say, `ProbLinCa` (`out`, $\texttt{rd}_p$, $\texttt{in}_p$) vs. `LinCa` (`out`, `rd`, `in`) – by defining compilers C as

$$C_{\texttt{LinCa}} = \begin{cases} \texttt{out} & \longmapsto \texttt{out} \\ \texttt{rd} & \longmapsto \texttt{rd}_p \\ \texttt{in} & \longmapsto \texttt{in}_p \end{cases} \qquad C_{\texttt{ProbLinCa}} = \begin{cases} \texttt{out} & \longmapsto \texttt{out} \\ \texttt{rd}_p & \longmapsto \texttt{rd} \\ \texttt{in}_p & \longmapsto \texttt{in} \end{cases}$$

and letting decoder D be compliant to the concurrent notion of observables given in [5]. Given such C and D, we can state that `ProbLinCa` *("modularly") embeds* `LinCa` and also that `LinCa` (modularly) embeds `ProbLinCa`, hence they are *(observational) equivalent* ($\equiv_\Psi$). Formally:

$$\texttt{ProbLinCa} \succeq \texttt{LinCa} \;\wedge\; \texttt{LinCa} \succeq \texttt{ProbLinCa} \quad \Longrightarrow \quad \texttt{ProbLinCa} \equiv_\Psi \texttt{LinCa}$$

However, process P *becomes* a probabilistic process due to the weighted-probability feature of $\texttt{rd}_p$, hence a probabilistic measure for P behaviour would be potentially available: however, it is not captured by ME. In the above example, for instance, such an additional bit of information would make it possible to assess that one state is "twice as probable as" the other. This is exactly the purpose of the *probabilistic modular embedding* (PME) we refine and extend in the remainder of this paper.

2.3 Formal Tools for Probability: The "Closure" Operator

In order to better ground the notion of PME as an extension of ME, and also to show its application to probabilistic coordination languages such as the aforementioned `ProbLinCa`, a proper formal framework is required.

In [11] a novel formalism is proposed that aims at dealing with the issue of open transition systems specification, requiring *quantitative information* to be attached to synchronisation actions at run-time—that is, based on the *environment* state during the computation. The idea is that of *partially closing* labelled transition systems via a process-algebraic *closure* operator ($\uparrow$), which associates quantitative values – e.g., probabilities – to admissible transition actions based upon a set of *handles* defined in an application-specific manner, dictating which quantity should be attached to which action. More precisely:

(i) actions labelling open transitions are equipped with handles;
(ii) the operator $\uparrow$ is exploited to compose a system to a specification G, associating at run-time each handle to a given value—e.g., a value $\in \mathbb{N}$;
(iii) quantitative informations with which to equip actions – e.g., probabilities $\in [0, 1]$ summing up to 1 – are computed from handle values for each enabled action, possibly based on the action context (environment);
(iv) quantitatively-labelled actions turn an open transition into a reduction, which then executes according to the quantitative information.

For instance, such operators are used as *restriction* operators in the case of `ProbLinCa` (Subsection 3.2) – in the same way as [12] for PCCS (Probabilistic Calculus of Communicating Systems) – to formally define the probabilistic interpretation of *observable actions*, informally given in Subsection 3.1, in the context of a tuple-based probabilistic language. In particular:

(i) handles coupled to actions (open transition labels) represent tuple templates associated to corresponding primitives;
(ii) handles listed in restriction term G represent tuples offered (as synchronisation items) by the tuple space;
(iii) restriction term G associates handles (tuples) to their weight in the tuple space;
(iv) restriction operator $\uparrow$ matches admissible synchronisations between processes and the tuple space, cutting out unavailable actions, and computes their associated probability distribution based upon handle-associated values.

3 Probabilistic Modular Embedding

3.1 Probabilistic Setting Requirements

In order to define the notion of *probabilistic modular embedding* (PME), we elaborate on the notion sketched in [9], starting from the informal definition of "embedding", then giving a precise characterisation to both words "easily" and "equivalently". Although the definition of "easily" given in Subsection 2.1 could be rather satisfactory in general, we prefer here to strengthen its meaning by narrowing its scope to asynchronous coordination languages and calculi: without limiting the generality of the approach, this allows us to make precise assumptions on the structure of programs.

A process can be said to be *easily* mappable into another if it requires:

(i) no extra-computations to mimic complex coordination operators;
(ii) no extra-coordinators (neither coordinated processes nor coordination medium) to handle suspensive semantics;
(iii) no unbounded extra-interactions to perform additional coordination.

Requirement *(i)* ensures absence of internal protocols in-between process-medium interactions, to emulate complex interaction primitives or behaviours—e.g., $\mathtt{in}_p$

probabilistic selection simulated by processes drawing random numbers. Requirement *(ii)* avoids proliferation of processes and media while translating a program into another, constraining such mappings to have the same number of processes and media. The last requirement complements the first in ensuring absence of complex interaction patterns to mimic complex coordination operators, such as the `in_all` global primitive as a composition of multiple `inp` (`in` predicative version)—which could be obtained by forbidding unbounded *replication* and *recursion* algebraic operators in compiler C. Altogether, the three requirements above represent a necessary constraint since our goal here is to focus on "pure coordination" expressiveness—that is, we intentionally focus on the sole expressiveness of coordination primitives, while abstracting away from the *algorithmic expressiveness* of processes and media.

The refined notion of "equivalently" is a bit more involved due to the very nature of a probabilistic process, that is, its intrinsic *randomness*. The notions of *observable behaviour* and *termination* are affected by such randomness, thus they could need to be re-casted in the probabilistic setting. Probabilistic processes, in fact, have their actions conditioned by probabilities, hence their observable transitions between reachable states are probabilistic, too—so, their execution is possible but never guaranteed. Therefore, also final states are reached by chance only, following a certain *probability path*, hence termination, too, should be equipped with its own associated probability—also in case of deadlock.

In order to address the above issues, PME improves ME by making the following properties about observable behaviour and termination available:

Probabilistic Observation. Observable actions performed by processes – e.g., `ProbLinCa` coordination primitives – should be associated with their *execution probability*. Such probability should depend on their run-time context, that is, synchronisation opportunities offered by the coordination medium. Then, compiler C should preserve transition probabilities and properly "aggregate" them along any *probabilistic trace*—that is, a sequence of probabilistic actions.

Probabilistic Termination. Final states of processes and media should be first defined as those states for which all outgoing transitions have probability 0. Furthermore, they should be refined with a *probabilistic reachability value*, that is, the probability of reaching that state from a given initial one. Then, decoder D should preserve such probabilities and determine how to compute them.

3.2 Formal Semantics

In the following, we provide a precise semantic characterisation for the PME requirements pointed out in Subsection 3.1 – that is, *probabilistic observation* and *probabilistic termination* – using `ProbLinCa` primitives as our running example, which may also be regarded as a generalisation of uniform primitives used in [9].

Probabilistic Observation. A single probabilistic observable transition step, deriving from the synchronisation between a `ProbLinCa` process and a `ProbLinCa` space – e.g. by using a $\texttt{in}_p$ –, can be formally defined as follows:

$$\texttt{in}_p(T).P \mid \langle t_1[w_1], .., t_n[w_n]\rangle \quad \overset{\mu(T,t_j)}{\longrightarrow}_{p_j} \quad P[t_j/T] \mid \langle t_1[w_1], .., t_n[w_n]\rangle \backslash t_j$$

where operator $\mu(T,t)$ denotes LINDA matching function, symbol $[\cdot/\cdot]$ stands for template substitution in process continuation, and operator $\backslash$ represents multiset difference, there expressing removal of tuple t_j from the tuple space.

By expanding such observable transition in its embedded reduction steps – that is, non-observable, silent transitions – we can precisely characterise the probabilistic semantics thanks to the $\uparrow$ operator:

$$\begin{gathered}
\texttt{in}_p(T).P \mid \langle t_1[w_1], .., t_n[w_n]\rangle \\
\xrightarrow{T} \\
\texttt{in}_p(T).P \mid \langle t_1[w_1], .., t_n[w_n]\rangle \uparrow \{(t_1, w_1), .., (t_n, w_n)\} \\
\hookrightarrow \\
\texttt{in}_p(T).P \mid \langle t_1[w_1], .., t_n[w_n]\rangle \uparrow \{(t_1, p_1), .., (t_j, p_j), .., (t_n, p_n)\} \\
\xrightarrow{t_j}_{p_j} \\
P[t_j/T] \mid \langle t_1[w_1], .., t_n[w_n]\rangle \backslash t_j
\end{gathered}$$

where $p_j = \frac{w_j}{\sum_{i=1}^{n} w_i}$ is the absolute probability of retrieving tuple t_j (with $j = 1..n$) assuming for the sake of simplicity that all tuples match template T.

The $\uparrow$ operator implicitly enforces a re-normalisation of probabilities based on available synchronisations offered by the tuple space – that is, which tuples in the space match the given template –, in the spirit of PCCS restriction operator used in [12] for the generative model of probabilistic processes. For instance, given the following *probabilistic* process P acting on `ProbLinCa` space S

$$P = \tfrac{1}{6}\texttt{in}_p(T).P + \tfrac{1}{2}\texttt{in}_p(T).P + \tfrac{1}{3}\texttt{rd}_p(T).P$$
$$S = \langle \texttt{t}_\texttt{l}[w], \texttt{t}_\texttt{r}[w]\rangle$$

where template T matches with both tuples $\texttt{t}_\texttt{l}$, $\texttt{t}_\texttt{r}$, we observe that an "experiment" $\texttt{in}_p$ could succeed with probability $\frac{2}{3} = \frac{1}{6} + \frac{1}{2}$, and that $\texttt{rd}_p$ could do so with probability $\frac{1}{3}$. Furthermore, if we suppose the branching choice after experiment $\texttt{in}_p$ to depend upon the consumed tuple, we could see the aforementioned re-normalisation by computing the probability to branch left ($\texttt{t}_\texttt{l}$ is returned), which is 0.25, and that of branching right ($\texttt{t}_\texttt{r}$ is returned), which is instead 0.75.

This addresses the first issue of probabilistic observation: we define *observable actions* as all those actions requiring synchronisation with the medium, then equip them with a probability of execution driven by available run-time synchronisation opportunities, and normalised according to the generative model interpretation enforced by $\uparrow$. Formally, we define the *probabilistic observation function* (Θ), mapping a process (W) into observables, as follows:

$$\Theta[W] = \left\{(\rho, W[\bar{\mu}]) \mid \quad (W, \langle\sigma\rangle) \longrightarrow^* (\rho, W[\bar{\mu}])\right\}$$

where ρ is a probability value $\in [0,1]$, $\bar{\mu}$ is a sequence of actual synchronisations – e.g. $\bar{\mu} = \mu(T_1, t_1), \ldots, \mu(T_n, t_n)$ – and σ is the space state—e.g. $\sigma = t_1, \ldots, t_n$.

Such definition is partial in the sense that we only know how to compute ρ for single-step transitions – that is, according to the $\uparrow$-dependent generative semantics –, in fact, it tells us nothing about how to compute ρ also for *observable traces*—that is, for sequences of observable actions. Nothing more than standard probability theory is needed here, stating that [13]:

(i) the cumulative probability of a *sequence* – that is, a *"dot"-separated* list – of probabilistic actions is the *product* of the probabilities of such actions;

(ii) the cumulative probability of a *choice* – that is, a *"+"-separated* list – of probabilistic actions is the *sum* of the probabilities of such actions.

Formally, we define the *sequence probability aggregation function* ($\bar{\nu}$) and the *choice probability aggregation function* (ν^+), mapping multiple probability values to a single one, as follows:

$$\bar{\nu} : W \times \langle\sigma\rangle \mapsto \rho \;\text{ where }\; \rho = \textstyle\prod_{j=0}^{n} \{p_j \mid (p_j, \mu_{\bar{\ell}}) \in \Theta[W = \bar{\ell}.W']\}$$
$$\nu^+ : W \times \langle\sigma\rangle \mapsto \rho \;\text{ where }\; \rho = \textstyle\sum_{j=0}^{n} \{p_j \mid (p_j, \mu_{\ell^+}) \in \Theta[W = \ell^+.W']\}$$

where $\bar{\ell}$ is a *sequence* of synchronisation actions – e.g., $\bar{\ell} = \mathtt{in}_p(T_1).\mathtt{rd}_p(T_2).\ldots$ – and ℓ^+ is a *choice* between synchronisation actions—e.g., $\bar{\ell}^+ = \mathtt{in}_p(T_1) + \mathtt{rd}_p(T_2) + \ldots$. By properly composing such aggregation functions, it is possible to compute $\Theta[W]$ for any process W and for any transition sequence $\longrightarrow^*$.

An example may help clarifying the above definitions. Let us consider the following process P and space S (sequence operator has priority w.r.t. choice):

$$P = \mathtt{in}_p(T).\big(\mathtt{rd}_p(T').P' + \mathtt{rd}_p(T').P''\big) + \mathtt{in}_p(T).P'$$
$$S = \langle \mathtt{t}_{11}[40], \mathtt{t}_{12}[30], \mathtt{t}_{r1}[20], \mathtt{t}_{r2}[10] \rangle$$

where template T may match either $\mathtt{t}_{11}$ or $\mathtt{t}_{r1}$ whereas T' may match either $\mathtt{t}_{12}$ or $\mathtt{t}_{r2}$—and branching structure is based on returned tuple as usual, that is, $\mathtt{t}_{11}$,$\mathtt{t}_{12}$ for left, $\mathtt{t}_{r1}$,$\mathtt{t}_{r2}$ for right. Applying function Θ to process P could lead to the following *observable states*:

$$\Theta[P] = (0.5, P'[\mu(T, \mathtt{t}_{11}), \mu(T', \mathtt{t}_{12})])$$
$$\Theta[P] = (0.1\bar{6}, P''[\mu(T, \mathtt{t}_{11}), \mu(T', \mathtt{t}_{r2})])$$
$$\Theta[P] = (0.\bar{3}, P'[\mu(T, \mathtt{t}_{r1})])$$

According to Θ, and using both aggregation functions $\bar{\nu}$ and ν^+, we can state that process P will eventually behave like P' – although with different substitutions – with a probability of $\simeq 0.83$, and like P'' with a probability of $\simeq 0.17$.

As a last note, one may consider that sequence probability aggregation function $\bar{\nu}$ will asymptotically tend to 0 as the length of the sequence $\bar{l}$ tends to infinity. This is unavoidable according to the basic probability theory framework adopted throughout this paper. One way to fix this aspect could be that of considering only the prefix sequence executed in loop by a process, then to associate that process not with the probability of n iterations of such loop, but with the probability of the looping prefix sequence solely—that is, with only 1 iteration of the loop. However, this concern is left for future investigation.

Probabilistic Termination. In order to define *probabilistic termination*, we should first adapt the classical notion of termination to the probabilistic setting. For this purpose, we define *ending states* as all those states for which either no more transitions are possible or all outgoing transitions have probability 0 to occur. Other than that, termination states can be enumerated as usual [5] to be $\tau = \texttt{success}, \texttt{failure}, \texttt{deadlock}$, plus the `undefined` state, which could be useful to distinguish *absorbing states* – that is, those states for which the probability of performing a self-loop transition is 1 – from deadlocks. Furthermore, such termination states have to be equipped with a *probabilistic reachability value*, according to Subsection 3.1.

Formally, we define reachability value $\rho_\perp$ and the *probabilistic termination state function* Φ as follows:

$$\Phi[W] = \left\{ (\rho_\perp, \tau) \mid \quad (W, \langle\sigma\rangle) \longrightarrow_\perp^* (\rho_\perp, \tau) \right\}$$

where subscript $\perp$ means a sequence of finite transitions leading to termination τ. By comparing this function with the observation function Θ, it can be noticed that Φ abstracts away from computation *traces* – that is, it does not keep track of synchronisations, hence substitutions, in term $W[\bar{\mu}]$ – focussing solely on termination states τ. However, when computing the value of $\rho_\perp$, the same aggregation functions $\bar{\nu}$ and ν^+ have to be used.

For instance, recalling process P used to test observation function Θ, changing space S configuration as follows:

$$P = \texttt{in}_p(T).\big(\texttt{rd}_p(T').P' + \texttt{rd}_p(T').P''\big) + \texttt{in}_p(T).P'$$
$$S = \langle \texttt{t}_{11}[40], \texttt{t}_{\texttt{r}1}[20] \rangle$$

where $P' \equiv P'' = \emptyset$ so to reach termination, the application of the probabilistic termination state function just defined would lead to the following *observable termination states*:

$$\Phi[P] = (0.\bar{6}, \texttt{deadlock}) \quad \vee \quad \Phi[P] = (0.\bar{3}, \texttt{success})$$

In particular, P deadlocks with probability $\frac{2}{3}$ if tuple $\texttt{t}_{11}$ is consumed, whereas succeeds with probability $\frac{1}{3}$ if tuple $\texttt{t}_{\texttt{r}1}$ is consumed in its stead.

Please notice that, unlike the case of endless sequences $\bar{l}$ highlighted at the end of Subsection 3.2, absorbing states cause no harm for our sequence probability aggregation function $\bar{\nu}$, since the probability value aggregated until reaching the absorbing state will be from now on always multiplied by 1—in the very end, making each iteration of the self-loop indistinguishable from the others.

4 PME vs. ME: Testing Expressiveness on Case Studies

`ProbLinCa` **vs.** `LinCa` We now recall the two processes P and Q acting on space S introduced in the example of Subsection 2.2:

$$P = \texttt{in}_p(T).\emptyset + \texttt{in}_p(T).\texttt{rd}_p(T').\emptyset \quad Q = \texttt{in}(T).\emptyset + \texttt{in}(T).\texttt{rd}(T').\emptyset$$
$$S = \langle \texttt{t}_\texttt{l}[20], \texttt{t}_\texttt{r}[10] \rangle$$

to repeat the embedding observation, this time under the assumptions of PME. As expected, we can now distinguish the *behaviour* of process P from that of process Q. In fact, by applying function Φ to both P and Q we get:

$$\Phi[P] = (0.\bar{6}, \texttt{success}) \text{ OR } (0.\bar{3}, \texttt{deadlock})$$
$$\Phi[Q] = (\bullet, \texttt{success}) \text{ OR } (\bullet, \texttt{deadlock})$$

where symbol $\bullet$ denotes "absence of information".

Therefore, only a "one-way" *encoding* can be now established between the two languages used above – again, `ProbLinCa` (`out`, $\texttt{rd}_p$, $\texttt{in}_p$) vs. `LinCa` (`out`, `rd`, `in`) – by defining compiler $C_{\texttt{LinCa}}$ as

$$C_{\texttt{LinCa}} = \begin{cases} \texttt{out} & \longmapsto & \texttt{out} \\ \texttt{rd} & \longmapsto & \texttt{rd}_p \\ \texttt{in} & \longmapsto & \texttt{in}_p \end{cases}$$

and making decoder D rely on observation function Θ and termination function Φ. Then we can state that `ProbLinCa` *probabilistically embeds* ($\succeq_p$) `LinCa`—but not the other way around. Formally, according to PME:

$$\texttt{ProbLinCa} \succeq_p \texttt{LinCa} \;\wedge\; \texttt{LinCa} \not\succeq_p \texttt{ProbLinCa} \quad\Longrightarrow\quad \texttt{ProbLinCa} \not\equiv_p \texttt{LinCa}$$

In the end, PME succeeds in telling `ProbLinCa` apart from `LinCa` (classifying `ProbLinCa` as *more expressive* than `LinCa`), whereas ME fails.

pKLAIM vs. KLAIM pKLAIM was introduced in [14] as a probabilistic extension to KLAIM [15], a kernel programming language for mobile computing. In KLAIM, processes as well as data can be moved across the network among computing environments: in fact, it features *(i)* a core LINDA with multiple tuple spaces, and *(ii) localities* as first-class abstractions to explicitly manage mobility and distribution-related aspects. pKLAIM extends such model by introducing probabilities in a number of different ways and according to the two levels of KLAIM formal semantics: *local* and *network* semantics. We here consider local semantics solely, because the network one can quickly become cumbersome, due to multiple probability normalisation steps [15], and is unnecessary for the purpose of showing the power of the PME approach.

The local semantics defines how a number of co-located processes interact with a tuple space, either local or remote. Here, probabilities are given by:

- a probabilistic choice operator $+_{i=1}^{n} p_i : P_i$;
- a probabilistic parallel operator $|_{i=1}^{n}\ p_i : P_i$;
- *probabilistic allocation environments*, formally defined as a partial map $\sigma : \mathit{Loc} \mapsto \mathit{Dist}(S)$ associating probability distributions on physical sites (S) to logical localities (Loc).

For the sake of clarity (and brevity), we consider the three probabilistic extensions introduced by pKLAIM separately—their combination is a trivial extension once that normalisation procedures [15] are accounted for.

First of all, we focus on the probabilistic choice operator. Let us suppose pKLAIM process P and KLAIM process Q are interacting with space s—refer to [14] for process syntax:

$$P = \tfrac{2}{3}\mathtt{in}(T)@s.\emptyset + \tfrac{1}{3}\mathtt{in}(T)@s.\mathtt{rd}(T)@s.\emptyset$$
$$Q = \mathtt{in}(T)@s.\emptyset + \mathtt{in}(T)@s.\mathtt{rd}(T)@s.\emptyset$$
$$s = \mathtt{out}(\mathtt{t})@\mathtt{self}.\emptyset \quad \equiv \quad s = \langle \mathtt{t} \rangle$$

where T matches the single tuple $\mathtt{t}$ and KLAIM notation for tuple space s is equivalent to our usual notation.

Both processes have a non-deterministic branching structure which cannot be distinguished by ME. In fact, according to any observation function Ψ defined based on [5], $\Psi[P] = \Psi[Q]$, that is, P and Q can reach the same final states:

$$\Psi[P] = (\mathtt{success}, \langle\,\rangle) \text{ OR } (\mathtt{deadlock}, \langle\,\rangle)$$
$$\Psi[Q] = (\mathtt{success}, \langle\,\rangle) \text{ OR } (\mathtt{deadlock}, \langle\,\rangle)$$

PME is instead sensitive to the probabilistic information available for pKLAIM process P, hence by applying the Φ function we get:

$$\Phi[P] = (0.\bar{6}, \mathtt{success}) \text{ OR } (0.\bar{3}, \mathtt{deadlock})$$
$$\Phi[Q] = (\bullet, \mathtt{success}) \text{ OR } (\bullet, \mathtt{deadlock})$$

Since the probabilistic parallel operator can be handled (almost) identically, we step onwards to the probabilistic allocation operator. Suppose, then, to be in the following network configuration:

$$P = \mathtt{in}(T)@l.\emptyset \quad Q = \mathtt{in}(T)@l.\emptyset$$
$$s_1 = \langle \mathtt{t} \rangle \quad s_2 = \langle\,\rangle \quad \sigma : l \mapsto \begin{cases} \frac{2}{3}s_1 \\ \frac{1}{3}s_2 \end{cases}$$

where function σ is defined only for the pKLAIM process P—whereas for KLAIM process Q the allocation function is unknown (e.g., implementation-dependent). Thus, the processes have is branching structure and are actually identical. However, the coexistence of two admissible allocation environments (s_1 and s_2) may impact the termination states of P and Q. In fact, by applying any observation function suitable to ME criteria [5], we get the following final states:

$$\Psi[P] = (\mathtt{success}, s_1 = \langle\,\rangle \wedge s_2 = \langle\,\rangle) \text{ OR } (\mathtt{deadlock}, s_1 = \langle \mathtt{t} \rangle \wedge s_2 = \langle\,\rangle)$$
$$\Psi[Q] = (\mathtt{success}, s_1 = \langle\,\rangle \wedge s_2 = \langle\,\rangle) \text{ OR } (\mathtt{deadlock}, s_1 = \langle \mathtt{t} \rangle \wedge s_2 = \langle\,\rangle)$$

Nevertheless, final states are the same for both P and Q, which cannot be distinguished. This happens because ME is *insensitive* to the probabilistic allocation function σ, having no ways to account for it.

PME provides instead "probability-sensitive" observation/termination functions, whose application produces the following final states:

$$\Phi[P] = (0.\bar{6}, \mathtt{success}) \text{ OR } (0.\bar{3}, \mathtt{deadlock})$$
$$\Phi[Q] = (\bullet, \mathtt{success}) \text{ OR } (\bullet, \mathtt{deadlock})$$

Unlike ME, PME tells pKLAIM process P apart from KLAIM process Q. Since the difference between P and Q is quantitative, not qualitative, only a quantitative embedding such as PME can successfully distinguish between the two.

***π_{pa}-calculus* vs. *π_a-calculus*.** The *π_{pa}-calculus* was introduced in [16] as a probabilistic extension to the *π_a-calculus* (asynchronous *π-calculus*) defined in [17]. In order to increase the expressive power of *π_a-calculus*, the authors propose a novel calculus, featuring a probabilistic guarded choice operator ($\sum_i p_i \alpha_i.P_i$), able to distinguish between probabilistic and purely nondeterministic behaviours. Whereas the former is due to a random choice performed by the process itself, the latter is associated to the arbitrary decisions made by an external process (scheduler).

Let us consider the following processes P and Q willing to synchronise with process S (playing the role of the tuple space)—refer to [16] for the syntax used:

$$P = \left(\tfrac{2}{3}x(y) + \tfrac{1}{3}z(y)\right).\emptyset \qquad Q = \left(x(y) + z(y)\right).\emptyset$$
$$S = \bar{x}y \quad \equiv \quad S = \{S_x = \langle y \rangle \bigcup S_z = \langle\,\rangle\}$$

where the last equivalence just aims at providing a uniform notation compared to the other case studies proposed—in particular, *π-calculus* channels can resemble KLAIM allocation environments.

For both P and Q, two are the admissible termination states, listed below as a result of the application of ME:

$$\Psi[P] = (\texttt{success}, \langle\,\rangle) \text{ OR } (\texttt{deadlock}, \langle y \rangle)$$
$$\Psi[Q] = (\texttt{success}, \langle\,\rangle) \text{ OR } (\texttt{deadlock}, \langle y \rangle)$$

As expected, they are indistinguishable despite the probabilistic information available for P, lost by the ME observation function.

As in previous cases, PME fills the gap:

$$\Phi[P] = (0.\bar{6}, \texttt{success}) \text{ OR } (0.\bar{3}, \texttt{deadlock})$$
$$\Phi[Q] = (\bullet, \texttt{success}) \text{ OR } (\bullet, \texttt{deadlock})$$

5 Related Works

One of the starting points of our work is the observation that modular embedding (ME) as defined in [5] does not suit probabilistic scenarios, since it does not consider probabilistic transitions—therefore probabilistic termination. In other words, when applied to a probabilistic process, ME can just point out its reachable termination states and the admissible transitions, and cannot say anything about their *quantitative* aspects—that is, probability of execution (transitions) and reachability (end states). Thus, in this paper we discussed how probabilistic modular embedding (PME) extends ME towards probabilistic models and languages, such as those defined in [12].

To the best of our knowledge, no other researches push the work by De Boer and Palamidessi [5] towards a probabilistic setting with a focus on coordination languages, in the same way as no other works try to connect the concept of "language embedding" with any of the probability models defined in [12].

In [18], the authors try to answer questions such as how to formalise probabilistic transition system, and how to extend non-probabilistic process algebras operators to the probabilistic setting. In particular, they focus on reactive models of probability – hence, models where pure non-determinism and probability coexist – and provide the notions of *probabilistic bisimulation*, *probabilistic simulation* (the asymmetric version of bisimulation), and *probabilistic testing preorders* (testing-based observation of equivalence), again applied to PCCS. Although targeted to the PCCS reactive model, their work is related to ours in the attempt to find a way to *compare* the relative expressiveness of different probabilistic languages. On the other hand, our approach is quite different because we adopted a *language embedding* perspective rather than a *process bisimulation* viewpoint. Whereas probabilistic bisimulation can prove the *observational equivalence* of different probabilistic models, it cannot detect which is the most expressive among them. However, we cannot exclude that a "two-way" probabilistic embedding relationship may correspond to a probabilistic bisimulation according to [18] definition of bisimulation—at least for reactive models.

In [19] the notion of *linear embedding* is introduced. Starting from the definition of ME in [5], the authors aim at *quantifying* "how much a language embeds another one", that is, "how much a given language is more expressive than another". To do so, they *(i)* take *linear vector spaces* as a semantic domain for a subset of LINDA-like languages – that is, considering `tell, get, ask, nask` primitives –; *(ii)* define an observation criteria associating to each program a linear algebra operator acting on such vector spaces; then *(iii)* quantify the difference in expressive power by computing the *dimension* of the linear algebras associated to each language. Although the possibility to *quantify* the relative expressive power of a set of languages is appealing, the work in [19] do consider neither probabilistic languages nor probabilistic processes, hence cannot be directly compared to ours. However, it still remains an interesting path to follow for further developments of the probabilistic embedding here proposed.

Last but not least, in [20] the authors apply the *Probabilistic Abstract Interpretation* (PAI) theory and its techniques to probabilistic transition systems, in order to formally define the notion of *approximate process equivalence*—that is, two probabilistic processes are equivalent "up to an error ε". As in [19], Di Pierro, Hankin, and Wiklicky adopt linear algebras to represent some semantical domain, but they consider probabilistic transition systems instead of deterministic ones. Therefore, they allow matrices representing algebraic operators to specify probability values $v \in [0, 1]$ instead of binary values $b = 0 \mid 1$. Then, by using the PAI framework and drawing inspiration from *statistical testing* approaches, they define the notion of *ε-bisimilarity*, which allows the minimum number of tests needed to accept the bisimilarity relation between two processes to be quantified with a given confidence. By examining such a number, a quantitative idea of the

statistical distance between two given sets of (processes) admissible behaviours can be inferred. Although quite different from ours, this work can be considered nevertheless as another opportunity for further improvement of PME: for instance, an enhanced version of PME may be able to detect some notion of approximate process equivalence.

6 Conclusion and Future Works

Starting from the notions of embedding and modular embedding, in this paper we refine and extend the definition of *probabilistic modular embedding* (PME) first sketched in [9], as a tool for modelling the expressiveness of concurrent languages and systems, in particular those featuring probabilistic mechanisms and exhibiting stochastic behaviours. We discuss its novelty with respect to the existing approaches in the literature, then show how PME succeeds in telling apart probabilistic languages from non-probabilistic ones, whereas standard ME fails. While apparently trivial, such a distinction was not possible with any other formal framework in the literature so far, to the best of our knowledge.

Furthermore, PME has seemingly the potential to compare the expressiveness of two probabilistic languages: for instance, the ability of PME of telling apart the different probabilistic processes models proposed in [12] is currently under investigation.

Acknowledgments. This work has been partially supported by the EU-FP7-FET Proactive project SAPERE – Self-aware Pervasive Service Ecosystems, under contract no. 256874.

References

1. Wegner, P.: Why interaction is more powerful than algorithms. Communications of the ACM 40(5), 80–91 (1997)
2. 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 1998), February 27-March 1, pp. 169–177. ACM, Atlanta (1998); Special Track on Coordination Models, Languages and Applications
3. Busi, N., Gorrieri, R., Zavattaro, G.: On the expressiveness of Linda coordination primitives. Information and Computation 156(1-2), 90–121 (2000)
4. Wegner, P., Goldin, D.: Computation beyond Turing machines. Communications of the ACM 46(4), 100–102 (2003)
5. de Boer, F.S., Palamidessi, C.: Embedding as a tool for language comparison. Information and Computation 108(1), 128–157 (1994)
6. Shapiro, E.: Separating concurrent languages with categories of language embeddings. In: 23rd Annual ACM Symposium on Theory of Computing (1991)
7. Omicini, A., Viroli, M.: Coordination models and languages: From parallel computing to self-organisation. The Knowledge Engineering Review 26(1), 53–59 (2011); Special Issue 01 (25th Anniversary Issue)

8. Omicini, A.: Nature-inspired coordination models: Current status, future trends. ISRN Software Engineering 2013, Article ID 384903, Review Article (2013)
9. Mariani, S., Omicini, A.: Probabilistic embedding: Experiments with tuple-based probabilistic languages. In: 28th ACM Symposium on Applied Computing (SAC 2013), Coimbra, Portugal, March 18-22, pp. 1380–1382 (2013) (Poster Paper)
10. Bravetti, M., Gorrieri, R., Lucchi, R., Zavattaro, G.: Quantitative information in the tuple space coordination model. Theoretical Computer Science 346(1), 28–57 (2005)
11. Bravetti, M.: Expressing priorities and external probabilities in process algebra via mixed open/closed systems. Electronic Notes in Theoretical Computer Science 194(2), 31–57 (2008)
12. van Glabbeek, R.J., Smolka, S.A., Steffen, B.: Reactive, generative, and stratified models of probabilistic processes. Information and Computation 121(1), 59–80 (1995)
13. Drake, A.W.: Fundamentals of Applied Probability Theory. McGraw-Hill College (1967)
14. Di Pierro, A., Hankin, C., Wiklicky, H.: Probabilistic KLAIM. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 119–134. Springer, Heidelberg (2004)
15. De Nicola, R., Ferrari, G., Pugliese, R.: KLAIM: A kernel language for agent interaction and mobility. IEEE Transaction on Software Engineering 24(5), 315–330 (1998)
16. Herescu, O.M., Palamidessi, C.: Probabilistic asynchronous pi-calculus. CoRR cs.PL/0109002 (2001)
17. Boudol, G.: Asynchrony and the Pi-calculus. Rapport de recherche RR-1702, INRIA (1992)
18. Bengt, J., Larsen, K.G., Yi, W.: Probabilistic extensions of process algebras. In Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra. Elsevier Science B.V., pp. 685–710 (2001)
19. Brogi, A., Di Pierro, A., Wiklicky, H.: Linear embedding for a quantitative comparison of language expressiveness. Electronic Notes in Theoretical Computer Science 59(3), 207–237 (2002); Quantitative Aspects of Programming Languages (QAPL 2001 @ PLI 2001)
20. Di Pierro, A., Hankin, C., Wiklicky, H.: Quantitative relations and approximate process equivalences. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 508–522. Springer, Heidelberg (2003)

ByteSTM: Virtual Machine-Level Java Software Transactional Memory

Mohamed Mohamedin, Binoy Ravindran, and Roberto Palmieri

ECE Dept., Virginia Tech, Blacksburg, VA, USA
{mohamedin,binoy,robertop}@vt.edu

Abstract. We present ByteSTM, a virtual machine-level Java STM implementation that is built by extending the Jikes RVM. We modify Jikes RVM's optimizing compiler to transparently support implicit transactions. Being implemented at the VM-level, it accesses memory directly, avoids Java garbage collection overhead by manually managing memory for transactional metadata, and provides pluggable support for implementing different STM algorithms to the VM. Our experimental studies reveal throughput improvement over other non-VM STMs by 6–70% on micro-benchmarks and by 7–60% on macro-benchmarks.

1 Introduction

Transactional Memory (TM) [13] is an attractive programming model for multicore architectures promising to help the programmer in the implementation of parallel and concurrent applications. The developer can focus the effort on the implementation of the application's business logic, giving the responsibility to managing concurrency to the TM framework. It ensures, in a completely transparent manner, properties difficult to implement manually like atomicity, consistency, deadlock-freedom and livelock-freedom. The programmer simply organizes the code identifying blocks to be executed as transactions (so called atomic blocks). TM overtakes the classical coarse grain lock-based implementation of concurrency, executing transactions optimistically and logging into a private part of the memory the results of read and write operations performed during the execution (respectively on the read- and write-set). Further TM is composable.

TM has been proposed in hardware (HTM; e.g., [8]), in software (STM; e.g., [17]), and in combination (HybridTM; e.g., [18]). HTM has the lowest overhead, but transactions are limited in space and time. STM does not have such limitations, but has higher overhead. HybridTM avoids these limitations.

Given STM's hardware-independence, which is a compelling advantage, we focus on STM. STM implementations can be classified into three categories: *library-based*, *compiler-based*, and *virtual machine-based*. Library-based STMs add transactional support without changing the underlying language, and can be classified into those that use *explicit* transactions [14,21,26] and those that use *implicit* transactions [5, 17, 23]. Explicit transactions are difficult to use. They

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 166–180, 2013.

support only transactional objects (i.e., objects that are known to the STM library by implementing a specific interface, being annotated, etc) and hence cannot work with external libraries. Implicit transactions, on the other hand, use modern language features (e.g., Java annotations) to mark code sections as `atomic`. Instrumentation is used to transparently add transactional code to the atomic sections (e.g., begin, transactional reads/writes, commit). Some implicit transactions work only with transactional objects [5, 23], while others work on any object and support external libraries [17].

Compiler-based STMs (e.g., [15, 16]) support implicit transactions transparently by adding new language constructs (e.g., `atomic`). The compiler then generates transactional code that calls the underlying STM library. Compiler-based STMs can optimize the generated code and do overall program optimization. On the other hand, compilers need external libraries' source code to instrument it and add transactional support. Usually, external libraries source code is not available. With managed run-time languages, compilers alone do not have full control over the VM. Thus, the generated code will not be optimized and may contradict with some of the VM features like the garbage collector (GC).

VM-based STMs, which have been less studied, include [1,7,12,27]. [12] is implemented in C inside the JVM to get benefits of the VM-managed environment, and uses an algorithm that does not ensure the opacity correctness property [11]. This means that inconsistent reads may occur before a transaction is aborted, causing unrecoverable errors in an unmanaged environment. Thus, the VM-level implementation choice is to prevent such unrecoverable errors, which are not allowed in a managed environment. [7] presented a new Java-based programming language called Atomos, and a concomitant VM where standard Java synchronization (i.e., `synchronized`, `wait/notify`) is replaced with transactions. However, in this work, transactional support relies on HTM.

Library-based STMs are largely based on the premise that it is better not to modify the VM or the compiler, to promote flexibility, backward compatibility with legacy code, and easiness to deploy and use. However, this premise is increasingly violated as many require some VM support or are being directly integrated into the language and thus the VM. Most STM libraries are based on annotations and instrumentation, which are new features in the Java language. For example, the Deuce STM library [17] uses a non-standard proprietary API (i.e., `sun.misc.Unsafe`) for performance gains, which is incompatible with other JVMs. Moreover, programming languages routinely add new features in their evolution for a whole host of reasons. Thus, as STM gains traction, it is natural that it will be integrated into the language and the VM.

Implementing STM at the VM-level allows many opportunities for optimization and adding new features. For example, the VM has direct access to memory, which allows faster write-backs to memory. The VM also has full control of the GC, which allows minimizing the GC's degrading effect on STM performance (see Subsection 2.5). Moreover, if TM is supported using hardware (as in [7]), then VM is the only appropriate level of abstraction to exploit that support for obtaining higher performance. (Otherwise, if TM is supported at a higher level, the GC will abort transactions when it interrupts them.) Also, VM memory

Table 1. Comparison of Java STM implementations

Feature	Deuce [17]	JVSTM [5]	ObjectFabric [21]	AtomJava [15]	DSTM2 [14]	Multiverse [26]	LSA-STM [23]	Harris & Fraser [12]	Atomos[1] [7]	Trans. monitors [27]	ByteSTM
Implicit transactions	√	√	X	√	X	X	√	√	√	√	√
All data types	√	X	X	√	X	X	X	√	√	√	√
External libraries	√	X	X	√ [2]	X	X	X	√	X [3]	√	√
Unrestricted atomic blocks	X	X	√	√	√	√	X	√	√	√	√
Direct memory access	√[4]	X	X	X	X	X	X	√	√	X	√
Field-based granularity	√	X	X	X	X	X	X	X	X	X	√
No GC overhead	√ [5]	X	X	X	X	X	X	√	√	X	√
Compiler support	X	X	X	√	X	X	X	√	√	√	√ & X [6]
Strong atomicity	X	X	√	√	X	X	X	X	√	X	X
Closed/Open nesting	X	√	√	X	X	X	X	X	√	X	X
Conditional variables	X	X	X	X	X	X	X	X	√	X	X

[1] Is a HybridTM, but software part is implemented inside a VM.
[2] Only if source code is available.
[3] It is a new language, thus no Java code is supported.
[4] Uses non-standard library.
[5] Uses object pooling, which partially solves the problem.
[6] ByteSTM can work with or without compiler support.

systems typically use a centralized data structure, which increases conflicts, degrading performance [4].

Motivated by these observations, we design and implement a VM-level STM: ByteSTM. In ByteSTM, a transaction can surround any block of code, and it is not restricted to methods. Memory bytecode instructions reachable from a transaction are translated so that the resulting native code executes transactionally. ByteSTM uniformly handles all data types (not just transactional objects), using the memory address and number of bytes as an abstraction, and thereby supports external libraries. It uses field-based granularity, which scales better than object-based or word-based granularity, and eliminates the GC overhead, by manually managing (e.g., allocation, deallocation) memory for transactional metadata. It can work without compiler support, which is only required if the new language construct `atomic` is used. ByteSTM has a modular architecture, which allows different STM algorithms to be easily plugged in (we have implemented three algorithms: TL2 [10], RingSTM [25], and NOrec [9]). Table 1 distinguishes ByteSTM from other STM implementations. The current release of ByteSTM does not support closed/open nesting, strong atomicity, or conditional variables. However, features like these are in the implementation roadmap.

ByteSTM is open-sourced and is freely available at `hydravm.org/bytestm`.

2 Design and Implementation

ByteSTM is built by modifying Jikes RVM [2] using the optimizing compiler. Jikes RVM is a Java research virtual machine and it is implemented in Java. Jikes RVM has two types of compilers: the Optimizing compiler and the Baseline compiler. The Baseline compiler simulates the Java stack machine and has no optimization. The Optimizing compiler does several optimizations (e.g., register allocation, inlining, code reordering). Jikes RVM has no interpreter, and bytecode must be compiled to native code before execution. Building the Jikes RVM with production configuration gives performance comparable to the HotSpot server JIT compiler [22].

In ByteSTM, bytecode instructions run in two modes: transactional and non-transactional. The visible modifications to VM users are very limited: two new instructions (`xBegin` and `xCommit`) are added to the VM bytecode instructions. These two instructions will need compiler modifications to generate the correct bytecode when `atomic` blocks are translated. Also, the compiler should handle the new keyword `atomic` correctly. To eliminate the need for a modified compiler, a simpler workaround is used: the method `stm.STM.xBegin()` is used to begin a transaction and `stm.STM.xCommit()` is used to commit the transaction. The two methods are defined empty and static in class `STM` in `stm` package.

ByteSTM is implicitly transactional: the program only specifies the start and end of a transaction and all memory operations (loads and stores) inside these boundaries are implicitly transactional. This simplifies the code inside the atomic block and also eliminates the need for making a transactional version for each memory load/store instruction, thereby keeping the number of added instructions minimal. When `xBegin` is executed, the thread enters the transactional mode. In this mode, all writes are isolated and the execution of the instructions proceeds optimistically until `xCommit` is executed. At that point, the transaction is compared against other concurrent transactions for a conflict. If there is no conflict, the transaction is allowed to commit and, only at this point, all transaction modifications become externally visible to other transactions. If the commit fails, all the modifications are discarded and the transaction restarts from the beginning.

We modified the Jikes optimizing compiler. Each memory load/store instruction (`getfield`, `putfield`, `getstatic`, `putstatic`, and all array access instructions) is replaced with a call to a corresponding method that adds the transactional behavior to it. The compiler inlines these methods to eliminate the overhead of calling a method with each memory load/store. The resulting behavior is that each instruction checks whether the thread is running in transactional or non-transactional mode. Thus, instruction execution continues transactionally or non-transactionally. The technique is used to translate the new instructions `xBegin` and `xCommit` (or replacing calls to `stm.STM.xBegin()` and `stm.STM.xCommit()` with the correct method calls).

Modern STMs [5,17,23] use automatic instrumentation. Java annotations are used to mark methods as `atomic`. The instrumentation engine then handles all code inside atomic methods and modifies them to run as transactions. This

conversion does not need the source code and can be done offline or online. Instrumentation allows using external libraries – i.e., code inside a transaction can call methods from an external library, which may modify program data [17].

In ByteSTM, code that is reachable from within a transaction is compiled to native code with transactional support. Classes/packages that will be accessed transactionally are input to the VM by specifying them on the command line. Then, each memory operation in these classes is translated by first checking the thread's mode. If the mode is transactional, the thread runs transactionally; otherwise, it runs regularly. Although doing such a check with every memory load/store operation increases overhead, our results show significant throughput improvement over competitor STMs (see Section 3).

Atomic blocks can be used anywhere (excluding blocks containing irrevocable operations such as I/O). It is not necessary to make a whole method atomic; any block can be atomic. External libraries can be used inside transactions without any change.

Memory access is monitored at the field level, and not at the object level. Field-based granularity scales well and eliminates false conflicts resulting from two transactions changing different fields of the same object [17].

2.1 Metadata

Working at the VM level allows changing the thread header without modifying program code. For each thread that executes transactions, the metadata added includes the read-set, the write-set, and other STM algorithm-specific metadata. Metadata is added to the thread header and is used by all transactions executed in the thread. Since each thread executes one transaction at a time, there is no need to create new data for each transaction, allowing reuse of the metadata. Also, accessing a thread's header is faster than Java's `ThreadLocal` abstraction.

2.2 Memory Model

At the VM-level, the physical memory address of each object's field can be easily obtained. Since ByteSTM is field-based, the address of each field is used to track memory reads and writes. A conflict occurs only if two transactions modified the same field of an object. Since arrays are objects in Java, memory accesses to arrays are tracked at the element level, which eliminates unnecessary aborts.

An object instance's field's absolute address equals the object's base address plus the field's offset. A static object's field's absolute address equals the global static memory space's address plus the field's offset. Finally, an array's element's absolute address equals the array's address plus the element's index in the array (multiplied by the element's size). Thus, our memory model is simplified as: base object plus an offset for all cases.

Using absolute addresses is limited to non-moving GC only (i.e., a GC which releases unreachable objects without moving reachable objects, like the mark-and-sweep GC). In order to support moving GC, a field is represented by its base object and the field's offset within that object. When the GC moves an object,

only the base object's address is changed. All offsets remain the same. ByteSTM's write-set is part of the GC root-set. Thus, the GC automatically changes the saved base objects' addresses as part of its reference updating phase.

To simplify how the read-set and the write-set are handled, we use a unified memory access scheme. At a memory load, the information needed to track the read includes the base object and the offset within that object of the read field. At a memory store, the base object, the field's offset, the new value, and the size of the value are the information used to track the write. When data is written back to memory, the write-set information (base object, offset, value, and length of the location) is used to store the committed values correctly. This abstraction also simplifies the code, as there is now no need to differentiate between different data types, as they are all handled as a sequence of bytes in the memory. The result is simplified code that handles all the data types the same, yielding faster execution.

2.3 Write-Set Representation

We found that using a complex data structure to represent read-sets and write-sets affects performance. Given the simplified raw memory abstraction used in ByteSTM, we decided to use simple arrays of primitive data types. This decision is based on two reasons. First, array access is very fast and has access locality, resulting in better cache usage. Second, with primitive data types, there is no need to allocate a new object for each element in the read/write set. (Recall that an array of objects is allocated as an array of references in Java, and each object needs to be allocated separately. Hence, there is a large overhead for allocating memory for each array element.) Even if object pooling is used, the memory will not be contiguous since each object is allocated independently in the heap.

Using arrays to represent the write-set means that the cost of searching an n-element write-set is $O(n)$. To obtain the benefits of arrays and hashing's speed, open-addressing hashing with linear probing is used. We used an array of size 2^n, which simplifies the modulus calculation.

We used Java's `System.identityHashCode` standard method and configured Jikes to use the memory address to compute an object's hash code. This method also handles object moving. We then add the field's offset to the returned hash code, and finally remove the upper bits from the result using bitwise *and* operation (which is equivalent to calculating the modulus): *address AND mask = address MOD arraySize*, where *mask = arraySize* - 1. For example, if *arraySize = 256*, then *hash(address) = address AND 0xFF*. This hashing function is efficient with addresses, as the collision ratio is small. Moreover, as program variables are aligned in memory, when a collision happens, there is always an empty cell after the required index because of the memory alignment gap (so linear probing will give good results). This way, we have a fast and efficient hashing function that adds little overhead to each array access, enabling $O(1)$-time searching and adding operations on large write-sets.

Iterating over the write-set elements by cycling through the sparse array elements is not efficient. We solve this by keeping a contiguous log of all the used indices, and then iterating on the small contiguous log entries.

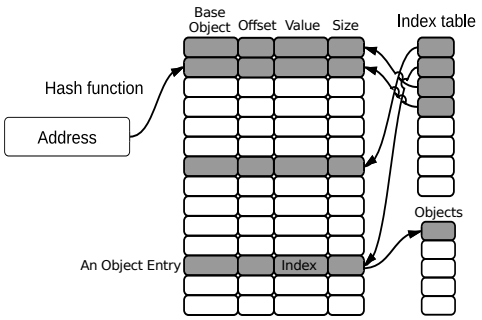


Fig. 1. ByteSTM's write-set using open address hashing

Open addressing has two drawbacks: memory overhead and rehashing. These can be mitigated by choosing the array size such that the number of rehashing is reduced, while minimizing memory usage. Figure 1 shows how ByteSTM's write-set is represented using open-addressing. In this figure, the field's memory address is hashed to determine the associated index in the array. Four entries are occupied in the shown array (identified by gray color) and they are scattered by the hashing function into different locations in the array. The index table contains all used indices in the main array contiguously for faster sequential access to the write-set entries. Reference fields (object entries) are handled differently as described in Subsection 2.5.

2.4 Atomic Blocks

ByteSTM supports atomic blocks anywhere in the code, excluding I/O operations and JNI native calls. When `xBegin` is executed, local variables are backed up. If a transaction is aborted, the variables are restored and the transaction can restart as if nothing has changed in the variables. This technique simplifies the handling of local variables since there is no need to monitor them.

ByteSTM has been designed to natively support opacity [11]. In fact, when an inconsistent read is detected in a transaction, the transaction is immediately aborted. Local variables are then restored, and the transaction is restarted by throwing an exception. The exception is caught just before the end of the transaction loop so that the loop continues again. Note that throwing an exception is not expensive if the exception object is preallocated. Preallocating the exception object eliminates the overhead of creating the stack trace every time the exception is thrown. Stack trace is not required for this exception object since it is used only for doing a long jump. The result is similar to `setjmp/longjmp` in C.

2.5 Garbage Collector

One major drawback of building an STM for Java (or any managed language) is the GC [19]. STM uses metadata to keep track of transactional reads and writes.

This requires allocating memory for the metadata and then releasing it when not needed. Frequent memory allocation (and implicit deallocation) forces the GC to run more frequently to release unused memory, increasing STM overhead.

Some STMs solve this problem by reducing memory allocation and recycling allocated memory. For example, [17] uses object pooling, wherein objects are allocated from, and recycled back to a pool of objects (with the heap used when the pool is exhausted). However, allocation is still done through the Java memory system and the GC checks if the pooled objects are still referenced.

Since ByteSTM is integrated into the VM, its memory allocation and recycling is done outside the control of the Java memory system: memory is directly allocated and recycled. STM's memory requirement, in general, has a specific lifetime. When a transaction starts, it requires a specific amount of metadata, which remain active for the transaction's duration. When the transaction commits, the metadata is recycled. Thus, manual memory management does not increase the complexity or overhead of the implementation.

The GC causes another problem for ByteSTM, however. ByteSTM stores intermediate changes in a write buffer. Thus, the program's newly allocated objects will not be referenced by the program's variables. The GC scans only the program's stack to find objects that are no longer referenced. Hence, it will not find any reference to the newly allocated objects and will recycle their memory. When ByteSTM commits a transaction, it will therefore be writing a dangling pointer. We solve this problem by modifying the behavior of adding an object to the write-set. Instead of storing the object address in the write-set entry value, the object is added to another array (i.e., an "objects array"). The object's index in the objects array is stored in the write-set entry value (Figure 1). Specifically, if an object contains another object (e.g., a field that is a reference), we cannot save the field value as a primitive type (e.g., the absolute address) since the address can be changed by the GC. The field value is therefore saved as an object in the objects array which is available to the set of roots that the GC scans. The write-set array is another source of roots. So, the write-set contains the base objects and the objects array contains the object fields within the base objects. This prevents the GC from reclaiming the objects. Our approach is compatible with all GC available in Jikes RVM and we believe that this approach is better than modifying a specific GC.

2.6 STM Algorithms

ByteSTM's modular architecture allows STM algorithms to be easily "plugged in." We implemented three algorithms: TL2 [10], RingSTM [25], and NOrec [9]. Our rationale for selecting these three is that, they are the best performing algorithms reported in the literature. Additionally, they cover different points in the performance/workload tradeoff space: TL2 is effective for long transactions, moderate number of reads, and scales well with large number of writes. RingSTM is effective for transactions with high number of reads and small number of writes. NOrec has a better performance with small number of cores and has no false conflicts since it validates by value.

Plugging a new algorithm in ByteSTM is straight forward. One needs to implement read barriers, write barriers, transaction start, transaction end, and any other algorithm specific helping methods. All these methods are in one class "STM.java". No prior knowledge of Jikes RVM is required and porting a new algorithm to ByteSTM requires only understanding ByteSTM framework.

3 Experimental Evaluation

To understand how ByteSTM, a VM-level STM, stacks up against non VM-level STMs, we conducted an extensive experimental study. Though a VM-level implementation may help improve STM performance, the underlying STM algorithm used also plays an important role, especially, when that algorithm's performance is known to be workload-dependent (see Subsection 2.6). Thus, the performance study wants also to investigate whether any performance gain from a VM-level implementation is *algorithm-independent*. It would also be interesting to understand whether some algorithms gained more than others, and if so, why. Thus, we compare ByteSTM against non-VM STMs, with the same algorithm inside the VM versus "outside" it.

Our competitor non-VM STMs include Deuce, ObjectFabric, Multiverse, and JVSTM. Since some of these STMs use different algorithms (e.g., Multiverse uses TL2's modified version; JVSTM uses a multi-version algorithm) or different implementations, a direct comparison between them and ByteSTM has some degree of unfairness. This is because, such a comparison will include many combined factors–e.g., ByteSTM's TL2 implementation is similar to Deuce's TL2 implementation, but the write-set and memory management are different. Therefore, it will be difficult to conclude that ByteSTM's (potential) gain is directly due to VM-level STM implementation. Therefore, we implemented a non-VM version using TL2, RingSTM and NOrec algorithms as Deuce *plug-ins*. Comparing ByteSTM with such a non-VM implementation reduces the factors in the comparison.

The non-VM implementations were made as close as possible to the VM ones. Offline instrumentation was used to eliminate online instrumentation overhead. The same open-address hashing write set was used. A large read-set and write-set were used so that they were sufficient for the experiments without requiring extra space. The sets were recycled for the next transactions, thereby needing only a single memory allocation and thus minimizing the GC overhead. We used Deuce for the non-VM implementation, since it has many of ByteSTM's features – e.g., it directly accesses memory and uses field granularity. Moreover, it achieved the best performance among all competitors (see results later in this section).

3.1 Test Environment

We used a 48-core machine, which has four AMD Opteron^{TM}Processors, each with 12 cores running at 1700 MHz, and 16 GB RAM. The machine runs Ubuntu Linux 10.04 LTS 64-bit. We used Jikes's production configuration (version 3.1.2),

which includes the Jikes optimizing compiler and the GenImmix GC [3] (i.e., a two-generation copying GC) and matches ByteSTM configurations. Since Deuce uses a non-standard proprietary API, `sun.misc.Unsafe`, which is not fully supported by Jikes, to run Deuce atop Jikes RVM, we added the necessary methods to Jikes RVM's `sun.misc.Unsafe` implementation (e.g., `getInt`, `putInt`, etc).

Our test applications include both micro-benchmarks (i.e., data structures) and macro-benchmarks. The micro-benchmarks include Linked-List, Skip-List, Red-Black Tree, and Hash Set. The macro-benchmarks include five applications from the STAMP suite [6]: Vacation, KMeans, Genome, Labyrinth, and Intruder. We used Arie Zilberstein's Java implementation of STAMP [28].

For the micro-benchmarks, we measured the transactional throughput (i.e., transactions committed per second). Thus, higher is better. For the macro-benchmarks, we measured the core program execution time, which includes transactional execution time. Thus, smaller is better. Each experiment was repeated 10 times, and each time, the VM was "warmed up" (i.e., we let the VM run for some time without logging the results) before taking the measurements. We show the average for each data point. Due to space constraints, we only show results for Linked-List, Red-Black Tree, Vacation, Intruder, and Labyrinth (see [20] for complete results).

3.2 Micro-Benchmarks

We converted the data structures from coarse-grain locks to transactions. The transactions contain all the critical section code in the coarse-grain lock version.

Each data structure is used to implement a sorted integer set interface, with set size 256 and set elements in the range 0 to 65536. Writes represent add and remove operations, and they keep the set size approximately constant during the experiments. Different ratios of writes and reads were used to measure performance under different levels of contention: 20% and 80% writes. We also varied the number of threads in exponential steps (i.e., 1, 2, 4, 8, ...), up to 48.

Linked List. Linked-list operations are characterized by high number of reads (the range is from 70 at low contention to 270 at high contention), due to traversing the list from the head to the required node, and a few writes (about 2 only). This results in long transactions. Moreover, we observed that transactions suffer from high number of aborts (abort ratio is from 45% to 420%), since each transaction keeps all visited nodes in its read-set, and any modification to these nodes by another transaction's add or remove operation will abort the transaction.

Figure 2 shows the results. ByteSTM has three curves: RingSTM, TL2, and NOrec. In all cases, ByteSTM/NOrec achieves the best performance since it uses an efficient read-set data structure based on open addressing hashing. ByteSTM/RingSTM comes next since it has no read-set and it uses a bloom filter as a read signature, but the performance is affected by bloom filter's false positives. This is followed by ByteSTM/TL2 which is affected by its sequential read-set. Deuce's performance is the best between other STM libraries. Other STMs perform similarly, and all of them have very low throughput. Non-VM

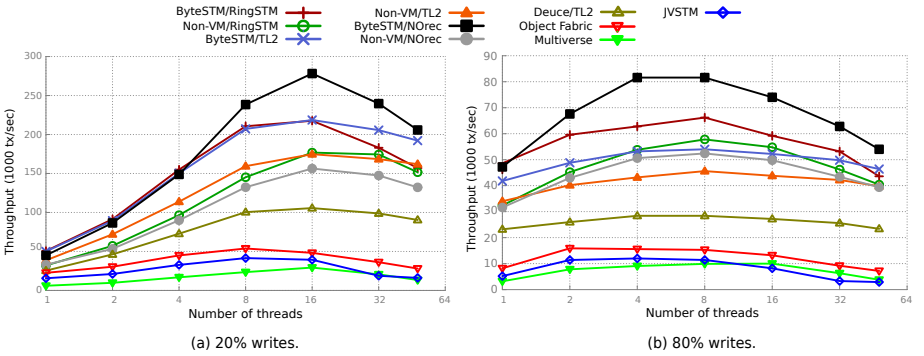


Fig. 2. Throughput for Linked-List

implementation of each algorithm performs in a similar manner but with lower throughput. ByteSTM outperforms non-VM implementations by 15–70%.

Since Deuce/TL2 achieved the best performance among all other STMs, for all further experiments, we use Deuce as a fair competitor against ByteSTM to avoid clutter, along with the non-VM implementations of TL2, RingSTM and NOrec. Accordingly, in the rest of the plots the curves of JVSTM, Multiverse and Object Fabric will be dropped (simplifying also the legibility).

Red-black Tree. Here, operations are characterized by small number of reads (15 to 30), and small number of writes (2 to 9), which result in short transactions.

Figure 3 shows the results. In all cases, ByteSTM/TL2 achieves the best performance and scalability due to the small number of reads. ByteSTM/NOrec does not scale well since it is based on one single global lock. Moreover, the small size of the tree creates a high contention between the threads. RingSTM's performance is similar to NOrec as it is based on a global data structure and has the added overhead of bloom's filter false positives.

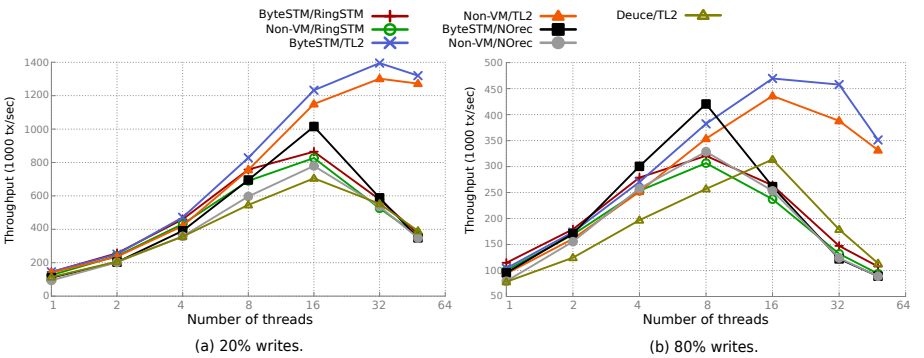


Fig. 3. Throughput of Red-Black Tree

In this benchmark, there is no big gap between ByteSTM and non-VM implementation due to the nature of the benchmark, namely very short transactions over a small data structure by large number of threads. ByteSTM outperforms non-VM implementation by 6–17%.

3.3 Macro Benchmark

Vacation. Vacation has medium-length transactions, medium read-sets, medium write-sets, and long transaction times (compared with other STAMP benchmarks). We conducted two experiments: low contention and high contention.

Figure 4 shows the results. Note that, here, the y-axis represents the time taken to complete the experiment, and the x-axis represents the number of threads. ByteSTM/NOrec has the best performance under both low and high contention conditions. The efficient read-set implementation contributed to its performance. But, it does not scale well. ByteSTM/RingSTM suffers from high number of aborts due to false positives and long transactions so it started with a good performance then degrades quickly. ByteSTM outperforms non-VM implementations by an average of 15.7% in low contention and 18.3% in high contention.

Intruder. The Intruder benchmark [6] is characterized by short transaction lengths, medium read-sets, medium write-sets, medium transaction times, and high contention.

Figure 5(a) shows the results. We observe that ByteSTM/NOrec achieves the best performance but does not scale. ByteSTM/RingSTM suffers from increased aborts due to false positives. ByteSTM/TL2 has a moderate performance. ByteSTM outperforms non-VM implementations by an average of 11%.

Labyrinth. The Labyrinth benchmark [6] is characterized by long transaction lengths, large read-sets, large write-sets, long transaction times, and very high contention.

Figure 5(b) shows the results. ByteSTM/NOrec achieves the best performance. ByteSTM/RingSTM suffers from extremely high number of aborts due

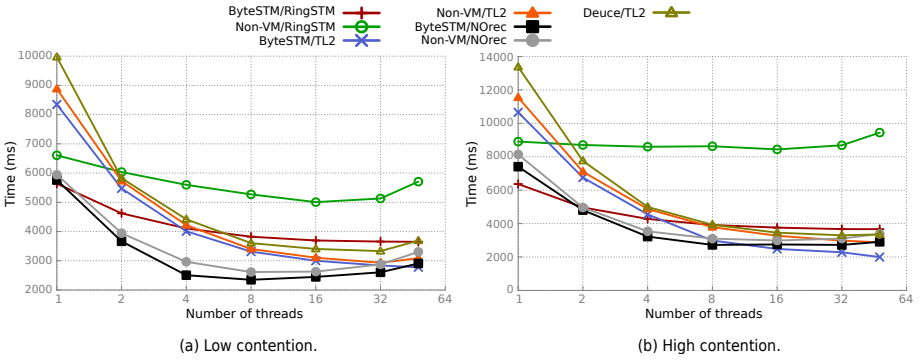


Fig. 4. Execution time under Vacation

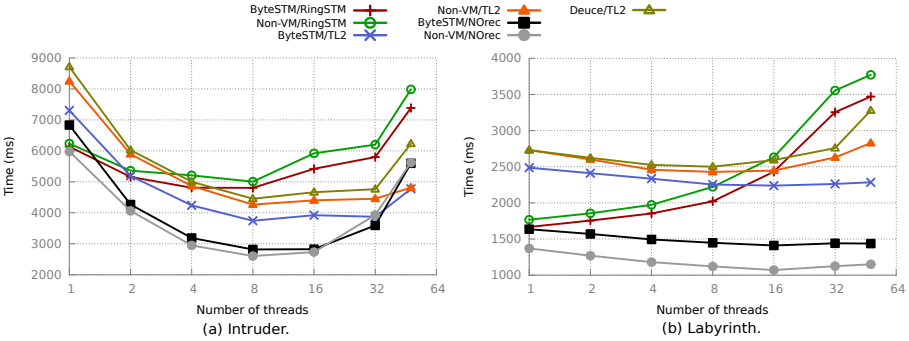


Fig. 5. Execution time under (a) Intruder, and (b) Labyrinth

to false positives and long transactions, and shows no scalability. All algorithms suffered from the high contention and has very low scalability. ByteSTM outperforms non-VM implementation by an average of 18.5%.

3.4 Summary

ByteSTM improves over non-VM implementations by an overall average of 30%: on micro-benchmarks, it improves by 6 to 70%; on macro-benchmarks, it improves by 7 to 60%. Moreover, scalability is better. ByteSTM, in general, is better when the abort ratio and contention are high.

RingSTM performs well, irrespective of reads. However, its performance is highly sensitive to false positives when writes increases. TL2 performs well when reads are not large due to its sequential read-set implementation. It also performs and scales well when writes increases. NOrec performs the best with behavior similar to RingSTM as it does not suffer from false positives, but it does not scale.

Algorithms that uses a global data structure and small number of CAS operations (like NOrec and RingSTM) have a bigger gain when implemented at VM-level. TL2 that uses a large number of locks (e.g., Lock table) and hence large number of CAS operations did not gain a lot from VM-level implementation.

4 Conclusions

Our work shows that implementing an STM at the VM-level can yield significant performance benefits. This is because at the VM-level: *i)* memory operations are faster; *ii)* the GC overhead is eliminated; *iii)* STM operations are embedded in the generated native code; *iv)* metadata is attached to the thread header.

For all these reasons the overhead of STM is minimized. Since the VM has full control over all transactional and non-transactional memory operations, features

such as strong atomicity and irrevocable operations (not currently supported) can be efficiently supported.

These optimizations are not possible at a library-level. A compiler-level STM for managed languages also cannot support these optimizations. Implementing an STM for a managed language at the VM-level is likely the most performant.

Next steps for ByteSTM are exploring compile time optimization specific for STM (i.e., STM optimization pass). And, modify the thread scheduler so that it will be STM aware and reduces the conflicts rate.

ByteSTM is open-sourced and is freely available at `hydravm.org/bytestm`. A modified version of ByteSTM is currently used in the HydraVM project [24], which is exploring automated concurrency refactoring in legacy code using TM.

Acknowledgments. This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385. A short version of this paper appeared as Brief Announcement at 2013 TRANSACT workshop. TRANSACT does not have archival proceedings and explicitly encourages resubmissions to formal venues.

References

1. Adl-Tabatabai, A.: The StarJIT compiler: A dynamic compiler for managed runtime environments. Intel Technology Journal (2003)
2. Alpern, B., Augart, S.: The Jikes research virtual machine project: building an open-source research community. IBM Syst. J. 44, 399–417 (2005)
3. Blackburn, S.M., McKinley, K.S.: Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In: PLDI (2008)
4. Bradel, B.J., Abdelrahman, T.S.: The use of hardware transactional memory for the trace-based parallelization of recursive Java programs. In: PPPJ (2009)
5. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Science of Computer Programming 63(2), 172–185 (2006)
6. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC (September 2008)
7. Carlstrom, B., McDonald, A., et al.: The Atomos transactional programming language. ACM SIGPLAN Notices 41(6), 1–13 (2006)
8. Christie, D., Chung, J., et al.: Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In: EuroSys, pp. 27–40 (2010)
9. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: Streamlining STM by Abolishing Ownership Records. In: PPoPP, pp. 67–78. ACM (2010)
10. Dice, D., Shalev, O., Shavit, N.N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
11. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP, pp. 175–184 (2008)
12. Harris, T., Fraser, K.: Language support for lightweight transactions. ACM SIGPLAN Notices 38(11), 388–402 (2003)
13. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd edn. Morgan and Claypool Publishers (2010)

14. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. ACM SIGPLAN Notices 41(10), 253–262 (2006)
15. Hindman, B., Grossman, D.: Atomicity via source-to-source translation. In: Workshop on Memory System Performance and Correctness, pp. 82–91 (2006)
16. Intel Corporation. Intel C++ STM Compiler (2009), http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/
17. Korland, G., Shavit, N., Felber, P.: Noninvasive concurrency with Java STM. In: MULTIPROG (2010)
18. Lie, S.: Hardware support for unbounded transactional memory. Master's thesis. MIT (2004)
19. Meawad, F.: Collecting transactional garbage. In: TRANSACT (2011)
20. Mohamedin, M., Ravindran, B.: ByteSTM: Virtual Machine-level Java Software Transactional Memory. Technical report, Virginia Tech. (2012), http://www.hydravm.org/hydra/chrome/site/pub/ByteSTM_tech.pdf
21. ObjectFabric Inc. ObjectFabric (2011), http://objectfabric.com
22. Paleczny, M., Vick, C., Click, C.: The Java Hotspot^{TM}Server Compiler. In: Java^{TM}Virtual Machine Research and Technology Symposium. USENIX (2001)
23. Riegel, T., Fetzer, C., Felber, P.: Snapshot isolation for software transactional memory. TRANSACT (2006)
24. Saad, M.M., Mohamedin, M., Ravindran, B.: HydraVM: extracting parallelism from legacy sequential code using STM. In: HotPar. USENIX (2012), hydravm.org
25. Spear, M.F., et al.: RingSTM: scalable transactions with a single atomic instruction. In: SPAA, pp. 275–284 (2008)
26. Veentjer, P.: Multiverse (2011), http://multiverse.codehaus.org
27. Welc, A., Jia, L., Hosking, A.L.: Transactional monitors for concurrent objects. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 518–541. Springer, Heidelberg (2004)
28. Zilberstein, A.: Java implementation of STAMP (2010), https://github.com/DeuceSTM/DeuceSTM/tree/master/src/test/jstamp

The Future of a Missed Deadline

Behrooz Nobakht[1,3], Frank S. de Boer[2], and Mohammad Mahdi Jaghoori[1]

[1] Leiden University
bnobakht@liacs.nl, m.jaghouri@lacdr.leidenuniv.nl
[2] Centrum Wiskunde en Informatica
frb@cwi.nl
[3] SDL Fredhopper
bnobakht@sdl.com

Abstract. In this paper, we introduce a real-time actor-based programming language and provide a formal but intuitive operational semantics for it. The language supports a general mechanism for handling exceptions raised by missed deadlines and the specification of application-level scheduling policies. We discuss the implementation of the language and illustrate the use of its constructs with an industrial case study from distributed e-commerce and marketing domain.

Keywords: actors, application-level scheduling, real-time, deadlines, futures, Java.

1 Introduction

In real-time applications, rigid deadlines necessitate stringent scheduling strategies. Therefore, the developer must ideally be able to program the scheduling of different tasks inside the application. Real-Time Specification for Java (RTSJ) [11,12] is a major extension of Java, as a mainstream programming language, aiming at enabling real-time application development. Although RTSJ extensively enriches Java with a framework for the specification of real-time applications, it yet remains at the level of conventional *multithreading*. The drawback of multithreading is that it involves the programmer with OS-related concepts like threads, whereas a real-time Java developer should only be concerned about high-level entities, i.e., objects and method invocations, also with respect to real-time requirements.

The actor model [9] and actor-based programming languages, which have reemerged in the past few years [24,3,10,14,26], provide a different and promising paradigm for concurrency and distributed computing, in which threads are transparently encapsulated inside actors. As we will argue in this paper, this paradigm is much more suitable for real-time programming because it enables the programmer to obtain the appropriate high-level view which allows the management of complex real-time requirements.

In this paper, we introduce an actor-based programming language Crisp for real-time applications. Basic real-time requirements include deadlines and

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 181–195, 2013.

timeouts. In Crisp, deadlines are associated with asynchronous messages and timeouts with futures [6]. Crisp further supports a general actor-based mechanism for handling exceptions raised by missed deadlines. By the integration of these basic real-time control mechanisms with the application-level policies supported by Crisp for scheduling of the messages inside an actor, more complex real-time requirements of the application can be met with more flexibility and finer granularity.

We formalize the design of Crisp by means of structural operational semantics [22] and describe its implementation as a full-fledged programming language. This implementation uses both the Java and Scala language with extensions of Akka library. We illustrate the use of the programming language with an industrial case study from SDL Fredhopper that provides enterprise-scale distributed e-commerce solutions on the cloud.

The paper continues as follows: Section 2 introduces the language constructs and provides informal semantics of the language with a case study in Section 2.1. Section 3 presents the operational semantics of Crisp. Section 4 follows to provide a detailed discussion on the implementation. The case study continues in this section with further details and code examples. Section 5 discusses related work of research and finally Section 6 concludes the paper and proposes future line of research.

2 Programming with Deadlines

In this section, we introduce the basic concepts underlying the notion of "deadlines" for asynchronous messages between actors. The main new constructs specify how a message can be sent with a deadline, how the message response can be processed, and what happens when a deadline is missed. We discuss the informal semantics of these concepts and illustrate them using a case study in Section 2.1.

Listing 1 introduces a minimal version of the real-time actor-based language Crisp. Below we discuss the two main new language constructs presented at lines (7) and (8).

How to send a message with a deadline? The construct

$$f = e_0 \,!\, m(\overline{e})\ \texttt{deadline}(e_1)$$

describes an asynchronous message with a deadline specified by e_1 (of type T_{time}). Deadlines can be specified using a notion of time unit such as millisecond, second, minute or other units of time. The caller expects the callee (denoted by e_0) to process the message within the units of time specified by e_1. Here processing a message means starting the execution of the process generated by the message. A deadline is missed if and only if the callee does not start processing the message within the specified units of time.

What happens when a deadline is missed? Messages received by an actor generate processes. Each actor contains one active process and all its other processes are queued. Newly generated processes are *inserted* in the queue according

$$
\begin{aligned}
C &::= \textbf{class}\ \mathsf{N}\ \textbf{begin}\ V^{?}\ \{M\}^{*}\ \textbf{end} && (1)\\
M_{sig} &::= \mathsf{N}(\overline{T\ x}) && (2)\\
M &::= \{M_{sig} == \{V\ ;\}^{?} S\} && (3)\\
V &::= \textbf{var}\ \{\{x\},^{+}\ :\ \mathsf{T}\ \{= e\}^{?}\},^{+} && (4)\\
S &::= x := e\ | && (5)\\
&::= x := \textbf{new}\ \mathsf{T}(e^{?})\ | && (6)\\
&:= f = e\ !\ m(\overline{e})\ \textbf{deadline}(e)\ | && (7)\\
&::= x := f.\textbf{get}(e^{?})\ | && (8)\\
&::= \textbf{return}\ e\ | && (9)\\
&::= S\ ;\ S\ | && (10)\\
&::= \textbf{if}\ (b)\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{end}\ | && (11)\\
&::= \textbf{while}\ (b)\ \{\ S\ \}\ | && (12)\\
&::= \textbf{try}\ \{S\}\ \textbf{catch}(\mathsf{T}_{\texttt{Exception}}\ x)\ \{\ S\ \} && (13)
\end{aligned}
$$

Fig. 1. A kernel version of the real-time programming language. The bold scripted **keywords** denote the reserved words in the language. The over-lined $\overline{v}$ denotes a sequence of syntactic entities v. Both local and instance variables are denoted by x. We assume distinguished local variables `this`, `myfuture`, and `deadline` which denote the actor itself, the unique future corresponding to the process, and its deadline, respectively. A distinguished instance variable `time` denotes the current time. Any subscripted type $\mathsf{T}_{specialized}$ denotes a *specialized* type of general type T; e.g. $\mathsf{T}_{\texttt{Exception}}$ denotes all "exception" types. A variable f is in T_{future}. N is a name (identifier) used for classes and method names. C denotes a class definition which consists of a definition of its instance variables and its methods; M_{sig} is a method signature; M is a method definition; S denotes a statement. We abstract from the syntax the side-effect free expressions e and boolean expressions b.

to an application-specific policy. When a *queued* process misses its deadline it is *removed* from the queue and a corresponding *exception* is recorded by its future (as described below). When the currently active process is terminated the process at the head of the queue is activated (and as such dequeued). The active process cannot be *preempted* and is forced to *run to completion*. In Section 4 we discuss the implementation details of this design choice.

How to process the response of a message with a deadline? In the above example of an asynchronous message, the *future* result of processing the message is denoted by the variable f which has the type of **Future**. Given a future variable f, the programmer can query the availability of the result by the construct

$$v = f.\textbf{get}(e)$$

The execution of the **get** operation terminates successfully when the future variable f contains the result value. In case the future variable f records an exception, e.g. in case the corresponding process has missed its deadline, the **get**

operation is *aborted* and the exception is *propagated*. Exceptions can be caught by `try-catch` blocks.

Listing 1. Using try-catch for processing future values

```
1 try {
2   x = f.get(e)
3   S_1
4 } catch(Exception x) {
5   S_2
6 }
```

For example, in Listing 1, if the **`get`** operation raises an exception control, is transferred to line (5); otherwise, the execution continues in line (3). In the `catch` block, the programmer has also access to the occurred exception that can be any kind of exception including an exception that is caused by a missed deadline. In general, any uncaught exception gives rise to abortion of the active process and is recorded by its future. Exceptions in our actor-based model thus are propagated by futures.

The additional parameter e of the get operation is of type T_{time} and specifies a *timeout*; i.e., the **`get`** operation will timeout after the specified units of time.

2.1 Case Study: Fredhopper Distributed Data Processing

Fredhopper is an SDL company since 2008 and a leading search, merchandising and personalization solution provider, whose products are uniquely tailored to the needs of online business. Fredhopper operates behind the scenes of more than 100 of the largest online sellers. The Fredhopper Access Server (FAS) provides access to high quality product catalogs. Typically deployments have about 10 explicit attribute values associated with a product over thousands of attribute dimensions. This challenging task involves working on difficult issues, such as the performance of information retrieval algorithms, the scalability of dealing with huge amounts of data and in satisfying large amounts of user requests per unit of time, the fault tolerance of complex distributed systems, and the executive monitoring and management of large-scale information retrieval operations. Fredhopper offers its services and facilities to e-Commerce companies (customers) as services (SaaS) over the cloud computing infrastructure (IaaS); which gives rise to different challenges in regards with resources management techniques and the customer cost model and service level agreements (SLA).

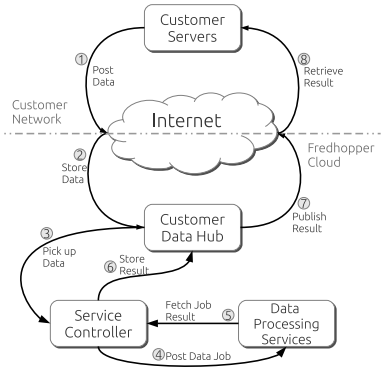


Fig. 2. Fredhopper's Controller life cycle for remote data processing

To orchestrate different services such as FAS or data processing, Fredhopper takes advantage of a service controller (a.k.a. Controller). Controller is responsible to passively manage different service installations for each customer. For instance, in one scenario, a customer submits their data along with a processing request to their data hub server. Controller, then picks up the data and initiates a data processing job (usually an ETL job) in a data processing service. When the data processing is complete, the result is again published to customer environment and additionally becomes available through FAS services. Figure 2 illustrates an example scenario that is described above.

In the current implementation of Controller, at Step 4, a data job instance is submitted to a remote data processing service. Afterwards, the future response of the data job is determined by a periodic remote check on the data service (Step 4). When the job is finished, Controller continues to retrieve the data job results (Step 5) and eventually publishes it to customer environment (Step 6).

In terms of system responsiveness, Step 4 may never complete. Step 4 failure can have different causes. For instance, at any moment of time, there are different customers' data jobs running on one data service node; i.e. there is a chance that a data service becomes overloaded with data jobs preventing the periodic data job check to return. If Step 4 fails, it leads the customer into an *unbounded waiting* situation. According to SLA agreements, this is *not* acceptable. It is strongly required that for any data job, the customer should be notified of the result: either a completed job with *success/failed* status, a job that is not completed, or a job with an unknown state. In other words, Controller should be able to guarantee that any data job request terminates.

To illustrate the contribution of this paper, we extract a closed-world simplified version of the scenario in Figure 2 from Controller. In Section 4, we provide an implementation-level usage of our work applied to this case study.

3 Operational Semantics

We describe the semantics of the language by means of a two-tiered labeled transition system: a local transition system describes the behavior of a single actor and a global transition system describes the overall behavior of a system of interacting actors. We define an actor state as a pair $\langle p, q\rangle$, where

- p denotes the current active process of the actor, and
- q denotes a queue of pending processes.

Each pending process is a pair (S, τ) consisting of the current executing statement S and the assignment τ of values to the *local* variables (e.g., formal parameters). The active process consists of a pair (S, σ), where σ assigns values to the local variables and additionally assigns values to the instance variables of the actor.

3.1 Local Transition System

The local transition system defines transitions among actor configurations of the form $\langle p, q, \phi\rangle$, where (p, q) is an actor state and for any object o identifying

a created future, ϕ denotes the shared heap of the created future objects, i.e., $\phi(o)$, for any future object o existing in ϕ, denotes a record with a field val which represents the return value and a boolean field $\mathsf{aborted}$ which indicates abortion of the process identified by o.

In the local transition system we make use of the following axiomatization of the occurrence of exceptions. Here $(S, \sigma, \phi) \uparrow v$ indicates that S raises an exception v:

- $(x = f.\mathtt{get}(), \sigma, \phi) \uparrow \sigma(f)$ where $\phi(\sigma(f)).\mathsf{aborted} = \mathtt{true}$,
- $\dfrac{(S, \sigma, \phi) \uparrow v}{\mathtt{try}\{S\}\mathtt{catch}(\mathsf{T}\ u)\{S'\} \uparrow v}$ where v is not of type T, and,
- $\dfrac{(S, \sigma, \phi) \uparrow v}{(S; S, \sigma, \phi)' \uparrow v}$.

We present here the following transitions describing internal computation steps (we denote by $\mathsf{val}(e)(\sigma)$ the value of the expression e in σ and by $f[u \mapsto v]$ the result of assigning the value v to u in the function f).

Assignment statement is used to assign a value to a variable:

$$\langle (x = e; S, \sigma), q, \phi \rangle \rightarrow \langle (S, \sigma[x \mapsto \mathsf{val}(e)(\sigma)]), q, \phi \rangle$$

Returning a result consists of setting the field val of the future of the process:

$$\langle (\mathtt{return}\ e; S, \sigma), q, \phi \rangle \rightarrow \langle (S, \sigma), q, \phi[\sigma(\mathtt{myfuture}).\mathsf{val} \mapsto \mathsf{val}(e)(\sigma)] \rangle$$

Initialization of `timeout` in `get` operation assigns to a distinguished (local) variable `timeout` its initial *absolute* value:

$$\langle (x = f.\mathtt{get}(e); S, \sigma), q, \phi \rangle \rightarrow \langle (x = f.\mathtt{get}(e); S, \sigma[\mathtt{timeout} \mapsto \mathsf{val}(e + \mathsf{time})(\sigma), q, \phi \rangle$$

The `get` operation is used to assign the value of a future to a variable:

$$\langle (x = f.\mathtt{get}(); S, \sigma), q, \phi \rangle \rightarrow \langle (S, \sigma[x \mapsto \phi(\sigma(f)).\mathsf{val}]), q, \phi \rangle$$

where $\phi(\sigma(f)).\mathsf{val} \neq \bot$.

Timeout is operationally presented by the following transition:

$$\langle (x = f.\mathtt{get}(); S, \sigma), q, \phi \rangle \rightarrow \langle (S, \sigma), q, \phi \rangle$$

where $\sigma(\mathtt{time}) < \sigma(\mathtt{timeout})$.

The `try-catch` block semantics is presented by:

$$\frac{\langle (S,\sigma),q,\phi\rangle \to \langle (S',\sigma'),q',\phi'\rangle}{\langle (\texttt{try}\{S\}\texttt{catch}(\texttt{T}\ x)\{S''\};S''',\sigma),q,\phi\rangle \to \langle (\texttt{try}\{S'\}\texttt{catch}(\texttt{T}\ x)\{S''\};S''',\sigma),q',\phi'\rangle}$$

Exception Handling. We provide the operational semantics of exception handling in a general way in the following:

$$\frac{(S,\sigma,\phi)\uparrow v}{\langle (\texttt{try}\{S\}\texttt{catch}(\texttt{T}\ x)\{S''\};S''',\sigma),q,\phi\rangle \to \langle (S'';S''',\sigma[x\mapsto v]),q,\phi\rangle}$$

where the exception v is of type `T`.

Abnormal termination of the active process is generated by an uncaught exception:

$$\frac{(S,\sigma,\phi)\uparrow v}{\langle (S;S',\sigma),q,\phi\rangle \to \langle (S'',\sigma'),q',\phi'\rangle}$$

where $q = (S'',\tau)\cdot q'$ and σ' is obtained from restoring the values of the local variables as specified by τ (formally, $\sigma'(x) = \sigma(x)$, for every instance variable x, and $\sigma'(x) = \tau(x)$, for every local variable x), and $\phi'(\sigma(\texttt{myfuture})).\texttt{aborted} = \texttt{true}$ ($\phi'(o) = \phi(o)$, for every $o \neq \sigma(\texttt{myfuture})$).

Normal termination is presented by:

$$\langle (E,\sigma),q,\phi\rangle \to \langle (S,\sigma'),q',\phi\rangle$$

where $q = (S,\tau)\cdot q'$ and σ' is obtained from restoring the values of the local variables as specified by τ (see above). We denote by E termination (identifying $S;E$ with S).

Deadline Missed. Let (S',τ) be some pending process in q such that $\tau(\texttt{deadline}) < \sigma(\texttt{time})$. Then

$$\langle (S,\sigma),q,\phi\rangle \to \langle p,q',\phi'\rangle$$

where q' results from q by removing (S',τ) and $\phi'(\tau(\texttt{myfuture})).\texttt{aborted} = \texttt{true}$ ($\phi'(o) = \phi(o)$, for every $o \neq \tau(\texttt{myfuture})$).

A message $m(\tau)$ specifies for the method m the initial assignment τ of its local variables (i.e., the formal parameters and the variables `this`, `myfuture`, and `deadline`). To model locally incoming and outgoing messages we introduce the following labeled transitions.

Incoming Message. Let the active process p belong to the actor $\tau(\texttt{this})$ (i.e., $\sigma(\texttt{this}) = \tau(\texttt{this})$ for the assignment σ in p):

$$\langle p,q,\phi\rangle \xrightarrow{m(\tau)} \langle p,\mathsf{insert}(q,m(\overline{v},d)),\phi\rangle$$

where $\mathsf{insert}(q, m(\tau))$ defines the result of inserting the process (S, τ), where S denotes the body of method m, in q, according to some application-specific policy (described below in Section 4).

Outgoing Message. We model an outgoing message by:

$$\langle (f = e_0 \,!\, m(\bar{e})\ \mathtt{deadline}(e_1); S, \sigma), q, \phi \rangle \xrightarrow{m(\tau)} \langle (S, \sigma[f \mapsto o]), q, \phi' \rangle$$

where

- ϕ' results from ϕ by extending its domain with a new future object o such that $\phi'(o).\mathsf{val} = \bot$[1] and $\phi'(o).\mathsf{aborted} = \mathtt{false}$,
- $\tau(\mathtt{this}) = \mathsf{val}(e_0)(\sigma)$,
- $\tau(x) = \mathsf{val}(e)(\sigma)$, for every formal parameter x and corresponding actual parameter e,
- $\tau(\mathtt{deadline}) = \sigma(\mathsf{time}) + \mathsf{val}(e_1)(\sigma)$,
- $\tau(\mathtt{myfuture}) = o$.

3.2 Global Transition System

A (global) system configuration S is a pair (Σ, ϕ) consisting of a set Σ of actor states and a global heap ϕ which stores the created future objects. We denote actor states by s, s', s'', etc.

Local Computation Step. The interleaving of local computation steps of the individual actors is modeled by the rule:

$$\frac{(s, \phi) \to (s', \phi')}{(\{s\} \cup \Sigma, \phi) \to (\{s'\} \cup \Sigma, \phi')}$$

Communication. Matching a message sent by one actor with its reception by the specified callee is described by the rule:

$$\frac{(s_1, \phi) \xrightarrow{m(\tau)} (s_1', \phi') \quad (s_2, \phi) \xrightarrow{m(\tau)} (s_2', \phi)}{(\{s_1, s_2\} \cup \Sigma, \phi) \to (\{s_1', s_2'\} \cup \Sigma, \phi')}$$

Note that only an outgoing message affects the shared heap ϕ of futures.

Progress of Time. The following transition uniformly updates the local clocks (represented by the instance variable `time`) of the actors.

$$(\Sigma, \phi) \to (\Sigma', \phi)$$

where

$$\Sigma' = \{\langle (S, \sigma'), q, \phi \rangle \mid \langle (S, \sigma), q, \phi \rangle \in \Sigma,\ \sigma' = \sigma[\mathtt{time} \mapsto \sigma(\mathtt{time}) + \delta]\}$$

for some positive δ.

[1] $\bot$ stands for "uninitialized".

4 Implementation

We base our implementation on Java's concurrent package: `java.util.concurrent`. The implementation consists of the following major components:

1. An extensible language API that owns the core abstractions, architecture, and implementation. For instance, the programmer may extend the concept of a scheduler to take full control of how, i.e., in what order, the processes of the individual actors are queued (and as such scheduled for execution). We illustrate the scheduler extensibility with an example in the case study below.
2. Language Compiler that translates the modeling-level programs into Java source. We use ANTLR [21] parser generator framework to compile modeling-level programs to actual implementation-level source code of Java.
3. The language is seamlessly integrated with Java. At the time of programming, language abstractions such as data types and third-party libraries from either Crisp or Java are equally usable by the programmer.

We next discuss the underlying deployment of actors and the implementation of real-time processes with deadlines.

Deploying Actors onto JVM Threads. In the implementation, each actor owns a main thread of execution, that is, the implementation does *not* allocate *one* thread per process because threads are *costly* resources and allocating to each process one thread in general leads to a poor performance: there can be an arbitrary number of actors in the application and each may receive numerous messages which thus give rise to a number of threads that goes beyond the limits of memory and resources. Additionally, when processes go into pending mode, their correspondent thread may be reused for other processes. Thus, for better performance and optimization of resource utilization, the implementation assigns a single thread for all processes inside each actor.

Consequently, at any moment in time, there is only one process that is executed inside each actor. On the other hand, the actors share a thread which is used for the execution of a watchdog for the deadlines of the queued processes (described below) because allocation of such a thread to each actor in general slows down the performance. Further this sharing allows the implementation to decide, based on the underlying resources and hardware, to optimize the allocation of the watchdog thread to actors. For instance, as long as the resources on the underlying hardware are abundant, the implementation decides to share as less as possible the watchdog thread. This gives each actor a better opportunity with higher precision to detect missed deadlines.

Implementation of Processes with Deadlines. A process itself is represented in the implementation by a data structure which encapsulates the values of its local variables and the method to be executed. Given a relative deadline

d as specified by a call we compute at run-time its absolute deadline (i.e. the expected starting time of the process) by

$$\texttt{TimeUnit.toMillis}(d) + \texttt{System.currentTimeMillis()}$$

which is a *soft* real-time requirement. As in the operational semantics, in the real-time implementation always the head of the process queue is scheduled for execution. This allows the implementation of a *default* earliest deadline first (EDF) scheduling policy by maintaining a queue ordered by the above absolute time values for the deadlines.

The important consequence of our non-preemptive mode of execution for the implementation is the resulting simplicity of thread management because preemption requires additional thread interrupts that facilitates the abortion of a process in the middle of execution. As stated above, a single thread in the implementation detects if a process has missed its deadline. This task runs periodically and to the end of all actors' life span. To check for a missed deadline it suffices to simply check for a process that the above absolute time value of its deadline is *smaller* than `System.currentTimeMillis()`. When a process misses its deadline, the actions as specified by the corresponding transition of the operational semantics are subsequently performed. The language API provides extension points which allow for each actor the definition of a customized watchdog process and scheduling policy (i.e., policy for enqueuing processes). The customized watchdog processes are still executed by a single thread.

Fredhopper Case Study. As introduced in Section 2.1, we extract a closed-world simplified version from Fredhopper Controller. We apply the approach discussed in this paper to use deadlines for asynchronous messages.

Listing 2 and 3 present the difference in the previous Controller and the approach in Crisp. The left code snippet shows the Controller that uses polling to retrieve data processing results. The right code snippet shows the one that uses messages with deadlines.

Listing 2. With polling

```
1  class DataProcessor begin
2    op process(d: Data) ==
3      var p := allocDataProcessor(d)
4      p ! process (d)
5      do {
6        s := p ! getStatus (d)
7        if (s <> nil)
8          var r := p ! getResults(d)
9          publishResult(r)
10       wait(TimeUnit.toSecond(1))
11     } while (true)
12 end
```

Listing 3. With deadlines

```
1  class DataProcessor begin
2    op process(d: Data) ==
3      var p := allocDataProcessor(d)
4      var D := estimateDeadline(d)
5      var f :=
6        p ! process (d) deadline (D)
7      try {
8        publishResult(f.get())
9      } catch (Exception x) {
10       if (f.isAborted)
11         notifyFailure(d)
12     }
13 end
```

When the approach in Crisp in the right snippet is applied to Controller, it is guaranteed that all data job requests are terminated in a *finite* amount of time. Therefore, there cannot be complains about never receiving a response for a specific data job request. Many of Fredhopper's customers rely on data jobs to eventually deliver an e-commerce service to their end users. Thus, to provide a guarantee to them that their job result is always published to their environment is critical to them. As shown in the code snippet, if the data job request is failed or aborted based on a deadline miss, the customer is still eventually informed about the situation and may further decide about it. However, in the previous version, the customer may never be able to react to a data job request because its results are never published.

In comparison to the Controller using polling, there is a way to express timeouts for future values. However, it does not provide language constructs to specify a deadline for a message that is sent to data processing service. A deadline may be simulated using a combination of timeout and periodic polling approaches (Listing 2). Though, this approach cannot guarantee eventual termination in all cases; as discussed before that Step 4 in Figure 2 may never complete. Controller is required to meet certain customer expectations based on an SLA. Thus, Controller needs to take advantage of a language/library solution that can provide a higher level of abstraction for real-time scheduling of concurrent messages. When messages in Crisp carry a deadline specification, Controller is able to guarantee that it can provide a response to the customer. This termination guarantee is crucial to the business of the customer.

Additionally, on the data processing service node, the new implementation takes advantage of the extensibility of schedulers in Crisp. As discussed above, the default scheduling policy used for each actor is EDF based on the deadlines carried by incoming messages to the actor. However, this behavior may be extended and replaced by a custom implementation from the programmer. In this case study, the priority of processes may differ if they the job request comes from specific customer; i.e. apart from deadlines, some customers have priority over others because they require a more real-time action on their job requests while others run a more relaxed business model. To model and implement this custom behavior, a custom scheduler is developed for the data processing node.

Listing 4. Data Processor class

```
1 class DataProcessor begin
2   var scheduler := new
        DataScheduler()
3   op process(d: Data) ==
4     // do process
5 end
```

Listing 5. Custom scheduler

```
1 class DataScheduler extends
      DefaultSchedulingManager {
2   boolean isPrior(Process p1,
        Process p2) {
3     if (p1.getCustomer().equals("A
          ")) {
4       return true;
5     }
6     return super.isPrior(p1, p2);
7   }
8 }
```

In the above listings, Listing 5 defines a custom scheduler that determines the priority of two processes with custom logic for specific customer. To use the custom scheduler, the only requirement is that the class `DataProcessor` defines a specific class variable called `scheduler` in Listing 4. The *custom scheduler* is picked up by Crisp core architecture and is used to schedule the queued processes. Thus, all processes from customer `A` have priority over processes from other customers no matter what their deadlines are.

We use Controller's logs for the period of February and March 2013 to examine the evaluation of Crisp approach. We define *customer satisfaction* as a property that represents the effectiveness of futures with deadline. For a customer c, the satisfaction can be denoted by $s = \frac{r_c^F}{r_c}$; in which r_c^F is the number of finished data processing jobs and r_c is the total number of requested data processing jobs from customer c. We extracted statistics for completed and never-ended data processing jobs from Controller logs (s_1). We replayed the logs with Crisp approach and measured the same property (s_2). We measured the same property for 180 customers that Fredhopper manages on the cloud. In this evaluation, a total number of about 25000 data processing requests were included. The results show 6% improvement in Table 1 (that amounts to around 1600 better data processing requests). Because of data issues or wrong parameters in the data processing requests, there are requests that still fail or never end and should be handled by a human resource.

s_1	s_2
88.71%	94.57%

Table 1. Evaluation Results

You may find more information including documentation and source code of Crisp at `http://nobeh.github.com/crisp`.

5 Related Work

The programming language presented in this paper is a real-time extension of the language introduced in [20]. This new extension features

- integration of asynchronous messages with deadlines and futures with timeouts;
- a general mechanism for handling exceptions raised by missed deadlines;
- high-level specification of application-level scheduling policies; and
- a formal operational semantics.

To the best of our knowledge the resulting language is the first implemented real-time actor-based programming language which formally integrates the above features.

In several works, e.g, [2] and [19], asynchronous messages in actor-based languages are extended with deadlines. However these languages do not feature futures with timeouts, a general mechanism for handling exceptions raised by missed deadlines or support the specification of application-level scheduling policies. Futures and fault handling are considered in the ABS language [13]. This

work describes recovery mechanisms for failed get operations on a future. However, the language does not support the specification of real-time requirements, i.e., no deadlines for asynchronous messages are considered and no timeouts on futures. Further, when a get operation on a future fails, [13] does not provide any context or information about the exception or the cause for the failure. Alternatively, [13] describes a way to "compensate" for a failed get operation on future. In [4], a real-time extension of ABS with scheduling policies to model distributed systems is introduced. In contrast to Crisp, Real-Time ABS is an executable *modeling* language which supports the explicit specification of the progress of time by means of duration statements for the *analysis* of real-time requirements. The language does not support however asynchronous messages with deadlines and futures with timeouts.

Two successful examples of actor-based programming languages are Scala and Erlang. Scala [10,1] is a hybrid object-oriented and functional programming language inspired by Java. Through the event-based model, Scala also provides the notion of continuations. Scala further provides mechanisms for scheduling of tasks similar to those provided by concurrent Java: it does not provide a direct and customizable platform to manage and schedule messages received by an individual actor. Additionally, Akka [25] extends Scala's actor programming model and as such provides a direct integration with both Java and Scala. Erlang [3] is a dynamically typed functional language that was developed at Ericsson Computer Science Laboratory with telecommunication purposes [5]. Recent developments in the deployment of Erlang support the assignment of a scheduler to each processor [17] (instead of one global scheduler for the entire application) but it does not, for example, support application-level scheduling policies. In general, none these languages provide a formally defined real-time extension which integrates the above features.

There are well-known efforts in Java to bring in the functionality of asynchronous message passing onto multicore including Killim [24], Jetlang [23], ActorFoundry [15], and SALSA [26]. In [15], the authors present a comparative analysis of actor-based frameworks for JVM platform. Most of these frameworks support futures with timeouts but do not provide asynchronous messages with deadlines, or a general mechanism for handling exceptions raised by missed deadlines. Further, pertaining to the domain of priority scheduling of asynchronous messages, these efforts in general provide a predetermined approach or a limited control over message priority scheduling. As another example, in [18] the use of Java Fork/Join is described to optimize mulicore applications. This work is also based on a *fixed priority* model. Additionally, from embedded hardware-software research domain, Ptolemy [7,16] is an actor-oriented open architecture and platform that is used to design, model and simulate embedded software. Their approach is hardware software co-design. It provides a platform framework along with a set of tools.

In general, existing high-level programming languages provide the programmer with little real-time control over scheduling. The state of the art allows specifying priorities for threads or processes that are used by the operating system,

e.g., Real-Time Specification for Java (RTSJ [11,12]) and Erlang. Specifically in RTSJ, [27] extensively introduces and discusses a framework for application-level scheduling in RTSJ. It presents a flexible framework to allow scheduling policies to be used in RTSJ. However, [27] addresses the problem mainly in the context of the standard multithreading approach to concurrency which in general does not provide the most suitable approach to distributed applications. In contrast, in this paper we have shown that an actor-based programming language provides a suitable formal basis for a fully integrated real-time control in distributed applications.

6 Conclusion and Future Work

In this paper, we presented both a formal semantics and an implementation of a real-time actor-based programming language. We presented how asynchronous messages with deadline can be used to control application-level scheduling with higher abstractions. We illustrated the language usage with a real-world case study from SDL Fredhopper along the discussion for the implementation. Currently we are investigating further optimization of the implementation of Crisp and the formal verification of real-time properties of Crisp applications using schedulability analysis [8].

References

1. Haller, P., Odersky, M.: Actors That Unify Threads and Events. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 171–190. Springer, Heidelberg (2007)
2. Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. In: FOCLASA, pp. 1–19 (2011)
3. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
4. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Lizeth Tapia Tarifa, S.: User-defined schedulers for real-time concurrent objects. Innovations in Systems and Software Engineering (2012)
5. Corrêa, F.: Actors in a new "highly parallel" world. In: Proc. Warm Up Workshop for ACM/IEEE ICSE 2010, WUP 2009, pp. 21–24. ACM (2009)
6. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
7. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity - the ptolemy approach. Proceedings of the IEEE 91(1), 127–144
8. Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Schedulability analysis using two clocks. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 224–239. Springer, Heidelberg (2003)
9. Smith, S.F., Agha, G.A., Mason, I.A., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming 7, 1–72 (1997)

10. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
11. JCP. RTSJ v1 JSR 1 (1998), http://jcp.org/en/jsr/detail?id=1
12. JCP. RTSJ v1.1 JSR 282 (2005), http://jcp.org/en/jsr/detail?id=282
13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
14. Johnsen, E.B., Owe, O.: An Asynchronous Communication Model for Distributed Concurrent Objects. Software and Systems Modeling 6(1), 39–58 (2007)
15. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: a comparative analysis. In: Proc. 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009, pp. 11–20. ACM (2009)
16. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems and Computers 12(03), 231–260 (2003)
17. Lundin, K.: Inside the Erlang VM, focusing on SMP. Presented at Erlang User Conference (November 13, 2008), http://www.erlang.se/euc/08/euc_smp.pdf
18. Maia, C., Nogueira, L., Pinho, L.M.: Combining rtsj with fork/join: a priority-based model. In: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2011, pp. 82–86. ACM, New York (2011)
19. Nielsen, B., Agha, G.: Semantics for an Actor-Based Real-Time Language. In: Fourth International Workshop on Parallel and Distributed Real-Time Systems (1996)
20. Nobakht, B., de Boer, F.S., Jaghoori, M.M., Schlatte, R.: Programming and deployment of active objects with application-level scheduling. In: ACM SAC (2012)
21. Terence Parr. Antlr, http://antlr.org/
22. Plotkin, G.D.: The origins of structural operational semantics. The Journal of Logic and Algebraic Programming 60-61(0), 3–15 (2004)
23. Rettig, M.: Jetlang Library (2008), http://code.google.com/p/jetlang/
24. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
25. TypeSafe. Akka (2010), http://akka.io/
26. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. SIGPLAN Not. 36, 20–34 (2001)
27. Zerzelidis, A., Wellings, A.: A framework for flexible scheduling in the RTSJ. ACM Trans. Embed. Comput. Syst. 10(1), 1–3 (2010)

Event Loop Coordination Using Meta-programming

Laure Philips*, Dries Harnie**, Kevin Pinte, and Wolfgang De Meuter

Software Languages Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{lphilips,dharnie,kpinte,wdmeuter}@vub.ac.be

Abstract. Event-based programming is used in different domains, ranging from user interface development to various distributed systems. Combining different event-based subsystems into one system forces the developer to manually coordinate the different event loops of these subsystems. This leads to a lot of excessive code and, in addition, some event loops are prey to lifecycle state changes. On mobile applications, for example, event loops can be shut down when memory runs low on the device. Current approaches take care of the communication problems between the different types of event loops, but become complex when trying to deal with lifecycle state changes. We propose a new coordination model, Elector, that allows two event loops to run separately, and introduce a novel kind of reference, called undead references. These references do not only allow communication between the event loops, but also handle lifecycle state changes in such a way that they do not influence other event loops.

Keywords: Event loops, Coordination model, Mobile platforms, Ambient-oriented programming.

1 Introduction

In traditional programming the control flow of a program is determined by its structure. To allow the program to react upon input in the of form events, the event-based programming paradigm can be used. The programmer registers observers or event handlers for different types of events and the *event loop* is responsible for detecting events and dispatching it to the observers. This programming paradigm is popular for developing user interfaces, where the program reacts upon input from the user. Event-based programming proves to be useful in distributed programming, where the source of an event and the matching event handler can live on different devices.

When developing a larger software system, composed out of different event-based subsystems, the programmer needs to manually coordinate these event loops.

* Funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).
** Prospective research for Brussels, Innoviris.

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 196–210, 2013.

Because the event loops have different characteristics, take for example the start-up time, it is up to the programmer to take care of the mismatch between the event loops. Current systems only address this issue partially; they do not take into account the lifecycle of event loops. Some event loops are prey to lifecycle state changes, meaning that an event loop can be shut down at any moment in time and maybe restarted afterwards. For example, mobile applications, which often use event loops, can be shut down when running in the background. Typically, the programmer can react upon lifecycle state changes, e.g. pause a playing video when the application is no longer visible to the user. These changes have consequences: a destroyed event loop is no longer accessible and other event loops in the system thus must be aware of the lifecycle state of that event loop.

In this paper, we introduce a coordination model called "Elector", which allows different types of event loops to be coordinated. The model enables programmers to concentrate on the logic of the program, instead of writing glue code for the event loops. The main concept behind Elector are its undead references, which transparently handle these lifecycle problems. We contribute the design and implementation of Elector, as a framework for the AmbientTalk language. We also provide a validation of our approach by comparing the code complexity of different versions of a representative application. Finally, we present a validation of Elector's performance measurements.

This paper is organised as follows: we first discuss related work in section 2 after which we introduce a motivating example (section 3). Afterwards we discuss the problems that arise when implementing this case study in section 4. In section 5 we present "Elector", together with a concrete implementation for Android and AmbientTalk. Section 6 evaluates the Elector model by comparing the different implementations of our motivating example. Finally we conclude this paper and present future work.

2 Related Work

In this section we discuss related work: other models that encapsulate one of the event loops and component-based software architectures, which are tailored towards systems with different sub-components.

2.1 Event Loop Encapsulation

Often when integrating different subsystems into a new and larger system, one ends up with excessive code size. This is because the subsystems make certain assumptions about the system in which they are used [6]. For example, when combining different event-based systems that are not compatible with each other, the programmer must adapt one or more event loops.

The models we discuss allow the coordination of event loops, by encapsulating their own native event loop inside an external one and allowing the native event loop to handle its events at a regular basis. As a consequence, when an event handler does not end immediately or not at all, the *entire* application is blocked

and not only one of the subparts. These models solve the communication issues related to combining event loops, but do not take into account the possible lifecycle state changes of these event loops.

POE (Perl Object Environment) [1] is an event-based Perl framework for reactive systems, cooperative multitasking and network applications. POE provides bridges for other event loops that normally need complete control, for example GTK. Such a bridge between an external event loop and the POE event loop runs the external event loop and uses timer functions to allow the POE event loop to handle its events periodically.

Tcl (Tool Command Language) [11] is a cross platform programming language that can be used in different domains: web applications, desktop applications, etc. The Tcl event loops provides mechanisms that allow the programmer to have a more fine-grained control over the event mechanism. This way, one can embed a Tcl event loop inside applications that have their own event loop.

2.2 Component-Based Software Architectures

ROS (Robot Operating System) [8] uses topics to exchange messages between nodes in a publish/subscribe manner. More concretely, nodes subscribe to a certain topic, while other nodes publish data on this relevant topic. ROS allows newly connected nodes to subscribe dynamically. While this decouples the subscribers and publishers, the nodes need to be running at the same time to send data to each other. This means that ROS supports decoupling in space , because it decouples the publishers and subscribers, but not decoupling in time, because the nodes need to be active at the same time in order to communicate [10]. This is necessary when taking into account that components in a system can have a different lifetime. In the case of event loops, Elector does not require one event loop to wait for the other event loop to be (re)started. Elector allows event loops to send messages to an event loop that is not available upon the time of sending, but guarantees that the message will be delivered when that event loop becomes available.

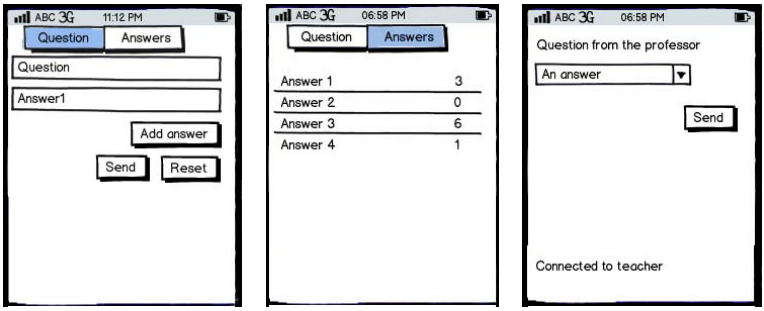
Java Beans [5] are reusable components that can be composed using a visual composition tool. It is made for reusability by supporting persistence, introspection, customisation through property editors, etc. Beans communicate through events, where a bean can be the source or the target of the event. Beans register their interest as a listener at another bean. This also reduces the inter-component coupling, but they are not decoupled in time, so beans need to be running at the same time to communicate.

3 Motivating Example

In this section we introduce an example scenario, which is used to abstract the problems that arise when combining event loops. Our case study is a clicker

application, a so-called personal response system that is used as a communication system during lectures. Every student receives a remote control, also called zapper or clicker, that allows them to select an answer when the teacher asks a question. The software on the teacher's computer collects the answers and the results are shown to the class by representing them in a graph. Such a clicker application increases the interactivity and feedback during lectures [4], involving all students and ensuring anonymity.

Nowadays, all students are equipped with smart phones, tablets, ... Therefore we use these mobile devices as a clicker device and use a mobile network to communicate between the students and the teacher. Our clicker application can be started for a teacher or for students, so first of all the user has to choose one of these roles. The teacher can send questions to the students, together with possible answers (shown in figures 1(a) and 1(b)). The student receives these questions and can send a selected answer to the teacher (shown in figure 1(c)).



(a) Teacher: question (b) Teacher: answers (c) Student

Fig. 1. Android views of the clicker application

We will implement this application on the Android mobile platform using the AmbientTalk programming language [10]. In the following sections we introduce AmbientTalk and Android and how they use event loops.

3.1 AmbientTalk

AmbientTalk [3,10] is a distributed programming language that is designed to solve the typical problems in mobile ad-hoc networks. A mobile ad-hoc network has no central infrastructure and the mobile devices that communicate with each other can move out of range. Moreover, the communicating parties can reconnect after a disconnection. AmbientTalk is based on the actor model [2], where every actor is represented by an event loop that communicates with the other event loops. AmbientTalk's communicating event loop model is tailored towards mobile ad-hoc networks: every event loop is independent, maintains its own state and communicates with other event loops by sending messages, thus events, to it. Moreover, network events like the discovery of actors are also handled by the event loop.

3.2 Android

An Android application is typically composed of different activities, where each activity represents the screen the user sees. An activity is the most important component of an Android application. Because the user can only see one screen at the time, only one activity can be active or in the foreground. Figure 1 shows the different screens the application consists of, each thus implemented by an activity.

The application components that run in the background can be either still running, but not visible, or can be destroyed. The programmer thus needs to take into account that an application running in the background can be destroyed, meaning all its state is lost. This is not only so for the Android platform, it is a characteristic shared by other mobile platforms like Apple iOS, BlackBerry OS, Windows Phone, etc. These mobile platforms define at least three lifecycle states for applications: active or running in the foreground, suspended or running the background and not running or destroyed. Thus, mobile applications are subject to lifecycle state changes, which are caused by the system or by the user navigating between applications.

When the lifecycle changes, the programmer can react by using callback methods. When for example an application that plays videos is suspended, the "pause" callback method can pause the video. When this application is destroyed, the programmer can store which video the user was watching.

If we take for example the student activity of the clicker application, we need to take into account that the reference to the corresponding AmbientTalk actor can be lost when the activity is restarted. Both event loops establish these references by implementing a registering method. In case the user is a student, the student activity starts the student actor at the beginning of the application. The actor retrieves a reference to the activity by calling its `registerAT` method and passing a reference to the student actor (shown in listing 1.2). Inside this registering method, the activity saves this reference to the actor and returns a reference to itself, as can be seen on line 1 of listing 1.1. On line 5 we see a simplified example of an event listener that gets called when the "send" button gets clicked. As can be seen on line 7, we need to check if the reference to the student actor is still valid. When the activity is restarted, we cannot restart the student actor, because the previous started one is still running.

```
1 StudentActivity registerAT(Student s) {
2   StudentActivity.student = s;
3   return this;
4 }
5 class Listener {
6  void onClick(View v) {
7    if (student != null) {
8    // send answer to teacher
9 }}}
```

Listing 1.1. Registering an actor (Java)

```
1 def gui := Android.registerAT(self)
```

Listing 1.2. Registering an activity (AmbientTalk)

Each application has a main thread, also called the UI thread, which cannot be blocked because it is responsible for processing lifecycle and user interaction events. Only the UI thread can alter view components, hence its name. When an event handler has a long running task to perform, it is up to the programmer to perform this task on a background or worker thread. As mentioned, these worker threads cannot update the user interface of the activity. To solve this, background threads have to use the `runOnUIThread` method for this purpose.

4 Problems

Using the building components of the AmbientTalk language and the Android platform we implemented a first version of the clicker application. The language integration is solved because AmbientTalk is implemented in Java (Android applications are implemented in Java) and the languages live in symbiosis with each other [9]. Therefore our solution does not take language integration into account, but focuses on the event loops. Several state-of-the art mechanisms exist to let code written in different languages interoperate, for example Mono, a framework for building cross-platform applications, integrates Java with languages of the .NET framework.

We give an overview of the problems that are encountered when combining different types of event loops. We can categorise these problems into lifecycle and communication problems. Lifecycle problems are caused by the different lifecycle of the event loops, while communication problems arise when the event loops send messages to one another.

Lifecycle Problems

Different Startup Time. When using two different event loops, chances are small that they have the same startup time. Because of this, one of the event loops needs to wait before communication between the two event loops can start. Therefore, it is up to the event loop that started later to initialise the communication, or to let the other event loop know it has started. When programming an application on Android using AmbientTalk, it is the task of the programmer to manually start up AmbientTalk and evaluate AmbientTalk actors. The mutual discovery is hard-coded by implementing registering methods in every event loop to establish references, as showed earlier in section 3.2.

Lifecycle State Changes. Event loops that suffer from lifecycle state changes can be destroyed, which means that events or messages sent to them are not processed. For instance, Android activities can be killed when the user navigates to another activity or when memory is low. But in our clicker application we need to take into account that actors and activities refer to each other. When an activity is killed, all state is lost, including the reference to the actor and maybe the AmbientTalk interpreter. When the activity is restarted, the programmer needs to check if AmbientTalk is still running, if the actors are still alive, etc. If so, the programmer must establish

a way to reconnect the new instance of the activity and the actor. If not, the actor needs to be restarted. There is currently no trivial way to handle this problem and the programmer must implement ad-hoc solutions for handling these lifecycle state changes.

References can Become Invalid. Because of the lifecycle state changes, the references to an event loop can become invalid when it is destroyed. The other event loop does not know the reference has become invalid. In the case of AmbientTalk and Android event loops, we have seen that the references between them cannot be restored, when the activity is recreated. A restarted activity is actually a new instance of that activity, so upon creation of an activity, we need a way to make actors point to the newly started activity. Communication sent to a destroyed activity is lost, which brings us to the communication problems.

Communication Problems

Communication can be Lost. This problem can arise in two different situations; when both event loops are not started yet, or when a reference to an event loop has become invalid. When a reference to an event loop is invalid, all messages sent to it are lost. When one of the event loops has not been started yet, the already started event loop has to wait before it can start to communicate with the other event loop. This is a consequence of the first problem. In case of Android and AmbientTalk event loops this means that Android's activities must wait when all the actors are started before they can send messages to it. Actors on their turn must be careful when sending messages to an activity, because it can be destroyed.

It is up to the programmer to establish a way for these event loops to communicate and handle each others lifecycle state changes. This leads to a lot of boilerplate code, which can be avoided as we discuss in the next section.

5 Elector

In this section we introduce the Elector (an acronym of Event Loop Coordination) model together with its implementation for the Android and AmbientTalk event loops. In Elector, references between different event loops are managed by undead references, which revive when a new event loop of the same type is started, hence their naming. On top of that, Elector wraps one of the event loops inside an event loop of the other type. The wrapped and wrapping event loop share the same lifecycle: when the wrapped one is killed, the wrapping is killed too. When the wrapped event loop becomes available again, so is its wrapper. Moreover, the wrapping event loop is the only one that has access to the wrapped event loop. The wrapping event loop thus routes incoming events directly to the wrapped event loop. Concretely, the wrapped and wrapping event loop refer directly to each other. Because they share the same lifecycle, this does not introduce the problems discussed before.

5.1 Model

Figure 2 shows how two different event loops are coordinated in Elector. We used AmbientTalk and Android event loops, but the model can be used for other types of event loops as well. Elector wraps the Android event loop inside an AmbientTalk event loop, which we call from now on the *wrapped* and *wrapping* event loops respectively. As we can see, the wrapping event loop refers to the

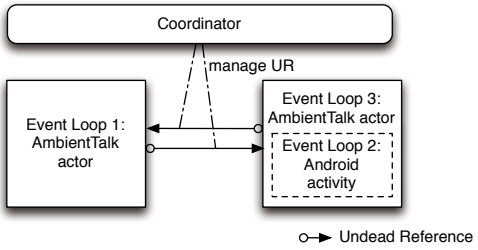


Fig. 2. Elector: the basic idea

other event loop of the same type by means of these undead references. The *coordinator* is responsible for managing the undead references and immediately returns a reference when asked for, even if the other event loops has not started yet. Undead references are the main concept of the model and they have the following characteristics:

Return immediately. When the coordinator is asked for an undead reference to a certain event loop, it returns one immediately, even when the referred event loop has not started yet. This way, event loops don't have to wait for the initialisation of the other event loop. In our example, the student actor retrieves an undead reference to the wrapping actor of the student activity, even when that activity is not started yet.

Remain accessible. Elector hides the state changes of an event loop behind undead references. Event loops that have an undead reference to an event loop that is hampered by lifecycle state changes, are not informed of this change and can keep communicating with the event loop as if it is still available. The student actor for example can continue using the undead reference it retrieved to the student activity, even when the activity is destroyed.

Rebind automatically. When an event loop is killed and subsequently restarted by the system, it is actually a new instance of that event loop. Undead references will rebind automatically to the new event loop. For example, after restarting the student activity all undead references to it transparently rebind with the new instance.

Buffer communication. Undead references route the communication to the actual event loop. When that event loop is inaccessible, the communication to it is buffered. When the event loop returns to a state that it can process messages, all the buffered ones are sent to it in the same order they were

received. In case of the clicker example, all messages sent to a wrapping actor for updating the user interface (e.g. displaying a question from the teacher) are sent when the wrapping actor and its wrapped activity are restarted.

By wrapping an event loop inside an event loop of another type, we can make use of the communication abstractions of that type of event loops. This results in an asynchronous way of communication between the event loops, which allows both of them to keep processing events, which is not the case of the models we discussed in section 2.

The coordinator creates the undead references, keeps track of them and also discovers all the event loops in the system. When an event loop is killed, the coordinator is informed and all undead references to that event loop start to buffer communication until a replacement event loop of the same type becomes available again. As we will see in the instantiation of Elector for Android and AmbientTalk, the coordinator should provide some guarantees that it is less likely to be killed. In our implementation, the coordinator is only killed when the whole application is destroyed, and thus all its subcomponents too.

5.2 Instantiation for Android and AmbientTalk

We implemented Elector for the Android and AmbientTalk event loops, where we choose to wrap the Android event loop inside an AmbientTalk event loop, being an AmbientTalk actor. There are several reasons for this choice: first of all, we can use the AmbientTalk discovery mechanism, which we discuss further on. Secondly, AmbientTalk allows event loops to communicate in an asynchronous and non-blocking way. Finally, AmbientTalk event loops don't suffer lifecycle state changes as much as Android event loops.

Coordinator. The coordinator is in this case an AmbientTalk actor that discovers all wrapping actors, together with an Android service that orchestrates all the activities of the application. In our case, the activities of the application connect with the service. When the state of an activity changes, it must inform the service of this state change. Other than keeping track of the lifecycle of the activities, we use this service to automatically start an AmbientTalk interpreter in the background and to load code in this interpreter. Each Android activity thus has an associated AmbientTalk actor, and the programmer has to follow a naming convention in order for the evaluation of the actors to be done transparently.

When an Android application uses an Android service, it retrieves a higher priority and it is less likely that the application is destroyed. Should it happen that the service is killed under extreme memory pressure, the AmbientTalk interpreter is destroyed too, together with the coordinator actor and all other running actors. When the service is restarted, the AmbientTalk interpreter is restarted too and the required actors will be re-evaluated.

For the discovery we use AmbientTalk's discovery mechanism, as illustrated in the following code snippets.

```
1 def CoordinatorActor := actor: {
2   whenever: Activity discovered: { |e|
3     // retrieve tag, manage references
4   };
5   Android.coordinator = self }
```

Listing 1.3. Discovery of an actor

```
1 deftype StudentGUI <: Activity;
2 def remoteInterface := object: {
3     // behaviour
4 };
5 export: remoteInterface as: StudentGUI
```

Listing 1.4. Publishing an actor

On line 2 of listing 1.3 we see how the coordinator actor discovers all actors that are categorised in the network as (subtypes of) the Activity tag. The construct `whenever:discovered:` is used for this purpose. When an actor with that tag is discovered, the coordinator actor receives a *far reference* to the discovered actor. The other code snippet (1.4) shows the other side: inside a wrapping actor, we can export its behaviour in the network using `export:as:`. Tags are made with the `deftype` keyword. On line 1 we create a tag StudentGUI, that is a subtype of the Activity tag. The StudentGUI tag is defined by the wrapping actor of the activity StudentActivity. We now discuss how these wrapping actors behave.

Wrapping Actors. Wrapping actors share the same lifecycle as their wrapped activity. More concretely, the wrapping actor is taken offline when the corresponding activity is destroyed, and taken back online when the activity restarts. The coordinator can react upon these network events and inform all undead references to that particular wrapping actor.

Next to sharing the same lifecycle with its Android activity, the wrapping actor communicates with that activity and also allows us to write most of the user interface code inside these wrapping actors instead of the activities. The communication between a wrapping actor and its wrapped activity can easily be achieved because of the symbiosis [9] between the AmbientTalk and Java language. When executing a method call inside an AmbientTalk actor, this method gets executed on the thread of that actor. Recall from section 3.2 that only the UI thread may update Android views and as a consequence we cannot directly update the view from AmbientTalk. Therefore, we introduce a new kind of actor: the UI Thread actor. This actor is responsible for the communication between actors and its activities, by allowing actors to post Runnables on the Android event loop. The wrapping actors can *ship off* their messages to this actor, which guarantees that they are executed on the right thread. The following code shows how the student's wrapping actor defines a method that shows whether the student is currently connected to the teacher.

```
1 def teacherStatus(status) {
2   UIThreadActor←runOnUIThread(getTypeTag(), "connection_state", runnable: {
3     def run(v) {  // method to be executed on UI Thread
4       v.setText(status);    // set the text
5       v.setVisibility(v.VISIBLE); // show the text view
6  }})}
```

This code sends an asynchronous message to the UIThreadActor, denoted by the left arrow. The first argument of `runOnUIThread` (line 2) message selects the correct Android activity. As a second argument, we pass the name of the view we want to alter and lastly, we pass the AmbientTalk equivalent of a Java Runnable, denoted by the `runnable` keyword. It defines a method `run` on line 3, that takes the Android view as an argument. This way of changing Android views is only the first step: we discuss an improvement in the next section.

Undead References. The final component of Elector are the undead references between an actor and a wrapping actor. AmbientTalk actors ask the coordinator for an undead reference to a wrapping actor using a tag, e.g. StudentGUI. The first time, the coordinator object makes a new one. When another actor already asked for an undead reference to that wrapping actor, the coordinator does not need to make a new reference, but reuses it. Figure 3 shows how an undead reference switches between two states: "forwarding" indicates the corresponding wrapping actor is available and in the "buffering" state the undead reference starts buffering all messages sent to it. Since the coordinator is informed of the availability of the wrapping actor, it is the task of the coordinator to switch the undead references between these states.

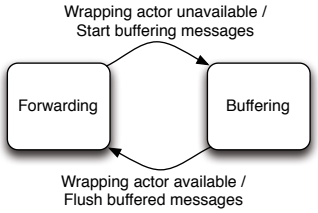


Fig. 3. State diagram for an undead reference

An undead reference is entirely programmed using AmbientTalk's reflective layer [7]. This layer allows us to catch asynchronous method calls on this reference and decide whether they need to be buffered (in case the corresponding wrapping actor is not available) or can be sent to the wrapping actor.

For example, the student actor can alter the text of the status view of the previous code snippet to "Not connected to teacher" when the student actor is started in the following way:

```
1 import /.at.android.undead_references;
2 deftype StudentGUI; // tag we want to retrieve
3 def guiRef := undeadRef(StudentGUI);   // ask undead reference from coordinator
4 guiRef←teacherStatus("Not connected to teacher");  // send asynchronous message
```

On line 4 we send an asynchronous message to the undead reference we retrieved on line 3. When the wrapping actor is already started, the message is immediately

forwarded to the wrapping actor. If not, the message is buffered and guaranteed to be delivered when the wrapping actor becomes available (again).

6 Preliminary Results

In this section we show the merits of the Elector model by comparing the ad-hoc implementation of the clicker application and the versions using the Elector model. But first of all, we validate the implementation of the Elector model by showing how it offers a base for a more polished version, adding more constructs that make it easier for the programmer.

6.1 Specific Improvements to Elector for Android and AmbientTalk

The implementation of the Elector model as presented in the previous section is a direct translation of the general model. This implementation solves all the problems that arise when combining two different event loops. Because the user interface code has shifted to the wrapping actors, the Android part of the application contains less code. Therefore, we introduce new constructs that ease the writing of the wrapping actors.

First of all, we introduce a `run:` construct that takes a block of code as an argument. Behind the scenes, this code is sent to the Android UI thread, hiding the UI Thread actor from the programmer. Similarly, the `listener:` construct can be used to create an event handler for Android views.

Most of the UI code for an Android application are operations like setText, setColor, setVisibility,... on an Android view. These methods alter the user interface, thus they must be executed on the UI thread of the application. We introduce the `getView` method that returns a reference to an Android View object. All setters on this reference are automatically forwarded to the UI Thread actor. The following code snippet shows how `getView` can be used by the student wrapping actor to display a question and possible answers from the teacher.

```
1 import jlobby.android.view:
2 def askQuestion(question, answers) {
3   def question_v := getView("question");
4   def previous := question_v.getText(); // retrieve current text of view
5   question_v.setText(question); // alter text
6   question_v.setVisibility(View.VISIBLE); // show the text
7 }
```

This way, the programmer can get attributes of the view and alter them inside AmbientTalk, without worrying about thread-safety.

The final extension is the support for futures [3]. AmbientTalk's event loops communicate in a non-blocking, asynchronous way, meaning that an asynchronous message send between event loops immediately returns. AmbientTalk's futures are placeholders for the actual result of a message execution. When the actual result is computed, the future is resolved and other actors can receive this result by installing an event handler on that future.

A common task in applications is to ask the user for some kind of input, and this is where futures are helpful. We extended Elector in such a way that futures can be resolved inside the code of a `run:` or `listener:` construct. This way, an AmbientTalk future can be resolved when e.g. a button is clicked. Vice versa, a text view can be linked to a future, filling in the view when the future becomes resolved.

6.2 Comparison of the Clicker Applications

We ended up with three versions of the clicker application: one without Elector, one that uses Elector and a final one that uses the extensions from previous section. The final version of the application spends less code on the mismatching concerns and contains less code overall. The first clicker version contains 512 lines of code, the second one 430 lines and the final version 367 lines of code, which is a decrease of 28% between the first and final one.

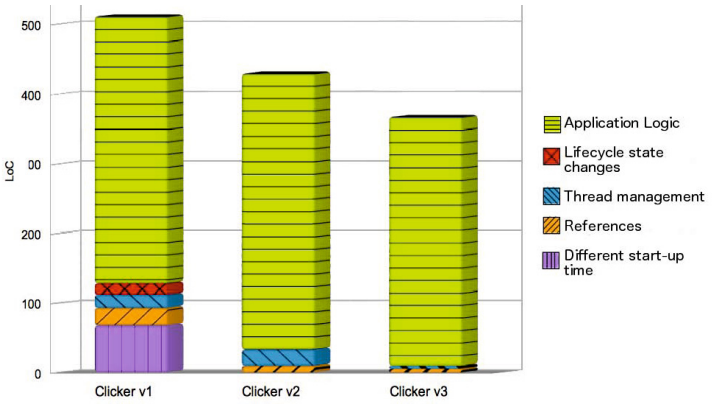


Fig. 4. Graph comparison for the first and final version of clicker

As we can see from the graphs in figure 4, Elector frees the programmer from dealing with the lifecycle of the event loops (lifecycle state changes and different start-up time). These issues are completely handled by Elector in the last two versions. Managing references and threads are reduced from 8,5% for the first version to 2,6% for the final one.

We also conducted memory and latency benchmarks. First, we measured the memory usage of the application at three different points: the choice between two roles, starting of the teacher, and starting of the student version. Starting the application uses 3.767 Mb for the first clicker and 3.671 Mb for the final one. For the teacher role we measured 5.112 Mb versus 9.293 Mb for the first and third version of Clicker respectively. For the student role these measurements are 4.82 Mb versus 7.384 Mb.

Secondly, we performed benchmarks to measure the latency of GUI updates in the final version of clicker (the second and third version are almost identical).

Therefore, we measured the latency between registering an event (clicking on a button) and the actual update of the GUI (in the Android UI code) (Scenario 1). We also measured the latency between registering the event and executing the `listener:` construct and from this point to the updating of the Android View (Scenario 2). From these experiments we obtained the following results (average and standard deviation of 200 experiments) in milliseconds:

	Scenario 1	Scenario 2	
		click → listener	listener → update
AVG +− STDEV	285.8 +− 39.73	4.55 +− 2.28	278.29 +− 47.57

Elector is a proof-of-concept implementation as a framework for AmbientTalk which we use to illustrate that our model eases the coordination of different event loops. When absolute performance is a necessity, Elector could be implemented in the AmbientTalk interpreter itself.

We can conclude that Elector had an impact on the clicker application: the programmer is freed from manually coordinating the event loops. As a side-effect, this has an impact on the code size and complexity, but moreover, more of the code now focuses on the logic of the program. This initial result is promising, but more studies are needed to further gauge the impact of Elector.

7 Conclusion and Future Work

We introduced a coordination model called Elector, that tackles the problems that arise when coordinating different event loops. Elector is targeted towards event loops that suffer from lifecycle changes (e.g. the event loops can be killed and restarted) but the model is suited for all sorts of event loops. In contrast to existing libraries, Elector allows the event loops to run separately, instead of letting one event loop take control over the other. Elector also does not change the source code of the event loops, but provides a bridge between them.

The main component of the Elector model are its undead references, the other components of the model manage these references. The undead references solve all the problems encountered when combining different event loops, namely the lifecycle and communication problems.

Using the implementation of the Elector model for the Android and AmbientTalk event loops, we evaluated the model by implementing a concrete case study: a clicker application. We compared the different versions of that application and concluded that Elector relieves the programmer from manually coordinating the event loops.

The Elector implementation is targeted towards event loops on one device, but we could extend Elector with *distributed undead references*. This way, a teacher could for example first demonstrate how the clicker application works by retrieving an undead reference to the user interfaces of the students. The other way around is

also possible: several people can retrieve an undead reference to one and the same user interface, e.g. for a brainstorming session.

An important task of the undead references of Elector is to buffer communication between event loops when one of them is not available (yet). The current implementation of Elector does not take duplication of messages into account, but we could easily support *smart buffering*. In order to achieve this, an annotation could be used, that allows the programmer to define the behaviour of messages in the buffer. So not only does Elector solve the problems that arise when combining different types of event loops, its undead references can be applied in other domains as well.

References

1. The POE Cookbook (January 2013), `http://poe.perl.org/?POE_Cookbook`
2. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge (1986)
3. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-Oriented Programming in AmbientTalk. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 230–254. Springer, Heidelberg (2006)
4. D'Inverno, R., Davis, H., White, S.: Using a Personal Response System for Promoting Student Interaction. Teaching Mathematics and its Applications 22(4), 163–169 (2003)
5. Englander, R.: Developing Java beans. O'Reilly & Associates, Inc, Sebastopol (1997)
6. Garlan, D., Allen, R., Ockerbloom, J.: Architectural Mismatch, or, Why its hard to build systems out of existing parts. In: Proceedings of the 17th International Conference on Software Engineering, Seattle, Washington, pp. 179–185 (1995)
7. Mostinckx, S., Van Cutsem, T., Timbermont, S., Tanter, E.: Mirages: behavioral intercession in a mirror-based architecture. In: Proceedings of the 2007 Symposium on Dynamic Languages, DLS 2007, pp. 89–100. ACM, New York (2007)
8. Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source Robot Operating System. In: ICRA Workshop on Open Source Software (2009)
9. Van Cutsem, T., Mostinckx, S., De Meuter, W.: Linguistic Symbiosis between Event Loop Actors and Threads. Computer Languages, Systems & Structures 35(1), 80–98 (2009)
10. Van Cutsem, T., Mostinckx, S., Gonzalez Boix, E., Dedecker, J., De Meuter, W.: AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. In: Proceedings of the XXVI International Conference of the Chilean Society of Computer Science, SCCC 2007, pp. 3–12. IEEE Computer Society, Washington, DC (2007)
11. Welch, B.B.: Practical programming in Tcl and TK, 2nd edn. Prentice Hall (1997)

Interactive Interaction Constraints*

José Proença and Dave Clarke

iMinds-DistriNet, Department of Computer Science,
KU Leuven, Belgium
{jose.proenca,dave.clarke}@cs.kuleuven.be

Abstract. Interaction constraints are an expressive formalism for describing coordination patterns, such as those underlying the coordination language Reo, that can be efficiently implemented using constraint satisfaction technologies such as SAT and SMT solvers. Existing implementations of interaction constraints interact with external components only in a very simple way: interaction occurs only *between* rounds of constraint satisfaction. What is missing is any means for the constraint solver to interact with the external world *during* constraint satisfaction.

This paper introduces *interactive interaction constraints* which enable interaction during constraint satisfaction, and in turn increase the expressiveness of coordination languages based on interaction constraints by allowing a larger class of operations to be considered to occur atomically. We describe how interactive interaction constraints are implemented and detail a number of strategies for guiding constraint solvers. The benefit of interactive interaction constraints is illustrated using two examples, a hotel booking system and a system of transactions with compensations. From a general perspective, our work describes how to open up and exploit constraint solvers as the basis of a coordination engine.

Keywords: interaction constraints, constraint satisfaction, coordination, Reo.

1 Introduction

Coordination languages facilitate the exchange of data between components (or services) externally to the operation of those components. One way of describing coordination patterns is by using *interaction constraints* [11], which originated as an approach to implementing Reo connectors [3]. In this approach, off-the-shelf constraint solvers such as Choco [18] and SAT and SMT solvers such as SAT4J [6] and Z3 [16] are used as the basis of the underlying coordination engine. The coordination engine operates in rounds, each of which proceeds by collecting constraints from components and the connector that coordinates them, and then solving the constraints. Components perform blocking reads and writes on ports, which are converted into constraints stating that they want to output

* This research is supported by the KU Leuven BOF-START project STRT1/09/031 DesignerTypeLab, Belgium, and by the FCT grant SFRH/BPD/91908/2012.

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 211–225, 2013.

or input data. A solution to the constraints describes how data flows between the components, after which some reads and writes may succeed. Each round is considered to be an *atomic* (or *synchronous*) *step*. Between rounds the states of the components and connectors may change.

One problem with the current state-of-the-art is that interaction occurs only *between* constraint solving rounds, not *during* rounds. It is impossible in Reo, for instance, to send data to a component and receive a result from it within the round, as no interaction with external components occurs during a round. This means that all data involved in the constraint is known at the start of a round, and consequently that the kinds of interaction that can be expressed using interaction constraints are limited. The challenge of introducing interaction with external components during a coordination round is that such interaction can produce externally observable behaviour. However, after solving the coordination constraints, these external observations may not correspond to what the coordination pattern aims to achieve. This implies that such actions will need to be undone after a round using rollbacks or compensations.

This paper reports on our work on *interactive interaction constraints*, which enhance a coordination engine to enable interaction during rounds. The underlying constraint solver can invoke external components several times during a round while searching for a valid solution to the coordination constraints. This can be seen as a form of negotiation. This kind of interaction can be incorporated into the visual notation of Reo using special *filter* and *transformer* channels. Filters have a predicate that is used to determine whether data flows through the channel and transformers modify the data passing through the channel by invoking a unary function. In our approach these could involve external interaction. For instance the predicate could consult an external database or by engaging in interaction with a user. Because the actual data that flows through a connector involves the solution of a potentially complex set of constraints, it is never clear when invoking a filter or a transformer whether the connector will commit to the chosen data. Thus, these channels must be implemented using a try-and-compensate mechanism, and hence the whole constraint satisfaction process becomes transactional.

Although originally stemming from research on Reo—indeed, the same visual notation can be used to describe connectors—our approach is much more general, as a greater range of coordination patterns can be implemented. Interactive interaction constraints are based on information unknown at compile time, the expressing a wider variety of coordination patterns than constraints over a fixed set of data types and operators. More generally, our approach falls within the implicit programming paradigm [17], wherein constraints specify the computation and SAT and SMT solvers perform the computation. The contribution of our work to this field is the use of constraint satisfaction to implement coordination patterns. More specifically, this paper deals with the problem of increasing the kinds of external interaction possible during constraint satisfaction.

Organisation of the Paper. The next section motivates the need for richer interaction model and identifies the main challenges. Section 3 describes the

language used to specify constraints, and Section 4 explains how they are solved. Details of the constraint-solving engine are described in Section 5. Section 6 presents how transactions are achieved with interactive interaction constraints, Section 7 presents related work, and Section 8 concludes.

2 The Need for Interaction

Interaction constraints were introduced as a bridge to providing an efficient and flexible implementation of the Reo coordination language [11]. Previous implementations based on compilation to constraint automata suffered from the problem that the entire behaviour of the connector needed to be known in advance [5]. Implementations based on connector colouring got around this problem, but these were initially not very efficient and were ultimately not very flexible as they were insensitive to data values [9]. Changizi et. al [7] extended the automata-based compilation approach with filters and transformers. These are handled by a SAT/SMT solver, though the choice of filters and transformers is limited to those expressible in the language of the solver. When building an automaton, all solutions for all states need to be found, and thus more work than necessary needs to be done and the approach is inflexible. Jongmans et al. [14] integrated external functionality by generating Java code corresponding to the automata-with-data-constraints model of Reo. The resulting code has an exponential number of formulas, without data transformations, that are checked sequentially. Interaction constraints improved on these implementations by exploiting the flexibility of constraints—again, limited only by the underlying solver—, and by permitting constraints to change dynamically and to be evaluated concurrently and partially, thereby increasing scalability as well [10].

One remaining problem is that the kind of external interaction available in the current interaction constraints-base engine is limited to blocking reads and writes. That is, considering Fig. 1, component Request interacts with the connector by performing a blocking write of some value. At some future time, this value may be accepted by the system and the write will proceed (or it may timeout). Dually, component Confirmed interacts by preforming a blocking read. At some future time, a value will become available and the read proceeds.

From Reo's perspective synchronicity corresponds to atomicity. Thus in Fig. 1 there are three possible ways data can flow (synchronously) through the connector: a request flows from Request via exactly one of $\mathsf{SrchHotel}_i$, which return a possible hotel room booking. Then it flows through filter Approve, which seeks approval from the user, transformer Book, which performs the booking, transformer Invoice, which handles the payment, filter Paid, which occurs when the payment succeeds, and finally to component Confirmed. Due to semantics of the synchronous drain (sd), dataflow through the connector is permitted only if Paid allows the data to pass. Now it is clear that such an atomic step corresponds to all steps of the hotel booking process succeeding. If any step fails, such as when the user does not approve the selection or if the payment is not made, then no flow occurs in the connector at all. To actually implement this requires a lot

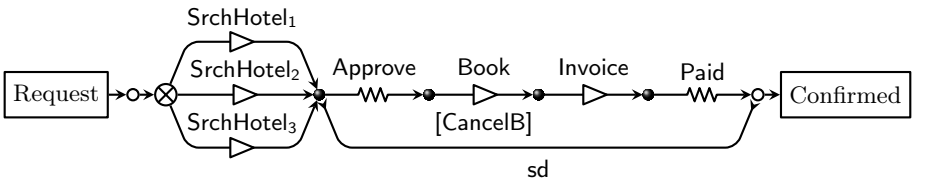


Fig. 1. Example of a Hotel Reservation workflow. Nodes (●) receive data from exactly one of their inputs, and replicate it to all of their outputs. The exclusive router (⊗) is a special node that forwards data to exactly one of its outputs. Transformer channels are represented using a triangle and filter channels using a zig-zag line.

more than interaction via blocking reads and writes, such as calling and getting results multiple times from the hotels, interacting with the user checking the payment, and retrying with different possible values from the hotels. These interactions can be realised using special functions and predicates that, whenever queried by the constraint solver, trigger calls to external components. Solving an interactive constraint might require such functions to be executed with different parameters until a valid solution is found. Whenever constraints imposed by individual elements of the connector are not satisfied, some externally observable actions, such as interaction with the user and calls to the booking site, might need to be undone, by performing a rollback or running compensation code.

Exploiting a constraint solver to implement a scenario such as the one above requires that a number of issues are properly handled:

- Constraints corresponding to the channels need to be evaluated in the right order, such as calling Approve before Book, to avoid situations where the solver guesses a value and wastefully calls some external function. External functions should only be given input that derives from concrete initial values.
- The constraint solver may (potentially, wastefully) invoke all of $\mathsf{SrchHotel}_1$, $\mathsf{SrchHotel}_2$, and $\mathsf{SrchHotel}_3$, although only one of them will be accepted. A better strategy is often to try one at a time.
- If an external function that produces an externally observable side-effect is called, such as Book, but subsequently the booking needs to be cancelled, then some compensation/rollback functionality (CancelB) needs to be performed to undo the side-effect.
- Functions that do not transform their data but do produce side-effects can be postponed until after a solution to the constraints has been found, as from the perspective of the constraint solver they are equivalent to a synchronous channel (what happens on both ends is the same).
- External functions and predicates need to be total to avoid blocking the constraint solver. Non-complying functions could easily be wrapped, such as by implementing a time-out and returning a 'no-result' value.
- Within a given round, external functions and predicates need to be deterministic to preserve the consistency of the constraint solving process. Non-complying functions can be memoized to behave as deterministic functions.

3 Coordination via Interaction Constraints

Our previous work on interaction constraints formed the basis of an efficient implementation of the Reo coordination language [11]. In this model, coordination patterns are described using logical formulas defined over two kinds of variables: *synchronisation* variables, which capture whether or not there is dataflow on a given port, and *data* variables, which describe the value that flows, when there is dataflow. Coordination takes place in rounds, and between rounds the constraints can change. From a high level perspective, each round corresponds to an atomic operation, and each solution to the interaction constraints gives the ports that synchronise and the data that flows between them.

Given a collection of ports $\mathcal{X}$, synchronisation variables $x \in \mathcal{X}$ range over booleans, and data variables $\widehat{x} \in \widehat{\mathcal{X}}$ range over a global data set $\mathbb{D}$. Formulas are defined as Dijkstra's guarded commands [12], given by the following grammar:

$$\begin{array}{rcll} \psi & ::= & \phi \to s \mid \psi_1\ \psi_2 \mid \top & \text{(formulas)} \\ \phi & ::= & x \mid P(\widehat{x}) \mid \phi_1 \wedge \phi_2 \mid \neg\phi & \text{(guards)} \\ s & ::= & \phi \mid \widehat{x} := d \mid \widehat{x}_1 := \widehat{x}_2 \mid \widehat{x}_1 := f(\widehat{x}_2) \mid s_1\,;\, s_2 & \text{(statements)} \end{array}$$

$\top$ is *true*, P is a unary predicate over data variables, and f is a unary total function. Constraint $\psi\ \psi$ is interpreted as $\psi \wedge \psi$, $s; s$ as $s \wedge s$, and $x := y$ as $x = y$. Other logical connectives can be encoded as usual. The grammar for guarded commands enforces data assignments to always be in positive positions, and ensures that the assignment operator $:=$ is asymmetric to capture the direction of dataflow. These features are not exploited in this paper, but required by the predicate abstraction technique described in Section 4.1.

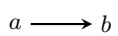
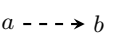
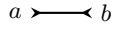
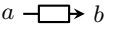
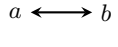
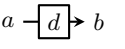
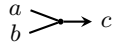
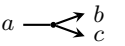
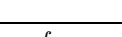
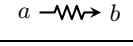
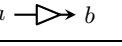
The novel addition of this paper is that functions and predicates need not be built-in functions of the logic and can be implemented externally. Thus, evaluating a function or predicate requires a call outside of the constraint solver. Furthermore, such functions and predicates may be side-effecting, and these side-effects may need to be undone. How this interactive and transactional evaluation of functions and predicates is handled is explained in detail in Section 4.

A Constraint-Based Encoding of Reo Channels. As an example of how interaction constraints are used, Table 1 recalls the encoding of the semantics of the most common Reo primitives [11]. For example, the LossySync can have dataflow on b only if data flows on a ($b \to a$), and, whenever both ports have dataflow, the data flowing on a is copied to b ($b \to \widehat{b} := \widehat{a}$). Included in the table are writers and readers: these capture the essential interaction behaviour of components that perform blocking reads and writes. The semantics of connector composition is given by conjunctively joining the constraints of its constituents.

4 Solving Interactive Constraints

As functions and predicates are defined externally, the constraint solving process requires external interaction to provide an interpretation for them as logical

Table 1. Constraint-based encoding of Reo Channel

Channel	Representation	Constraints	Channel	Representation	Constraints
Sync		$a \leftrightarrow b$ $b \rightarrow \widehat{b} := \widehat{a}$	LossySync		$b \rightarrow a$ $b \rightarrow \widehat{b} := \widehat{a}$
SyncDrain		$a \leftrightarrow b$	FIFO-E		$\neg b$
SyncSpout		$a \leftrightarrow b$	FIFO-F(d)		$\neg a$ $b \rightarrow \widehat{b} := d$
Merger		$c \leftrightarrow (a \vee b)$ $\neg(a \wedge b)$ $a \rightarrow \widehat{c} := \widehat{a}$ $b \rightarrow \widehat{c} := \widehat{b}$	Replicator		$a \leftrightarrow b$ $a \leftrightarrow c$ $a \rightarrow \widehat{b} := \widehat{a};$ $\widehat{c} := \widehat{a}$
Filter(P)		$b \rightarrow \widehat{b} := \widehat{a}$ $(a \wedge P(\widehat{a})) \leftrightarrow b$	Transf(f)		$a \leftrightarrow b$ $b \rightarrow \widehat{b} := f(\widehat{a})$
Writer(d)		$a \rightarrow \widehat{a} := d$	Reader		$\top$

formulas. This section provides a model of how this can be done. Firstly, before constraint solving, constraints are reduced to a boolean formula using a notion of predicate abstraction based on the data dependencies of each predicate. This avoids prematurely computing such predicates and functions. Then, during constraint solving the solver will evaluate predicates and functions on a per-need basis. In addition, compensations need to be performed, but as these are independent of constraint solving, their discussion is postponed until Section 5.

4.1 Pre-processing

Formulas over boolean and data variables are encoded into formulas only over boolean variables using a notion of predicate abstraction in two steps (full details are available in a separate report [21]):

1. starting from predicates, calculate all data variables that may contribute to its value by tracing back to sources of data; and
2. replace data variables by new boolean variables, one for each predicate that is *reachable*, based on the sets calculated on the previous step.

These steps are illustrated using the examples in Fig. 2. In the first step, every path backwards from a predicate to a data source is determined based on the data assignments in formulas, yielding paths numbered from 1 to 5. Each of these paths is denoted by a new boolean variable. In the second step, data variables are replaced by boolean variables. Each new variable captures whether the associated predicate holds when applied to the given data. For instance, path 2 is represented by variable $\widehat{x}_{R.f}$, which denotes whether $R(f(\widehat{x}))$ holds. In the underlying constraints, $\widehat{x}$ is replaced by 4 new boolean variables $\widehat{x}_P$, $\widehat{x}_Q$,

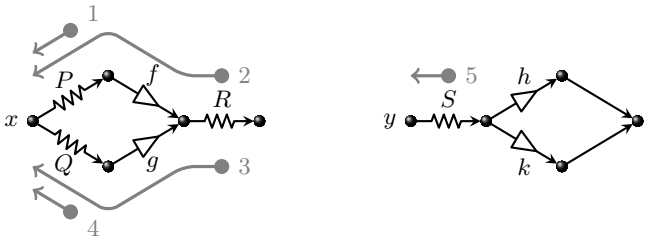


Fig. 2. Simple examples to illustrate predicate abstraction

$\widehat{x}_{R.f}$ and $\widehat{x}_{R.g}$. Similarly $\widehat{y}$ is replaced by $\widehat{y}_S$, and so forth. Data assignments are modified to work on these new variables. For example, given variables $\widehat{x}_{P.f}$ and $\widehat{y}_{P.f}$, the encoding of $\widehat{y} := \widehat{x}$ includes $\widehat{y}_{P.f} := \widehat{x}_{P.f}$. Given variables $\widehat{x}_{P.f}$ and $\widehat{z}_P$, the encoding of $\widehat{z} := f(\widehat{x})$ includes $\widehat{z}_P := \widehat{x}_{P.f}$. Predicates in formulas are replaced by the corresponding variable: for instance, $P(\widehat{x})$ is replaced by $\widehat{x}_P$. Finally, data assignments $\widehat{x} := d$ are encoded as a conjunction of special constraints, called *external predicates*:

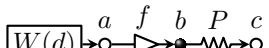
$$\bigwedge_{p \text{ reaches } x} \mathsf{XPred}(p, x, d) \qquad \text{(usage of external predicates)}$$

The reachability mentioned in the formula above is the same as in Fig. 2, and, in this case, p ranges over $R.f$, $R.g$, P and Q. Each $\mathsf{XPred}(p, x, d)$ in this formula is a wrapper guarding the evaluation of $p(d)$. This constraint is equivalent to the one below, but it is evaluated during constraint solving using dedicated functions.

$$\neg x \ \wedge \ \neg y \ \wedge \ \widehat{x}_p = \mathsf{eval}(p(d)) \qquad \text{(interpretation of an external predicate)}$$

where y is the port to which the predicate p is applied, after dropping all the associated functions, and eval performs the computations needed to evaluate a predicate and its associated functions. For instance, the external predicate $\mathsf{XPred}(R.f, x, d)$ is interpreted as $\neg x \ \wedge \ \neg r \ \wedge \ \widehat{x}_{R.f} = \mathsf{eval}(R(f(d))$, where r is the port between the transformer f and the filter R. This means that the value of $\widehat{x}_{R.f}$ only reflects the result of $R(f(d))$ when both x and r have dataflow.

For example, the original and abstracted constraints of the connector on the right are, respectively:



$$\begin{array}{c} a \ \rightarrow \ \widehat{a} := d \qquad b \ \rightarrow \ \widehat{b} := f(\widehat{a}) \\ a \leftrightarrow b \quad c \ \rightarrow \ \widehat{c} := \widehat{b} \quad (b \wedge P(\widehat{b})) \ \leftrightarrow \ c \end{array} \quad \Big| \quad \begin{array}{c} a \rightarrow \mathsf{XPred}(P, b, d) \qquad b \ \rightarrow \ \widehat{b}_P := \widehat{a}_{P.f} \\ a \leftrightarrow b \qquad (b \wedge \widehat{b}_P) \ \leftrightarrow \ c \end{array}$$

Since no predicate reaches c, no variables are created for c and $\widehat{c} := \widehat{b}$ is dropped.

An alternative to using predicate abstraction is to use an encoding into constraints over a fixed domain that are used as hashes for the actual values or functions that need to be computed. Following this line of thought, we are currently investigating the advantages of encoding into integer constraints, making a tradeoff between the number of variables used at the cost of a more expensive underlying constraint solver. These ideas are left as future work.

4.2 Evaluation Model

Fig. 3 presents a model of constraint solving over booleans, following the style of Apt [2], adapted to our setting. These rules rely on two core functions: *propagate* and *satisfy* ($\models$), which will be defined later for external predicates to control when functions and predicates are evaluated.

$$\frac{\langle c_1, \ldots, c_n \ ; \ V_1, x \mapsto \{\top, \bot\}\rangle}{\langle c_1^x, \ldots, c_n^x \ ; \ V_{n+1}^x\rangle \ \ \langle c_1^{-x}, \ldots, c_n^{-x} \ ; \ V_{n+1}^{-x}\rangle} \ (\textsc{branch}) \qquad \begin{array}{l} \textit{where, for } 1 \le i \le n: \\ (c_i^x \ , V_{i+1}^x) = \textit{propagate}(c_i, V_i^x) \ ; \\ (c_i^{-x}, V_{i+1}^{-x}) = \textit{propagate}(c_i, V_i^{-x}) \end{array}$$

$$\frac{\langle c, C \ ; \ V\rangle}{\langle c', C \ ; \ V'\rangle} \ (\textsc{propagate}) \quad \textit{where } (c', V') = \textit{propagate}(c, V)$$

$$\frac{\langle c, C \ ; \ V\rangle}{\langle C \ ; \ V\rangle} \ (\textsc{satisfy}) \quad \textit{if } V \vDash c$$

$$\frac{\langle c, C \ ; \ V\rangle \ \overline{\langle C' \ ; \ V'\rangle}}{\overline{\langle C' \ ; \ V'\rangle}} \ (\textsc{prune}) \quad \textit{if } V \nvDash c$$

Fig. 3. Semantics of the constraint solver. $V^x = V[x \mapsto \top]$ and $V^{-x} = V[x \mapsto \bot]$, where $V[x \mapsto b]$ denotes the update of V by mapping x to $\{b\}$, for $b \in \{\bot, \top\}$.

Formally, a CSP over booleans is a pair $\langle C; V\rangle$ of constraints and variable domains. The constraints are initially the conjunctive set of guarded commands of a connector. The variable domains initially map each variable to $\{\top, \bot\}$. Constraint satisfaction proceeds by branching over variables and by simplifying constraints based on the current domains of variables. Branching over a variable x means creating two new CSPs, one assuming x is true and one that x is false. Simplification of constraints is performed by a *propagate* function, which takes a constraint c and a variable domain V and builds, when possible, a simpler constraint c' and a smaller domain variable V' where some variables become instantiated with $\{\top\}$ or $\{\bot\}$. Satisfaction of a constraint c is determined using the operator $\models$, based on the instantiated variables of domain V.

The implementation of *propagate* and *satisfaction* for external predicates $\mathsf{XPred}(P, x, d)$ are as follows:

$$\textit{propagate}(\mathsf{XPred}(P,x,d), V) = \begin{cases} (\top, V) & \textbf{if } V(x) = \bot \textbf{ or } V(y) = \bot \\ (\bot, V) & \textbf{if } V(x) = V(y) = \top \textbf{ and } \mathsf{eval}(P(d)) \notin V(\widehat{x}_P) \\ (\top, V[\widehat{x}_P \mapsto \top]) & \textbf{if } V(x) = V(y) = \top \textbf{ and } \top \in V(\widehat{x}_P) \textbf{ and } \mathsf{eval}(P(d)) \\ (\top, V[\widehat{x}_P \mapsto \bot]) & \textbf{if } V(x) = V(y) = \top \textbf{ and } \bot \in V(\widehat{x}_P) \textbf{ and } \neg\mathsf{eval}(P(d)) \\ (\mathsf{XPred}(P,x,d), V) & \textbf{otherwise} \end{cases}$$

$$V \vDash \mathsf{XPred}(P,x,d) = \begin{cases} \top & \textbf{if } V(x) = \bot \textbf{ or } V(y) = \bot \\ \widehat{y}_P = \mathsf{eval}(P(d)) & \textbf{if } V(x) = V(y) = \top \\ \textit{unknown} & \textbf{otherwise} \end{cases}$$

Observe that if $V \vDash c$ returns *unknown*, then neither $V \vDash c$ nor $V \nvDash c$ hold.

The solver will still control when to instantiate any of the variables used by XPred, which are guided using a search strategy (see the next section).

5 Implementation

We have implemented a prototype coordination engine that handles external predicates based on the Choco constraint solver.[1] It focuses on the boolean satisfaction problem of the formulas obtained via predicate abstraction. Contrary to most other SAT and SMT solvers,[2] Choco allows user-defined function and predicates, and the customisation of strategies. These capabilities are exploited to control the evaluation of external predicates. As a running example we use a simple implementation of the Hotel Reservation system (Fig. 1). External predicates are implemented to read from and write to the command line.

5.1 Caching and Compensating

To avoid redoing complex calculations or performing the same query twice, whenever a function or predicate is evaluated its result is cached. This ensures that all functions are deterministic in any given round. Similarly, functions and predicates that require human interaction are executed at most once per argument, per round.

After each round of constraint solving the cache is cleared. During this process the engine checks which values contributed to the solution. Every cached value that did not contribute is reverted using its associated compensation, if it exists.

5.2 Strategies

By default Choco uses a branching strategy that selects the next variable to be analysed based on its current domain size, the number of uninstantiated constraints involving the variable, and the sum of some counters associated with these constraints [18]. For each selected variable, Choco starts by assigning the smallest value (*false* in the case of booleans), increasing it whenever necessary.

We propose the use of an alternative strategy, built using Choco strategy constructors,[3] that selects variables using a fixed order and assigns *false* values before assigning *true*. This order of variables is produced based on the possible paths of dataflow, which have been partially calculated during the pre-processing of the constraints (Section 4.1). The intuition is that if a path with dataflow that satisfies the constraints is found, the external predicates on the remaining paths do not need to be evaluated. More concretely, we use the following guidelines:

[1] `http://www.emn.fr/z-info/choco-solver/`

[2] We also experimented with SAT4J and Z3 solvers, among others.

[3] These constructors are called `AssignVar`, `StaticVarOrder`, and `IncreasingDomain`.

– Approximate the data dependency graph and linearise the resulting ordering.
– Branch first over synchronisation variables and only later over the rest.

In the Hotel Reservation system, this strategy can avoid the booking, invoicing, and payment of hotels that are not approved. The choice of which function and predicate is evaluated first is non-deterministic: it depends on the order of concatenation of traversals during the linearisation process. In this case, a possible linearisation is: $req \leftarrow h_1 \leftarrow h_1 out \leftarrow hs \leftarrow ap \leftarrow bk \leftarrow inv \leftarrow paid \leftarrow h_3 \leftarrow h_3 out \leftarrow h_2 \leftarrow h_2 out$. Thus the constraint solver will first select and branch the variable $h_2 out$, then the variable h_2, and so on. The data variables are selected only after the synchronous variables. The ports h_i and $h_i out$ are the two ports of the filter $\mathsf{SrchHotel}_i$, and the ports $ap, bk, inv, paid$ succeed the channels Approve, Book, Invoice, and Paid, respectively.

5.3 Scala Implementation

We have developed a set of libraries for the Scala language[4] to easily specify interactive constraints.Scala is fully interoperable with Java, hence our libraries can also be used to define and run connectors using Java.

```
object HotelReservation extends App {

  case class Req(val content:String)

  def srchHotel(i:Int) =
        Function("SearchHotel-"+i){
    case r:Req => i match {
      case 1 => List("F1","Ibis","Mercury")
      case 2 => List("B&B","YHostel")
      case _ => List("HotelA","HotelB")
    }
  }
  val approve = Predicate("approve"){
    case l:List[String] =>
      println("approve: "+l.mkString(",
            ")+". [y,n]")
      readChar() == 'y'
  }
  val book = Function("book"){
    case l : List[String] =>
      println("Options: "+l.mkString(", ")+
        ". Which one? (1.."+l.length+")")
      val res = readInt()
      l(res-1)
  }
  val cancelB = Function("cancelB"){
    case x => println("canceling "+x+".")
  }
  val invoice = Function("invoice"){
    case x => println("invoice for "+x+".")
  }
```

```
  val pay = Predicate("paid"){
    case x => if (x == "Ibis") {
      println("paid for Ibis")
      true
    }
    else {
      println("not paid for "+x)
      false
    }
  }



  // Connector definition
  val connector =
    writer("req",List(Req("req1"),
          Req("req2"))) ++
    nexrouter("req",List("h1","h2","h3")) ++
    transf("h1","h1o",srchHotel(1)) ++
    transf("h2","h2o",srchHotel(2)) ++
    transf("h3","h3o",srchHotel(3)) ++
    nmerger(List("h1o","h2o","h3o"),"hs") ++
    filter("hs","ap",approve) ++
    sdrain("hs","ap") ++
    transf("ap","bk",book,cancelB) ++
    monitor("bk","inv",invoice) ++
    filter("inv","paid",paid) ++
    reader("paid",5)

  connector.run()
}
```

Listing 1: Scala code for the Hotel Reservation system

[4] http://www.scala-lang.org

The code for the Hotel Reservation system is presented in Listing 1. The code consists of a single object `HotelReservation`, which defines a `Request` inner class, a method or constant value that returns an instance of a `Predicate` or `Function` for each predicate and function, and a connector defined as the composition of 14 sub-connectors. The sub-connector `monitor` is a channel with a function that has side-effects but does not transform data. Instances of the `Predicate` and `Function` classes can be equally created using class inheritance, defining the methods `check` and `calculate`, respectively. The last line of the listing starts the connector running. This triggers the consecutive execution of rounds until a state with no solutions is reached. The code that interacts with the user via command line is highlighted. The documentation of the API can be found online.[5]

6 Example: Transactional Connectors

This section presents three example connectors that coordinate transactions with compensations. The examples are based on a chain of pairs (F_i, F_i^{-1}), where F_i is some operation and F_i^{-1} is a compensation that undoes the effect of F_i. A successful transaction will pass data through each F_i in succession. If any intermediate step fails, then all compensations up to that point need to be run, in reverse order. Thus, if F_1 and F_2 succeed, but F_3 fails, then F_3^{-1}, F_2^{-1}, and F_1^{-1} need to be run. The first example is based on traditional Reo connectors and external components. Traditionally in Reo, external components operate asynchronously and no external interaction occurs during the constraint solving process. The second example is an adaptation of the first where the asynchronous external components are replaced with synchronous transformer channels that encapsulate the external interaction. The third example internalises the compensation behaviour so that it is only accessible to the engine.

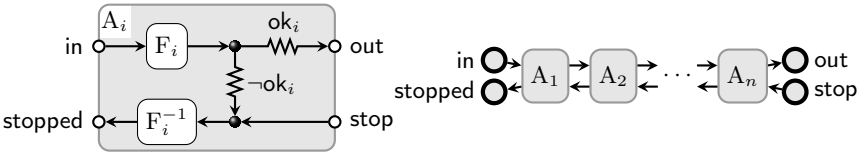


Fig. 4. Asynchronous transactions in Reo

Fig. 4 presents a traditional Reo connector for coordinating the transaction and its compensation. The left-hand side shows how to coordinate a pair of components (F_i, F_i^{-1}), and the right-hand side shows how these can be composed sequentially to form a larger transaction. Each connector A_i passes data input on port in to F_i. In a subsequent step, F_i returns a result. If this satisfies the filter ok_i, the value is passed to out, otherwise it is passed to F_i^{-1}. In addition, a value

[5] `http://people.cs.kuleuven.be/~jose.proenca/reopp/doc`

can come from port stop and be passed to F_i^{-1} to indicate that the transaction failed upstream. They key point is that all F_i and F_i^{-1} are asynchronous as far as the Reo connector is concerned. Consequently, each part of the chain runs in a separate round, and nothing guarantees that all parts will run. Thus the connector does not really enforce that the transaction is atomic.

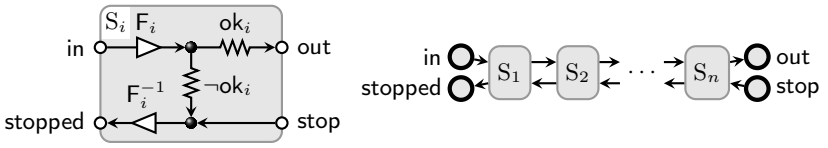


Fig. 5. Synchronous transactions via transformers

Fig. 5 presents a revised version of the connector that uses synchronous transformer channels F_i and F_i^{-1} instead of components, as in the previous example. These transformers perform the same external operations as their counterparts above, but now they can be handled by the constraint engine. A consequence of the fact that they are synchronous is that the entire connector S_i is synchronous. Indeed, the entire chain in the right-hand side of the figure is synchronous, thus atomicity is regained.

Fig. 6. Synchronous transactions with built-in compensations

But we can do better. Fig. 6 shows an improved version. In this version, each transformer is modified so that the compensation action F^{-1} is built into the transformer and is run by the engine whenever the transaction fails. The semantics of the connector is that it permits flow on all ports and only on all ports. So the only possible flows permitted are the ones where each transaction succeeds. In cases where this is not possible, such as when some ok_i is false, the engine rollbacks all transformers through which data has passed by running their compensations.

The main differences between the three approaches are summarised as follows: the first approach takes a multiple number of rounds to complete the transaction, while the second and third approaches take only one round; and the running of the compensation is handled by the connector in the first and second approaches, but by the engine in the third approach. Two consequences of having multiple rounds are that the intermediate steps are observable to the external world and the transaction may get stuck in the middle. By compressing everything into a single round, these problems are avoided, and the only observables are completed transactions (modulo the fact that some actions are

undone using compensations). Having the connector handle the running of the compensations is potentially error-prone, even though it introduces a degree of flexibility. Handling compensations within the engine simplifies the connector and shifts responsibility for correctness to the engine.

An alternative approach to using Reo to coordinate (long-running) transactions was presented by Kokash et al. [15]. This resembles the first approach above, though the connector was more complicated as it also permitted the transaction to be cancelled externally.

7 Related Work

Traditional approaches to implementing Reo [5,7,14] are based on pre-computing an automaton describing all future behaviour of the Reo connector. This typically performs more work than is necessary and is rather inflexibility, specifically because it eliminates all intensional information about the connector. Our approach is based on dynamically generating and solving logical constraints [11,8]. This permits more control over intensional aspects during runtime, which allows more refined interaction with external components than was previously possible.

Montanari and Rossi express coordination as a constraint satisfaction problem, in a similar and general way [20]. They view networks as graphs, and use the tile model to distinguish between synchronisation and sequential composition of the coordination pieces. In our approach, we explore a more concrete coordination model, which not only captures the semantics of Reo, but also extends it with external interaction, not found in Montanari and Rossi's work.

Minsky and Ungureanu introduced the Law-Governed Interaction (LGI) mechanism [19], implemented in the Moses toolkit. The mechanism targets distributed coordination of heterogeneous actors, enforcing laws that are defined using constraints in a Prolog-like language. The main innovation is the enforcement of laws by certified controllers that are not centralised. Their laws, as opposed to our approach, are not global, allowing them to achieve good performance, while compromising the scope of the constraints. Communication between actors governed by laws and actors outside LGI is possible, but not the execution of side-effecting code while checking constraints.

Abreu and Fiadeiro explore the coordination of interactions in service-oriented systems using SRML, a Service Modelling Language [1]. Services are linked with each other by connecting ports and referring to the protocol used to connect them. SRML operate at the abstraction level of business modelling, using asynchronous message passing and explicitly supporting service discovery. Our approach differs from theirs as in our work orchestration is guided by a constraint solving process, interaction with services is transactional, and our global constraints express more coordination patterns.

Our work falls within the implicit programming paradigm. Köksal et al. proposed similarly to integrate the power of declarative SAT/SMT solvers non-intrusively into sequential, imperative programs [17]. In contrast to this work, our approach targets coordination languages, and depends upon a constraint solver enhanced with interaction as a side-effect.

Faltings et al. [13] explore interactive constraint satisfaction, which is a framework for open constraint satisfaction in a distributed environment that enables constraints to be added on-the-fly. There are a number of key differences between our work and theirs. The first is that our work focuses on the coordination of components, separating the computation and coordination aspects, whereas they aim purely at constraint satisfaction. Secondly, our work allow functions and predicates to be defined externally to the constraint solver, whereas their approach allows instead on-the-fly constraint generation. The former requires the management of compensations for any external interaction that is not committed to, where as the latter does not.

8 Conclusion and Future Work

This paper expanded upon the use of constraint solving as the basis of an engine for coordinating components by introducing support for external interaction during the constraint solving process. For this to make sense in terms of externally observable behaviour, certain calls to functions and predicates required rollback or compensation to undo their effect. This means that the rounds of constraint solving become transactional. In contrast to previous implementation approaches for Reo connectors, our approach increases the degree of external interaction possible in a connector, and transactional behaviour can be expressed much more concisely as a consequence. In effect, we have lifted Reo's notion of synchrony as atomicity to synchrony as transactional atomicity. This means that Reo's synchronous connector semantics can be used to (visually) express transactional behaviour, and interactive interaction constraints supply the bridge to the underlying implementation.

As future work we first plan to experiment with heuristics to better guide the constraint solver. One approach is to avoid the pre-processing phase by reducing the original constraints over arbitrary data values to an SMT problem of a simple theory. This approach will allow some external functions and predicates to be internalised within the engine, thereby avoiding the need for rollback/compensation. Secondly, we will combine the techniques described in this paper with our earlier work on partial connector colouring [10], which optimises the constraint satisfaction process by admitting partial solutions to the constraints. This optimisation was experimentally demonstrated to increase scalability of the engine. Finally, we plan to integrate our implementation into the existing open source ECT tools for Reo [4], thereby making it available to developers.

References

1. Abreu, J., Fiadeiro, J.L.: A coordination model for service-oriented interactions. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 1–16. Springer, Heidelberg (2008)
2. Apt, K.: Principles of Constraint Programming. Cambridge University Press (2003)
3. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)

4. Arbab, F., Koehler, C., Maraikar, Z., Moon, Y.-J., Proença, J.: Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. In: Proceedings of FACS (2008)
5. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Science of Computer Programming 61(2), 75–113 (2006)
6. Berre, D.L., Parrain, A.: The Sat4j library, release 2.2. JSAT 7(2-3), 59–64 (2010)
7. Changizi, B., Kokash, N., Arbab: A constraint-based method to compute semantics of channel-based coordination models. In: ICSEA: Proceedings of the International Conference on Software Engineering Advances (2012)
8. Clarke, D.: Coordination: Reo, nets, and logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 226–256. Springer, Heidelberg (2008)
9. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: Synchronisation and context dependency. Science of Computer Programming 66(3), 205–225 (2007)
10. Clarke, D., Proença, J.: Partial connector colouring. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 59–73. Springer, Heidelberg (2012)
11. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Channel-based coordination via constraint satisfaction. Science of Computer Programming 76 (2011)
12. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)
13. Faltings, B., Macho-Gonzalez, S.: Open constraint programming. Artificial Intelligence 161(1-2), 181–208 (2005)
14. Jongmans, S.-S.T.Q., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Automatic code generation for the orchestration of web services with reo. In: De Paoli, F., Pimentel, E., Zavattaro, G. (eds.) ESOCC 2012. LNCS, vol. 7592, pp. 1–16. Springer, Heidelberg (2012)
15. Kokash, N., Arbab, F.: Applying Reo to service coordination in long-running business transactions. In: Shin, S.Y., Ossowski, S. (eds.) SAC, pp. 1381–1382. ACM (2009)
16. Köksal, A.S., Kuncak, V., Suter, P.: Scala to the power of Z3: Integrating smt and programming. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 400–406. Springer, Heidelberg (2011)
17. Köksal, A.S., Kuncak, V., Suter, P.: Constraints as control. SIGPLAN Not. 47(1), 151–164 (2012)
18. Laburthe, F., Jussien, N.: CHOCO solver documentation (August 2012), http://sourceforge.net/projects/choco/files/choco/2.1.5/choco-2.1.5/choco-doc-2.1.5.pdf
19. Minsky, N.H., Ungureanu, V.: Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. ACM Transactions on Software Engineering and Methodology 9(3), 273–305 (2000)
20. Montanari, U., Rossi, F.: Modeling process coordination via tiles, graphs, and constraints. 3rd Biennial World Conference on Integrated Design and Process Technology 4, 1–8 (1998)
21. Proença, J., Clarke, D.: Solving data-sensitive coordination constraints. CW Reports CW637, Department of Computer Science, KU Leuven (February 2013)

Towards Distributed Reactive Programming

Guido Salvaneschi, Joscha Drechsler, and Mira Mezini

Technische Universität Darmstadt
lastname@informatik.tu-darmstadt.de

Abstract. Reactive applications is a wide class of software that responds to user input, network messages, and other events. Recent research on reactive languages successfully addresses the drawbacks of the Observer pattern – the traditional way reactive applications are implemented in the object-oriented setting – by introducing time-changing values and other ad-hoc programming abstractions.

However, those approaches are limited to local settings, but most applications are distributed. We highlight the research challenges of distributed reactive programming and present a research roadmap. We argue that distributed reactive programming not only moves reactive languages to the distributed setting, but is a promising concept for middleware and distributed systems design.

Keywords: Functional-reactive Programming, Scala, Event-driven Programming.

1 Introduction

Reactive applications respond to user input, packets from the network, new values form sensors and other external or internal events. Traditionally, reactive applications are implemented by using the Observer design pattern. The disadvantages of this style, however, have been studied for a long time now [7,15,13]. The major points of criticism include verbosity, lack of static reasoning, and inversion of the logic flow of the application.

A first solution proposed by researchers is to provide language-level support for the Observer pattern. For example, in C# classes can expose events to clients beside fields and methods. Other languages that provide advanced event systems include Ptolemy [16] – which supports event quantification – and EScala [9] – which supports event combination and implicit events.

A different approach is adopted by *reactive languages* that directly represent time-changing values and remove inversion of control. Among the others, we mention Fr-Time [7] (Scheme), FlapJax [15] (Javascript), AmbientTalk/R [12] (Smalltalk) and Scala.React [13] (Scala). This idea originates from functional reactive programming, which explored the semantics of time-changing values in the setting of (strict) functional languages. Another source of inspiration for reactive languages can be found in illustrious ancestors like the Esterel [4] and Signal [10] dataflow languages that focus on realtime constraints. Reactive languages are a promising solution and active research is ongoing in this field.

Reactive languages require careful design to avoid inconsistencies – referred to as *glitches* [15] – that can arise in the update process of time-changing values. All reactive

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 226–235, 2013.

languages proposed so far do not provide these guarantees in the distributed setting. However, applications rarely span over a single host. Rather, most applications are distributed – the client-server architecture being the simplest case.

In summary, distributed reactive programming is an open challenge. Existing reactive languages take into account distribution in the sense that they are capable of transmitting events or propagating function calls among remote hosts that independently run a reactive system. In this way, however, consistency properties, locally enforced by a reactive programming system, are not guaranteed over reactive data flows that cross hosts [2]. We show that this limitation can lead to serious errors in the behavior of applications. Despite reactive programming having huge potential as a means to coordinate complex computing systems in a simple way, it struggles to succeed in this context. We argue that the aforementioned limitation is one reason for this.

The contributions of this paper are the following:

- We analyze the drawbacks of the existing approaches and motivate the need for reactive languages that uphold their guarantees in distributed applications.
- We propose a research roadmap for distributed reactive programming, including a first solution that demonstrates the feasibility of this goal and can be used for further design refinement and optimization.

Our vision is that distributed reactive programming can be a new way of structuring distributed systems. It centers around the concept of *Remote Reactives*, remote entities characterized by time-changing values, which clients can compose to model reactive computations and effects that span across multiple hosts.

2 Motivation

In this section, we introduce the basic concepts of reactive programming, show its advantages over traditional techniques, and describe the challenges of achieving the same result in a distributed setting.

Reactive Programming in a Nutshell. Reactive languages provide dedicated abstractions to model time-changing values. Time-changing values are usually referred to as *behaviors* [7] or *signals* [13]. For example, in Figure 1 we show a code snippet in Scala.React [13] that highlights the mouse cursor when it hits the borders of a window. In Scala.React, signals are time-changing values that are updated automatically; vars are time-changing values that are directly modified. In the example, the `position` signal models the time-changing value of the mouse position (Line 1). In Line 2, the `showCursorChoice` var is declared, which is changed by the user settings (not shown) to enable or disable the highlighting feature. In the rest of the paper, we generically refer to vars and signals as *reactives*. Developers can build expressions upon reactives. In Line 3 we create the `isMouseOnBorders` signal. It depends on the `position` signal and evaluates to true if the current position overlaps the window border. The key point is that reactive values that depend on other reactive values are automatically updated when any of their dependencies change. For example, when the position of the mouse changes,

```
1 val position: Signal[(Int,Int)] = GUI.getMouse().position
2 val showCursorChoice: Var[Boolean] = Var(false)
3 val isMouseOnBorders = Signal{ overlap(position(), GUI.getWindowBorders) }
4 val shouldHighlight = Signal{ isMouseOnBorders() && showCursorChoice() }
5 observe(shouldHighlight) { value : Boolean => setHighlightVisible(value) }
```

Fig. 1. An example of reactive programming

any expression associated with the `position` signal is reevaluated. In our example, the value of `isMouseOnBorders` is recalculated. This in turn causes the expression defined at Line 4, which combines both the `isMouseOnBorders` signal and the `showCursorChoice` var, to be reevaluated. Finally, every time this value changes, it is used to update the highlighting state through the `observe` statement in Line 5.

A detailed comparison of reactive programming with the Observer pattern is out of the scope of this paper. The advantages of this style over the Observer pattern have been extensively discussed in literature. The interested readers can refer for example to [7,15,13]. The main points are summarized hereafter. Callbacks typically have no return type and change the state of the application via side effects. Instead, reactives enforce a more functional style and return a value. As a result, types can guide developers and prevent errors. More importantly, differently from callbacks, reactives are composable, enhancing maintainability and software reuse. Finally, callbacks invert the logic of the application. Instead, reactives express dependencies in a direct way, improving readability and reducing the boilerplate code for callback registration.

Challenges of the Distributed Setting. As we have shown, reactive programming is an appealing solution to implement reactive applications. As such, it is desirable to move the advantages of reactive programming to a distributed setting, in the same way as, historically, functions were generalized to remote procedure calls and local events inspired publish-subscribe distributed middleware.

To discuss the challenges of this scenario, we present SimpleEmail, a minimal email application. In SimpleEmail, the server keeps the list of the emails received for an account. The client can request the emails that were received within the last n days and contain a given word and display them on the screen. The client highlights the desired word in the text of each email. Since an email text can span over multiple screen pages, it can be the case that there is no highlighted word on the current page, even if the server returned that email. To make this case less confusing for the user, the client displays a warning message. A proof-of-concept implementation is presented in Figure 2. We assume that reactive values can be shared with other hosts by publishing them on a registry, similarly to Java RMI remote objects. These objects are referred to as *Remote Reactives* (RRs). Clients can obtain a reference to RRs from the registry. For simplicity, we do not distinguish between remote vars and remote signals and we assume that a RR can be changed only by the client that publishes it.

The client (Figure 2a) publishes a RR that holds the word to look for (Line 2a) and another one with the number of days n (Line 4a). These reactives are automatically updated when the user inserts a new word or a new number in the GUI. The server (Figure 2b) retrieves the word and the days (Lines 5b and 7b) and uses these values to

filter the stored emails (Lines 9b-14b). It publishes the RR that holds the result (Line 16b), which is in turn looked up by the client (Line 7a) and displayed in the GUI (Line 9a). The client additionally combines the filtered list of emails with the searched word to determine whether the word is visible on the first page (signal `hInFirstPage`, Line 12a). This value is then used to decide, whether or not to display the alert (Line 16a) informing the user that the search term is only visible on a different page.

Figure 3 models the dependencies among the reactive values in the SimpleEmail application (the c – respectively s – subscript refers to the client – respectively server – copy of a RR). Suppose, the user now inputs a different word. In the program, this would update the $\texttt{days}_c$ reactive. This change would be propagated to the server's instance $\texttt{days}_s$ of the reactive, where it causes the reevaluation of the $\texttt{filteredEmails}_s$ reactive. The new list of emails is then filtered with the current $\texttt{word}_s$ search term and the result is propagated back to the client to updates the display. Now suppose that, instead of $\texttt{days}_c$, $\texttt{word}_c$ is updated. This change is likely to propagate to $\texttt{hInFirstPage}_c$ first, because the propagation of the value to $\texttt{word}_s$ requires network communication. As a result, $\texttt{hInFirstPage}_c$ is evaluated with the *new* word, but the *old* list of emails. This can result in the word not being found on the current page and, in consequence, the warning message being displayed. Only later, when the change propagated through $\texttt{word}_s$, $\texttt{filteredEmails}_s$ and back to $\texttt{filteredEmails}_c$, $\texttt{hInFirstPage}_c$ reevaluates to false and the warning is switched back off.

In summary, spurious executions can be generated, depending on the evaluation order of the values in the graph. In our example, this issue only led to the erroneous display of a warning message. However, it is easy to see that in a real system that involves physical actuators, temporary spurious values can have more serious consequences. For example, in a train control system, a rail crossing could be temporarily opened, letting cars cross the railway while a train is approaching.

Temporary inconsistencies due to the propagation order are a known problem in reactive languages, commonly referred to as *glitches* [7]. Current reactive languages achieve glitch freedom by keeping the dependency graph of the reactive values topologically sorted; reevaluations are triggered in the correct order, avoiding spurious values. More details on this technique can be found in [15,13]. Current reactive languages, however, enforce glitch freedom only locally, but give no guarantees when the communication spans over several hosts as in the presented example. The fundamental point is, that true distributed reactive programming cannot be achieved by naively *connecting the dots* among single (individually glitch-free) reactive applications. On the contrary, dedicated logic must ensure glitch freedom as a global property of the entire system. Re-using topological sorting in the distributed setting would however force a single-threaded, centralized execution of updates across the entire application – an approach that we consider unacceptable. Next to finding and implementing a suitable remoting mechanism for reactives, developing an acceptable solution for ensuring glitch freedom in a distributed reactive application is the main challenge in supporting distributed reactive programming.

```
1  val word: Signal[String] = GUI.wordInput
2  publishRR{"word", word}
3  val days: Signal[Int] = GUI.daysInput
4  publishRR{"days", days}
5
6  val filteredEmails: Signal[List[Email]]] =
7    lookupRR("filteredEmails")
8
9  observe(filteredEmails) { showEmails(_); }
10 observe(word) { setHighlightedWord(_); }
11
12 val hInFirstPage : Signal[Boolean] = Signal{
13   isInFirstPage(word(), filteredEmails())
14 }
15
16 observe(hInFirstPage) {
17   setShowHighlightOnNextPageWarning(_)
18 }
```

(a)

```
1  val allEmails : Signal[List[Email] =
2    Database.emails
3
4  val word: Signal[String] =
5    lookupRR{"word"}
6  val days: Signal[Int] =
7    lookupRR{"days"}
8
9  val filteredEmails: Signal[List[Email]] =
10   Signal{ allEmails().filter( email =>
11     email.date < (Date.today() − days())
12     &&
13     email.text.contains(word())
14   ) }
15
16 publishRR{"filteredEmails", filteredEmails}
17
18
```

(b)

Fig. 2. The SimpleEmail application. Client-side (a) and server-side (b).

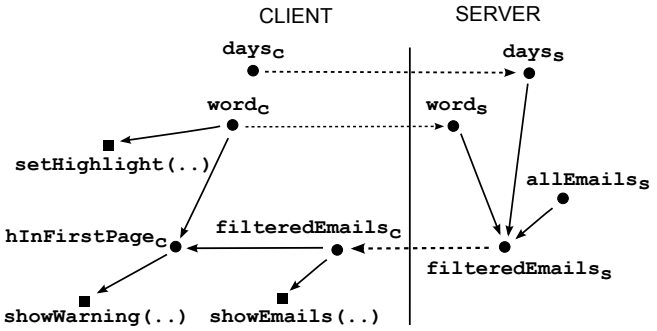


Fig. 3. Dependency graph among reactive values in the SimpleEmail application. Dashed arrows indicate over-network dependencies, square nodes indicate observers with side effects.

3 A Research Roadmap

In this section, we design an initial solution for distributed reactive programming, describe our plans to improve upon it, and present possible application domains.

Distributed Reactive Programming. We formulate our solution as a distributed transaction [18], for which well-known algorithms are available (e.g. distributed 2PL). Since persistency is not necessarily required, an implementation can leverage distributed software transactional memories [17], which have been proved to efficiently scale up to large-scale clusters [5]. A promising alternative for the implementation of our system is the automatic code generation from a specification. The advantage of such an approach is that an efficient and correct-by-construction distributed implementation is automatically derived from a formal model [11,6]. A more detailed comparison with automatic generation of distributed applications can be found in Section 4.

The following operations are a sufficient, minimal set of primitive operations to drive a distributed reactive system and must be executed transactionally. Without loss of generality, we only consider the case in which all the nodes in the distributed graph are remote. Possible optimizations can take advantage of the fact that local reactives are not visible by other participants.

A Participant Updates a RR Value. The participant starts a distributed transaction. The change is propagated across the distributed graph, each participant updating the RRs inside the same transaction. In case of a conflict between two transactions executing at the same time, one transaction rolls back, restoring the previous values of all affected reactives, and retries its execution. For this reason, the system requires separation of purely functional update propagation among reactives and side effects in the same style as Scala.React. The execution of the code preforming side-effects is deferred to after the transaction has successfully committed, thereby ensuring glitch freedom even in case a transaction has to be rolled back. Topological sorting of the dependency graph could still be used inside the update transaction to prevent multiple recalculations of the same functional parts. Doing so would, however, imply forcing all hosts' local update threads that are involved in the transaction to emulate a single-threaded execution. Ignoring the topological order would exploit parallel processing capabilities of multiple hosts and could even be used locally on multicore systems, but at the cost of multiple reevaluations. An ideal solution would be designing a new algorithm, which incorporates both concurrent updates of independent reactives (those that are incomparable in the topological order) and prevention of multiple recalculations of affected reactives. It is our goal to provide this hybrid solution.

The case of a *participant establishing a new dependency*, either independently or as the result of a reevaluation, can be considered a subcase of updating a reactive. Instead of updating the reactive's value, the participant updates the set of dependencies of the reactive. It is even possible, to update both at once. This must be done transactionally as well, to avoid interference with other changes. In addition, all graph modifications must be checked to prevent the introduction of dependency cycles.

A Participant Reads a RR Value. Reading must also be transactional, to avoid inconsistent state between subsequent reads of different reactive values – some of which could have been updated by a change propagation in the meantime. This can easily be achieved by executing all the reads inside a transaction.

The solution sketched above allows a certain degree of parallelism between transactions that do not interfere. In this sense, it is already an improvement over transferring the algorithms available in literature to a distributed setting by adopting a naive, centralized execution, regardless of how much independencies in the topological ordering are exploited. Our research plan is to use this solution as the first step to improve upon. For example, as sketched above, it requires finding a performant way to check the dependency graph for cycles. Based on the assumption that dependency changes are less frequent than value updates, using a centralized master that keeps a representation of the entire dependency graph would be a simple way to enable performant execution of this validation while still providing an improvement over executing all updates from a centralized coordinator. Still, this would imply a single point of failure and prevent concurrent changes to the dependency graph. Our plan includes exploring better design

options that allow more decentralized approaches. Also, we expect to identify various trade-offs between provided guarantees and processing speed, some of which are described in the following paragraphs.

As long term goals, we envisage three application scenarios: Web applications, enterprise systems, and computing systems in an open environment. The remainder of this section elaborates our visions for each of these.

Web Applications. The solution presented so far provides a general model that is valid on distributed systems with an arbitrary number of hosts. This model directly applies when several browsers *share* resources among each other through a server[1]. However, we expect that in typical web applications only a client and a server share the RRs. While the aforementioned model proves the feasibility of our research vision, we expect that in a client-server model it can be significantly simplified, reducing the number of messages sent over network and gaining in performance. For example, Javascript applications enforce a single thread model, which solves the problem of synchronizing between concurrent changes for free.

Enterprise Distributed Systems. We envisage a scenario in which distributed reactive programming can be used to implement large-scale enterprise systems. In those environments, the consistency constraints required by distributed reactive programming should be provided by a dedicated middleware in the same way e.g. the JMS middleware provides *guaranteed delivery* and *once-and-only-once* semantics.

This research direction must take into account that enterprise software often does not run in isolation, but operates inside containers, like Tomcat or JBoss. For example, containers can provide persistence for Enterprise Java Beans (EJB). In a similar way, RRs can be modeled by EJBs, whose consistency is supported by the container

Highly Dynamic and Unreliable Environments. Distributed systems pose a number of challenges that include slowdown of network communication, node failures, communication errors and network partitioning. Those issues have been studied for a long time by the distributed systems community. An open challenge is to define proper behavior of a reactive system in such conditions. Proper language abstractions should support dealing with those cases with clear semantics. For example, in case of network partitioning, the system should be able to restructure the distributed graph with only the available hosts, but this clearly has an impact on the semantics of the application.

Another aspect we plan to explore is to relax the constraints as a possible reaction to network performance degradation. Reactive programming has been successfully used in the field of ambient-oriented programming [12], where systems are dynamic and unreliable and links can become slow. We envisage a scenario in which glitch-freedom guarantees can occasionally be switched off, in case maintaining them becomes too onerous. Similarly to before, proper abstractions should be introduced to deal with this case.

Evaluation A primary contribution of our research is to provide a new conceptual tool to support the design and the implementation of distributed software. The achievement

[1] For example, the AtomizeJS distributed software transactional memory supports this functionality, `http://atomizejs.github.com/`

of such a goal is hard to measure numerically, so we mostly expect feedback from the scientific community. More concretely, we plan to empirically evaluate our findings along two dimensions: On the one hand, we want to evaluate the impact of our solution on the design of a distributed system. Since these phenomena have already been observed in the local setting, we expect that static metrics can significantly improve, including reduced lines of code, reduction of the number of callbacks, and increased code reuse. On the other hand, we plan to evaluate the performances of our solution. Our goal here is not to make it competitive with models that require less stringent consistency guarantees, like publish-subscribe systems or complex event processing engines, but to provide a solution with significant design advantages at a minimum performance penalty.

4 Related Work

Due to space limitations, related work cannot be discussed extensively. Several languages implement concepts from reactive programming in various flavors. The interested reader can refer to [2] for an overview. None of them provides glitch-freedom in the distributed setting. Among the existing reactive languages, AmbientTalk/R [12] is the closest to our approach, and, to the best of our knowledge, it is the only one that has been specifically designed to support distribution in a form similar to remote reactives. In the remainder of this section, we point to the main research areas related to reactive programming and summarize their relation with the subjects discussed in this paper.

Dataflow languages, like Esterel [4] and Signal [10] focus on provable time and memory bounds. Differently form reactive languages and functional reactive programming, reactive abstractions are typically second-order, to support compilation to efficient finite state machines.

Self-adjusting computation [1] automatically derives an incremental version of an algorithm. It mostly focuses on efficiency and algorithmic complexity of the incremental solution, while reactive programming focuses on language abstractions for time-changing values.

The Implementation of distributed algorithms has been explored in form of code generation from formal specifications of I/O automatons [11]. A similar approach [6] generates the distributed programs from a description based on BIP (Behavior, Interaction and Priority) models [3]. We believe that the core of a middleware for distributed reactive programming can be conveniently specified in such a model and automatically generated. Consistency properties like glitch freedom would be provided through the synchronized state transitions offered by petri nets. However, we expect that the application code that builds on top of such a middleware (i.e. on remote reactives) is still written in a “traditional” language. As such, we believe it to be beneficial to allow programmers the implementation of all parts of their application seamlessly in their regular language in the style of reactive programming.

Publish-subscribe systems [8] leverage inversion of control to loosely couple interacting systems. Differently from publish-subscribe systems, we want to enforce a more tight coupling in the change propagation, to transfer the advantages of the local reactive semantics into the distributed setting.

Complex event processing (CEP) [14] is about performing queries over streams of data. Similarly to reactive languages, changes in the data result in an update to dependent values – in CEP, query results. Differently from reactive programming, CEP expresses dependencies via SQL-like queries that are usually expressed as strings and not integrated into the language.

Acknowledgments. This work has been supported by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant No. 16BY1206E and by the European Research Council, grant No. 321217.

References

1. Acar, U.A., Ahmed, A., Blume, M.: Imperative self-adjusting computation. In: POPL 2008, pp. 309–322. ACM (2008)
2. Bainomugisha, E., Lombide Carreton, A., Van Cutsem, T., Mostinckx, S., De Meuter, W.: A survey on reactive programming. In: ACM Comput. Surv. (2013) (To appear)
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: Fourth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2006, pp. 3–12. IEEE (2006)
4. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming 19(2), 87–152 (1992)
5. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2008, pp. 247–258. ACM, New York (2008)
6. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. In: Distributed Computing, pp. 1–27 (2012)
7. Cooper, G.H., Krishnamurthi, S.: Embedding dynamic dataflow in a call-by-value language. In: ESOP, pp. 294–308 (2006)
8. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
9. Gasiunas, V., Satabin, L., Mezini, M., Núñez, A., Noyé, J.: EScala: modular event-driven object interactions in Scala. In: AOSD 2011, pp. 227–240. ACM (2011)
10. Gautier, T., Le Guernic, P., Besnard, L.: SIGNAL: A declarative language for synchronous programming of real-time systems. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, Springer, Heidelberg (1987)
11. Georgiou, C., Lynch, N., Mavrommatis, P., Tauber, J.A.: Automated implementation of complex distributed algorithms specified in the ioa language. International Journal on Software Tools for Technology Transfer (STTT) 11(2), 153–171 (2009)
12. Lombide Carreton, A., Mostinckx, S., Van Cutsem, T., De Meuter, W.: Loosely-coupled distributed reactive programming in mobile ad hoc networks. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 41–60. Springer, Heidelberg (2010)
13. Maier, I., Odersky, M.: Deprecating the Observer Pattern with Scala.react. Technical report (2012)
14. Margara, A., Cugola, G.: Processing flows of information: from data stream to complex event processing. In: Proceedings of the 5th ACM International Conference on Distributed Event-Based System, DEBS 2011, pp. 359–360. ACM, New York (2011)

15. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for ajax applications. In: OOPSLA 2009, pp. 1–20. ACM (2009)
16. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 155–179. Springer, Heidelberg (2008)
17. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing 10(2), 99–116 (1997)
18. Weikum, G., Vossen, G.: Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. Morgan Kaufmann Publishers Inc., San Francisco (2001)

Typing Progress in Communication-Centred Systems

Hugo Torres Vieira and Vasco Thudichum Vasconcelos

LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

Abstract. We present a type system for the analysis of progress in session-based communication centred systems. Our development is carried out in a minimal setting considering classic (binary) sessions, but building on and generalising previous work on progress analysis in the context of conversation types. Our contributions aim at underpinning forthcoming works on progress for session-typed systems, so as to support richer verification procedures based on a more foundational approach. Although this work does not target expressiveness, our approach already addresses challenging scenarios which are unaccounted for elsewhere in the literature, in particular systems that interleave communications on received session channels.

1 Introduction

In today's ever-growing cloud infrastructure of computation, communication is more and more of crucial importance. Communication is still of crucial importance even when considering multi-core machines, where processes may interact via shared memory, since such machines will inevitably have to communicate among them. It is then of vital importance to introduce mechanisms that support the development of *correct* communicating programs, given their wide dissemination, ranging from critical services, such as medical or financial, to those helping people connect with family and friends. Focusing on communication, there are at least two fundamental correctness properties one may be interested in: that interacting parties follow a given communication protocol (*fidelity*) and that the interaction does not reach a deadlock (*progress*).

Verification procedures, such as type systems, that statically ensure communicating programs enjoy the properties above can prove to be cost-effective, as they save on software maintenance by preventing bugs from the start, and can help to expedite the software development process. Several type systems have been proposed that single out programs with *correct* communication behaviour, out of which we distinguish session types, introduced by Honda et al. [8, 9]. Session types focus on ensuring *fidelity* in systems where (single-threaded) protocols are carried out between two parties, such as, e.g., a client and a server. Session types are by now widely adopted as the basis of pragmatic typing disciplines, targeting operating system design [5], middleware communication protocols [16] and distributed object-oriented programming [7], just to mention a few. Session types have also been generalised so as to consider multiparty interactions [3, 10].

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 236–250, 2013.

Building on session types, a number of works were proposed that ensure progress (e.g., [2–4]). This has proven to be a challenging task, in particular regarding the expressiveness of the approaches, even when considering sophisticated typing mechanisms. In this paper we present a typing discipline that distils the basic ingredients necessary to prove progress, building on (classic) session types. This work is not to be viewed as an exercise on expressiveness, capturing every conceivable communication pattern: it fails to do so, even if we are able to address challenging system configurations. Instead, our (far more ambitious) goal is to introduce a foundational approach that will hopefully underpin the development of forthcoming works on progress for communication centred-programming based on session types.

At the basis of our development is the idea that a communication-centred program that enjoys progress is embodied with a natural ordering of communications (or *events*). Building on this idea, and trying to use as few ingredients as possible, we unify the notions of event and session type so as to characterise the sessions and the ordering of events in a combined and novel way. To achieve this we add to each communication in a session type an annotation that allows to identify the (abstract) communication event that is associated to the (concrete) communication action described by the session type. Then, together with a separate notion of overall ordering of events we are able to distinguish systems that enjoy fidelity and progress.

Remarkably we are able to address challenging configurations (unaccounted for elsewhere in the literature) where processes interleave communications in received session channels, for instance to communicate on other sessions or to initiate new ones. As an example, think of a (service) broker that must interleave communications with a client and a server (not to mention other local or remote resources), using (two binary) session channels that where shared (communicated) among the three parties. Our approach generalises previous work on progress in the context of conversation types [3], a session-based type system that addresses multiparty interaction. Also, differently from other session-types based approaches, our type system works directly on the (standard) π-calculus [13], exploiting notions from [3] and from a previous work on session types [17].

Motivating Examples. We illustrate our development by visiting a couple of simple examples. Consider the system shown in Fig. 1 that specifies a basic interaction. The `Client` process creates a new name *chat* and sends it on channel *service* which is read by `Server`. The synchronisation on *service* allows the client and the server to share a private channel (*chat*) where the *session* will take place. After synchronizing on *service*, the client proceeds by receiving from channel *chat* a text message, after which, it sends another text message again on channel *chat*. The `Server` process is (continuously $\star$) waiting to receive in *service* a channel, which instantiates message parameter y. Upon reception, it will then send a text message on that channel, after which, to avoid waiting for the reply, it delegates the rest of the session interaction on y by sending y on channel *handle*. The `Slave` process is defined so as to (also continuously) receive a channel name

$$
\begin{aligned}
\texttt{Client} &\triangleq (\nu \mathit{chat})\mathit{service}!\mathit{chat}.\mathit{chat}?s.\mathit{chat}!\text{“}\texttt{bye}\text{”}\\
\texttt{Server} &\triangleq \star\mathit{service}?y.y!\text{“}\texttt{hello}\text{”}.\mathit{handle}!y\\
\texttt{Slave} &\triangleq \star\mathit{handle}?z.z?s\\
\texttt{System} &\triangleq \texttt{Client} \mid \texttt{Server} \mid \texttt{Slave}
\end{aligned}
$$

Fig. 1. Greeting service code

from *handle*, and then receiving a text message in that channel. The `System` is defined as the parallel composition of the three processes.

Notice we described the system using type information (e.g., “text message”), taking *fidelity* for granted. Also, in the examples we use basic types (such as string) although our technical development focuses exclusively on *channel* types.

It is straightforward to see that the system enjoys progress, as the interaction supported by *chat* does not deadlock. However, we may already use this simple example to convey some intuition on our typing approach. Consider for instance process `Slave` that receives a channel name from *handle* and then communicates in the received channel. We may characterise this usage of channel *handle* with type $?\,(?\,\texttt{String})$, where ? specifies reception, which may be read as “receives a channel used to receive a string”, just like in a regular session type.

We distinguish (session) interactions carried out exactly once (*linear*, no races) and service definitions that support several interactions (*shared* or unrestricted) by annotating the type of *handle* accordingly, i.e., $\mathsf{un}\,?\,(\mathsf{lin}\,?\,\texttt{String})$, which adds to the description that *handle* can be used zero or more times while the received channel must be used exactly once (linearly).

Up to now we are using standard session type notions to characterise the usage of a channel (see [17]). Building on the standards, we add information that allows to characterise (in an abstract way) the moment in time when the communication is to take place: the *event*. We say that the communication in *handle* corresponds to some event in time e_1, while the reception of the string corresponds to event e_2, and obtain the type $e_1\,\mathsf{un}\,?\,(e_2\,\mathsf{lin}\,?\,\texttt{String})$. Notice that e_2 necessarily takes place *after* e_1 (hence, e_1 and e_2 are different events), to represent which we use $e_1 \prec e_2$ (read “e_1 happens before e_2”).

We are then able to characterise processes with both information on the channel usage and on the expected overall *ordering* of events, in particular `Slave` is charaterized by the typing assumption $\mathit{handle}\colon e_1\,\mathsf{un}\,?\,(e_2\,\mathsf{lin}\,?\,\texttt{String})$ and ordering of events $e_1 \prec e_2$. Notice that we do not refer to z in the type since z is a name bound to the reception ($\mathit{handle}?z$). However, and crucially to our approach, we do mention the event where z is involved (e_2), and register it both in the message type $e_2\,\mathsf{lin}\,?\,\texttt{String}$ and in the event ordering $e_1 \prec e_2$. This will allow to crosscheck whether the name sent in *handle* has a corresponding type so as to keep a sound (overall) ordering. Following the same lines we may characterise process `Server` by the following typing assumptions.

$$
\begin{aligned}
\mathit{handle} &: e_1\,\mathsf{un}\,!\,(e_2\,\mathsf{lin}\,?\,\texttt{String})\\
\mathit{service} &: e_3\,\mathsf{un}\,?\,(e_4\,\mathsf{lin}\,!\,\texttt{String}.e_2\,\mathsf{lin}\,?\,\texttt{String})
\end{aligned}
$$

Notice that `Server`'s usage of channel *handle* is *dual* to that of `Slave` (`Server` outputs ! and `Slave` receives ?). Notice also that `Server` and `Slave` agree that

$$
\begin{array}{rcl}
\texttt{Client} & \triangleq & (\nu\, chat)service!chat.chat?s.chat!\text{“}\texttt{bye}\text{”} \\
\texttt{Proxy} & \triangleq & \star service?y.(\nu s)masterservice!s.s!y \\
\texttt{Master} & \triangleq & \star masterservice?z.z?y.y!\text{“}\texttt{hello}\text{”}.handle!y \\
\texttt{Slave} & \triangleq & \star handle?z.z?s \\
\texttt{System} & \triangleq & \texttt{Client} \mid \texttt{Proxy} \mid \texttt{Master} \mid \texttt{Slave}
\end{array}
$$

Fig. 2. Greeting via proxy service code

handle is to be used unrestrictedly (un). Furthermore, both processes agree on the moment in time when the communication in *handle* will take place (e_1). Lastly, both processes also agree on the message type $e_2\,\mathsf{lin}\,?\,\texttt{String}$, hence `Server` also knows that the channel sent in *handle* is involved in e_2.

We also have that *service* is used as an unrestricted input, associated to event e_3, with message type $e_4\,\mathsf{lin}\,!\,\texttt{String}.e_2\,\mathsf{lin}\,?\,\texttt{String}$. The message type captures that the name received from *service* will be used to output a string (event e_4) and *after* (denoted as . like in session types) used to receive a string (event e_2). The last part is realized via the delegation of the session channel in *handle*, where the delegated usage is given by the message type associated to *handle*.

The ordering of events that `Server` expects is $e_3 \prec e_4 \prec e_1 \prec e_2$, since the process first receives from *service* (e_3), then in the session channel (e_4) and then outputs in *handle* (e_1), delegating the reception usage of the session channel (e_2) which necessarily takes places after the channel is sent (hence $e_1 \prec e_2$). Notice that `Server` interleaves communications in the received channel y and in *handle*, addressed by our characterisation based on the fact that the ordering of events (intertwined with channel types) mentions the events associated to communicated names.

As for the characterisation of `Client` we have that it uses *service* with the dual usage with respect to `Server` (`Server` inputs ? while `Client` outputs !) and expects ordering $e_3 \prec e_4 \prec e_2$ since it first outputs in *service* (e_3), then sequentially inputs (e_4) and outputs (e_2) in the session channel. Notice that although the session channel (*chat*) is private to the `Client`, the expected ordering mentions events where the private name is involved.

Using the usage and ordering information that characterise `Server` and `Slave` we may characterise the system $\texttt{Server} \mid \texttt{Slave}$. Channel usage is sound since the two usages of shared name *handle* are dual as mentioned before. The ordering of events is sound since gathering $e_1 \prec e_2$ and $e_3 \prec e_4 \prec e_1 \prec e_2$ does not introduce cycles in the overall order. Likewise for `System` since adding the characterisation of the `Client` we have consistent usages (dual in *service*) and a sound total ordering (obtained by gathering $e_3 \prec e_4 \prec e_1 \prec e_2$ and $e_3 \prec e_4 \prec e_2$). We are then able to show that `System` enjoys (not only *fidelity* but also) progress.

We now consider a slight variation of the previous scenario, illustrated in Fig. 2. The `Client` and `Slave` processes are exactly the same with respect to Fig. 1. Between them we find a `Proxy` and a `Master`, where the `Proxy` is used as an intermediary between `Client` and `Master`. The `Proxy` process starts a session with the client (via *service*) and then starts another session with `Master` (via *masterservice*), where the latter is used just to delegate the session channel

$$
\begin{array}{lll}
P, Q ::= \mathbf{0} & \text{(Inaction)} & \mid\ x!y.P \quad \text{(Output)} \\
\mid\ P \mid Q & \text{(Parallel)} & \mid\ x?y.P \quad \text{(Input)} \\
\mid\ (\nu x)P & \text{(Restriction)} & \mid\ \star x?y.P \quad \text{(Replicated Input)}
\end{array}
$$

Fig. 3. Syntax of processes

associated to the interaction with the client (notice that the client and the proxy only interact in the *service* synchronisation). The `Master` starts a session (via *masterservice*) and receives in it the session channel used to interact with the client. After that the `Master` process starts interacting with the `Client` via the received channel (from then on just like the `Server` in the previous example).

We may also single out the `System` shown in Fig. 2 using our type system to show it enjoys progress. Notice the `Master` process interleaves communications in two received names, one in a session "initiation" and the other in a inner session delegation, which presents no further challenge to our type system since both cases are handled uniformly. Such configuration is unaccounted for in the reference works on progress for sessions [2, 4].

In the rest of the paper we present our technical development, starting by the definition of the process model, followed by the presentation of the type system and associated results. To finish we discuss related and future work.

2 Process Model

In this section we present the syntax and semantics of the language of processes, a fragment of the π-calculus [13, 15]. The syntax of processes is given in Fig. 3, considering an infinite set of names Λ ($x, y, \ldots \in \Lambda$). The (so-called) static fragment of the process model is given by the inaction $\mathbf{0}$ that represents a process with no behaviour, the parallel composition of processes $P \mid Q$ that represents a process where P and Q are simultaneously (concurrently) active, or the name restriction $(\nu x)P$ that represents a process that has a "private" name x (x is bound in $(\nu x)P$). The dynamic (active) fragment of the language is given by the communication prefixes: $x!y.P$ represents a process that outputs name y in channel x and then continues as specified by P; $x?y.P$ represents a process that receives a name from channel x and then proceeds as P (y is bound in $x?y.P$); $\star x?y.P$ represents a replicated input, i.e., a process able to continuously receive a name from channel x and proceed as P.

In order to keep the setting as simple as possible, we decided not to allow specifying alternative behaviour via summation, $+$. We believe however that our development can be extended to consider summation along non-surprising lines.

The semantics of the language is defined via structural congruence and reduction relations, to define which we introduce some (standard) notions. We denote by $fn(P)$ the set of free names that occur in P. Also, we denote by $P \equiv_\alpha Q$ that P and Q are equal up to a renaming of bound names. By $P\{x \leftarrow y\}$ we present the process obtained by replacing all free occurrences of x in P by y.

$$P \mid \mathbf{0} \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) \qquad P \equiv_\alpha Q \implies P \equiv Q$$
$$(\nu x)\mathbf{0} \equiv \mathbf{0} \qquad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \qquad P_1 \mid (\nu x)P_2 \equiv (\nu x)(P_1 \mid P_2) \ \ (x \notin fn(P))$$

Fig. 4. Structural congruence

$$\frac{}{x?y.P \mid x!z.Q \to P\{y \leftarrow z\} \mid Q}\text{(R-Com)} \qquad \frac{P \to Q}{(\nu x)P \to (\nu x)Q}\text{(R-New)}$$
$$\frac{}{\star x?y.P \mid x!z.Q \to \star x?y.P \mid P\{y \leftarrow z\} \mid Q}\text{(R-Rep)} \qquad \frac{P \to P'}{P \mid Q \to P' \mid Q}\text{(R-Par)}$$
$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}\text{(R-Cong)}$$

Fig. 5. Reduction relation

Structural congruence is defined as the least congruence over processes that satisfies the rules in Fig. 4. Structural congruence allows to specify equivalent classes of processes and supports the definition of the reduction relation focusing on the interactions of the (basic) representatives of the equivalent classes.

The reduction relation over processes is given by the rules in Fig. 5, capturing how processes "evolve" via communication. Rule (R-Com) captures the interaction between an output and an input, where the output emits a name that is received in the input (notice the communicated name z replaces the bound input parameter in the continuation P). Rule (R-Rep) follows the same lines with respect to the communication behaviour, the difference is that in the resulting state the input is again ready to further synchronise. Aiming at a simple subject congruence result, we do not consider the (original to the pi calculus) structural congruence rule that introduces parallel copies of the replicated process, defining instead unbounded behaviour via rule (R-Rep). The remaining rules close the relation under language contexts — (R-New) for name restriction and (R-Par) for parallel composition — and under structural equivalence classes.

3 Type System

In this section we present our type system, starting by introducing strict partial orders, a crucial notion that allows us to single out well-formed communication dependency structures of processes. We then present our type language which unifies the notions of events and of session types descriptions, and our type system where processes are characterised via their usage of channels (as usual) and of their ordering of events. Finally, we present our results, namely typing preservation under reduction (Theorem 1) and progress (Theorem 2).

The idea of ordering events to guarantee progress is not new (cf., [2, 4, 11, 12]) and seems in fact an excellent mechanism to single out sound communication

$$
\begin{array}{llll}
p, p_1, p_2, \ldots & ::= & ! & \text{(Output)} \\
& \mid & ? & \text{(Input)} \\
& \mid & \tau & \text{(Synchronisation)} \\
L, L_1, L_2, \ldots & ::= & \mathsf{end} & \text{(No interaction)} \\
& \mid & e\,\mathsf{lin}\,p\,T.L & \text{(Session)}
\end{array}
\qquad
\begin{array}{llll}
T, T_1, T_2, \ldots & ::= & L & \text{(Linear)} \\
& \mid & e\,\mathsf{un}\,p\,T & \text{(Shared)} \\
\Gamma, \Gamma_1, \Gamma_2, \ldots & ::= & \cdot & \text{(Empty)} \\
& \mid & \Gamma, x\colon T & \text{(Assumption)}
\end{array}
$$

Fig. 6. Syntax of types and typing contexts

dependency structures. In our approach, we introduce event orderings and session types in a combined and uniform way, aiming at minimising the number of ingredients required to prove progress of communicating processes. We thus find the strict partial orders defined next at the root of our development.

Strict Partial Orders. A strict (or irreflexive) partial order $\prec$ over a set $\mathcal{E}$ is a binary relation that is asymmetric (hence irreflexive) and transitive. We call $\mathcal{E}$ the *set of events*, and we let $e, e_1, e_2, \ldots$ range over $\mathcal{E}$. Furthermore we distinguish two events, and call them end and $\top$, that form the "leaves" of the communication dependency tree (since strict partial orders do not admit reflexive pairs, we require two elements to form the "leaf" pair of the relation).

We make use of the following notions on partial orders. The *support* of a partial order $\prec$ is the set of elements of $\mathcal{E}$ that occur in $\prec$, defined as follows.

$$supp(\prec) \triangleq \{e \mid \exists e_1.(e_1, e) \in \prec \vee (e, e_1) \in \prec\}$$

The support is used in our typing rules so as to allow us to pick "fresh" events, i.e., events that are not referred to by the relation. The relation obtained by *adding the least event* e to $\prec$ is denoted by $e + \prec$. Notice that if e is not in the support of $\prec$ and $\prec$ is a strict partial order, then so is $e + \prec$.

$$e + \prec \triangleq \prec \cup \{(e, e_1) \mid e_1 \in supp(\prec)\}$$

The relation obtained by *removing an element* e from $\prec$ is denoted by $\prec \setminus e$. Notice that if $\prec$ is a strict partial order, then so is $\prec \setminus e$.

$$\prec \setminus e \triangleq \{(e_1, e_2) \mid (e_1, e_2) \in \prec \wedge e \neq e_1 \wedge e_2 \neq e\}$$

The strict partial order $\prec$ relation obtained by the *union of two strict partial orders* $\prec_1$ and $\prec_2$ is denoted by $\prec_1 \uplus \prec_2$. Notice that $\uplus$ is a partial operation (undefined when the plain union of the relations introduces cycles). We use $\uplus$ to gather the communication dependency structures of, e.g., two parallel processes.

Types. Having defined the notion of strict partial orders of events we proceed to the presentation of the type language whose syntax is given in Fig. 6. Our types extend session types [8, 9] with event annotations, and also exploit notions introduced in previous work on session types [17] and conversation types [3].

We use *polarities*, p, to capture communication capabilities: ! captures the output capability, ? captures the input capability and τ captures a synchronisation pair (cf., [3]). Types are divided in two main classes, shared and linear. The

former captures the exponential usage of channels, i.e., channels where (communication) races are admissible (intuitively, think of services that may be simultaneously provided and requested by several sites). The latter captures linear usage of channels, where no races are allowed (in the service analogy, the single-threaded protocol between server and client in a service instance). A shared type $e\,\mathsf{un}\,p\,T$ specifies a polarity p that captures a communication capability, an event e so as to create the association between the communication action and the abstract notion of event, and an argument type T that prescribes the delegated behaviour of the communicated channel. The description of a linear type $e\,\mathsf{lin}\,p\,T.L$ follows the same lines, except for the continuation L which specifies the behaviour that takes place after the action captured by $e\,\mathsf{lin}\,p\,T$. Linear types are terminated in end, meaning no further interaction is allowed on the channel.

Notice that our types build on the notion of abstract events, differently from other related approaches (cf., [2–4]) that resort to channel names (and communication labels) to order communication actions. Notice also our types structurally resemble "classic" session types, differing in the introduction of the event identifier, crucial to our approach, the polarity annotation that allows us to avoid polarised channels, and the linear/unrestricted annotation that allows us to avoid separate typing contexts for shared and linear channels and separate typing rules for linear/unrestricted argument type of communications.

We next define some auxiliary notions over types used in our typing rules. The set of elements of $\mathcal{E}$ that occur in a type T is denoted by $events(T)$. Notice that events in messages (argument types) are not included.

$$events(T) \triangleq \begin{cases} e \cup events(L) & \text{if } T = e\,\mathsf{lin}\,p\,T_1.L \\ \{\mathsf{end}\} & \text{if } T = \mathsf{end} \\ \{e\} & \text{if } T = e\,\mathsf{un}\,p\,T_1 \end{cases}$$

The binary relation over $\mathcal{E}$ present in a type T is denoted by $T\!\downarrow$. If each linear prefix in T has a distinct event e then it is immediate that $T\!\downarrow$ is a strict partial order. We use $T\!\downarrow$ to single out the order of events prescribed by a type, which essentially is a (single) chain of events in the case that T is a linear type.

$$T\!\downarrow \triangleq \begin{cases} e + (L\!\downarrow) & \text{if } T = e\,\mathsf{lin}\,p\,T_1.L \\ \{(\mathsf{end}, \top)\} & \text{if } T = \mathsf{end} \\ \{(e, \top)\} & \text{if } T = e\,\mathsf{un}\,p\,T_1 \end{cases}$$

We introduce a predicate that is true for types that do not specify pending communication actions.

$$matched(T) \triangleq \begin{cases} matched(L) & \text{if } T = e\,\mathsf{lin}\,\tau\,T_1.L \\ \mathsf{true} & \text{if } T = \mathsf{end} \text{ or } T = e\,\mathsf{un}\,?\,T_1 \\ \mathsf{false} & \textit{otherwise} \end{cases}$$

Matched linear communication actions are captured by τ annotated types. As for shared communication actions, we focus only on unmatched output actions and thus only shared inputs are matched. We will clarify this notion in the definition of splitting (Fig. 8) and in the characterisation of active processes in the context of our main result (Theorem 2), for now it suffices to say that matched shared communications are ?-annotated.

Typing contexts. The syntax of typing contexts is given in Fig. 6. We assume by convention that, in a typing context $\Gamma, x\colon T$, name x does not occur in Γ. Also, we use Γ_{end} to abbreviate a typing context $\cdot, x_1\colon \mathsf{end}, \ldots, x_k\colon \mathsf{end}$ for some $k \geq 0$ and $x_1, \ldots, x_k$. We introduce some auxiliary predicates over typing contexts to single out typing contexts that refer only to unrestricted and matched communications.

We denote by Γ_{un} contexts that include only shared communications, defined as $\Gamma_{un} ::= \cdot \mid \Gamma'_{un}, x\colon \mathsf{end} \mid \Gamma'_{un}, x\colon e\,\mathsf{un}\,!\,T$. Such contexts are used to qualify the exponential resources that a replicated input may use. Since more than one copy of the continuation of a replicated input may be simultaneously active, there must be no linear behaviour present (to avoid communication races). We also exclude shared inputs in Γ_{un} so as to avoid nested replicated inputs. Intuitively, if we admit nested service definitions then, to guarantee progress, we would also require that every service is called at least once (in such way activating all nested service definitions) or characterise progress of open systems by inserting them in the "right" context (cf., [4]). We focus on closed systems where all communications are matched, i.e., typed in *matched* contexts. We lift the matched predicate over types to typing contexts in the expected way: we write $\mathit{matched}(\cdot, x_1\colon T_1, \ldots, x_k\colon T_k)$ if $\mathit{matched}(T_i)$ for all i such that $1 \leq i \leq k$.

Splitting and Conformance. We now introduce two notions crucial to our development, namely *splitting* (inspired by [1]) that explains how behaviour can be decomposed and safely distributed to distinct parts of a process (e.g., to the branches of a parallel composition), and *conformance* that captures the desired relation between typing contexts and strict partial orders.

We say a typing context Γ conforms to a strict partial order $\prec$, denoted by $\mathit{conforms}(\Gamma, \prec)$, if all event orderings prescribed by the types in Γ are contained in $\prec$, thus ensuring that the events associated with the communication actions described by the types are part of the overall ordering.

$$\mathit{conforms}(\Gamma, \prec) \triangleq \begin{cases} \mathsf{true} & \text{if } \Gamma = \cdot \\ \mathit{conforms}(\Gamma_1, \prec) & \text{if } \Gamma = \Gamma_1, x\colon T \text{ and } T{\downarrow} \subseteq \prec \\ \mathsf{false} & \mathit{otherwise} \end{cases}$$

Splitting is defined for both types and typing contexts, defined via three operations over linear types, shared types and typing contexts. We write $T = T_1 \circ T_2$ to mean that type T is split in types T_1 and T_2, and likewise for $\Gamma = \Gamma_1 \circ \Gamma_2$. Linear type splitting, shared type splitting and context splitting are given by the rules in Figs. 7–9. Linear type splitting supports the decomposition of a synchronised, τ, session type (including continuation) in the respective dual capabilities !, ?, via rule (L-Dual-1) and its symmetric (L-Dual-2). Notice $L = L_1 \circ L_2$ is defined only when $\mathit{matched}(L)$. Essentially, linear type splitting allows to decompose a matched session type in its two *dual* counterparts (see, e.g., [6]).

Shared type splitting decomposes shared communication capabilities in two distinct ways, depending on whether the polarity of the type to be split is ? or !. A shared input is split in a shared input and either in an output or another input, via rules (S-In-1) and its symmetric (S-In-2). Intuitively, an input that is decomposed in two inputs allows to type processes that separately offer the

$$\frac{}{\mathsf{end} = \mathsf{end} \circ \mathsf{end}}\text{(L-End)}$$

$$\frac{L = L_1 \circ L_2}{e\,\mathsf{lin}\,\tau\,T.L = e\,\mathsf{lin}\,!\,T.L_1 \circ e\,\mathsf{lin}\,?\,T.L_2}\text{(L-Dual-1)}$$

$$\frac{L = L_1 \circ L_2}{e\,\mathsf{lin}\,\tau\,T.L = e\,\mathsf{lin}\,?\,T.L_1 \circ e\,\mathsf{lin}\,!\,T.L_2}\text{(L-Dual-2)}$$

Fig. 7. Linear type splitting

$$\frac{p \in \{?, !\}}{e\,\mathsf{un}\,?\,T = e\,\mathsf{un}\,?\,T \circ e\,\mathsf{un}\,p\,T}\text{(S-In-1)}$$

$$\frac{p \in \{?, !\}}{e\,\mathsf{un}\,?\,T = e\,\mathsf{un}\,p\,T \circ e\,\mathsf{un}\,?\,T}\text{(S-In-2)}$$

$$\frac{}{e\,\mathsf{un}\,!\,T = e\,\mathsf{un}\,!\,T \circ e\,\mathsf{un}\,!\,T}\text{(S-Out)}$$

Fig. 8. Shared type splitting

$$\frac{}{\cdot = \cdot \circ \cdot}\text{(C-Empty)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon T = \Gamma_1, x\colon T \circ \Gamma_2}\text{(C-Left)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon T = \Gamma_1 \circ \Gamma_2, x\colon T}\text{(C-Right)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad T = T_1 \circ T_2}{\Gamma, x\colon T = \Gamma_1, x\colon T_1 \circ \Gamma_2, x\colon T_2}\text{(C-Split)}$$

Fig. 9. Context splitting

input capability (e.g., a service that is provided by two distinct sites), and an input that is decomposed in an output and an input allows to type processes that offer the dual communication capabilities (e.g., a service provider and a service client). A shared output is split in two shared outputs — rule (S-Out) — which, intuitively, allows to type processes that offer the output capability separately (e.g., like two service clients). Input capabilities may be further split so as to "absorb" several output capabilities and be distributed in several input capabilities, and output capabilities may also be further split to be distributed in several output capabilities. Notice type splitting (both linear and shared) preserves the argument types so as to guarantee the dual communication actions agree on the type of the communication.

Context splitting allows to split a context in two distinct ways: context entries either go into the left or the right outgoing contexts — rules (C-Left) and its symmetric (C-Right) — or they go in both contexts — rule (C-Split). The latter form lifts the (type) behaviour distribution to the context level, while the former allows to delegate the entire behaviour to a part of the process, leaving no behaviour to the other part. To lighten notation we use $\Gamma_1 \circ \Gamma_2$ to represent Γ such that $\Gamma = \Gamma_1 \circ \Gamma_2$ (if such Γ exists). Notice that, given Γ_1 and Γ_2, there is at most one Γ such that $\Gamma = \Gamma_1 \circ \Gamma_2$.

Typing System. We may now present our type system which characterises processes in terms of their usage of channels and of their overall ordering of events, as captured by judgement $\Gamma; \prec \vdash P$ where Γ describes channel usage and $\prec$ gives the ordering of events. We say process P is well-typed if $\Gamma; \prec \vdash P$ is the conclusion of a derivation using the rules in Fig. 10.

$$\frac{}{\Gamma_{\mathsf{end}}; \{(\mathsf{end}, \top)\} \vdash \mathbf{0}} \quad \text{(T-Inact)}$$

$$\frac{\Gamma_1; \prec_1 \vdash P \qquad \Gamma_2; \prec_2 \vdash Q}{\Gamma_1 \circ \Gamma_2, \prec_1 \cup \prec_2 \vdash P \mid Q} \quad \text{(T-Par)}$$

$$\frac{\Gamma, x\colon T; \prec \vdash P \qquad \mathit{matched}(T)}{\Gamma; \prec \vdash (\nu x)P} \quad \text{(T-New)}$$

$$\frac{\Gamma, x\colon L, y\colon T; \prec \vdash P \qquad e \notin \mathit{supp}(\prec)}{\Gamma, x\colon e\,\mathsf{lin}\,?\,T.L; e + \prec \vdash x?y.P} \quad \text{(T-LIn)}$$

$$\frac{\Gamma, x\colon L; \prec \vdash P \qquad e \notin (\mathit{supp}(\prec) \cup \mathit{events}(T))}{(\Gamma, x\colon e\,\mathsf{lin}\,!\,T.L) \circ y\colon T; e + (\prec \cup T\downarrow) \vdash x!y.P} \quad \text{(T-LOut)}$$

$$\frac{\Gamma_{un}, y\colon T; \prec \vdash P \qquad e \notin \mathit{supp}(\prec)}{\Gamma_{un}, x\colon e\,\mathsf{un}\,?\,T; e + \prec \vdash \star x?y.P} \quad \text{(T-UIn)}$$

$$\frac{\Gamma; \prec \vdash P \qquad e \notin (\mathit{supp}(\prec) \cup \mathit{events}(T))}{(\Gamma, x\colon e\,\mathsf{un}\,!\,T) \circ y\colon T; e + (\prec \cup T\downarrow) \vdash x!y.P} \quad \text{(T-UOut)}$$

Fig. 10. Typing rules

We comment on the rules in Fig. 10. Rule (T-Inact) types the inactive process with a context that associates end to (any set) of channel names and with the "leaf" ordering (the relation with just one pair $(\mathsf{end}, \top)$). Rule (T-Par) types parallel composition by typing each branch with a slice of the context, obtained via splitting, and with a sub-ordering (such that the union of the sub-orderings is a strict partial order). So, the two branches of the parallel composition may freely refer different channels but they must agree in a sound overall ordering. Rule (T-New) types name restriction by typing the restricted name with a matched type (no unmatched communications). Notice the ordering expected for the interactions in the restricted name is kept in the conclusion, so as to characterise the abstract communication dependencies of the process, which includes (an abstraction of) the communication dependencies of bound names.

Communication prefixes are typed in separate rules depending on the type of the subject of the communication — notice however that mixing linear and shared types in the same typing context avoids introducing rules that depend on the type of the object of the communication. Rule (T-LIn) types the input on a channel x with linear usage by typing the continuation process considering the continuation session type L for x, the argument type T for the input variable y and ordering $\prec$. We single out a fresh event e with respect to the continuation ($e \notin \mathit{supp}(\prec)$) that is used to specify the type of the input, together with the respective ? polarity, argument type T and continuation L. We build a new order from $\prec$ by setting e as the least element (given by $e+\prec$) since any communication in the continuation depends on the input (hence is greater than e). Notice that the communications in the continuation include the ones that involve the received name, characterised by T and ordered by $\prec$. Notice also that $\prec$ is recorded in the conclusion, so as to (also) capture the communication dependencies involving

the received name. This is crucial to our approach so as to address processes that interleave communications in received names.

Rule (T-LOut) types the output in a channel x with linear usage by typing the continuation process with the continuation session type L and ordering $\prec$. The conclusion records the type of the output using a fresh event e with respect to the continuation (ordered by $\prec$) and also with respect to the type delegated in the communication T ($e \notin \mathit{supp}(\prec) \cup \mathit{events}(T)$). The (linear) session type in the conclusion is thus specified using e, the argument type T, the respective ! polarity and continuation L. Event e is also recorded in the ordering of the output as the minimum event $e + (\prec \uplus T\downarrow)$, since any communication in the continuation, along with any delegated communication capabilities, depends on the output (hence, are greater than e). We use $T\downarrow$ to extract the (chain of) events prescribed by T. The conclusion records the delegated type T via splitting $(\Gamma, x\colon e\,\mathsf{lin}\,!\,T.L) \circ (y\colon T)$ as y may be used (dually to T) in Γ.

The description of rule (T-UOut) follows the same lines, the only differences is that a shared type captures the output and there is no continuation usage for channel x. We rule out uses of x in the continuation to exclude processes that offer the input in the continuation of an output (at the cost of excluding processes that perform more than one shared output over the same channel in sequence). Our rationale for shared communications is that at least one (replicated) shared input matches all corresponding outputs, so the input cannot be activated *after* the output (to avoid cluttering the rules this led us to also exclude two outputs in sequence, a configuration which is not problematic per se). Rule (T-UIn) types shared inputs that are necessarily replicated, so as to support the rationale that a shared input is able to match all respective inputs. Since more than one copy of the continuation of the input may be simultaneously active we require the resources *shared* by all copies to be shared outputs (Γ_{un}).

The reason why we exclude (nested) shared inputs, as explained earlier, is to avoid the situation where a shared output is blocked due to a shared input (of lesser order) that is blocking the matching shared input (of greater order) which is not matched. To avoid forcing that all shared inputs are matched we exclude nested shared inputs in general. Notice however that the argument type T may be linear or shared, in which case T may actually specify a shared input (a nested input that is activated via interaction). Since the behaviour of the name received in the input is only "published" via a corresponding output, this particular case of nested shared inputs is naturally supported.

The main restriction of the presented work is the absence of a general form of recursion, which, conceivably, involves considering the repetition of the overall ordering throughout the unfolding, handled e.g., via a dedicated typing context that we believe can be engineered in conformance to our approach.

Results. We may now present our results, namely that typing is preserved under reduction (Theorem 1) and that a specific class of well-typed processes (those where all communications are matched) enjoy progress (Theorem 2). We start by mentioning some auxiliary results, namely that we may show that conformance is ensured between the typing context and the strict partial order in all derivations,

a sanity check that ensures the conditions imposed by our rules (e.g., picking freshness of events) are enough to keep conformance invariant in a derivation. We may also show two standard results used in the proof of Theorem 1, namely that typing is preserved under structural equivalence and under name substitution.

Before presenting our first main result (Theorem 1) we introduce two auxiliary notions that characterise reduction of contexts and of strict partial orders. As expected from a behavioural type system, as processes evolve so must the types that characterise the processes. Reduction for contexts is defined as follows.

$$\cdot \to \cdot \qquad \Gamma, x\colon e \,\mathsf{lin}\, p\, T.L \to \Gamma, x\colon L \qquad \Gamma_1 \to \Gamma_2 \implies \Gamma_1, x\colon T \to \Gamma_2, x\colon T$$

A context reduces if it holds an assumption on a linear type prefix, which reduces to the continuation so as to mimic the analogous behaviour in processes. Also, the empty context reduces (to the empty context) so as to capture synchronisations in processes on restricted channels and on channels with shared usage, thus introducing reflexivity in context reduction since no change is required to capture such synchronisations. Reduction for partial orders is defined as follows.

$$\prec \to \prec \qquad e \in \mathit{supp}(\prec) \implies \prec \to \prec \setminus e$$

Strict partial order reduction is also reflexive. This allows to capture synchronisations that *depend* on shared inputs. Notice that the ordering for shared inputs is kept invariant via reduction (since the replicated process is kept throughout reduction), thus capturing synchronisations that depend on shared inputs (as they will take place repeatedly for each activation of the continuation of the shared input). Reduction is also defined by removing an event of the ordering, so as to capture *one shot* synchronisations. Since such synchronisations may depend on shared outputs, they are not necessarily associated with the minimum event in the ordering. We may now present our first main result, where $\Gamma_1; \prec_1 \to \Gamma_2; \prec_2$ denotes $\Gamma_1 \to \Gamma_2$ and $\prec_1 \to \prec_2$.

Theorem 1 (Preservation). *If* $\Gamma_1; \prec_1 \vdash P_1$ *and* $P_1 \to P_2$ *then* $\Gamma_1; \prec_1 \to \Gamma_2; \prec_2$ *and* $\Gamma_2; \prec_2 \vdash P_2$.

The proof follows by induction on the length of the derivation of $P_1 \to P_2$ in expected lines. The theorem says that typing is preserved under process reduction, up to a reduction in the context and ordering. Fidelity is an immediate consequence of Theorem 1, as usual. We now turn our attention to the result on progress. In order to define "live" processes (processes that should reduce) we introduce the (standard) notion of *active contexts*, noted $\mathcal{C}[\cdot]$, defined as $\mathcal{C}[\cdot] ::= \cdot \mid (P \mid \mathcal{C}[\cdot]) \mid (\nu x)\mathcal{C}[\cdot]$. A context $\mathcal{C}[\cdot]$ is a process with a hole $\cdot$ under a number of parallel compositions and restrictions. We say a process P is *active* if it has a (non-replicated) communication prefix in an active context, defined as follows $\mathit{active}(P) \triangleq \exists \mathcal{C}[\cdot], x, y, Q.\ P \equiv \mathcal{C}[x!y.Q] \vee P \equiv \mathcal{C}[x?y.Q]$. Notice the definition of active process rules out replicated inputs. So, we consider *stable* processes (processes that do not reduce but are not errors) to be processes where a number of (replicated) shared inputs are active. We now state our second main result that says an active and well-typed (*matched*) process reduces.

Theorem 2 (Progress). *If* $\mathit{active}(P)$, $\Gamma; \prec \vdash P$ *and* $\mathit{matched}(\Gamma)$ *then* $P \to P'$.

The proof follows by induction on the size of $\prec$. The proof invariant is that for every event either there is a synchronisation pair of *lesser* order or every active prefix of lesser order is a replicated (shared) input. Theorem 2 attests our typing discipline ensures progress of active processes, including processes that interleave communications on received channels.

4 Concluding Remarks

We have presented a typing discipline for the analysis of progress in session-based communication-centred systems. Our work exploits notions introduced in [3] (e.g., the τ polarity) and in [1] (e.g., the splitting relation), allowing to type systems specified in standard π-calculus. This is in contrast with related approaches, where session channels are equipped with polarities (see, e.g., [6]) or where channels have two endpoints (see, e.g., [17]), or or where sessions are established via specialised initiation primitives (see, e.g., [9]). Also, we uniformly handle communication *of* linear and shared channels (when they are the object of the communication) via lin and un annotations introduced in [17]. However, this is not the case for communications *on* linear and shared channels (when they are the subject of the communication).

A cornerstone of our development is the progress analysis technique introduced in [3], where message types already specify the orderings expected for the communicated names, thus providing the basic support for the interleaving of communications on received names. We depart from [3] by unifying the channel usage and event ordering in a single type analysis. Moreover, our orderings build on abstract events and do not refer channel identities (nor labels) differently from [2–4], which allows us to relate events in received names (via an abstract event) with others. This is crucial to address the interleaving of received names in a more general way, allowing us to address scenarios out of reach of the above mentioned works [2–4]. By combining session types and events in the same type language, inspired by [14], we are able to rely on usual session-based reasoning. Our approach differs from the preliminary ideas presented in [14] that combines session types with a typing discipline that relies on type simulation [11], in that our verification system is completely syntax driven and does not rely on extra-imposed conditions (neither on type simulation nor on model-checking).

In our approach, the sound communication dependency structure is captured in a minimal way, via a (strict) partial order of events, which combined with the event-equipped session types, allow us to single out systems that enjoy progress. We acknowledge that our development does not address full-fledged recursion. However, the principles we use can conceivably be lifted to consider recursion, considering the repetition of the event ordering (handled by a dedicated typing context, as usual) or (well-founded) infinite orderings, an engineering exercise we leave to future work. We also plan to use the ideas presented in this paper to type progress in multiparty conversations.

On a pragmatic (vital) level, we may show that the type checking procedure is decidable (considering bound names are type annotated) and we are confident

that a type inference procedure can be extracted from our type system. While decidability attests the type system is worth mentioning, type inference makes it more interesting. It supports the verification of systems without burdening the development process, thus contributing to a cost-effective increase of reliability.

Acknowledgments. We acknowledge support of the project PTDC/EIA-CCO/117513/2010 and thank Pedro Baltazar and Luís Caires for fruitful discussions.

References

1. Baltazar, P., Caires, L., Vasconcelos, V.T., Vieira, H.T.: A type system for flexible role assignment in multiparty communicating systems. In: Proceedings of the TGC 2012. LNCS. Springer (to appear, 2013)
2. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
3. Caires, L., Vieira, H.T.: Conversation types. Theor. Comput. Sci. 411(51-52), 4399–4440 (2010)
4. Dezani-Ciancaglini, M., de'Liguoro, U., Yoshida, N.: On Progress for Structured Communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 257–275. Springer, Heidelberg (2008)
5. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G.C., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in singularity OS. In: Proceedings of the EuroSys 2006, pp. 177–190. ACM (2006)
6. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Inf. 42(2-3), 191–225 (2005)
7. Gay, S.J., Vasconcelos, V.T., Ravara, A., Gesbert, N., Caldeira, A.Z.: Modular session types for distributed object-oriented programming. In: Proceedings of the POPL 2010, pp. 299–312. ACM (2010)
8. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
9. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
10. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of the POPL 2008, pp. 273–284. ACM (2008)
11. Kobayashi, N.: A type system for lock-free processes. Inf. Comput. 177(2), 122–159 (2002)
12. Lynch, N.A.: Fast allocation of nearby resources in a distributed system. In: Proceedings of the STOC 1980, pp. 70–81. ACM (1980)
13. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I and II. Inf. Comput. 100(1), 1–77 (1992)
14. Padovani, L.: From lock freedom to progress using session types. In: Proceedings of the PLACES (to appear, 2013)
15. Sangiorgi, D., Walker, D.: The π-Calculus: A Theory of Mobile Processes. Cambridge University Press (2001)
16. Vallecillo, A., Vasconcelos, V.T., Ravara, A.: Typing the behavior of software components using session types. Fundam. Inform. 73(4), 583–598 (2006)
17. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. 217, 52–70 (2012)

Author Index