

Giovanna Di Marzo Serugendo  
Michele Loreti (Eds.)

LNCS 10852

# Coordination Models and Languages

20th IFIP WG 6.1 International Conference, COORDINATION 2018  
Held as Part of the 13th International Federated Conference  
on Distributed Computing Techniques, DisCoTec 2018  
Madrid, Spain, June 18–21, 2018, Proceedings



ifip



Springer

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

David Hutchison

*Lancaster University, Lancaster, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Friedemann Mattern

*ETH Zurich, Zurich, Switzerland*

John C. Mitchell

*Stanford University, Stanford, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

C. Pandu Rangan

*Indian Institute of Technology Madras, Chennai, India*

Bernhard Steffen

*TU Dortmund University, Dortmund, Germany*

Demetri Terzopoulos

*University of California, Los Angeles, CA, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Gerhard Weikum

*Max Planck Institute for Informatics, Saarbrücken, Germany*


More information about this series at <http://www.springer.com/series/7408>


Giovanna Di Marzo Serugendo · Michele Loreti (Eds.)

# Coordination Models and Languages

20th IFIP WG 6.1 International Conference, COORDINATION 2018  
Held as Part of the 13th International Federated Conference  
on Distributed Computing Techniques, DisCoTec 2018  
Madrid, Spain, June 18–21, 2018  
Proceedings

*Editors*

Giovanna Di Marzo Serugendo   
University of Geneva  
Geneva  
Switzerland

Michele Loreti   
Università degli Studi di Firenze  
Florence  
Italy

ISSN 0302-9743                      ISSN 1611-3349 (electronic)  
Lecture Notes in Computer Science  
ISBN 978-3-319-92407-6              ISBN 978-3-319-92408-3 (eBook)  
<https://doi.org/10.1007/978-3-319-92408-3>

Library of Congress Control Number: 2018944392

LNCS Sublibrary: SL2 – Programming and Software Engineering

© IFIP International Federation for Information Processing 2018

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG  
part of Springer Nature  
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

# Foreword

The 13th International Federated Conference on Distributed Computing Techniques (DisCoTec) took place in Madrid, Spain, during June 18–21, 2018. The DisCoTec series is one of the major events sponsored by the International Federation for Information Processing (IFIP). It comprises three conferences:

- COORDINATION, the IFIP WG6.1 International Conference on Coordination Models and Languages (the conference celebrated its 20th anniversary in 2018)
- DAIS, the IFIP WG6.1 International Conference on Distributed Applications and Interoperable Systems (the conference is in its 18th edition)
- FORTE, the IFIP WG6.1 International Conference on Formal Techniques for Distributed Objects, Components and Systems (the conference is in its 38th edition)

Together, these conferences cover a broad spectrum of distributed computing subjects, ranging from theoretical foundations and formal description techniques to systems research issues. Each day of the federated event began with a plenary speaker nominated by one of the conferences.

In addition to the three main conferences, two satellite events took place during June 20–21, 2018:

- ICE, the Workshop on Interaction and Concurrency Experience (in its 11th edition)
- FADL, Workshop on Foundations and Applications of Distributed Ledgers (this was the first year that the workshop took place)

I would like to thank the Program Committee chairs of the different events for their help and cooperation during the preparation of the conference and the Steering Committee of DisCoTec for its guidance and support. The organization of DisCoTec 2018 was only possible thanks to the dedicated work of the Organizing Committee, including the organization chairs, Jesús Correas and Sonia Estévez (Universidad Complutense de Madrid, Spain), the publicity chair, Ivan Lanese (University of Bologna/Inria, Italy), the workshop chairs, Luis Llana and Ngoc-Thanh Nguyen (Universidad Complutense de Madrid, Spain and Wroclaw University of Science and Technology, Poland, respectively), the finance chair, Mercedes G. Merayo (Universidad Complutense de Madrid, Spain), and the webmaster, Pablo C. Cañizares (Universidad Complutense de Madrid, Spain). Finally, I would like to thank IFIP WG6.1 for sponsoring this event, Springer’s *Lecture Notes in Computer Science* team for their support and sponsorship, and EasyChair for providing the reviewing infrastructure.

# Preface

This volume contains the papers presented at COORDINATION 2018: the 20th IFIP WG 6.1 International Conference on Coordination Models and Languages held during June 18–21, 2018, in Madrid, Spain. The conference was co-located with FORTE and DAIS, as part of the DisCoTec federated conferences on distributed computing techniques.

The conference is the premier forum for publishing research results and experience reports on software technologies for collaboration and coordination in concurrent, distributed, and complex systems. The key focus of the conference is the quest for high-level abstractions that can capture interaction patterns and mechanisms occurring at all levels of the software architecture, up to the end-user domain. COORDINATION called for high-quality contributions on the usage, study, formal analysis, design, and implementation of languages, models, and techniques for coordination in distributed, concurrent, pervasive, multi-agent, and multicore software systems.

The Program Committee (PC) of COORDINATION 2018 consisted of 23 top researchers from 12 different countries. In all, 26 submissions were received out of 29 submitted abstracts. All submissions were reviewed by four 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 PC. At the end of the review process, 12 papers were accepted.

The selected papers constituted a program covering a varied range of topics and techniques related to system coordination, including: actor-based coordination, tuple-based coordination, agent-oriented techniques, constraints-based coordination, and finally coordination based on shared spaces. Five of the accepted papers are surveys. This was a new category of submission considered this year to celebrate the 20th edition of COORDINATION. These papers describe important results and successful stories that originated in the context of COORDINATION. The program was further enhanced by an invited talk by Franco Zambonelli from Università degli Studi di Modena-Reggio Emilia (Italy).

The success of COORDINATION 2018 was due to the dedication of many people. We thank the authors for submitting high-quality papers, the PC and their sub-reviewers, for their careful reviews, and lively discussions during the final selection process, and the publicity chair, Francesco Tiezzi, for helping us advertise the CFP. 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 Organizing Committee from Universidad Complutense de Madrid, led by Manuel Núñez, for its contribution in making the logistic aspects of COORDINATION 2018 a success.

# Organization

## Steering Committee

Farhad Arbab	CWI and Leiden University, The Netherlands
Gul Agha	University of Illinois at Urbana-Champaign, USA
Dave Clarke	Uppsala University, Sweden
Wolfgang De Meuter	Vrije Universiteit Brussels, Belgium
Rocco De Nicola	IMT - School for Advanced Studies, Italy
Tom Holvoet	KU Leuven, Belgium
Jean-Marie Jacquet	University of Namur, Belgium
Christine Julien	The University of Texas at Austin, USA
Eva Kühn	Vienna University of Technology, Austria
Alberto Lluch Lafuente	Technical University of Denmark, Denmark
Mieke Massink	ISTI-CNR, Pisa, Italy
Jose Proenca	University of Minho, Portugal
Rosario Pugliese	Università degli Studi di Firenze, Italy
Marjan Sirjani	Malardalen University, Norway and Reykjavik University, Iceland
Carolyn Talcott	SRI International, USA
Vasco T. Vasconcelos	University of Lisbon, Portugal
Mirko Viroli	Università di Bologna, Italy
Gianluigi Zavattaro (Chair)	Università di Bologna, Italy

## Program Committee

Gul Agha	University of Illinois at Urbana-Champaign, USA
Luis Barbosa	University of Minho, Portugal
Jacob Beal	BBN Technologies, USA
Simon Bliudze	Inria, France
Carlos Canal	University of Málaga, Spain
Giovanna Di Marzo Serugendo (Chair)	University of Geneva, Switzerland
Vashti Galpin	University of Edinburgh, UK
Jean-Marie Jacquet	University of Namur, Belgium
Eva Kühn	Vienna University of Technology, Austria
Alberto Lluch Lafuente	Technical University of Denmark, Denmark
Michele Loreti (Chair)	Università di Camerino, Italy
Maxime Louvel	BAG-ERA, France
Mieke Massink	ISTI-CNR, Pisa, Italy
Hernan Melgratti	Universidad de Buenos Aires, Argentina
Andrea Omicini	Università di Bologna, Italy
Sascha Ossowski	University Rey Juan Carlos, Spain



Luca Padovani	Università di Torino, Italy
Rosario Pugliese	Università degli Studi di Firenze, Italy
Marjan Sirjani	Malardalen University, Norway and Reykjavik University, Iceland
Carolyn Talcott	SRI International, USA
Mirko Viroli	Università di Bologna, Italy
Nobuko Yoshida	Imperial College London, UK
Gianluigi Zavattaro	Università di Bologna, Italy

### **Additional Reviewers**

Castellan, Simon	Miculan, Marino
Charalambides, Minas	Neykova, Romyana
Ciancia, Vincenzo	Pianini, Danilo
Ciccozzi, Federico	Plyukhin, Dan
Cimini, Matteo	Re, Barbara
Crass, Stefan	Roldán, Christian
Forcina, Giorgio	Scalas, Alceste
Jaber, Mohamad	Sesum-Cavic, Vesna
Jafari, Ali	Sharifi, Zeinab
Langerak, Rom	Tiezzi, Francesco
Latella, Diego	Tonello, Nicola
Margheri, Andrea	Toninho, Bernardo

# Contents

Space-Time Universality of Field Calculus . . . . .	1
<i>Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli</i>	
Foundations of Coordination and Contracts and Their Contribution to Session Type Theory . . . . .	21
<i>Mario Bravetti and Gianluigi Zavattaro</i>	
Twenty Years of Coordination Technologies: State-of-the-Art and Perspectives . . . . .	51
<i>Giovanni Ciatto, Stefano Mariani, Maxime Louvel, Andrea Omicini, and Franco Zambonelli</i>	
On Multiplicities in Tuple-Based Coordination Languages: The Bach Family of Languages and Its Expressiveness Study . . . . .	81
<i>Denis Darquennes, Jean-Marie Jacquet, and Isabelle Linden</i>	
A Formal Approach to the Engineering of Domain-Specific Distributed Systems . . . . .	110
<i>Rocco De Nicola, Gianluigi Ferrari, Rosario Pugliese, and Francesco Tiezzi</i>	
Rule-Based Form for Stream Constraints . . . . .	142
<i>Kasper Dokter and Farhad Arbab</i>	
Forward to a Promising Future . . . . .	162
<i>Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo</i>	
Aggregation Policies for Tuple Spaces . . . . .	181
<i>Linas Kaminskis and Alberto Lluch Lafuente</i>	
Distributed Coordination Runtime Assertions for the Peer Model . . . . .	200
<i>eva Kühn, Sophie Therese Radschek, and Nahla Elaraby</i>	
Active Objects for Coordinating BSP Computations (Short Paper). . . . .	220
<i>Gaétan Hains, Ludovic Henrio, Pierre Leca, and Wijnand Suijlen</i>	
Boosting Transactional Memory with Stricter Serializability . . . . .	231
<i>Pierre Sutra, Patrick Marlier, Valerio Schiavoni, and François Trahay</i>	

From Field-Based Coordination to Aggregate Computing . . . . .	252
<i>Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini</i>	
<b>Author Index</b> . . . . .	281



# Space-Time Universality of Field Calculus

Giorgio Audrito<sup>1,2(✉)</sup>, Jacob Beal<sup>3</sup>, Ferruccio Damiani<sup>1,2</sup>, and Mirko Viroli<sup>4</sup>

<sup>1</sup> Dipartimento di Informatica, University of Torino, Turin, Italy  
{giorgio.audrito,ferruccio.damiani}@unito.it

<sup>2</sup> Centro di Competenza per il Calcolo Scientifico, University of Torino, Turin, Italy

<sup>3</sup> Raytheon BBN Technologies, Cambridge, MA, USA  
jakebeal@ieee.org

<sup>4</sup> DISI, University of Bologna, Cesena, Italy  
mirko.viroli@unibo.it

**Abstract.** Recent work in the area of coordination models and collective adaptive systems promotes a view of distributed computations as functional blocks manipulating data structures spread over space and evolving over time. In this paper, we address expressiveness issues of such computations, and specifically focus on the *field calculus*, a prominent emerging language in this context. Based on the classical notion of *event structure*, we introduce the *cone Turing machine* as a ground for studying computability issues, and first use it to prove that field calculus is *space-time universal*. We then observe that, in the most general case, field calculus computations can be rather inefficient in the size of messages exchanged, but this can be remedied by an encoding to nearly similar computations with slower information speed. We capture this concept by a notion of *delayed space-time universality*, which we prove to hold for the set of message-efficient algorithms expressible by field calculus. As a corollary, it is derived that field calculus can implement with message-size efficiency all self-stabilising distributed algorithms.

**Keywords:** Distributed computing · Computability · Field calculus

## 1 Introduction

A traditional viewpoint in the engineering of coordination systems is to focus on the primitives by which a single coordinated device (or agent) interacts with others, either by point-to-point interaction, broadcast, or by means of some sort of mediator (a shared space, a channel, an orchestrator, and the like). A global

---

This work has been partially supported by: EU Horizon 2020 project HyVar ([www.hyvar-project.eu](http://www.hyvar-project.eu)), GA No. 644298; ICT COST Action IC1402 ARVI ([www.cost-arvi.eu](http://www.cost-arvi.eu)); Ateneo/CSP D16D15000360005 project RunVar ([runvar-project.di.unito.it](http://runvar-project.di.unito.it)). This document does not contain technology or technical data controlled under either U.S. International Traffic in Arms Regulation or U.S. Export Administration Regulations.

coordination system is then designed as a protocol or workflow of interaction “acts”, regulating synchronisation of computational activities and the exchange of information through messages (among the many, e.g., see [8,25]).

However, a number of recent works originated in the context of distributed intelligent systems (swarm intelligence, nature-inspired computing, multi-agent systems, self-adaptive and self-organising systems), and then impacting coordination models and languages as well, promote a higher abstraction of spatially-distributed collective adaptive systems. In these approaches, system coordination is expressed in terms of how the “collective” actually carries on an overall task, designed in terms of a spatio-temporal data structure to be produced as “output”. Works such as [4,18] survey from various different viewpoints the many approaches that fall under this umbrella, and which we can classify in the following categories: methods that simplify programming of a collective by abstracting individual networked devices (e.g., TOTA [28], Hood [38], chemical models [36]), spatial patterns and languages (e.g., Growing Point Language [13], Origami Shape Language [30]), tools to summarise and stream information over regions of space and time (e.g., TinyDB [27] and Cougar [39]), and finally space-time computing models, e.g. targeting parallel computing (e.g., StarLisp [24], systolic computing [19]).

More recently, field-based computing through the field calculus [14,15] and the generalised framework of aggregate programming [5,34] combine and generalise over the above approaches, by viewing a distributed computation as a pure function, neglecting explicit indication of message-passing and rather focussing on the manipulation of data structures, spread over space and evolving over time. This is achieved by a small set of constructs equipped with a functional composition model that well supports the construction of complex system specifications. More generally, we see the field-calculus in terms of an evolution of distributed systems programming towards higher and higher declarative abstractions.

Some questions then naturally arise: which notions of universality emerge out of such a view of distributed computation? how can we characterise the expressiveness of a set of constructs used as building blocks to program distributed systems? how may non-functional aspects affect such notions? Classical Turing computability is not directly applicable to space-time distributed computations, as it does not capture relevant aspects such as the availability of information at given devices at given moments of time.

In this paper we address these issues by introducing the notions of *cone Turing machine* and *space-time computability*, and use them to prove a universality result for the field calculus—this notion of universality differs from others previously introduced for the field calculus [6], as it is performed in a discrete model rather than a continuous one, and it is more strongly connected to classical Turing computability. We also inspect efficiency aspects, since they deeply affect the “practical” notion of expressiveness: we find examples of space-time functions that would be realised only by field calculus programs that are “message-size-inefficient” (simply, *message-inefficient* henceforth)—i.e., that would rely on increasingly big messages as time passes and information moves around.

Sym.	Meaning	Sym.	Meaning	Sym.	Meaning
$\epsilon$	event identifier	$<$	causality relation	$\mathbf{V}$	set of comput. values
$\epsilon_{\top}$	maximal event	$\rightsquigarrow$	neighbouring relation	$\mathbf{V}(\mathbf{E})$	set of s/t values in $\mathbf{E}$
$\delta$	device identifier	$\upharpoonright$	restriction	$\mathbf{V}(\ast)$	set of s/t values
$E$	set of events	$\rightarrow$	partial function	$\mathbf{V}(\top)$	set of cone s/t values
$\mathbf{E}$	event structure	$\text{LC}(\epsilon)$	past light cone of $\epsilon$	$\Phi$	space-time value
$\mathbb{E}$	augmented event struct.	$\text{CD}(\epsilon)$	set of connected devices	$\mathbf{f}$	space/time function
$\mathbf{ES}$	set of event structures	$A^*$	finite sequences from $A$	$\mathbf{e}$	expression
$\mathbf{EC}$	set of cone event struct.	$\text{TM}_{\text{cone}}$	cone Turing machine	$\mathbf{D}$	set of device identifiers

**Fig. 1.** Table of symbols and notations used throughout this paper.

However, we also find that for each such message-inefficient function there exists a “delayed” version with nearly similar behaviour: it features somewhat slower information speed across devices but can be implemented in a message-efficient way by field calculus. We capture this concept in terms of a stricter notion of *delayed space-time universality*, a property that holds for the set of message-efficient field calculus programs. As a corollary, we also derive an *effective self-stabilisation universality* result, stating that the field calculus is able to provide a message-efficient implementation for *any* self-stabilising distributed algorithm [15].

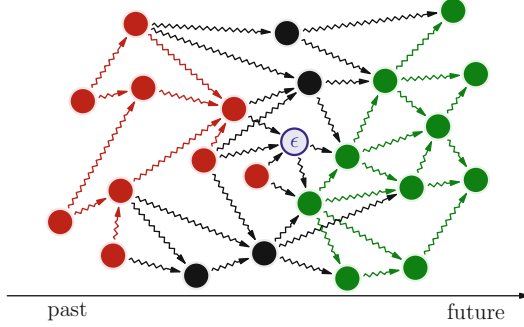
The remainder of this paper is organised as follows: Sect. 2 gives the main definitions of computability this paper is built upon; Sect. 3 introduces the field calculus and proves its universality; Sect. 4 shows a message-efficient but delayed encoding of computations, and discusses the notion of delayed space-time universality; Sect. 5 reviews related works and Sect. 6 concludes with final remarks. Figure 1 summarises the symbols and notations used throughout this paper.

## 2 Space-Time Computability

In order to ground a definition of “Turing-like” computability for distributed functions, two main ingredients are required: a mathematical space of functions, abstracting the essence of distributed computations, and a set of criteria for discarding *impossible* computations. The former can be achieved by translating the main features of distributed computations into mathematical objects: in this case, atomic computing events with communication through message passing. The latter can be achieved by combining physical requirements (i.e., causality) with classical computability requirements [6]. Accordingly, Sect. 2.1 formalises a space of distributed functions, and Sect. 2.2 introduces a Turing-like machine  $\text{TM}_{\text{cone}}$  to ground computability.

### 2.1 Denotation of Space-Time Computations

A number of models have been proposed over the decades to ground distributed computations, each with a different viewpoint and purpose. Most of them boil



**Fig. 2.** A sample event structure, split in events  $\epsilon'$  in the causal past of  $\epsilon$  ( $\epsilon' < \epsilon$ , in red), events in the causal future ( $\epsilon < \epsilon'$ , in green) and concurrent (non-ordered, in black). (Color figure online)

down to two main ingredients: computational *events*, where actual actions take place, and *messages* that are exchanged to carry information between different events. These concepts can be formalised by the notion of *event structure* [23].

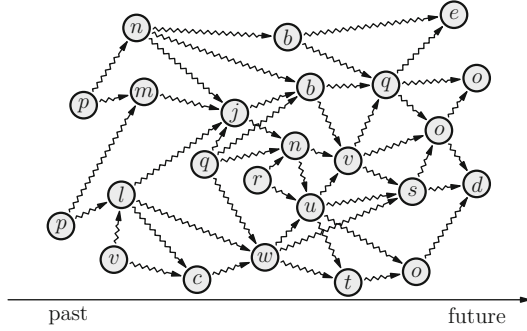
**Definition 1 (Event Structure).** *An event structure  $\mathbf{E} = \langle E, \rightsquigarrow, < \rangle$  is a countable set of events  $E$  together with a neighbouring relation  $\rightsquigarrow \subseteq E \times E$  and a causality relation  $< \subseteq E \times E$ , such that the transitive closure of  $\rightsquigarrow$  forms the irreflexive partial order  $<$  and the set  $\{\epsilon' \in E \mid \epsilon' < \epsilon\}$  is finite for each  $\epsilon$  (i.e.,  $<$  is locally finite). We call **ES** the set of all such event structures.*

Note that the transitive closure condition on  $\rightsquigarrow$  also implies that  $\rightsquigarrow$  is asymmetric and irreflexive. A sample event structure is depicted in Fig. 2, showing how these relations partition events into “causal past,” “causal future,” and “concurrent” subsets with respect to any given event  $\epsilon$ —that is, respectively, events from which information can potentially be carried to  $\epsilon$  in a message, those to which information from  $\epsilon$  can be carried, and events informationally isolated from  $\epsilon$ . Since  $<$  is uniquely induced by  $\rightsquigarrow$ , we shall feel free to omit it whenever convenient, or use its weak form  $\leq$ .<sup>1</sup> Notice that since  $<$  is required to be irreflexive,  $\rightsquigarrow$  has to be an acyclic relation, thus inducing a directed acyclic graph (DAG) structure on  $E$ . In fact,  $\mathbf{E}$  can be thought of as a DAG with a “neighbouring” relation (modelling message passing) and a “reachability” relation (modelling causal dependence). This kind of structure is also compatible with spaces of events equipped with special or general relativity metrics, considering  $\epsilon_1 \rightsquigarrow \epsilon_2$  to be possible only if  $\epsilon_1$  is in its causal past, i.e., is contained in or precedes the past light cone<sup>2</sup> of  $\epsilon_2$ .

Notice that information about which device or devices might be performing that actual computation at each event is completely abstracted away: event

<sup>1</sup> The weak form of a partial order is defined as  $x \leq y$  iff  $x < y$  or  $x = y$ .

<sup>2</sup> In relativity, the past light cone of an event  $\epsilon_2$  comprises all events  $\epsilon_1$  such that photons produced by  $\epsilon_1$  reach the position of  $\epsilon_2$  at the time when  $\epsilon_2$  happens.



**Fig. 3.** Representation of a space-time value  $\Phi$  of literals.

structures aim to model which data may be available at every computational step, no matter on what device the computation may be happening. Thus a series of computations on the same device (whether it is fixed or mobile) can still be accurately modelled by a sequence of events  $\epsilon_1, \dots, \epsilon_n$  such that  $\epsilon_i \rightsquigarrow \epsilon_{i+1}$ , in which message passing is implemented simply by keeping data available on the device for subsequent computations.<sup>3</sup>

The notion of event structure dates back several decades [23], and it has been used to relate many different distributed computation paradigm, such as Petri nets [31] or the actor model [21]: we now use them as a ground for *space-time universality*, building on these previous works. Even though the definition of event structure is usually given just in terms of the causality relation, we have also included the neighbouring relation since it is able to capture message passing details, which are usually needed to interpret actual distributed programs.

The notion of event structure is abstract, but well-suited to ground a semantics for *space-time computations*, intended as “elaborations of distributed data in a network of related events”: the causality ordering of events abstracts time, while the presence of concurrent events abstract spatial dislocation. Following [22], in the remainder of this paper overbar notation denotes metavariables over sequences and the empty sequence is  $\bullet$ : e.g., we use  $\overline{\Phi}$  for the sequence  $\Phi_1, \dots, \Phi_n$ . Similarly, formulas with sequences are duplicated for each set of sequence elements (sequences are assumed to have the same length): e.g.,  $\overline{\Phi}(\epsilon) = \overline{v}$  is a shorthand for  $\Phi_1(\epsilon) = v_1, \dots, \Phi_n(\epsilon) = v_n$ .

**Definition 2 (Space-Time Values).** *Let  $V$  be a denumerable universe of allowed computational values and  $\mathbf{E}$  be a given event structure. A space-time value  $\Phi$  in  $\mathbf{E}$  is an annotation of the graph  $\mathbf{E}$  with labels in  $V$ , that is, a tuple  $\Phi = \langle \mathbf{E}, f \rangle$  with  $f : E \rightarrow V$ , taking  $E$  as the set of events in  $\mathbf{E}$ .*

<sup>3</sup> Note that a computation in this model may be an arbitrarily complex action, so long as it is local: our later formulation will take each event to be an atomic execution of an entire round of a potentially complex program.



**Definition 3 (Space-Time Functions).** Let  $V(\mathbf{E}) = \{\langle \mathbf{E}, f \rangle \mid f : E \rightarrow V\}$  be the set of space-time values in an event structure  $\mathbf{E}$ , and  $V(*) = \bigcup_{\mathbf{E} \in \mathbf{ES}} V(\mathbf{E})$  be the set of all space-time values in any event structure. Then, an  $n$ -ary space-time function in  $\mathbf{E}$  is a partial map<sup>4</sup>  $\mathbf{f} : V(\mathbf{E})^n \rightarrow V(\mathbf{E})$  and an  $n$ -ary space-time function is a partial map  $\mathbf{f} : V(*)^n \rightarrow V(*)$ , defined only for arguments belonging to a same  $V(\mathbf{E})$  and such that for any  $\bar{\Phi}$  in  $V(\mathbf{E}) \cap \text{dom}(\mathbf{f})$ ,  $\mathbf{f}(\bar{\Phi}) \in V(\mathbf{E})$ .

A sample space-time value is depicted in Fig. 3. Notice that space-time values can be used to model data not only *spatially* distributed across devices, but also *temporally* distributed across time. In this way time-evolving inputs and, most importantly, intermediate results of computations (which are naturally time-dependent) are easily represented, attaining maximal generality while ensuring composability of behaviour. Furthermore, since space-time functions  $\mathbf{f}$  are *partial* maps, undefined values for  $\mathbf{f}(\bar{\Phi})$  can model computations that are non-halting or otherwise failing on some event. We assume that a non-halting computation does not constitute a proper event, as it is not “observable” from the external world. The partial outcome of a computation  $\mathbf{f}$  that is non-halting on some event  $\epsilon \in \mathbf{E}$  can still be recovered by restricting  $\mathbf{E}$  to the largest  $\mathbf{E}' \subseteq \mathbf{E}$  on which  $\mathbf{f}$  is defined.

Most space-time functions, however, are not feasible in the physical world due to two main obstacles: inconsistencies between the causality relation and the required behaviour of the function (non-causal functions), and violation of classical constraints on computability (super-Turing functions). We shall see in the following subsection how to implement these two restrictions.

## 2.2 Cone Turing Machine

In order to define causality of a space-time function, it is necessary for the output value in each event  $\epsilon$  to depend only on input values in events  $\epsilon'$  which *may have influenced*  $\epsilon$ , that is, such that  $\epsilon' \leq \epsilon$ . This concept of causality can be captured by the definitions of *event cone* and *cone function*.<sup>5</sup>

**Definition 4 (Event Cone).** An event cone is an event structure  $\mathbf{E}$  with a distinguished unique element  $\epsilon_{\top}$  which is the  $<$ -maximum in  $\mathbf{E}$ , i.e., such that  $\forall \epsilon \in E \ \epsilon \leq \epsilon_{\top}$ . We call  $\mathbf{EC}$  the set of all such event structures.

**Definition 5 (Restriction).** Given an event structure  $\mathbf{E}$  and an event  $\epsilon \in E$ , the  $\epsilon$ -cone in  $\mathbf{E}$ , also called the restriction of  $\mathbf{E}$  to (the causal past of)  $\epsilon$ , is defined as:

$$\mathbf{E} \upharpoonright \epsilon = \langle E \upharpoonright \epsilon, \rightsquigarrow \cap (E \upharpoonright \epsilon)^2, < \cap (E \upharpoonright \epsilon)^2 \rangle$$

where  $E \upharpoonright \epsilon = \{\epsilon' \in E \mid \epsilon' \leq \epsilon\}$ . Analogously, given  $\bar{\Phi} \in V(\mathbf{E})$  and  $\epsilon \in E$ , the restriction of  $\bar{\Phi}$  to  $\epsilon$  is  $\bar{\Phi} \upharpoonright \epsilon = \langle E \upharpoonright \epsilon, \rightsquigarrow \cap (E \upharpoonright \epsilon)^2, f \cap (E \upharpoonright \epsilon) \times V \rangle$ .

<sup>4</sup> With  $A \rightarrow B$  we denote the space of partial functions from  $A$  into  $B$ .

<sup>5</sup> These concepts are closely linked to the notion of causality in physics and its definition in terms of light cones.

For example, the event structure formed by red events and by event  $\epsilon$  in Fig. 2 is an event cone, which is the restriction of the whole structure to  $\epsilon$ .

**Definition 6 (Cone Function).** *Let  $V(\top) = \bigcup_{\mathbf{E} \in \mathbf{EC}} V(\mathbf{E})$  be the set of all space-time values in any event cone. Then, an  $n$ -ary cone function is a partial map  $\mathbf{f}_\top : V(\top)^n \rightarrow V$  defined only for arguments  $\bar{\Phi}$  belonging to a same  $V(\mathbf{E})$ , and the space-time function  $\mathbf{f} : V(*)^n \rightarrow V(*)$  induced by such  $\mathbf{f}_\top$  is such that given  $\bar{\Phi} \in V(\mathbf{E})$ ,  $\mathbf{f}(\bar{\Phi}) = \langle \mathbf{E}, f \rangle$  where  $f(\epsilon) = \mathbf{f}_\top(\bar{\Phi} \upharpoonright \epsilon)$ .<sup>6</sup>*

Notice that the output of a cone function is not a map over space, but a single value in  $V$ , i.e., the value computed at the event  $\epsilon_\top$  on the basis of the history accessible to it in the cone. Note also that when inducing a space-time function, the same cone function is assumed to be applied in each event. However, space-time computations that apply different functions at different times can still be modelled by a single function with an extra input selecting the appropriate behaviour for each event. Since cone functions are able to represent any computation from causally-available inputs, causal space-time functions are precisely those which are induced by cone functions. Thus, computable space-time functions are those induced by a cone function that is *computable*, in the sense that it can be computed by a Turing Machine that operates over cones:

**Definition 7 (Cone Turing Machine).** *Let  $A$  be an alphabet,  $\pi : V \rightarrow A^*$  and<sup>7</sup>  $\pi^\top : V(\top) \rightarrow A^*$  be injective encodings of  $V$  and  $V(\top)$ . A cone Turing machine  $TM_{\text{cone}}^{\mathbf{f}}$  is a deterministic Turing machine with  $n+1$  tapes which given in its input tapes encodings  $\pi^\top(\bar{\Phi})$  of a sequence of space-time values in an event cone  $\mathbf{E}$ , writes in its output tape an encoding  $\pi(\mathbf{v})$  of a value in  $V$  (if it terminates). The cone function  $\mathbf{f}_\top$  induced by  $TM_{\text{cone}}^{\mathbf{f}}$  is such that  $\mathbf{f}_\top(\bar{\Phi}) = \mathbf{v}$  if and only if  $TM_{\text{cone}}^{\mathbf{f}}$  terminates with output  $\pi(\mathbf{v})$  given inputs  $\pi^\top(\bar{\Phi})$ . The space-time function  $\mathbf{f}$  induced by  $TM_{\text{cone}}^{\mathbf{f}}$  is the one induced by the corresponding  $\mathbf{f}_\top$ .*

The specific choice of Turing machine formalisation in this definition is not significant, as all Turing machine formalisations are equivalent for purposes of determining computability, except insofar as its formulation simplifies connection with the field calculus in subsequent sections. The cone Turing machine can be accepted as a ground for space-time computability, since it processes all causally available data in each event in a Turing-complete way. Thus, a space-time function can be defined *computable* as per the following definition.

**Definition 8 (Discrete Space-Time Computability).** *Let  $\mathbf{f} : V(*)^n \rightarrow V(*)$  be an  $n$ -ary space-time function. We say that  $\mathbf{f}$  is computable if and only if there exists a cone Turing machine  $TM_{\text{cone}}^{\mathbf{f}}$  which induces  $\mathbf{f}$ .*

**Definition 9 (Space-Time Universality).** *A programming model (e.g., the field calculus) is space-time universal if and only if it is able to compute every space-time function that can be computed by a cone Turing machine.*

<sup>6</sup> We remark that whenever  $\mathbf{f}_\top(\bar{\Phi} \upharpoonright \epsilon)$  is undefined for some  $\epsilon$  (the computation has not halted), we take  $\mathbf{f}(\bar{\Phi})$  to be undefined as well.

<sup>7</sup> We denote with  $A^*$  the set of finite sequences of values from  $A$ .

### 3 Universality of Field Calculus

The field calculus is a tiny functional calculus capturing the essential elements of *field computations*, much as  $\lambda$ -calculus [12] captures the essence of functional computation and FJ [22] the essence of class-based object-oriented programming. Among other uses, it has been used to define reusable blocks of adaptive distributed algorithms [3, 5], and to define robustness properties [32, 34]. The defining property of *computational fields* is that they allow us to consider a computation from two different viewpoints: under a *global* viewpoint, a field is a distributed data structure manipulated by a network of devices, while under a *local* viewpoint it is just a single value, computed by a device on the basis of information gathered from neighbours. The translation between the two viewpoints is deterministic and automatic, abstracting away message-passing primitives.

Section 3.1 briefly presents the syntax and semantics of the field calculus from the local viewpoint, a detailed account of which can be found in [14, 15]. Section 3.2 extends the “event structure” formalism presented in Sect. 2, enabling convenient formalisation of properties used in the remainder of this paper. Section 3.3 shows that the field calculus is space-time universal, while outlining some inefficiencies that may occur in translating programs into it.

#### 3.1 Field Calculus: Syntax and Semantics

We now present first-order field calculus [14] with a syntax inspired by recent DSL implementations [11] (in place of the prior Scheme-like formulation in [14]), plus a brief overview of its semantics under a *local* viewpoint. In our model, individual devices undergo computation in (local) asynchronous rounds (one per event): in each round, a device sleeps for some time, wakes up, gathers information about messages received from neighbours while sleeping, performs an evaluation of the program, and finally emits a message to all neighbours with information about the outcome of computation before going back to sleep.

The overall evolution of a network of devices is represented operationally through a small-step transition system  $\xrightarrow{act}$  on network configurations  $N = \langle Env; \Psi \rangle$ , where *Env* models the environmental conditions (i.e., network topology, inputs of sensors on each device) and  $\Psi$  models the overall status of the devices in the network at a given time (as a map from device identifiers to environments  $\Theta$ ).

Two types of transitions are considered: *device firings*  $N \xrightarrow{\delta} N'$ , modelling a computational round performed by a device  $\delta$ , and *environment changes*  $N \xrightarrow{env} N'$ , modelling any change in sensor data or network topology *Env*. Such a sequence of transitions can be mapped to a corresponding event structure, comprising an event  $\epsilon$  for each  $\xrightarrow{\delta}$  transition, with neighbouring relations  $\rightsquigarrow$  according to the network topology determined by  $\xrightarrow{env}$  transitions. More precisely, an event  $\epsilon$  on device  $\delta$  (corresponding to a transition  $\langle Env; \Psi_1 \rangle \xrightarrow{\delta} \langle Env; \Psi_2 \rangle$ ) has a neighbouring relation  $\rightsquigarrow$  to the first event  $\epsilon'$  on  $\delta'$  after  $\epsilon$  if and only if  $\delta$  is connected to  $\delta'$  in the network topology *Env*.

$P ::= \bar{F} e$	program
$F ::= \text{def } d(\bar{x}) \{e\}$	function declaration
$e ::= x \mid v \mid \text{let } x = e \text{ in } e \mid f(\bar{e})$ $\mid \text{if}(e_1)\{e_2\}\{e_3\} \mid \text{nbr}\{e\} \mid \text{rep}(e)\{(x)=e\}$	expression
$v ::= \ell \mid \phi$	value
$\ell ::= c(\bar{\ell})$	local value
$\phi ::= \bar{\delta} \mapsto \bar{\ell}$	neighbouring field value
$f ::= d \mid b$	function name

**Fig. 4.** Syntax of field calculus.

Operational semantics within a single device is formalised by the judgement “ $\delta; \Theta \vdash e_{\text{main}} \Downarrow \theta$ ”, to be read “expression  $e_{\text{main}}$  evaluates to  $\theta$  on  $\delta$  with respect to environment  $\Theta$ ”, where  $\theta$  is an ordered tree of values tracking the results of all evaluated sub-expressions of  $e_{\text{main}}$ , and  $\Theta$  is a map from neighbour devices  $\delta_i$  (possibly including  $\delta$  itself) to the  $\theta_i$  produced in their last firing. Mapped into our language of space-time computation,  $\theta$  is the value of a space-time function at event  $\epsilon$  and  $\Theta$  is the set of values of that function at events  $\epsilon' \rightsquigarrow \epsilon$ .

Figure 4 presents the syntax of field calculus.<sup>8</sup> A program  $P$  consists of a sequence of function declarations and of a main expression  $e$ . A function declaration  $F$  defines a (possibly recursive) function, with  $d$  the function name,  $\bar{x}$  the parameters and  $e$  the body. An expression  $e$  can be:

- a variable  $x$ , either a function formal parameter or local to a **let**- or **rep**-expression;
- a value  $v$ , either a *local value* (e.g., numbers, literals) defined through data constructors  $c(\bar{\ell})$ , or a *neighbouring field value*  $\phi$  (a map  $\bar{\delta} \mapsto \bar{\ell}$  from neighbours to local values) which is allowed to appear in intermediate computations but *not* in source programs;
- a **let**-expression **let**  $x = e_0$  **in**  $e$ , which is evaluated by computing the value  $v_0$  of  $e_0$  and then yielding as result the value of the expression obtained from  $e$  by replacing all the occurrences of the variable  $x$  with the value  $v_0$ ;
- a function call  $f(\bar{e})$ , where  $f$  can be a *declared function*  $d$  or a *built-in function*  $b$ , such as accessing sensors, mathematical and logical operators, or data structure operations;
- a conditional branching **if**( $e_1$ ) $\{e_2\}\{e_3\}$ , where  $e_1$  is a Boolean expression;
- a **nbr**-expression **nbr**{ $e$ }, modelling neighbourhood interaction and producing a neighbouring field value  $\phi$  that represents an “observation map” from neighbours to their latest evaluation of  $e$ ;
- or a **rep**-expression **rep**( $e_1$ ) $\{(x)=e_2\}$ , evolving a local state through time by evaluating an expression  $e_2$ , substituting variable  $x$  with the value calculated

<sup>8</sup> Note field calculus has also been extended to support higher-order functions [14, 15]: since this calculus is a proper subset and the space and time operations are identical, all results for this calculus apply to the higher-order formulation as well.

for the **rep**-expression at the previous computational round (in the first round  $\mathbf{x}$  is substituted with the value of  $\mathbf{e}_1$ ). Although this first-order version of the calculus does not model anonymous functions (differently from the higher-order version in [15]),  $(\mathbf{x})\Rightarrow\mathbf{e}_2$  can be understood as an anonymous function with parameter  $\mathbf{x}$  and body  $\mathbf{e}_2$ .

Values associated with data constructors  $\mathbf{c}$  of arity zero (e.g., literal values) are written by omitting the empty parentheses, i.e., we write  $\mathbf{c}$  instead of  $\mathbf{c}()$ . In case  $\mathbf{b}$  is a binary built-in operator, we allow infix notation to enhance readability: e.g., we shall sometimes write  $1 + 2$  for  $+(1, 2)$ .

A correct matching between messages exchanged and **nbr** sub-expressions is ensured by a process called *alignment*, which navigates the value-trees  $\theta$  of neighbours in the environment  $\Theta$  as sub-expressions of the main expression  $\mathbf{e}_{\text{main}}$  are accessed. This process interacts subtly with branching statements  $\text{if}(\mathbf{e}_1)\{\mathbf{e}_2\}\{\mathbf{e}_3\}$ : since no matching of messages from different **nbr** sub-expressions is allowed, computation of  $\mathbf{e}_2$  in devices that selected the first branch cannot interact with devices computing  $\mathbf{e}_3$ . This effectively splits the computation into two fully isolated sub-networks (devices evaluating  $\mathbf{e}_1$  to **True**, and those evaluating it to **False**).

### 3.2 Augmented Event Structures

In field calculus, as in most distributed computing paradigms, the semantics is *device-dependent*: in particular, neighbouring links  $\rightsquigarrow$  connecting subsequent events on the same device (state preservation) have a different role than links connecting events on different devices (message passing). This choice reflects practical implementation details of distributed computing networks, but it is not captured by the abstract concept of *event structure* (Definition 1).

However, it is still possible to use the framework in Sect. 2 for the field calculus. In fact, a function  $\mathbf{f}$  in field calculus always corresponds to a space-time function (Definition 3) with a number of extra input arguments (modelling environmental information) in each event:

- the device  $\delta$  where the event takes place;<sup>9</sup>
- local sensor information (e.g., time clock, temperature, etc.);
- relational sensor information (e.g., physical distance from other devices).

Note that relational sensor information is just a special case of local sensor information, in which the value returned is a map over neighbouring events.

Due to the special role played by these extra input parameters, it will be convenient to consider an event structure together with its associated environmental inputs to state the properties that will be investigated in the next sections:

**Definition 10 (Augmented Event Structure).** *An augmented event structure is a tuple  $\mathbb{E} = \langle \mathbf{E}, \Phi \rangle$  consisting of an event structure  $\mathbf{E}$  together with a number of space-time values  $\Phi$  (including device information).<sup>10</sup>*

<sup>9</sup> We assume that device identifiers  $\delta$  are taken among a denumerable set  $\mathbf{D}$ .

<sup>10</sup> We assume that a finite number of devices may occur in augmented event structures.

```

// previous round value of v
def older(v, null) {
  1st(rep (pair(null, null)) { (old) => pair(2nd(old), v) })
}
// gathers values from causal past events into a labelled DAG
def gather(node, dag) {
  let old = older(dag, dag_empty()) in
  let next = dag_join(unionhood(old, nbr{dag}), node) in
  if (next == node) { dag } {
    gather(node, dag_union(dag, next))
  }
}
def f_field(e, v...) {
  f( gather(dag_node(e, v...), dag_node(e, v...)) )
}

```

**Fig. 5.** Translation `f_field` of a Turing computable cone function `f` into field calculus, given event information as additional input.

When functions are interpreted in augmented event structures, the provided space-time values are then supplied as inputs to the functions (or indirectly used to define sensor built-in functions).

### 3.3 Space-Time Universality

As outlined in Sect. 3.1, the field calculus operational semantics is defined through a set of syntax-directed rules, involving data available in (and computed by) events in the causal past of each firing event. Since the cone Turing machine can process inputs from arbitrary events in the causal past of the current event in a Turing-complete way, it follows that every space-time function that is computable for the field calculus is also computable for the cone Turing machine. Conversely, in order for the field calculus to be space-time universal it needs to be (i) Turing-complete for fully local computations and (ii) able to gather values from arbitrary events in the causal past. Condition (i) is easily matched by the field calculus, as it is assumed that built-in functions on local values, together with branching-expressions and recursive function declarations, provide the required Turing-completeness. Condition (ii) holds as shown by the following theorem.

**Definition 11 (Rank).** *The rank of  $\epsilon$  in  $\mathbf{E}$  is the maximum length  $\text{rank}(\epsilon)$  of a path  $\epsilon_1 \rightsquigarrow \epsilon_2 \rightsquigarrow \dots \rightsquigarrow \epsilon$  ending in  $\epsilon$ .*

**Definition 12 (Distance).** *The distance of  $\epsilon' < \epsilon$  from  $\epsilon$  is the minimum length  $n$  of a path  $\epsilon' = \epsilon_0 \rightsquigarrow \dots \rightsquigarrow \epsilon_n = \epsilon$  connecting them.*

**Theorem 1 (Field Calculus Space-Time Universality).** *Let  $f$  be a Turing computable cone function. Then there is a field calculus function `f_field` that produces the same outputs as  $f$  in any augmented event structure including event information.*

*Proof (sketch).* Figure 5 shows a possible translation, assuming event information  $\mathbf{e}$  as additional input. Function `gather` collects values of its arguments from causal past events into a labelled DAG, which is fed to the cone function `f`. The code is based on the following built-in functions, which we assume to be available.<sup>11</sup>

`pair(v1, v2)`: constructs a pair  $\langle v_1, v_2 \rangle$ .

`1st(v)`, `2nd(v)`: returns the first (resp. second) element of a pair  $v$ .

`dag_empty()`: returns an empty DAG structure.

`dag_node( $\epsilon, \bar{v}$ )`: constructs a DAG consisting in a single node  $\epsilon$  with labels  $\bar{v}$ .

`dag_union( $G_1, G_2$ )`: returns  $G_1 \cup G_2$ , merging duplicate entries of a same event.

`dag_join( $G, n$ )`: returns  $G$  with an added node  $n$ , connected to each sink of  $G$ .

`unionhood( $G, \phi$ )`: computes the union of  $G$  with each neighbour graph in  $\phi$ .

Whenever `gather` is called, it computes in `next` the result of joining the current `node` with neighbour `dag` values. If no neighbour is aligned to the current function call, `next = node` hence `dag` is returned. Otherwise, a recursive call is made with the enlarged graph `dag ∪ next`. Every event performs strictly more recursive calls than all of its neighbour events, so the recursion depth in event  $\epsilon$  is  $\text{rank}(\epsilon)$ . On the  $n$ -th recursive call, values from events at distance  $\leq n$  are computed and fed to the following recursive calls. Thus, `gather` collects values from all events  $\epsilon' < \epsilon$ .

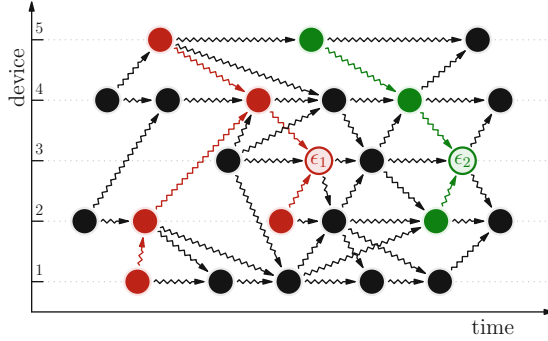
## 4 Delayed and Stabilising Universality

In this section we address an efficiency shortcoming of the field calculus that concerns the size of messages exchanged across devices (Sect. 4.1), and accordingly draw more tailored notions of universality: a notion of *delayed universality* that relaxes temporal constraints of computation by which efficiency in message size can be practically recovered (Sect. 4.2) and *stabilising universality* that focusses on the ability of expressing all stabilising computations (Sect. 4.3).

### 4.1 On Message-Size Efficiency and Delays in Field Calculus

As shown in the proof of Theorem 1, function `gather` performs  $\text{rank}(\epsilon)$  recursive calls in each event  $\epsilon$ , broadcasting an increasing part of the past event cone  $\mathbf{E} \upharpoonright \epsilon$  through expression `nbr{dag}` in each of them. Thus, the total messages exchanged have  $O(|\mathbf{E} \upharpoonright \epsilon| \cdot \text{rank}(\epsilon))$  size, which is larger by a factor of  $\text{rank}(\epsilon)$  than what would be necessary in other distributed models of computation. In fact, a Turing machine would be able to receive full cone reconstructions  $\mathbf{E} \upharpoonright \epsilon'$  from neighbour events, join them and in turn broadcast a final reconstructed value by uniting them. This is not possible in field calculus due to its alignment mechanism: message exchange is bound to `nbr`-expressions, which first *send* a

<sup>11</sup> All those functions except for `unionhood` are totally local, hence can be implemented through any Turing-complete set of built-in functions (e.g. the minimal `zero`, `-`, `<`).



**Fig. 6.** Past light cones of events  $\epsilon_1$  and  $\epsilon_2$  in a sample augmented event structure. Note that the  $\epsilon_1$  (red) light cone includes device 2 twice, due to the break in state memory, while the  $\epsilon_2$  (green) light cone does not contain device 1, since all states in the event cone of  $\epsilon_2$  can take a path that includes state memory. (Color figure online)

message and then *receive* a response, whereas the previous procedure would require a program to first *receive* data in order to compute the message to be *sent*. This obstacle can be circumvented only by `nbr` nesting (as in `f_field`), which leads to the larger message size. Not all field calculus computations require such nesting, though, only those requiring communication without delay, i.e., that access information in the past light cone of an event, as defined in the following.

**Definition 13 (Past Light Cone).** *Let  $\mathbb{E}$  be an augmented event structure,  $\epsilon$  be an event. The past light cone  $LC(\epsilon)$  of  $\epsilon$  is the set of  $\epsilon'$  such that  $\epsilon' < \epsilon$  and no path  $\epsilon' = \epsilon_0 \rightsquigarrow \dots \rightsquigarrow \epsilon_n = \epsilon$  passes through two events  $\epsilon_i, \epsilon_j$  on a same device.*

Figure 6 represents the past light cone of two given events. Intuitively, events are in the past light cone of  $\epsilon$  if they are barely able to reach  $\epsilon$ , i.e., any delay of information propagation (i.e., “waiting” one round in a device) would break connection with them. In this case, communication is more fragile since `rep` constructs are of no use: each form of communication enabled by `rep` constructs requires waiting at least one round on a device. For a message to be exchanged from events  $\epsilon' \in LC(\epsilon)$ , a field calculus program needs to execute a number of nested `nbr` statements at least equal to their relative distance, each of them contributing to the overall message size.

## 4.2 Delayed Universality of the Field Calculus

For events sufficiently far from the past light cone, a slower and more light-weight pattern of data collection with respect to `nbr` nesting can also be effective [3, 5]: the combined use of `nbr` and `rep` statements, as in the following.

```
rep (initial) { (old) => combine(old, nbr{old}) }
```



```

// gathers values from past events into a labelled tree with marked nodes
def gather(e, v...) {
  rep (dag_empty()) { (old) =>
    // for each neighbour, link its old DAG with the neighbour event
    let neigh = nbr{ dag_join(old, dag_node(e, False, v...)) } in
    // merge the obtained trees, and link the result with the current event
    dag_join(unionhood(old, neigh), dag_node(e, True, v...))
  }
}
// step to successor event e' of e in G, if it exists and the gathering is complete up to e'
def next_completed_event(G, e) {
  if (last_event(G, e) or not dag_true(dag_restrict(G, next_event(G, e))) { e } {
    next_event(G, e)
  }
}
def f_delayed(e, v...) {
  let G = gather(e, v...) in
  let delayed_event = rep (e) { (old) => next_completed_event(G, old) } in
  f ( dag_restrict(G, delayed_event) )
}

```

**Fig. 7.** Delayed translation `f_delayed` of a Turing computable cone function `f` into field calculus, given event information as additional input. Notice that a single `nbr` statement (line 5) is executed.

In this highly common pattern of field calculus, data from the previous event on the same device is combined with data from the *preceding event* to each neighbour event. The data flow induced by this pattern is necessarily slower than that of `nbr` nesting, however, it only requires a single `nbr` statement and hence messages carrying on a single data—with no expansion with rank. As we shall show in Theorem 2, this pattern is also able to mimic the behaviour of any space-time computable function with a finite *delay*, provided that the event structure involved is *persistent* and *fair*.

**Definition 14 (Persistence).** *An augmented event structure  $\mathbb{E}$  is persistent if and only if for each device  $\delta$ , the events  $\epsilon$  in  $\mathbb{E}$  corresponding to that device form a totally ordered  $\rightsquigarrow$ -chain.*

**Definition 15 (Fairness).** *An augmented event structure  $\mathbb{E}$  is fair if and only if for each event  $\epsilon_0$  and device  $\delta$ , there exists an event  $\epsilon$  on  $\delta$  such that  $\epsilon_0 < \epsilon$ .*

Notice that only countably infinite event structures can be fair.

**Definition 16 (Delayed Functions).** *Let  $f, g : V(*)^n \rightarrow V(*)$  be  $n$ -ary space-time functions. We say that  $g$  is a delay of  $f$  if and only if for each persistent and fair event structure  $\langle \mathbb{E}, \overline{\Phi} \rangle$  there is a surjective and weakly increasing<sup>12</sup> map  $\pi : E \rightarrow E$  such that  $g(\overline{\Phi})(\epsilon) = f(\overline{\Phi})(\pi(\epsilon))$  for each  $\epsilon$ .*

**Theorem 2 (Field Calculus Effective Delayed Universality).** *Let  $f : V(*)^n \rightarrow V(*)$  be a computable space-time function. Then there exists a field calculus function `f_delayed` which executes a single `nbr` statement and computes a space-time function  $g : V(*)^n \rightarrow V(*)$  which is a delay of  $f$ .*

<sup>12</sup> A map  $\pi : E \rightarrow E$  is weakly increasing if and only if  $\epsilon_1 < \epsilon_2 \Leftrightarrow \pi(\epsilon_1) < \pi(\epsilon_2)$ .

*Proof (sketch).* Figure 7 shows a possible translation, assuming event information  $\mathbf{e}$  as additional input. Function `gather` collects input values from past events into a labelled DAG, with an additional boolean label indicating whether all neighbours of the event are present in the graph. The DAG is then restricted to the most recent event for which the whole past event cone has already been gathered, and finally fed to the cone function  $\mathbf{f}$ . The code is based on the same built-in functions used in Theorem 1, together with the following.

`last_event( $G, \epsilon$ )`: true iff  $\epsilon$  has no successor event in the same device in  $G$ .  
`next_event( $G, \epsilon$ )`: returns the event  $\epsilon'$  following  $\epsilon$  in the same device in  $G$ .  
`dag_restrict( $G, \epsilon$ )`: returns the restriction  $G \upharpoonright \epsilon$ .  
`dag_true( $G$ )`: true iff every node in  $G$  has `True` as first label.

We assume all these functions are available, and that operator `unionhood` prefers label `True` against `False` when merging nodes with different labels.

Since the event structure is fair and persistent, data flow between devices is possible in many different ways: for any event  $\epsilon$  and device  $\delta$ , we can find a path  $\epsilon \rightsquigarrow \dots \rightsquigarrow \epsilon_\delta$  ending in  $\delta$  such that no two consecutive  $\rightsquigarrow$  crossing different devices are present. Thus, data about  $\epsilon$  is eventually gathered in  $\epsilon_\delta$ .

The delay implied by the above translation is proportional to the hop-count diameter of the network considered: in fact, a transmission path is delayed by one round for every device crossing in it. In most cases, this delay is sufficiently small for the translation to be fruitfully used in practical applications [2, 3].

### 4.3 Stabilising Universality of the Field Calculus

Since field calculus is able to efficiently perform computations with a certain delay, it means that it can also efficiently perform those computations whose goal is expressed by the spatial *computation limit* to be eventually reached, as defined by the well-known classes of *stabilising* and *self-stabilising* programs [32].

**Definition 17 (Stabilising Values).** *A space-time value  $\Phi$  in an augmented event structure  $\mathbb{E}$  is stabilising if and only if for each device  $\delta$ , there exists an event  $\epsilon_0$  on  $\delta$  such that for each subsequent  $\epsilon > \epsilon_0$  on  $\delta$ ,  $\Phi(\epsilon) = \Phi(\epsilon_0)$ .*

*The limit  $\lim(\Phi)$  of a stabilising value  $\Phi$  is the map  $m : \mathbf{D} \rightarrow \mathbf{V}$  such that  $m(\delta) = v$  if for all  $\epsilon$  on  $\delta$  after a certain  $\epsilon_0$ ,  $\Phi(\epsilon) = v$ .*

**Definition 18 (Stabilising Structures).** *Given an event  $\epsilon$  in an augmented event structure  $\mathbb{E}$ , the set  $\text{CD}(\epsilon)$  of connected devices in  $\epsilon$  is:*

$$\text{CD}(\epsilon) = \{\delta \mid \exists \epsilon_\delta \text{ on } \delta \text{ such that } \epsilon_\delta \rightsquigarrow \epsilon\}.$$

*An augmented event structure  $\langle \mathbf{E}, \bar{\Phi} \rangle$  is stabilising if and only if it is fair, persistent and both  $\bar{\Phi}$  and  $\text{CD}$  (interpreted as a space-time value) are stabilising.*

**Definition 19 (Stabilising Functions).** *An  $n$ -ary space-time function  $\mathbf{f} : \mathbf{V}(\ast)^n \rightarrow \mathbf{V}(\ast)$  is stabilising if and only if given any stabilising augmented event structure  $\langle \mathbf{E}, \bar{\Phi} \rangle$ , the output  $\mathbf{f}(\bar{\Phi})$  is stabilising. Two stabilising functions  $\mathbf{f}, \mathbf{g} : \mathbf{V}(\ast)^n \rightarrow \mathbf{V}(\ast)$  are equivalent if and only if given any stabilising augmented event structure  $\langle \mathbf{E}, \bar{\Phi} \rangle$ , their outputs have the same limits  $\lim(\mathbf{f}(\bar{\Phi})) = \lim(\mathbf{g}(\bar{\Phi}))$ .*

**Theorem 3 (Delayed to Stabilising).** *Let  $\mathbf{f} : \mathbf{V}(\ast)^n \rightarrow \mathbf{V}(\ast)$  be a stabilising space-time function, and  $\mathbf{g}$  be a delay of  $\mathbf{f}$ . Then  $\mathbf{g}$  is stabilising and equivalent to  $\mathbf{f}$ .*

*Proof.* Let  $\delta$  be any device, and  $\epsilon$  be the first event on  $\delta$  such that the output of  $\mathbf{f}$  has stabilised to  $\mathbf{v}$  on  $\delta$  after  $\epsilon$ . Let  $\pi : E \rightarrow E$  be the function such that  $\mathbf{g}$  is a delay of  $\mathbf{f}$  as in Definition 16, and let  $\epsilon'$  be such that  $\pi(\epsilon') = \epsilon$  by surjectivity of  $\pi$ . Then  $\mathbf{g}$  stabilises to  $\mathbf{v}$  on  $\delta$  after  $\epsilon'$ , concluding the proof.

Combining Theorems 2 and 3, we directly obtain the following corollary.

**Corollary 1 (Field Calculus Effective Stabilising Universality).** *Let  $\mathbf{f} : \mathbf{V}(\ast)^n \rightarrow \mathbf{V}(\ast)$  be a computable and stabilising space-time function. Then there exists a field calculus function `f.stabilising` which executes a single `nbr` statement and computes a space-time function  $\mathbf{g} : \mathbf{V}(\ast)^n \rightarrow \mathbf{V}(\ast)$  which is equivalent to  $\mathbf{f}$ .*

Stabilisation guarantees that a limit exists, but in general such a limit could highly depend on “transient environmental changes”. A stronger property, more useful in practical applications is *self-stabilisation* [1, 20, 26, 35], additionally guaranteeing full-independence to transient changes as defined in the following.

**Definition 20 (Self-Stabilising Functions).** *An  $n$ -ary space-time function  $\mathbf{f} : \mathbf{V}(\ast)^n \rightarrow \mathbf{V}(\ast)$  is self-stabilising if and only if it is stabilising and given two stabilising event structures  $\langle \mathbf{E}, \bar{\Phi}^1 \rangle$  and  $\langle \mathbf{E}, \bar{\Phi}^2 \rangle$  such that  $\lim(\bar{\Phi}^1) = \lim(\bar{\Phi}^2)$  and  $\lim(\text{CD}^1) = \lim(\text{CD}^2)$ , we have that  $\lim(\mathbf{f}(\bar{\Phi}^1)) = \lim(\mathbf{f}(\bar{\Phi}^2))$ .*

Since self-stabilising functions are a subclass of stabilising functions, stabilising universality trivially implies self-stabilising universality.

## 5 Related Work

Studying the expressiveness of coordination models is a traditional topic in the field of coordination models and languages. As such, a good deal of literature exists that we here classify and compare with the notions defined in this paper.

A first thread of papers, which forms the majority of the available works, study expressiveness of coordination models using a traditional approach of concurrency theory based on the following conceptual steps: (i) isolating coordination primitives of existing models, (ii) developing a core calculus formalising how their semantics affect the coordination space (production/reception of

messages, triggering of events or process continuations, injection/extraction of data-items/tuples), and finally *(iii)* drawing a bridge between the core calculus rewrite behaviour with the input/output behaviour of Turing machines, to inspect universality or compare expressiveness of different sets of primitives.

Notable examples of this approach include the study of expressiveness of Linda coordination primitives in [8], of event notification in data-driven languages in [9], of movement constructs in mobile environments in [10], and of timed coordination models in [25]. A slightly different approach is taken in [17], where the focus is expressiveness of a language for expressing coordination rules to program “the space of interaction”: the methodology is similar, but here expressiveness neglects the behaviour of coordinated entities, focussing just on the shared-space enacting coordination.

Other approaches start instead from the consideration that the dimension of interaction may require a more sophisticated machinery than comparison against Turing machines. A classical position paper following this line is Peter Wegner’s work in [37], which however did not turn into successful frameworks to study interaction models. Modular embedding is proposed in [16] as an empowering of standard embedding to compare relative expressiveness of concurrent languages, which has been largely used as a tool by the community studying the theory of concurrency.

The approach to universality and expressiveness presented in this paper sits in between the two macro approaches above. On the one hand, our notion of expressiveness is strictly linked to the classic Turing framework, and focusses on the global computation that a system of coordinated entities can carry on. Critically, however, it is based on denoting computations as *event structures*, a long-standing notion used to formalise distributed systems of various sorts [23, 29]. In this paradigm, each single node has the power of a Turing machine, all nodes execute the same behaviour, and what matters is the resulting spatio-temporal configuration of events, which describes the overall system execution and not just its final outcome. A somewhat similar stance is taken in [6, 7], in which field computations are considered as providing a space-time continuous effect, obtained with limit of density of devices and frequency of their operation going to infinity—an approach that we plan to soon connect with the one presented here.

## 6 Conclusions

In this paper, we proposed the *cone Turing machine* as a foundation for space-time computability in distributed systems based on event structures, and use it to study the expressiveness of field calculus. Field calculus is proved universal: but in practice, some computations can be ineffective for they would need exchange of messages with increasing size as time passes. By a form of abstraction which releases some constraints on temporal execution (i.e., accepting some delay), field calculus is shown instead to be both universal and message-size efficient. As a key corollary, we proved that field calculus can efficiently implement self-stabilising

computations, a class of computations which lately received considerable interest [3, 20, 26, 33, 35].

In the future, we plan to further investigate the interplay of expressiveness and efficiency for relevant classes of distributed algorithms, both in a discrete and continuous setting, with the goal of designing new declarative programming constructs for distributed systems.

## References

1. Altisen, K., Corbineau, P., Devismes, S.: A framework for certified self-stabilization. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 36–51. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-39570-8\\_3](https://doi.org/10.1007/978-3-319-39570-8_3)
2. Audrito, G., Casadei, R., Damiani, F., Viroli, M.: Compositional blocks for optimal self-healing gradients. In: Self-Adaptive and Self-Organizing Systems (SASO), 2017, pp. 91–100. IEEE (2017)
3. Audrito, G., Damiani, F., Viroli, M.: Optimally-self-healing distributed gradient structures through bounded information speed. In: Jacquet, J.-M., Massink, M. (eds.) COORDINATION 2017. LNCS, vol. 10319, pp. 59–77. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-59746-1\\_4](https://doi.org/10.1007/978-3-319-59746-1_4)
4. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, Chap. 16, pp. 436–501. IGI Global (2013)
5. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. *IEEE Comput.* **48**(9), 22–30 (2015)
6. Beal, J., Viroli, M., Damiani, F.: Towards a unified model of spatial computing. In: 7th Spatial Computing Workshop (SCW 2014) (2014)
7. Beal, J., Viroli, M., Pianini, D., Damiani, F.: Self-adaptation to device distribution in the Internet of Things. *TAAS* **12**(3), 12:1–12:29 (2017)
8. Busi, N., Gorrieri, R., Zavattaro, G.: On the expressiveness of Linda coordination primitives. *Inf. Comput.* **156**(1–2), 90–121 (2000)
9. Busi, N., Zavattaro, G.: On the expressiveness of event notification in data-driven coordination languages. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 41–55. Springer, Heidelberg (2000). [https://doi.org/10.1007/3-540-46425-5\\_3](https://doi.org/10.1007/3-540-46425-5_3)
10. Busi, N., Zavattaro, G.: On the expressive power of movement and restriction in pure mobile ambients. *Theor. Comput. Sci.* **322**(3), 477–515 (2004)
11. Casadei, R., Viroli, M.: Towards aggregate programming in Scala. In: First Workshop on Programming Models and Languages for Distributed Computing, PMLDC 2016, pp. 5:1–5:7. ACM, New York (2016)
12. Church, A.: A set of postulates for the foundation of logic. *Ann. Math.* **33**(2), 346–366 (1932)
13. Coore, D.: Botanical computing: a developmental approach to generating inter connect topologies on an amorphous computer. Ph.D. thesis, MIT, Cambridge, MA, USA (1999)
14. Damiani, F., Viroli, M., Beal, J.: A type-sound calculus of computational fields. *Sci. Comput. Program.* **117**, 17–44 (2016)
15. Damiani, F., Viroli, M., Pianini, D., Beal, J.: Code mobility meets self-organisation: a higher-order calculus of computational fields. In: Graf, S., Viswanathan, M. (eds.) FORTE 2015. LNCS, vol. 9039, pp. 113–128. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-19195-9\\_8](https://doi.org/10.1007/978-3-319-19195-9_8)

16. Deboer, F., Palamidessi, C.: Embedding as a tool for language comparison. *Inf. Comput.* **108**(1), 128–157 (1994)
17. Denti, E., Natali, A., Omicini, A.: On the expressive power of a language for programming coordination media. In: ACM SAC 1998, pp. 169–177 (1998)
18. Dobson, S., Denazis, S., Fernández, A., Gäiti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. *TAAS* **1**(2), 223–259 (2006)
19. Engstrom, B.R., Cappello, P.R.: The SDEF programming system. *J. Parallel Distrib. Comput.* **7**(2), 201–231 (1989)
20. Faghieh, F., Bonakdarpour, B., Tixeuil, S., Kulkarni, S.: Specification-based synthesis of distributed self-stabilizing protocols. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 124–141. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-39570-8\\_9](https://doi.org/10.1007/978-3-319-39570-8_9)
21. Hewitt, C., Bishop, P.B., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: IJCAI 1973, pp. 235–245 (1973)
22. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* **23**(3), 396–450 (2001)
23. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978)
24. Lasser, C., Massar, J., Miney, J., Dayton, L.: Starlisp Reference Manual. Thinking Machines Corporation, Cambridge (1988)
25. Linden, I., Jacquet, J., Bosschere, K.D., Brogi, A.: On the expressiveness of timed coordination models. *Sci. Comput. Program.* **61**(2), 152–187 (2006)
26. Lluch-Lafuente, A., Loret, M., Montanari, U.: Asynchronous distributed execution of fixpoint-based computational fields. *CoRR* abs/1610.00253 (2016)
27. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.* **36**, 131–146 (2002)
28. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: the TOTA approach. *ACM Trans. Softw. Eng. Methodol.* **18**(4), 1–56 (2009)
29. Mattern, F., et al.: Virtual time and global states of distributed systems. *Parallel Distrib. Algorithms* **1**(23), 215–226 (1989)
30. Nagpal, R.: Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics. Ph.D. thesis, MIT, Cambridge, MA, USA (2001)
31. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. *Theor. Comput. Sci.* **13**, 85–108 (1981)
32. Pianini, D., Beal, J., Viroli, M.: Improving gossip dynamics through overlapping replicates. In: Lluch Lafuente, A., Proença, J. (eds.) COORDINATION 2016. LNCS, vol. 9686, pp. 192–207. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-39519-7\\_12](https://doi.org/10.1007/978-3-319-39519-7_12)
33. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. *ACM Trans. Model. Comput. Simul.* **28**(2), 16:1–16:28 (2018)
34. Viroli, M., Beal, J., Damiani, F., Pianini, D.: Efficient engineering of complex self-organising systems by self-stabilising fields. In: Self-Adaptive and Self-Organizing Systems (SASO), 2015, pp. 81–90. IEEE (2015)
35. Viroli, M., Damiani, F.: A calculus of self-stabilising computational fields. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 163–178. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-662-43376-8\\_11](https://doi.org/10.1007/978-3-662-43376-8_11)

36. Viroli, M., Pianini, D., Montagna, S., Stevenson, G., Zambonelli, F.: A coordination model of pervasive service ecosystems. *Sci. Comput. Program.* **110**, 3–22 (2015)
37. Wegner, P.: Why interaction is more powerful than algorithms. *Commun. ACM* **40**(5), 80–91 (1997)
38. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*. ACM Press (2004)
39. Yao, Y., Gehrke, J.: The cougar approach to in-network query processing in sensor networks. *SIGMOD Record* **31**, 9–18 (2002)



# Foundations of Coordination and Contracts and Their Contribution to Session Type Theory

Mario Bravetti<sup>(✉)</sup> and Gianluigi Zavattaro<sup>(✉)</sup>

Department of Computer Science and Engineering & Focus Team, Inria,  
University of Bologna, Bologna, Italy  
{mario.bravetti,gianluigi.zavattaro}@unibo.it

**Abstract.** We briefly recall results obtained in twenty years of research, spanning across the old and the new millennium, on the expressiveness of coordination languages and on behavioural contracts for Service-Oriented Computing. Then, we show how the techniques developed in those contexts are currently contributing to the clarification of aspects that were unclear about session types, in particular, asynchronous session subtyping that was considered decidable since 2009, while it was proved to be undecidable in 2017.

## 1 Introduction

Shared dataspace and the so-called generative communication paradigm [27] attracted a lot of attention since the initial years of research about foundations of coordination models and languages. Linda [18], probably the most popular language based on this coordination model, is based on the idea that concurrent processes interact via a shared dataspace, the so-called Tuple Space (TS for short), where the information needed to coordinate the activities are introduced and retrieved. After its insertion in the TS, a datum becomes equally accessible to all processes, but it is bound to none. In this way, the interaction among concurrent processes is decoupled in space and time, principles useful in the development of modular and scalable concurrent and distributed systems.

Concerning foundational studies on Linda-like coordination languages, it appeared immediately clear that techniques borrowed from the tradition of concurrency theory could be naturally applied. At the first two editions of the Coordination conference, two process calculi based on Linda were proposed by De Nicola and Pugliese [22] and by Busi et al. [14]. In particular, the latter started a line of research on the expressiveness of Linda-like coordination primitives that exploited, besides process calculi, also Petri nets. For instance, Petri nets were used in [13], to prove that a Linda process calculus with input, output and test-for-absence is *not* Turing complete if the semantics for output is *unordered*, i.e., there is an unpredictable delay between the execution of an output and the actual availability of the emitted datum in the TS. It is interesting to



recall that, on the other hand, the same calculus is Turing complete if an *ordered* semantics is considered, i.e. an emitted datum is immediately available in the TS after the corresponding output is executed. Turing completeness was proved by showing an encoding of a Turing powerful formalism, namely Random Access Machines (RAMs), which is a computational model based on registers that can be incremented, decremented and tested for emptiness.

The success of the Linda coordination model was witnessed by the development, at the end of the 90s, of Linda-based middlewares from main ICT vendors like IBM and Sun Microsystems, which proposed T-Spaces and JavaSpaces, respectively. The basic Linda coordination model was extended with primitives for event notification, time-out based data expiration, and transactions. The techniques for evaluating the expressive power of Linda languages had to become more sophisticated to cope with these additional primitives. In particular, Petri nets with transfer and reset arcs [23] were adopted to cope also with the new coordination mechanisms for event notification [16] and for temporary data [15].

During the initial years of the new millennium, Service-Oriented Computing (SOC) emerged as an alternative model for developing communication-based distributed systems. In particular, the large diffusion of Web Services called for the development of new languages and techniques for service composition. The idea, at the basis of SOC, is to conceive an ecosystem of services that expose operations that can be combined to realize new applications. To support this idea, it is necessary for the services to be equipped with an interface that, besides describing the offered operations and the format of the exchanged messages, defines the conversation protocols, i.e., the expected flow of invocations of the operations. These interfaces, in particular the specification of the conversation protocol, are also called *behavioural contracts*.

Process calculi contributed to the development of theories for behavioural contracts. This line of research was initiated by Carpineti et al. [17], for the case of client-server composition, and by Bravetti and Zavattaro [6], for multiparty service compositions. The latter is particularly significant for the so-called service choreographies, i.e., systems in which there exists no central orchestrator, responsible for invoking all the other services in the system, because services reciprocally interact. Behavioural contract theories focused mainly on the investigation of appropriate notions of correctness for service compositions (i.e., define when a system based on services is free from communication errors) and on the characterization of notions of compatibility between services and behavioural contracts (i.e., define when a service conforms to a given behavioural contract).

For multiparty composition, a fairness based notion of correctness, called *compliance*, was proposed for the first time in [6]: a system is correct if, whatever state can be reached, there exists a continuation of the computation that yields a final state in which *all* services have successfully completed. Given the notion of compliance, it is possible to define also a natural notion of *refinement* for behavioural contracts: a refinement is a relation among contracts such that, given a set of compliant contracts  $C_1, \dots, C_n$ , each contract  $C_i$  can be *independently* replaced by any of its possible refinements  $C'_i$ , and the overall system obtained

by composition of  $C'_1, \dots, C'_n$  is still compliant. Contract refinement can then be used to check whether a service conforms with a behavioural contract: it is sufficient to verify if the communication behaviour of the service refines the behavioural contract. This, in fact, implies that such service can be safely used wherever a service is expected with the behaviour specified by the contract.

A negative result in the theory of behavioural contracts is that, in general, the union of two refinement relations is not guaranteed to be itself a refinement. This implies the impossibility to define a maximal notion of refinement. For this reason, most of the effort in the line of research on behavioural contracts initiated in [6] has been dedicated to the identification of interesting subclasses of contracts for which the maximal refinement exists. Such classes are: contracts with *output persistence* [6] (i.e. output actions cannot be avoided when a state is entered in which they are ready to be executed), contract refinement preserving *strong compliance* [7] (i.e. as soon as an output is ready to be executed, a receiver is guaranteed to be ready to receive it), and *asynchronously communicating* contracts [10] (i.e. communication is mediated by fifo buffers). In the first two of these three cases, it has been also possible to provide a sound algorithmic characterization of the corresponding maximal refinements.

To the best of our knowledge, characterizing algorithmically the maximal contract refinement in case of asynchronous communication is still an open problem. The main source of difficulty derives from the fact that, due to the presence of unbounded communication buffers, systems of asynchronously communicating contracts are infinite-state, even if contracts are finite-state. In the light of this difficulty, we tried to take inspiration from work on session types, where asynchronous communication has been investigated since the seminal work by Honda et al. [29] (recipient of the most influential POPL'08 paper award). Session types can be seen as a simplification of contracts obtained by imposing some limitations: there are only two possible choices, *internal* choice among distinct outputs and *external* choice among distinct inputs.

The counterpart of contract refinement in the context of session types is *subtyping* [26]. If we consider asynchronous communication, both contract refinement and session subtyping can admit a refinement/subtype to perform the communication actions in a different order. For instance, given a contract/type that performs an input followed by an output, a refinement/subtype can *anticipate* the output before the input, because such output can be *buffered* and actually received afterwards. Asynchronous session subtyping was already studied by Mostrous et al. in [38], where also an algorithm for checking subtyping was presented. Upon studying this algorithm we noticed an error in its proof of termination: if, while checking subtyping, the buffer grows unboundedly, the proposed procedure does not terminate. Subsequently, Bravetti et al. [4] and Lange and Yoshida [33] independently proved that asynchronous session subtyping is actually undecidable. Our experience in the modeling of Turing complete formalism (see the above discussion about encoding RAMs in the Linda process calculus) helped in finding an appropriate Turing powerful model to be encoded in terms of asynchronous session subtyping. In particular, we were able to present

a translation from a Queue Machine  $M$  (a computational model similar to push-down automata, but with a queue instead of a stack) to a pair of session types that are in asynchronous subtyping relation if and only if  $M$  does not terminate. Then, undecidability of asynchronous session subtyping directly follows from the undecidability of the halting problem for Queue Machines.

These negative results opened the problem of identifying significant classes of session types for which asynchronous subtyping can be decided. Currently, the most interesting fragments have been identified in [4, 5, 33]. In the former, an algorithm is presented for the case in which one of the two types is completely deterministic, i.e. all choices –both internal and external– have one branch only. In the latter, we have considered single-out (and single-in) session types, meaning that in both types to be checked all internal choices (resp. external choices) have one branch only. In the design of our algorithm we have been inspired by our expertise in the analysis of the expressiveness of Linda process calculi. In particular, the analysis techniques in Petri nets with transfer and reset arcs (our tools to prove decidability results) are based on the notion of well quasi ordering (wqo): while generating an infinite sequence of elements, it is guaranteed to eventually generate an element that is greater –with respect to the wqo– of an already generated element. Similarly to the procedure in [38], our algorithm checks a sequence of judgements, but differently from [38], termination is guaranteed because there exists a wqo on judgements, and the algorithm can terminate when a judgement greater than an already checked one is considered.

*Outline of the Paper.* The remainder of this paper is divided in three main parts: in Sect. 2 we discuss the techniques used to investigate the expressive power of the Linda coordination model; in Sect. 3 we recall the main results concerning behavioural contracts; and in Sect. 4 we present recent results on session types for asynchronous communication. The additional Sect. 5 reports some concluding remarks.

## 2 Process Calculi to Study the Expressiveness of Linda

Several Linda process calculi have been proposed in the literature, like PAL [22], LLinda [21], and the Linda calculi in [12, 14], just to mention some of them (published in the proceedings of the first two editions of the Coordination conference). Those calculi have been defined for several purposes: investigate from a foundational viewpoint coordination models and languages, develop novel formal analysis techniques for coordinated systems, or drive the implementation of fully-fledged coordination languages. In this section, we focus on process calculi used to prove results about the expressive power of primitives for Tuple Spaces. Different techniques have been adopted, spanning from Turing completeness (used, e.g., in [13]) to modular embeddings (used, e.g., in [12]). To give the reader a more precise idea of such techniques, in particular the former, we report some results taken from [13, 16].

We start by introducing a Linda calculus with four basic primitives: (i) *out* to introduce a new datum into a shared repository called dataspace, (ii) *in*

**Table 1.** The transition system for processes (symmetric rule of PAR omitted).

$$\text{PRE} : \frac{j \in I}{\sum_{i \in I} \alpha_i.P_i \xrightarrow{\alpha_j} P_j} \quad \text{REPL} : !\alpha.P \xrightarrow{\alpha} !\alpha.P \mid P \quad \text{PAR} : \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

to consume one datum available in the repository, (iii) *notify* to register the interest in future emissions of a specific datum (when such datum will be emitted a new instance of a given process will be spawned), and (iv) *tfa* to test for the absence of a specific datum. This calculus is Turing complete (result taken from [13]) while the fragment without *tfa* is not (result taken from [16]). The proof of Turing completeness is by reduction from Random Access Machines, while the non Turing completeness of the considered fragment was proved in [16] by resorting to a (non Turing complete) variant of Petri nets, namely Petri nets with transfer arcs (see, e.g., [23]); here, we present an alternative proof technique based on well quasi orderings (which are actually used to prove decidability results for Petri nets with transfer arcs).

We now formally report the definition of a Linda-based process calculus, starting from the syntax of processes.

**Definition 1 (Linda Processes).** *Let Name, ranged over by  $a, b, \dots$ , be a denumerable set of names. Processes are defined by the following grammar:*

$$\begin{aligned} \alpha &::= in(a) \mid out(a) \mid notify(a, P) \mid tfa(a) \\ P &::= \sum_{i \in I} \alpha_i.P_i \mid !\alpha.P \mid P \mid P \end{aligned}$$

The basic process actions are  $in(a)$  and  $out(a)$  denoting the consumption or emission, respectively, of one instance of datum  $a$  from/into the shared dataspace. Two additional primitives are considered:  $notify(a, P)$  to register a listener interested in future emissions of the datum  $a$  (the reaction to such event will be the spawning of process  $P$ ) and  $tfa(a)$  to test for the absence of the datum  $a$ . The term  $\sum_{i \in I} \alpha_i.P_i$  denotes a process ready to perform any of the action  $\alpha_i$ , and then proceed by executing the corresponding continuation  $P_i$ . We use  $\mathbf{0}$  to denote such process in case  $I = \emptyset$ , and we will usually omit trailing  $\mathbf{0}$ . The replicated process  $!\alpha.P$  performs an initial action  $\alpha$  and then spawns the continuation  $P$  by keeping  $!\alpha.P$  in parallel. Two parallel processes  $P$  and  $Q$  are denoted with  $P \mid Q$ . In the following we will use the notation  $\prod_{i \in I} P_i$  to denote the parallel composition of processes indexed on the set of indexes  $I$ .

We now formalize the operational semantics for Linda processes in terms of a transition system with four kinds of labels:  $in(a)$ ,  $out(a)$ ,  $notify(a, P)$ , and  $tfa(a)$ . The transition system is the least one satisfying the axioms and rules reported in Table 1. The PRE rule simply allows a sum process to execute one of its initial actions and then continue with the corresponding continuation. REPL allows  $!\alpha.P$  to execute  $\alpha$ , spawn an instance of the continuation  $P$ , and keep  $!\alpha.P$  in parallel. Finally, PAR allows a parallel process to execute an action.

We now move to the syntax and semantics of systems, in which processes are equipped with a dataspace and a multiset of registered listeners.

**Table 2.** The reduction relation for systems (brackets in singletons are omitted).

$$\begin{array}{c}
\frac{P \xrightarrow{in(a)} P'}{\langle P, \mathcal{S} \uplus a, \mathcal{L} \rangle \rightarrow \langle P', \mathcal{S}, \mathcal{L} \rangle} \quad \frac{P \xrightarrow{notify(a,Q)} P'}{\langle P, \mathcal{S}, \mathcal{L} \rangle \rightarrow \langle P', \mathcal{S}, \mathcal{L} \uplus (a, Q) \rangle} \\
\frac{P \xrightarrow{out(a)} P' \quad \mathcal{L} = \{(a, P_i) \mid i \in I\} \uplus \{(b_j, Q_j) \mid \forall j. b_j \neq a\}}{\langle P, \mathcal{S}, \mathcal{L} \rangle \rightarrow \langle P' \mid \prod_{i \in I} P_i, \mathcal{S} \uplus a, \mathcal{L} \rangle} \\
\frac{P \xrightarrow{tfa(a)} P' \quad a \notin \mathcal{S}}{\langle P, \mathcal{S}, \mathcal{L} \rangle \rightarrow \langle P', \mathcal{S}, \mathcal{L} \rangle}
\end{array}$$

**Definition 2 (Linda Systems).** A system  $S$  is a tuple  $\langle P, \mathcal{S}, \mathcal{L} \rangle$  where  $P$  is a process,  $\mathcal{S}$  is the dataspace (i.e. a multiset over  $Name$ ), and  $\mathcal{L}$  are the registered listeners (i.e. a multiset of pairs  $(a, P)$ ).

The semantics of systems is defined by the minimal transition system satisfying the rules in Table 2. The transitions for systems allow processes to (i) consume data from the shared dataspace, (ii) register a new listener, (iii) introduce a new datum in the shared dataspace with the corresponding spawning of the processes in the interested listeners, and (iv) test for the absence of one datum in the dataspace. We use  $\uplus$  to denote multiset union.

In the following, we will consider *termination* and *divergence* of systems.

**Definition 3 (Termination and Divergence).** Given a system  $S$ , we say that  $S$  terminates, denoted  $S \downarrow$ , if there exists at least an ending computation, i.e., there exist  $S_1, \dots, S_n$  such that  $S \rightarrow S_1 \rightarrow S_2 \dots \rightarrow S_n \not\rightarrow$ , where  $S_n \not\rightarrow$  means that there exists no system  $S'$  s.t.  $S_n \rightarrow S'$ . We say that  $S$  diverges, denoted  $S \uparrow$ , if there exists at least one infinite computation, i.e., there exist one infinite sequence of systems  $S_1, \dots, S_n, \dots$  such that  $S \rightarrow S_1 \rightarrow S_2 \dots \rightarrow S_n \rightarrow \dots$ .

## 2.1 Turing Completeness

The Turing completeness of the Linda calculus was proved in [13]. Actually, the calculus in that paper considered a variant of the *tfa* primitive called *inp*, which is a non-blocking version of *in*: *inp* has two possible continuations  $P$  and  $Q$ , the first one activated in case the consumption succeeds, and a second one activated if the message of interest is absent. This primitive coincides with the process  $in(a).P + tfa(a).Q$ . The proof of Turing completeness was based on an encoding of Random Access Machines (RAMs) [42], a well known register-based Turing powerful formalism. Here, we rephrase that encoding by exploiting *in*, *out* and *tfa*, without using *notify*. For this reason, in this subsection, we do not consider listeners and denote systems simply with pairs  $\langle P, \mathcal{S} \rangle$  of processes and dataspace.

A RAM, denoted in the following with  $R$ , is a computational model composed of a finite set of registers  $r_1, \dots, r_n$ , that can hold arbitrarily large natural numbers, and of a program composed by indexed instructions  $(1 : I_1), \dots, (m : I_m)$ ,

that is a sequence of numbered instructions, like arithmetic operations (on the contents of registers) or conditional jumps. An internal state of a RAM is given by  $(i, c_1, \dots, c_n)$  where  $i$  is the program counter indicating the next instruction to be executed, and  $c_1, \dots, c_n$  are the current contents of the registers  $r_1, \dots, r_n$ , respectively.

The computation starts from the first instruction ( $1 : I_1$ ) and terminates when the program counter points to an undefined instruction. In other terms, the initial configuration is  $(1, 0, \dots, 0)$  and the computation continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction outside the valid range  $1, \dots, m$  is reached.

Formally, we indicate by  $(i, c_1, \dots, c_n) \rightarrow_R (i', c'_1, \dots, c'_n)$  the fact that the configuration of the RAM  $R$  changes from  $(i, c_1, \dots, c_n)$  to  $(i', c'_1, \dots, c'_n)$  after the execution of the  $i$ -th instruction.

In [36] it is shown that the following two instructions are sufficient to model every recursive function:  $(i : Succ(r_j))$  to add 1 to the content of register  $r_j$ ;  $(i : DecJump(r_j, s))$  that, if the content of register  $r_j$  is not zero, decreases it by 1 and goes to the next instruction, otherwise jumps to instruction  $s$ .

We start presenting how to encode RAM instructions into Linda processes:

$$\begin{aligned} \llbracket (i : Succ(r_j)) \rrbracket &= !in(p_i).out(r_j).out(p_{i+1}) \\ \llbracket (i : DecJump(r_j, s)) \rrbracket &= !in(p_i).(in(r_j).out(p_{i+1}) + tfa(r_j).out(p_s)) \end{aligned}$$

The idea is to represent the content of the register  $r_j$  with a corresponding number of instances of the datum  $r_j$  in the dataspace. The program counter is modeled by a datum  $p_i$  indicating that the  $i$ -th instruction is the next one to be executed. The modeling of the  $i$ -th instruction always starts with the consumption of the  $p_i$  datum. An increment instruction on  $r_j$  simply produces one datum  $r_j$ , while a *DecJump* instruction either consumes one datum  $r_j$  or tests for the absence of such datum. After these operations, the subsequent program counter datum is emitted.

We now present the full definition of our encoding. Let  $R$  be a RAM with  $m$  instructions, and let  $(i, c_1, \dots, c_n)$  be one of its configurations. With

$$\llbracket (i, c_1, \dots, c_n) \rrbracket_R = \langle \prod_{1 \leq i \leq m} \llbracket (i : I_i) \rrbracket, \{p_i, \underbrace{r_1, \dots, r_1}_{c_1 \text{ times}}, \dots, \underbrace{r_n, \dots, r_n}_{c_n \text{ times}}\} \rangle$$

we denote the system representing the configuration  $(i, c_1, \dots, c_n)$ .

We now formally recall the correctness of the encoding (proved in [13]) from which we conclude the Turing completeness of the Linda calculus.

**Theorem 1.** *Let  $R$  be a RAM. Given  $\rightarrow^3$  as a notation for three successive reductions of systems, we have that:*

- *Soundness:* if  $\llbracket (i, c_1, \dots, c_n) \rrbracket_R \rightarrow^3 Q$  then there exists a unique configuration  $(j, c'_1, \dots, c'_n)$  such that  $Q = \llbracket (j, c'_1, \dots, c'_n) \rrbracket_R$  and  $(i, c_1, \dots, c_n) \rightarrow_R (j, c'_1, \dots, c'_n)$
- *Completeness:* if  $(i, c_1, \dots, c_n) \rightarrow_R (j, c'_1, \dots, c'_n)$  then  $\llbracket (i, c_1, \dots, c_n) \rrbracket \rightarrow^3 \llbracket (j, c'_1, \dots, c'_n) \rrbracket$

As a consequence of the Turing completeness of the Linda calculus, we have that both termination and divergence are undecidable, namely, given a system  $S$ , it is in general undecidable whether  $S \downarrow$  or  $S \uparrow$ . Notice that Turing completeness does not depend on *notify*, as this primitive is not used in the modeling of RAMs reported in the previous subsection.

## 2.2 Decidability of Divergence in the Calculus Without *tfa*

We now focus on the expressive power of *tfa*, by investigating whether the calculus continues to be Turing complete even if we remove such primitive. Hence, we consider the fragment of the Linda calculus without the *tfa* primitive. We will observe that this fragment is no longer Turing powerful, as divergence turns out to be decidable, namely, given a system  $S$ , it is always possible to decide whether  $S \uparrow$ . This result was proved in [16] by presenting an encoding of a Linda calculus with *in*, *out*, and *notify* into Petri nets with transfer arcs: an extension of place/transition Petri nets for which properties like coverability, or the existence of an infinite firing sequence, are still decidable. In this section we present a novel alternative proof (at least for this considered Linda calculus) inspired by the theory of well structured transition systems (WSTS) [24]: we first define an ordering on systems configurations which is proved to be a well quasi ordering, and then we show that the operational semantics is compatible with such ordering.

We start by recalling the notion of well quasi ordering (see, e.g., [24]).

**Definition 4.** *A reflexive and transitive relation on a set  $X$  is called quasi ordering. A well quasi ordering (wqo) is a quasi ordering  $(X, \leq)$  such that, for every infinite sequence  $x_1, x_2, \dots$ , there exist  $i < j$  with  $x_i \leq x_j$ .*

In the following we will use the following well known results for wqo:

- Consider a finite set  $S$  and the set of its multisets  $\mathcal{M}(S)$ . We have that multiset inclusion is a well quasi ordering for the latter, namely  $(\mathcal{M}(S), \subseteq)$  is a wqo, where  $\subseteq$  denotes multiset inclusion.
- Consider  $k$  well quasi orderings  $(X_1, \leq_1), \dots, (X_k, \leq_k)$ . Let  $\Pi$  be the cartesian product  $X_1 \times \dots \times X_k$  and  $\leq^k$  be the natural extension of the orderings  $\leq_1, \dots, \leq_k$  to  $\Pi$ , i.e.,  $(x_1, \dots, x_k) \leq^k (y_1, \dots, y_k)$  if and only if  $x_1 \leq_1 y_1, \dots, x_k \leq_k y_k$ . We have that  $(\Pi, \leq^k)$  is a wqo.

We now recall, taking it from [24], the notion of compatibility<sup>1</sup> of a transition system w.r.t. an ordering.

**Definition 5.** *A transition system  $(X, \rightarrow)$  is compatible with respect to an ordering  $(X, <)$  if, given two states  $s, t \in X$  of the transition system such that  $s < t$  and  $s \rightarrow s'$  for some  $s'$ , then there exists  $t'$  such that  $s' < t'$  and  $t \rightarrow t'$ .*

<sup>1</sup> The compatibility notion used in this paper is named *strict* compatibility in [24].

A known result from the theory of WSTS (Theorem 4.6 in [24]) is that it is decidable to establish the existence of an infinite computation in a transition system compatible with a wqo. To exploit this result, we now define a wqo on systems  $\langle P, \mathcal{S}, \mathcal{L} \rangle$ . To do this we will make use of the above result stating that multiset inclusion over multisets with a finite domain is a wqo. This result can be directly applied to dataspaces  $\mathcal{S}$  and registered listeners  $\mathcal{L}$  as they are multisets, while this is not possible on processes  $P$  that are terms with a given syntax. Hence, it is necessary to give an interpretation of such terms  $P$  as multisets: this can be obtained by extracting from a term  $P$  the multiset of sequential or replicated processes constituting  $P$ . Formally, given the process  $P$  we define inductively, on its syntactic structure, the multiset  $m(P)$  as follows:

$$m\left(\sum_{i \in I} \alpha_i.P_i\right) = \left\{ \sum_{i \in I} \alpha_i.P_i \right\} \quad m(!\alpha.P) = \{!\alpha.P\} \quad m(P|Q) = m(P) \uplus m(Q)$$

We are now ready to define the following ordering on systems:

$$\langle P, \mathcal{S}, \mathcal{L} \rangle \leq_S \langle P', \mathcal{S}', \mathcal{L}' \rangle \Leftrightarrow m(P) \subseteq m(P') \wedge \mathcal{S} \subseteq \mathcal{S}' \wedge \mathcal{L} \subseteq \mathcal{L}'$$

We now prove that this ordering  $\leq_S$  on systems is a wqo for the set of systems that are reachable from a given initial system.

**Proposition 1.** *Let  $S_0 = \langle P_0, \mathcal{S}_0, \mathcal{L}_0 \rangle$  be an initial system and let  $Sys$  be the set of systems that are reachable from  $S_0$  according to the reduction relation defined in Table 2. We have that  $(Sys, \leq_S)$  is a wqo.*

*Proof.* Let  $S = \langle P, \mathcal{S}, \mathcal{L} \rangle$  be a system reachable from the initial system  $S_0 = \langle P_0, \mathcal{S}_0, \mathcal{L}_0 \rangle$ . It is easy to see that the data in  $\mathcal{S}$  and the listeners in  $\mathcal{L}$  are taken from finite domains given, respectively, by the parameters of the primitives *out* and *notify* occurring in the initial process  $P_0$  (plus data or listeners already available in  $\mathcal{S}_0$  or  $\mathcal{L}_0$ ). We now observe that also the sequential or replicated processes in  $m(P)$  are taken from a finite domain. In fact,  $P$  is the parallel composition of terms that already occur in the initial process  $P$  or in the listeners in  $\mathcal{L}_0$ . This is because the reduction relation defined in Table 2 does not generate new processes, but simply consumes initial actions in front of already available sequential or replicated processes or spawns processes already present in listeners.

Hence (for the first of the two well known results recalled for wqo) we can conclude that multiset inclusion is a wqo for the multisets  $m(P)$  associated to the reachable processes  $P$ , as well as for the reachable dataspaces  $\mathcal{S}$  and listeners  $\mathcal{L}$ . The ordering  $\leq_S$  is the natural ordering for the cartesian product of these last three wqo, hence it is also a wqo (for the second of the recalled results).  $\square$

We now observe that the reduction relation for systems defined in Table 2 is compatible with the ordering  $\leq_S$ .

**Proposition 2.** *Let  $S, S'$  and  $T$  be three systems, in which the *tfa* primitive does not occur, such that  $S \rightarrow S'$  and  $S \leq_S T$ . We have that there exists a system  $T'$  such that  $T \rightarrow T'$  and  $S' \leq_S T'$ .*



*Proof.* We first observe that, for every pair of processes  $P$  and  $P'$  such that  $m(P) \subseteq m(P')$ , if  $P \xrightarrow{\alpha} Q$  then there exists also  $Q'$  such that  $P' \xrightarrow{\alpha} Q'$  and  $m(Q) \subseteq m(Q')$ . This can be proved by induction on the structure of  $P$ . If  $P = \sum_{i \in I} \alpha_i.R_i$  (resp.  $P = !\alpha.R$ ) then  $P'$  is the parallel composition of terms including also  $P$ . Let  $Q$  be the process in the r.h.s. of the transition inferred on  $P$  by the rule PRE (resp. REPL) in Table 1. Such transition can be inferred (by the same rule plus possible successive applications of PAR) also on  $P'$ : let  $Q'$  be the process in the r.h.s. of such transition. As the same PRE (resp. REPL) rule is initially applied, we have that  $Q'$  is the parallel composition of processes including also  $Q$ , hence  $m(Q) \subseteq m(Q')$ . In the inductive case, we have that the last rule applied in the transition  $P \xrightarrow{\alpha} Q$  is PAR, and the thesis directly follows from the inductive hypothesis on the transition used in the premise of the application of PAR.

Consider now three systems  $S$ ,  $S'$  and  $T$ , in which the *tfa* primitive does not occur, such that  $S \leq_S T$  and  $S \rightarrow S'$ . The latter implies that, given  $S = \langle P, \mathcal{S}_S, \mathcal{L}_S \rangle$  and  $S' = \langle P', \mathcal{S}_{S'}, \mathcal{L}_{S'} \rangle$ , there exists  $\alpha$  such that  $P \xrightarrow{\alpha} P'$ . Consider now  $T = \langle Q, \mathcal{S}_T, \mathcal{L}_T \rangle$ . Having  $S \leq_S T$ , we also have  $m(P) \subseteq m(Q)$ . For the above observation, then also  $Q \xrightarrow{\alpha} Q'$  for a process  $Q'$  such that  $m(P') \subseteq m(Q')$ . Given this transition  $Q \xrightarrow{\alpha} Q'$ , by the rules in Table 1, we also have that  $T \rightarrow T' = \langle Q', \mathcal{S}_{T'}, \mathcal{L}_{T'} \rangle$  with  $S' \leq_S T'$ . The latter is a consequence of  $m(P') \subseteq m(Q')$ ,  $\mathcal{S}_{S'} \subseteq \mathcal{S}_{T'}$  and  $\mathcal{L}_{S'} \subseteq \mathcal{L}_{T'}$ . The last two statements follow from the fact that  $\mathcal{S}_{T'}$  (resp.  $\mathcal{L}_{T'}$ ) is obtained from  $\mathcal{S}_T$  (resp.  $\mathcal{L}_T$ ) by applying the same modification applied to  $\mathcal{S}_S$  (resp.  $\mathcal{L}_S$ ) to obtain  $\mathcal{S}_{S'}$  (resp.  $\mathcal{L}_{S'}$ ); hence multiset inclusion  $\mathcal{S}_S \subseteq \mathcal{S}_T$  (resp.  $\mathcal{L}_S \subseteq \mathcal{L}_T$ ) is preserved.  $\square$

We can finally conclude with our decidability result.

**Theorem 2.** *Let  $S = \langle P, \mathcal{S}, \mathcal{L} \rangle$  be a system in which the *tfa* primitive does not occur. It is decidable whether  $S \uparrow$ .*

*Proof.* Direct consequence of Propositions 1 and 2 and the result taken from the theory of WSTS (Theorem 4.6 in [24]) recalled above.  $\square$

We conclude by recalling other related results about the un/decidability of  $S \downarrow$ , i.e. the existence of a terminating computation. In [16] it is proved that  $S \downarrow$  is undecidable in the fragment of the Linda calculus, without *tfa*, considered in this subsection: hence we conclude that the Linda calculus with *in*, *out* and *notify* is in between decidability and undecidability, namely,  $S \uparrow$  is decidable while  $S \downarrow$  is not. The undecidability proof consists of an encoding of RAMs which is nondeterministic: the Linda system corresponding to a RAM could have several alternative computations, but all the computations that are not faithful w.r.t. the modeled RAM are guaranteed to be divergent. Hence, the Linda system has a terminating computation if and only if the corresponding RAM terminates.

In the same paper [16], it is also discussed that if we remove also the primitive *notify* from the calculus, also  $S \downarrow$  becomes decidable. This follows from the possibility to faithfully encode the fragment of the calculus with only *in* and *out* into classical place/transition Petri nets, for which it is possible to decide the

reachability of any marking from which no other transitions can be fired. This additional result allows us to conclude that adding the primitive *notify* strictly increases the expressive power of the Linda calculus with only *in* and *out*.

### 3 Behavioural Contracts

Behavioural contracts are used to describe the message-passing behaviour of processes. The adoption of process calculi to the specification and analysis of behavioural contracts was initiated by Fournet et al. [25], who proposed to specify contracts as CCS-like processes. They also defined a notion of conformance between processes and contracts following a substitution principle: a process conforms to a contract if it can replace it in any context without adding additional stuck behaviour. Contract have been subsequently studied in the context of service oriented computing: contracts for client-service interaction have been proposed by Carpineti et al. [17] and then independently extended along different directions by, e.g., Bravetti and Zavattaro (see e.g. [6–8]) by Laneve and Padovani [32], by Castagna et al. [19], and Barbanera and de'Liguoro [1].

All such theories of contracts introduce, under different assumptions, notions of *contract refinements* that can be seen as generalizations of the notion of conformance initially studied in [25]: a contract refines another one if it can safely replace it in any possible context. To give to the reader an idea of such techniques, here we report a contract theory discussed in [8, 9], for synchronous communication, and [10], for the asynchronous case. In particular, the latter represents the unique contract theory, to the best of our knowledge, specifically tailored to asynchronous communication.

More precisely, the contract theory that we present is based on the following ingredients: the notion of correct contract composition, the definition of contract refinement, and its algorithmic characterization. The theory considers both synchronous and asynchronous communication, excluding the algorithmic characterization which is available only for the synchronous case.

We start by presenting the formal definition of behavioural contracts as it appears in [2]. Contracts can be seen as a representation of the communication actions that can be performed at a certain location over the network. We assume a denumerable set of action names  $\mathcal{N}$ , ranged over by  $a, b, c, \dots$  and a denumerable set  $Loc$  of location names, ranged over by  $l, l', l_1, \dots$ . We use  $\tau \notin \mathcal{N}$  to denote an internal (unsynchronizable) action. Contracts are denoted adopting a basic process algebra with prefixes over  $\{\tau, a, \bar{a}_l \mid a \in \mathcal{N}, l \in Loc\}$ , denoting internal, input, and output action, respectively. Notice that a destination location is specified for outputs. Such a process algebra is a simple extension of basic CCS [35] with successful termination denoted by “**1**” (whereas the traditional null process “**0**” denotes a failure or a deadlock).

**Definition 6 (Behavioural Contracts).** *We consider a denumerable set of contract variables  $Var$  ranged over by  $X, Y, \dots$ . The syntax of contracts is defined by the following grammar*

**Table 3.** Semantic rules for contracts (symmetric rules omitted).

$$\begin{array}{c}
\mathbf{1} \xrightarrow{\checkmark} \mathbf{0} \qquad \alpha.C \xrightarrow{\alpha} C \\
\frac{C \xrightarrow{\lambda} C'}{C+D \xrightarrow{\lambda} C'} \qquad \frac{C\{\text{rec}X.C/X\} \xrightarrow{\lambda} C'}{\text{rec}X.C \xrightarrow{\lambda} C'}
\end{array}$$

$$\begin{array}{c}
C ::= \mathbf{0} \mid \mathbf{1} \mid \alpha.C \mid C+C \mid X \mid \text{rec}X.C \\
\alpha ::= \tau \mid a \mid \bar{a}_l
\end{array}$$

where  $\text{rec}X._$  is a binder for the process variable  $X$  denoting recursive definition of processes. We assume that in a contract  $C$  all process variables are bound. In the following we will omit trailing “ $\mathbf{1}$ ” when writing contracts.

The operational semantics of contracts is defined in terms of a transition system labeled over  $L = \{a, \bar{a}_l, \tau, \checkmark \mid a \in \mathcal{N}, l \in \text{Loc}\}$ , ranged over by  $\lambda, \lambda', \dots$ , obtained by the rules in Table 3 (plus a symmetric rule for choice). We use the notation  $C\{-/_-\}$  to denote syntactic replacement. Semantic rules are the standard ones, apart from that of term  $\mathbf{1}$ , which performs a  $\checkmark$  transition denoting successful termination. The semantics of a contract  $C$  yields a *finite-state* labeled transition system,<sup>2</sup> whose states are the contracts reachable from  $C$ .

We now present a simple example of a contract describing an authentication service that repeatedly performs two kinds of task: (i) the authentication of clients by receiving their username and password, and (ii) the request to an external account service for update of the list of the registered users.

$$\begin{array}{c}
\text{rec}X.(\text{username.password}.\overline{\text{accepted}}_{\text{client}}.X + \overline{\text{failed}}_{\text{client}}.X) \\
+ \overline{\text{updateAccounts}}_{\text{accountServer}}.\text{newAccounts}.X
\end{array}$$

The contract indicates a repeated choice between the two possible tasks. The first task is activated by the reception of an invocation on *username*. In this case, a *password* should subsequently be received and then two possible answers are sent back to the *client*: either *accepted* or *failed*. The second task is activated by sending a request for update to the *accountServer*. In this case, the *newAccounts* are subsequently received.

In the following we will study independent contract refinement. As already anticipated in the Introduction under *synchronous* communication a maximal independent contract refinement that preserves compliance does not exist. In [6] we showed that this is a consequence of the symmetry between input and output actions and that a possible solution, for synchronous communication, is to resort to *output persistent* contracts; thus breaking such a symmetry.

<sup>2</sup> As for basic CCS [35] finite-stateness is an obvious consequence of the fact that the process algebra does not include static operators, like parallel or restriction.

**Definition 7 (Output Persistence).** Consider the following notation:  $C \xrightarrow{\lambda}$  means  $\exists C' : C \xrightarrow{\lambda} C'$  and, given a (possibly empty) sequence of labels  $w = \lambda_1 \lambda_2 \cdots \lambda_{n-1} \lambda_n$ , we use  $C \xrightarrow{w} C'$  to denote the sequence of transitions  $C \xrightarrow{\lambda_1} C_1 \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_{n-1}} C_{n-1} \xrightarrow{\lambda_n} C'$ . A contract  $C$  is output persistent if, for any  $C'$  such that  $C \xrightarrow{w} C'$  and  $C' \xrightarrow{\bar{a}_l}$ , the following holds:  $C' \not\xrightarrow{\checkmark}$  and, if  $C' \xrightarrow{\alpha} C''$  with  $\alpha \neq \bar{a}_l$ , then also  $C'' \xrightarrow{\bar{a}_l}$ .

The output persistence property states that once a contract decides to execute an output, its actual execution is mandatory in order to successfully complete the execution of the contract. This property typically holds in languages for the description of service orchestrations (see e.g. WS-BPEL [40]) in which output actions cannot be used as guards in external choices (see e.g. the pick operator of WS-BPEL which is an external choice guarded on input actions).

The previous example of the authentication server (which is not output persistent) can be rephrased as follows to be used in a synchronous setting:

$$\begin{aligned} & \text{rec}X.(\text{username.password}.\overline{(\tau.\text{accepted}_{client}.X + \tau.\overline{\text{failed}_{client}.X})} \\ & \quad + \tau.\overline{\text{updateAccounts}_{accountServer}.newAccounts.X}) \end{aligned}$$

Notice that, in this new version of the example, we have simply added an internal action  $\tau$  in front of outputs occurring in choices. This guarantees that, at the moment the choice is to be resolved, the output action is not yet ready to be executed: it becomes available only after the  $\tau$  and, then, its eventual execution is mandatory.

In the remainder, when we consider synchronous communication, we will restrict to output persistent contracts.

### 3.1 Synchronous Contract Composition

Synchronous systems are formed by the parallel composition of contracts.

**Definition 8 (Synchronous Systems).** The syntax of synchronous systems is defined by the following grammar

$$P ::= [C]_l \quad | \quad P \parallel P$$

We assume systems to be such that: (i) every contract subterm  $[C]_l$  occurs in  $P$  at a different location  $l$  and (ii) no output action with destination  $l$  is syntactically included inside a contract subterm occurring in  $P$  at the same location  $l$ , i.e. actions  $\bar{a}_l$  cannot occur inside a subterm  $[C]_l$  of  $P$ .

A contract located at location  $l$  is denoted with  $[C]_l$ . Located contracts can be combined in parallel with the operator  $P \parallel P$ .

System operational semantics is defined by the rules in Table 4 plus symmetric rules and a rule lifting  $\tau$  transitions to located contracts like the one for  $\checkmark$ . Transition system labels, still ranged over by  $\lambda, \lambda', \dots$ , are now taken from the

set  $\{\bar{a}_{sr}, a_{sr}, \tau, \surd \mid a \in \mathcal{N}; s, r \in Loc\}$ , where:  $\bar{a}_{sr}$  ( $a_{sr}$ , resp.) denotes a potential output (input, resp.) with the sender being at location  $s$  and the receiver at location  $r$ ;  $\tau$  denotes a synchronization or a move performed internally by one contract in the system and  $\surd$  denotes successful termination.

**Table 4.** Synchronous system semantics ( $\tau$  lifting rule and symmetric rules omitted).

$$\begin{array}{c}
\frac{C \xrightarrow{\bar{a}_r} C'}{[C]_s \xrightarrow{\bar{a}_{sr}} [C']_s} \qquad \frac{C \xrightarrow{a} C'}{[C]_r \xrightarrow{a_{sr}} [C']_r} \qquad \frac{C \xrightarrow{\surd} C'}{[C]_l \xrightarrow{\surd} [C']_l} \\
\\
\frac{P \xrightarrow{\lambda} P'}{P \parallel Q \xrightarrow{\lambda} P' \parallel Q} \quad \lambda \neq \surd \qquad \frac{P \xrightarrow{\bar{a}_{sr}} P' \quad Q \xrightarrow{a_{sr}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \qquad \frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P \parallel Q \xrightarrow{\surd} P' \parallel Q'}
\end{array}$$

### 3.2 Asynchronous Contract Composition

In asynchronous systems contracts are equipped with an input message queue.

**Definition 9 (Asynchronous Systems).** *The syntax of asynchronous systems is defined by the following grammar*

$$\begin{array}{l}
P ::= [C, \mathcal{Q}]_l \mid P \parallel P \\
\mathcal{Q} ::= \epsilon \mid a^l :: \mathcal{Q}
\end{array}$$

We assume asynchronous systems to be such that: (i) and (ii) of Definition 8 (with  $[C, \mathcal{Q}]_l$  replacing  $[C]_l$ ) hold true.

Terms  $\mathcal{Q}$  denote message queues. They are sequences of messages, each one denoted with  $a^l$  where  $a$  is the action name and  $l$  is the location of the sender. We use “ $\epsilon$ ” to denote the empty message queue. Trailing  $\epsilon$  are usually left implicit, and we use “ $::$ ” also as an operator over the syntax: if  $\mathcal{Q}$  and  $\mathcal{Q}'$  are  $\epsilon$ -terminated queues, according to the syntax above, then  $\mathcal{Q} :: \mathcal{Q}'$  means appending the two queues into a single  $\epsilon$ -terminated list. Therefore, if  $\mathcal{Q}$  is a queue, then  $\epsilon :: \mathcal{Q}$ ,  $\mathcal{Q} :: \epsilon$ , and  $\mathcal{Q}$  are syntactically equal. In the following, when we talk about asynchronous contract systems, we will use the shorthand  $[C]_l$  to stand for  $[C, \epsilon]_l$ .

Asynchronous system operational semantics is defined by the rules in Table 5 plus the rules for the parallel operator of Table 4. In Table 5 we assume a rule lifting  $\tau$  transitions to located contracts like the one for output (without action subscripts) and that  $b^l \in \mathcal{Q}$  holds true if and only if  $b^l$  syntactically occurs inside  $\mathcal{Q}$ . This notation is used in the premise of the novel  $\tau$  synchronization rule that represents the consumption of an  $a$  message from the queue by removal of the oldest  $a$  one.

As an example consider the system:  $[\bar{a}_s.\bar{b}_s]_r \parallel [b.a]_s$ . After executing the two outputs, the system evolves to  $[1]_r \parallel [b.a, a^r :: b^r]_s$ . The receiver is now ready to

**Table 5.** Asynchronous system semantics ( $\tau$  lifting rule and rules for parallel omitted).

$$\begin{array}{c}
 \frac{C \xrightarrow{\bar{a}_r} C'}{[C, \mathcal{Q}]_s \xrightarrow{\bar{a}_{sr}} [C', \mathcal{Q}]_s} \quad [C, \mathcal{Q}]_r \xrightarrow{a_{sr}} [C, \mathcal{Q} :: a^s]_r \quad \frac{C \xrightarrow{\check{v}} C'}{[C, \epsilon]_l \xrightarrow{\check{v}} [C', \epsilon]_l} \\
 \\
 \frac{C \xrightarrow{a} C' \quad b^l \in \mathcal{Q} \Rightarrow b \neq a}{[C, \mathcal{Q} :: a^s :: \mathcal{Q}]_r \xrightarrow{\tau} [C', \mathcal{Q} :: \mathcal{Q}]_r}
 \end{array}$$

consume the two messages stored in the queue, thus reaching  $[1]_r \parallel [1]_s$ . This means that the two messages are consumed in the opposite order of reception.

Notice that the information about the sender attached to queue messages is actually not used by the operational semantic rules in Table 5: even if omitted we would have obtained the same transitions. Nevertheless, we decided to use the same queue syntax as in [10] to be more adherent to reality, where messages can be distinguished e.g. depending on the sender. As a matter of fact, in [10], this information is used to produce, instead of  $\tau$  actions, more informative labels that include denotation of the sender-receiver (this makes it possible to establish conformance w.r.t. a given choreographical specification).

### 3.3 Contract Refinement

We now recall the formal definition of independent contract refinement that preserves correct composition of contracts in both the synchronous and asynchronous cases. With  $P \xrightarrow{\tau}^* P'$  we denote the existence of a (possibly empty) sequence of  $\tau$ -labeled transitions starting from the system  $P$  and leading to  $P'$ .

**Definition 10 (Correct Contract Composition – Compliance).** *A system  $P$  is a correct contract composition, denoted  $P \downarrow$ , if for every  $P'$  such that  $P \xrightarrow{\tau}^* P'$ , there exists  $P''$  such that  $P' \xrightarrow{\tau}^* P''$  and  $P'' \xrightarrow{\check{v}}$ .*

Intuitively, a system composed of contracts is correct if any possible computation may guarantee completion, i.e. it can be extended to reach a successfully terminated computation (in the asynchronous case this means that all queues are empty). In this case, such contracts are called *compliant*. An example of contract composition that is correct (both in the synchronous and asynchronous case) is  $[\bar{a}_{l_3}]_{l_1} \parallel [\bar{b}_{l_3}]_{l_2} \parallel [a.b]_{l_3}$ . Another example is  $[\bar{a}_s.\bar{b}_s]_r \parallel [b.a]_s$  considered above, which is correct only in the asynchronous case.

We are now ready to define the notion of *contract refinement*. Given a contract  $C$ , we use  $oloc(C)$  to denote the set of locations used as destinations in all the output actions occurring inside  $C$ .

**Definition 11 (Independent Refinement).** *A pre-order  $\leq$  over contracts is an independent refinement if, for any  $n \geq 1$ , contracts  $C_1, \dots, C_n$  and*

$C'_1, \dots, C'_n$  such that  $\forall i. C'_i \leq C_i$ , and distinguished location names  $l_1, \dots, l_n \in \text{Loc}$  such that  $\forall i. \text{oloc}(C_i) \cup \text{oloc}(C'_i) \subseteq \{l_j \mid 1 \leq j \leq n \wedge j \neq i\}$ , we have:

$$([C_1]_{l_1} \parallel \dots \parallel [C_n]_{l_n}) \downarrow \Rightarrow ([C'_1]_{l_1} \parallel \dots \parallel [C'_n]_{l_n}) \downarrow$$

An independent refinement pre-order formalizes the possibility to replace in a correct contract composition every contract with one of its refinements, with the guarantee that the new system is still correct. In [6] it is shown that in the synchronous case, in the absence of the output persistence assumption, it could happen that given two independent refinement pre-orders, their union is no longer an independent refinement pre-order. In other words, there exists no maximal independent refinement pre-order.

On the contrary, if we restrict to output persistent contracts or we consider asynchronous communication, we have that the maximal independent refinement pre-order exists: it can be achieved by considering a coarser form of refinement in which, given any system composed of a set of contracts, refinement is applied to one contract only (thus leaving the others unchanged). This form of refinement, that we call *compliance testing* [11], is a form of testing where both the test and the system under test must reach success. Given a system  $P$ , we use  $\text{loc}(P)$  to denote the subset of  $\text{Loc}$  of the locations of contracts syntactically occurring inside  $P$ .

**Definition 12 (Refinement Relation).** *A contract  $C'$  is a refinement of a contract  $C$  denoted  $C' \preceq C$ , if and only if for all  $l \in \text{Loc}$  and system  $P$  such that  $l \notin \text{loc}(P)$  and  $l \notin \text{oloc}(C) \cup \text{oloc}(C') \subseteq \text{loc}(P)$ , we have:*

$$([C]_l \parallel P) \downarrow \Rightarrow ([C']_l \parallel P) \downarrow$$

**Theorem 3 (Maximal Independent Refinement).** *There exists a maximal independent refinement  $\leq$  pre-order and it corresponds to the (compliance testing based) refinement relation “ $\preceq$ ”.*

### 3.4 Properties of Contract Refinement

We now discuss some properties of contract refinement and also show a sound characterization that is decidable for the synchronous case. We use  $I(C)$  ( $O(C)$ , resp.) to stand for the set of names  $a$  (located names  $a_l$ , resp.) of input actions  $a$  (output actions  $\bar{a}_l$ , resp.) syntactically occurring in  $C$ . Given  $O \subseteq \{a_l \mid a \in \mathcal{N} \wedge l \in \text{Loc}\}$ , we assume  $\bar{O}$  to stand for  $\{\bar{a}_l \mid a_l \in O\}$ .

We first observe that the refinement relation  $\preceq$  allows input on new names (and unreachable outputs on new names) to be added in refined contracts.

**Theorem 4 (Refinements with Extended Inputs and Outputs).** *Let  $C, C'$  be contracts. Both of the following hold*

$$\begin{aligned} C' \{ \mathbf{0} / \alpha . C'' \mid \alpha \in I(C') - I(C) \} \preceq C &\Leftrightarrow C' \preceq C \\ C' \{ T / \alpha . C'' \mid \alpha \in O(C') - O(C) \} \preceq C &\Leftrightarrow C' \preceq C \end{aligned}$$

where  $T$  is:  $\mathbf{0}$  in the synchronous case,  $\tau.\mathbf{0}$  in the asynchronous case.

This theorem is a direct consequence of queue based communication (in the asynchronous case) and output persistence (in the synchronous case): a subcontract  $C'$  cannot perform reachable outputs that were not included in the potential outputs of the supercontract  $C$ ; and, similarly, a compliant test  $P$  of a contract  $[C]_l$  cannot perform reachable outputs directed to  $l$  that  $C$  cannot receive (e.g. in the asynchronous case  $[a]_l \parallel [\bar{a}_l + \bar{b}_l]_{l'}$  is not a correct contract composition). From this theorem we can derive two fundamental properties of the maximal independent refinement pre-order: external choices on inputs can be extended, e.g.  $a + b \preceq a$ ; while internal choices on outputs can be reduced, e.g.  $\bar{a}_l \preceq \bar{a}_l + \bar{b}_l$  in the asynchronous case and  $\tau.\bar{a}_l \preceq \tau.\bar{a}_l + \tau.\bar{b}_l$  in the synchronous one, because the lefthand term is more deterministic (typical property in testing).

We now focus on determining an algorithmic sound characterization of the synchronous contract refinement relation. This is achieved by resorting to the theory of fair testing, called *should-testing* [41]. As a side result we also have that the refinement relation  $\preceq$  is coarser than fair testing preorder. We denote with  $\preceq_{test}$  the *should-testing* pre-order defined in [41] where we consider  $\surd$  to be included in the set of actions of terms under testing as any other action ( $\surd$  is treated as a normal action and not as the special action representing success of tests in [41]). In order to resort to the theory should-testing, we define a normal form for contracts  $C$ , denoted with  $\mathcal{NF}(C)$ , that corresponds to terms of the language in [41] (mainly a matter of replacing  $\mathbf{1}$  with a  $\surd$  action, see [8] for details).

**Theorem 5 (Resorting to Fair Testing).** *Let  $C, C'$  be contracts. We have*

$$\mathcal{NF}(C' \{ \mathbf{0} / \alpha.C'' \mid \alpha \in I(C') - I(C) \}) \preceq_{test} \mathcal{NF}(C) \quad \Rightarrow \quad C' \preceq C$$

The opposite implication does not hold in general. This can be easily seen by considering *uncontrollable* contracts, i.e. contracts for which there is no compliant test. For instance the contract  $\mathbf{0}$ , any other contract  $a.b.\mathbf{0}$  or  $c.d.\mathbf{0}$  or more complex examples like  $a + a.b$ . These contracts are all equivalent according to our refinement relation, but of course not according to fair testing. Notice that such uncontrollable contracts have completely different traces: this means that trace pre-order is *not* coarser than our refinement relation.

## 4 Session Types

In this section we move to session types, in particular we report about our study of asynchronous session subtyping. Session types [28, 29] are types for controlling the communication behaviour of processes over channels. In a very simple but effective way, they express the pattern of sends and receives that a process must perform. They are, therefore, similar to behavioural contracts, but more constrained in the kind of behaviours they can express. Since they can guarantee freedom from some basic programming errors, session types are becoming popular with many main stream language implementations, e.g., Haskell [34], Go [39] or Rust [30]. In [38] session subtyping is introduced for asynchronous



communication and it is also stated that it is decidable. Recently it has been proven that, on the contrary, it is undecidable. Here we present such an undecidability result [4] and the decidability result in [5], where the largest known decidable fragment is introduced. In particular, we recall the basic definitions of session types and synchronous and asynchronous session subtyping. We then report the undecidability proof in [4]. Finally, we present the fragment of *single-out* (and *single-in*) session types, for which we show asynchronous subtyping to be decidable [5]. The techniques for these (un)decidability results can be seen as improvements of those developed for Linda process calculi: reduction from Turing complete computational models and exploitation of well quasi orderings.

#### 4.1 Session Subtyping

Session subtyping, which is the counterpart for session types of refinement for behavioural contracts, was first introduced by Gay and Hole [26] for a session-based  $\pi$ -calculus where communication is synchronous. Session subtyping of [26] is endowed with covariant/contravariant properties that correspond to those we observed on behavioural contract refinement: internal choices on outputs can be reduced, while external choices on inputs can be extended. To the best of our knowledge, Mostrous et al. [38] were the first to adapt the notion of session subtyping to an asynchronous setting. Their computation model is a session  $\pi$ -calculus with asynchronous communication that makes use of session queues for maintaining the order in which messages are sent. Based on such a model they introduce the idea of *output anticipation*, which is also a main feature of our theory in [4,5] that we present here. Mostrous and Yoshida [37] extended the notion of asynchronous subtyping to session types for the higher-order  $\pi$ -calculus. They also observed that their definition of asynchronous subtyping allows for *orphan messages*, i.e. sent messages which are never consumed from the session queue. Orphan messages are, instead, prohibited with the definition of subtyping given by Chen et al. [20]: they show that such a definition is both sound and complete w.r.t. type safety and orphan message freedom.

We start with the formal syntax of binary session types, adopting a simplified notation (used, e.g., in [4,5]) without dedicated constructs for sending an output/receiving an input. We instead represent outputs and inputs directly inside choices. More precisely, we consider output selection  $\oplus\{l_i : T_i\}_{i \in I}$ , expressing an internal choice among outputs, and input branching  $\&\{l_i : T_i\}_{i \in I}$ , expressing an external choice among inputs. Each possible choice is labeled by a label  $l_i$ , taken from a global set of labels  $L$ , followed by a session continuation  $T_i$ . Labels in a branching/selection are assumed to be pairwise distinct.

**Definition 13 (Session Types).** *Given a set of labels  $L$ , ranged over by  $l$ , the syntax of binary session types is given by the following grammar:*

$$T ::= \oplus\{l_i : T_i\}_{i \in I} \quad | \quad \&\{l_i : T_i\}_{i \in I} \quad | \quad \mu t.T \quad | \quad \mathbf{t} \quad | \quad \mathbf{end}$$

*A session type is single-out if, for all of its subterms  $\oplus\{l_i : T_i\}_{i \in I}$ ,  $|I| = 1$ ; it is single-in if, for all of its subterms  $\&\{l_i : T_i\}_{i \in I}$ ,  $|I| = 1$ .*

In the sequel, we leave implicit the index set  $i \in I$  in input branchings and output selections when it is already clear from the denotation of the types. Note also that we abstract from the type of the message that could be sent over the channel, since this is orthogonal to our theory. Types  $\mu\mathbf{t}.T$  and  $\mathbf{t}$  denote standard tail recursion for recursive types. We assume recursion to be guarded: in  $\mu\mathbf{t}.T$ , the recursion variable  $\mathbf{t}$  occurs within the scope of an output or an input type. In the following, we will consider closed terms only, i.e., types with all recursion variables  $\mathbf{t}$  occurring under the scope of a corresponding definition  $\mu\mathbf{t}.T$ . Type **end** denotes the type of a channel that can no longer be used.

For session types, we define the usual notion of duality: given a session type  $T$ , its dual  $\bar{T}$  is defined as:  $\oplus\{l_i : T_i\}_{i \in I} = \&\{l_i : \bar{T}_i\}_{i \in I}$ ,  $\&\{l_i : T_i\}_{i \in I} = \oplus\{l_i : \bar{T}_i\}_{i \in I}$ , **end** = **end**,  $\bar{\mathbf{t}} = \mathbf{t}$ , and  $\mu\mathbf{t}.T = \mu\mathbf{t}.\bar{\bar{T}}$ . In the sequel, we say that a relation  $\mathcal{R}$  on session types is *dual closed* if  $(S, T) \in \mathcal{R}$  implies  $(\bar{T}, \bar{S}) \in \mathcal{R}$ .

We start by considering a *synchronous* subtyping relation, similar to that of Gay and Hole [26] but, to be more consistent with contracts, following a process-oriented instead of a channel-based approach.<sup>3</sup> Moreover, following [38], we consider a generalized version of unfolding that allows us to unfold recursions  $\mu\mathbf{t}.T$  as many times as needed.

**Definition 14 ( $n$ -unfolding).**

$$\begin{aligned} \text{unfold}^0(T) &= T & \text{unfold}^1(\oplus\{l_i : T_i\}_{i \in I}) &= \oplus\{l_i : \text{unfold}^1(T_i)\}_{i \in I} \\ \text{unfold}^1(\mu\mathbf{t}.T) &= T\{\mu\mathbf{t}.T/\mathbf{t}\} & \text{unfold}^1(\&\{l_i : T_i\}_{i \in I}) &= \&\{l_i : \text{unfold}^1(T_i)\}_{i \in I} \\ \text{unfold}^1(\mathbf{end}) &= \mathbf{end} & \text{unfold}^n(T) &= \text{unfold}^1(\text{unfold}^{n-1}(T)) \end{aligned}$$

**Definition 15 (Synchronous Subtyping,  $\leq_s$ ).**  $\mathcal{R}$  is a synchronous subtyping relation whenever  $(T, S) \in \mathcal{R}$  implies that:

1. if  $T = \mathbf{end}$  then  $\exists n \geq 0$  such that  $\text{unfold}^n(S) = \mathbf{end}$ ;
2. if  $T = \oplus\{l_i : T_i\}_{i \in I}$  then  $\exists n \geq 0$  such that  $\text{unfold}^n(S) = \oplus\{l_j : S_j\}_{j \in J}$ ,  $I \subseteq J$  and  $\forall i \in I. (T_i, S_i) \in \mathcal{R}$ ;
3. if  $T = \&\{l_i : T_i\}_{i \in I}$  then  $\exists n \geq 0$  such that  $\text{unfold}^n(S) = \&\{l_j : S_j\}_{j \in J}$ ,  $J \subseteq I$  and  $\forall j \in J. (T_j, S_j) \in \mathcal{R}$ ;
4. if  $T = \mu\mathbf{t}.T'$  then  $(T'\{T/\mathbf{t}\}, S) \in \mathcal{R}$ .

$T$  is a synchronous subtype of  $S$ , written  $T \leq_s S$ , if there is a synchronous subtyping relation  $\mathcal{R}$  such that  $(T, S) \in \mathcal{R}$ .

Two types  $T$  and  $S$  are related by  $\leq_s$ , whenever  $S$  is able to simulate  $T$  with output and input types enjoying covariance and contravariance properties, respectively. Notice the asymmetric use of unfolding between the left- and righthand terms  $T$  and  $S$ : in  $T$  recursion is always unfolded once, while in  $S$  many unfoldings can be needed in order to expose the starting operator of  $T$ .

As already discussed, subtyping is the counterpart of contract refinement in the context of session types. Consider, for instance,

$$\&\{a : \mathbf{end}, b : \mathbf{end}\} \leq_s \&\{a : \mathbf{end}\} \quad \oplus\{a : \mathbf{end}\} \leq_s \oplus\{a : \mathbf{end}, b : \mathbf{end}\}$$

<sup>3</sup> Differently from our definitions, in the channel-based approach of Gay and Hole [26] subtyping is covariant on branchings and contra-variant on selections.

that hold for input contravariance and output covariance. These examples of subtypings precisely correspond to those of contract refinements commented in Sect. 3.4. Note that, while in the case of contracts they were obtained as a consequence of considering the maximal independent refinement, in the theory of session types they are taken by definition.

We now consider the standard notion of *asynchronous* subtyping  $\leq$  introduced by Chen et al. [20], which enjoys orphan message freedom; we consider the simple rephrasing based on dual closeness we introduced in [5]. In the definition of  $\leq$  we use the following notion of input context.

**Definition 16 (Input Context).** *An input context  $\mathcal{A}$  is a session type with multiple holes defined by the syntax:  $\mathcal{A} ::= []^n \mid \&\{l_i : \mathcal{A}_i\}_{i \in I}$ .*

*The holes  $[]^n$ , with  $n \in \mathbb{N}^+$ , of an input context  $\mathcal{A}$  are assumed to be consistently enumerated, i.e. there exists  $m \geq 1$  such that  $\mathcal{A}$  includes one and only one  $[]^n$  for each  $n \leq m$ . Given types  $T_1, \dots, T_m$ , we use  $\mathcal{A}[T_k]^{k \in \{1, \dots, m\}}$  to denote the type obtained by filling each hole  $k$  in  $\mathcal{A}$  with the corresponding term  $T_k$ .*

**Definition 17 (Asynchronous Subtyping,  $\leq$ ).**  *$\mathcal{R}$  is an asynchronous subtyping relation whenever it is dual closed and  $(T, S) \in \mathcal{R}$  implies 1., 3., and 4. of Definition 15, plus the following modified version of 2.:*

2. if  $T = \oplus\{l_i : T_i\}_{i \in I}$  then  $\exists n \geq 0, \mathcal{A}$  such that
  - $\mathbf{unfold}^n(S) = \mathcal{A}[\oplus\{l_j : S_{k_j}\}_{j \in J_k}]^{k \in \{1, \dots, m\}}$ ,
  - $\forall k \in \{1, \dots, m\}. I \subseteq J_k$  and
  - $\forall i \in I, (T_i, \mathcal{A}[S_{k_i}]^{k \in \{1, \dots, m\}}) \in \mathcal{R}$

*$T$  is an asynchronous subtype of  $S$ , written  $T \leq S$ , if there is an asynchronous subtyping relation  $\mathcal{R}$  such that  $(T, S) \in \mathcal{R}$ .*

We now explain the modified version of Rule 2. and its impact on the obtained subtyping relation. Concerning the adopted notation, for each hole  $k$  of the input context  $\mathcal{A}$  (which is at the beginning of the righthand term  $S$  after any needed unfolding), we take  $l_j$ , with  $j \in J_k$ , to be the labels of the output selection in the hole. Moreover, we use  $S_{k_j}$  to denote the type reached after output  $l_j$  in the hole  $k$ . An important characteristic of asynchronous subtyping (formalized by Rule 2. above) is the following one. In a subtype output selections can be anticipated so to bring them before the input branchings that in the supertype occur in front of them. For example

$$\oplus\{l : \&\{l_1 : T_1, l_2 : T_2\}\} \leq \&\{l_1 : \oplus\{l : T_1\}, l_2 : \oplus\{l : T_2\}\}$$

where the output selection with label  $l$  (occurring in the supertype) is anticipated w.r.t. the input branching with labels  $l_1$  and  $l_2$  (such an output selection is present in *all* its input branches). As already discussed in the Introduction, output anticipation reflects the fact that we are considering asynchronous communication protocols in which messages are stored in queues. In this setting, it is safe to replace a peer that follows a given protocol with another one following a

modified protocol where outputs are anticipated: in fact, the difference is simply that such outputs will be stored earlier in the communication queue.

As a further example, consider the types  $T = \mu \mathbf{t}.\&\{l : \oplus\{l : \mathbf{t}\}\}$  and  $S = \mu \mathbf{t}.\&\{l : \&\{l : \oplus\{l : \mathbf{t}\}\}\}$ . We have  $T \leq S$  by considering an infinite subtyping relation including pairs  $(T', S')$ , with  $S'$  being  $\&\{l : S\}$ ,  $\&\{l : \&\{l : S\}\}$ ,  $\&\{l : \&\{l : \&\{l : S\}\}\}$ ,  $\dots$ ; that is, the effect of each output anticipation is that a new input  $\&\{l : \_ \}$  is accumulated in the initial part of the r.h.s. It is worth to observe that every accumulated input  $\&\{l : \_ \}$  is eventually consumed in the simulation game (orphan message freedom), but the accumulated inputs grows unboundedly.

## 4.2 Undecidability of Asynchronous Subtyping

The proof of undecidability of the asynchronous subtyping relation, taken from [4], is by reduction from the acceptance problem for queue machines.

**Definition 18 (Queue machine).** *A queue machine  $M$  is defined by a six-tuple  $(Q, \Sigma, \Gamma, \$, s, \delta)$  where:  $Q$  is a finite set of states;  $\Sigma \subset \Gamma$  is a finite set denoting the input alphabet;  $\Gamma$  is a finite set denoting the queue alphabet (ranged over by  $A, B, C, X$ );  $\$ \in \Gamma - \Sigma$  is the initial queue symbol;  $s \in Q$  is the start state;  $\delta : Q \times \Gamma \rightarrow Q \times \Gamma^*$  is the transition function.*

We now formally define queue machine computations.

**Definition 19 (Queue machine computation).** *A configuration of a queue machine is an ordered pair  $(q, \gamma)$  where  $q \in Q$  is its current state and  $\gamma \in \Gamma^*$  is the queue ( $\Gamma^*$  is the Kleene closure of  $\Gamma$ ). The starting configuration on an input string  $x$  is  $(s, x\$)$ . The transition relation  $\rightarrow_M$  over configurations  $Q \times \Gamma^*$ , leading from a configuration to the next one, is defined as follows. For any  $p, q \in Q$ ,  $A \in \Gamma$  and  $\alpha, \gamma \in \Gamma^*$  we have  $(p, A\alpha) \rightarrow_M (q, \alpha\gamma)$  whenever  $\delta(p, A) = (q, \gamma)$ . A machine  $M$  accepts an input  $x$  if it eventually terminates on input  $x$ , i.e. it reaches a blocking configuration with the empty queue (notice that, as the transition relation is total, the unique way to terminate is by emptying the queue). Formally,  $x$  is accepted by  $M$  if  $(s, x\$) \rightarrow_M^* (q, \epsilon)$  where  $\epsilon$  is the empty string and  $\rightarrow_M^*$  is the reflexive and transitive closure of  $\rightarrow_M$ .*

Queue machines are Turing complete, see [31] (p. 354) and [4].

Our goal is to construct a pair of types, say  $T$  and  $S$ , from a given queue machine  $M$  and a given input  $x$ , such that:  $T \leq S$  if and only if  $x$  is not accepted by  $M$ . Intuitively, type  $T$  encodes the finite control of  $M$ , i.e., its transition function  $\delta$ , starting from its initial state  $s$ . And type  $S$  encodes the machine queue that initially contains  $x\$$ , where  $x$  is the input string  $x = X_1 \cdots X_n$  of length  $n \geq 0$ . The set of labels  $L$  for such types  $T$  and  $S$  is  $M$ 's queue alphabet  $\Gamma$ .

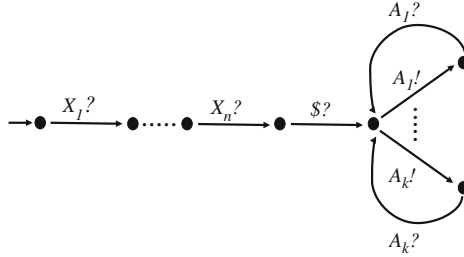
Formally, the queue of a machine is encoded into a session type as follows:

**Definition 20 (Queue Encoding).** *Let  $M = (Q, \Sigma, \Gamma, \$, s, \delta)$  be a queue machine and let  $C_1 \cdots C_m \in \Gamma^*$ , with  $m \geq 0$ . We define:*

$$\llbracket C_1 \cdots C_m \rrbracket = \&\{C_1 : \dots \&\{C_m : \mu\mathbf{t} \oplus \{A : \&\{A : \mathbf{t}\}\}_{A \in \Gamma}\}\}$$

Given a configuration  $(q, \gamma)$  of  $M$ , the encoding of the queue  $\gamma = C_1 \cdots C_m$  is thus defined as  $\llbracket C_1 \cdots C_m \rrbracket$ .

Note that whenever  $m = 0$ , we have  $\llbracket \epsilon \rrbracket = \mu\mathbf{t} \oplus \{A : \&\{A : \mathbf{t}\}\}_{A \in \Gamma}$ . Observe that we are using a slight abuse of notation: in both output selections and input branchings, labels  $l_A$ , with  $A \in \Gamma$ , are simply denoted by  $A$ .



**Fig. 1.** Session type encoding the initial queue  $X_1 \cdots X_n \$$

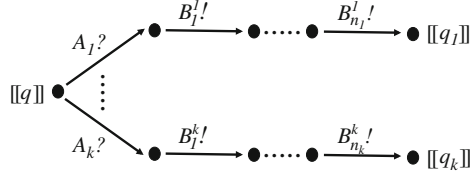
Figure 1 contains a graphical representation of the queue encoding with its initial content  $X_1 \cdots X_n \$$ . In order to better clarify our development, we graphically represent session types as labeled transition systems (in the form of communicating automata [3]), where an output selection  $\oplus\{l_i : T_i\}_{i \in I}$  is represented as a choice among alternative output transitions labeled with “ $l_i!$ ”, and an input branching  $\&\{l_i : T_i\}_{i \in I}$  is represented as a choice among alternative input transitions labeled with “ $l_i?$ ”. Intuitively, we encode a queue containing symbols  $C_1 \cdots C_m$  with a session type that starts with  $m$  inputs with labels  $C_1, \dots, C_m$ , respectively. Thus, in Fig. 1, we have  $C_1 \cdots C_m = X_1 \cdots X_n \$$ . After such sequence of inputs, representing the current queue content, there is a recursive type representing the capability to enqueue new symbols. Such a type repeatedly performs an output selection with one choice for each symbol  $A_i$  in the queue alphabet  $\Gamma$  (with  $k$  being the cardinality of  $\Gamma$ ), followed by an input labeled with the same symbol  $A_i$ .

We now give the definition of the type modelling the finite control of a queue machine, i.e., the encoding of the transition function  $\delta$ .

**Definition 21 (Finite Control Encoding).** Let  $M = (Q, \Sigma, \Gamma, \$, s, \delta)$  be a queue machine and let  $q \in Q$  and  $\mathcal{S} \subseteq Q$ . We define:

$$\llbracket q \rrbracket^{\mathcal{S}} = \begin{cases} \mu\mathbf{q} \&\{A : \oplus\{B_1^A : \dots \oplus \{B_{n_A}^A : \llbracket q' \rrbracket^{\mathcal{S} \cup \{q\}}\}\}\}_{A \in \Gamma} & \text{if } q \notin \mathcal{S} \text{ and } \delta(q, A) = (q', B_1^A \cdots B_{n_A}^A) \\ \mathbf{q} & \text{if } q \in \mathcal{S} \end{cases}$$

The encoding of the transition function of  $M$  is then defined as  $\llbracket s \rrbracket^{\emptyset}$ .



(for  $\Gamma = \{A_i \mid i \leq k\}$  and  $\delta(q, A_i) = (q_i, B_1^i \cdots B_{n_i}^i)$  for every  $i$ )

**Fig. 2.** Session type encoding a finite control.

The encoding of the finite control is a recursively defined term with one recursion variable  $\mathbf{q}$  for each state  $q \in Q$  of the machine. Above,  $\llbracket q \rrbracket^{\mathcal{S}}$  is a function that, given a state  $q$  and a set of states  $\mathcal{S}$ , returns a type representing the possible behaviour of the queue machine starting from state  $q$ . Such behaviour consists of first reading from the queue (input branching on  $A \in \Gamma$ ) and then writing on the queue a sequence of symbols  $B_1^A, \dots, B_{n_A}^A$ . The parameter  $\mathcal{S}$  is necessary for managing the recursive definition of this type. In fact, as the definition of the encoding function is itself recursive, this parameter keeps track of the states that have been already encoded (see example below). In Fig. 2, we report a graphical representation of the Labelled Transition System corresponding to the session type that encodes the queue machine finite control, i.e. the transition function  $\delta$ . Each state  $q \in Q$  is mapped onto a state  $\llbracket q \rrbracket$  of a session type, which performs an input branching with a choice for each symbol in the queue alphabet  $\Gamma$  (with  $k$  being the cardinality of  $\Gamma$ ). Each of these choices represents a possible symbol that can be read from the queue. After this initial input branching, each choice continues with a sequence of outputs labeled with the symbols that are to be inserted in the queue (after the symbol labeling that choice has been consumed). This is done according to function  $\delta$ , assuming that  $\delta(q, A_i) = (q_i, B_1^i \cdots B_{n_i}^i)$ , with  $n_i \geq 0$ , for all  $i$  in  $\{1, \dots, k\}$ . After the insertion phase, state  $\llbracket q_i \rrbracket$  of the session type corresponding to state  $q_i$  of the queue machine is reached.

Notice that, queue insertion actually happens in the encoding because, when the encoding of the finite control performs an output of a  $B$  symbol, the encoding of the queue must mimic such an output, possibly by anticipating it. This has the effect of adding an input on  $B$  at the end of the sequence of initial inputs of the queue machine encoding.

**Theorem 6.** *Given a queue machine  $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ , an input string  $x$ , and the two types  $T = \llbracket s \rrbracket^{\emptyset}$  and  $S = \llbracket x \$ \rrbracket$ , we have that  $M$  accepts  $x$  iff  $T \not\leq S$ .*

### 4.3 Decidability of Single-Out/Single-In Asynchronous Subtyping

We now show decidability of asynchronous session subtyping over single-out/single-in session types. The full technical machinery can be found in [5].

We start by giving a procedure (an algorithm that does not necessarily terminate) for the general subtyping relation, which we showed to be undecidable. Such a procedure is inspired by the one proposed by Mostrous et al. [38]

for asynchronous subtyping in multiparty session types. In order to do so, we introduce two functions on the syntax of types. The function `outDepth` calculates how many unfoldings are necessary for bringing an output outside a recursion (in every possible input path). If that is not possible, the function is undefined (denoted by  $\perp$ ). As an example consider, for any  $T_1$  and  $T_2$ ,  $\text{outDepth}(\oplus\{l_1 : T_1, l_2 : T_2\}) = 0$ . On the other hand, consider the type  $T_{ex} = \&\{l_1 : \mu t. \oplus\{l_2 : T_1\}, l_3 : \mu t. \&\{l_4 : \mu t'. \oplus\{l_5 : T_2\}\}\}$ : we have,  $\text{outDepth}(T_{ex}) = 2$ . We then define `outUnf()`, a variant of the unfolding function given in Definition 14, which unfolds only where it is necessary, in order to reach an output. The function above differs from `unfoldn`: for example,  $\text{unfold}^2(T_{ex})$  would unfold twice both subterms  $\mu t. \oplus\{l_2 : T_1\}$  and  $\mu t. \&\{l_4 : \mu t'. \oplus\{l_5 : T_2\}\}$ . On the other hand, applying `outDepth` to the same term would unfold once the term reached with  $l_1$  and twice the one reached with  $l_3$ . In the subtyping procedure defined below we make use of `outUnf()` in order to have that recursive definitions under the scope of an output are never unfolded. This guarantees that during the execution of the procedure, even if the set of reached terms could be unbounded, all the subterms starting with an output are taken from a bounded set of terms. This is important to guarantee termination of the algorithm that we are going to define as an extension of the procedure described below.

*Subtyping Procedure.* An environment  $\Sigma$  is a set containing pairs  $(T, S)$ , where  $T$  and  $S$  are types. Judgements are triples of the form  $\Sigma \vdash T \leq_a S$  which intuitively read as “in order to succeed, the procedure must check whether  $T$  is a subtype of  $S$ , provided that pairs in  $\Sigma$  have already been visited”. Our *subtyping procedure*, applied to the types  $T$  and  $S$ , consists of deriving the state space of our judgments using the rules in Fig. 3 bottom-up starting from the initial judgement  $\emptyset \vdash T \leq_a S$ . More precisely, we use the transition relation  $\Sigma \vdash T \leq_a S \rightarrow \Sigma' \vdash T' \leq_a S'$  to indicate that if  $\Sigma \vdash T \leq_a S$  matches the conclusions of one of the rules in Fig. 3, then  $\Sigma' \vdash T' \leq_a S'$  is produced by the corresponding premises. The procedure explores the reachable judgements according to this transition relation. We give highest priority to rule `Asmp`, thus ensuring that at most one rule is applicable.<sup>4</sup> The idea behind  $\Sigma$  is to avoid cycles when dealing with recursive types. Rules `RecR1` and `RecR2` deal with the case in which the type on the right-hand side is a recursion and must be unfolded. If the type on the left-hand side is not an output then the procedure simply adds the current pair to  $\Sigma$  and continues. On the other hand, if an output must be found, we apply `RecR1` which checks whether such output is available. Rule `Out` allows nested outputs to be anticipated (when not under recursion) and condition  $(\mathcal{A} \neq [\ ]^1) \Rightarrow \forall i \in I. \& \in T_i$  (inspired by [20]) makes sure there are no orphan messages. In fact, this condition implies that if there is some output which is anticipated in the subtype w.r.t. some inputs, in every continuation of the subtype there are input actions that will eventually reproduce also the input behaviour of the supertype.

<sup>4</sup> The priority of `Asmp` is sufficient because all the other rules are alternative, i.e., given a judgement  $\Sigma \vdash T \leq_a S$  there are no two rules different from `Asmp` that can be both applied.

$$\begin{array}{c}
(\mathcal{A} \neq []^1) \Rightarrow \forall i \in I. \& \in T_i \\
\forall n. I \subseteq J_n \quad \forall i \in I. \Sigma \vdash T_i \leq_a \mathcal{A}[S_{ni}]^n \\
\hline
\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_a \mathcal{A}[\oplus\{l_j : S_{nj}\}_{j \in J_n}]^n \quad \text{Out} \\
\\
\frac{J \subseteq I \quad \forall j \in J. \Sigma \vdash T_j \leq_a S_j}{\Sigma \vdash \&\{l_i : T_i\}_{i \in I} \leq_a \&\{l_j : S_j\}_{j \in J}} \text{In} \qquad \frac{}{\Sigma \vdash \mathbf{end} \leq_a \mathbf{end}} \text{End} \\
\\
\frac{}{\Sigma, (T, S) \vdash T \leq_a S} \text{Asmp} \qquad \frac{\Sigma, (\mu\mathbf{t}.T, S) \vdash T\{\mu\mathbf{t}.T/\mathbf{t}\} \leq_a S}{\Sigma \vdash \mu\mathbf{t}.T \leq_a S} \text{RecL} \\
\\
\frac{T = \mathbf{end} \vee T = \&\{l_i : T_i\}_{i \in I} \quad \Sigma, (T, \mu\mathbf{t}.S) \vdash T \leq_a S\{\mu\mathbf{t}.S/\mathbf{t}\}}{\Sigma \vdash T \leq_a \mu\mathbf{t}.S} \text{RecR}_1 \\
\\
\frac{\text{outDepth}(S) \geq 1 \quad \Sigma, (\oplus\{l_i : T_i\}_{i \in I}, S) \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_a \text{outUnf}(S)}{\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_a S} \text{RecR}_2
\end{array}$$

**Fig. 3.** A Procedure for Checking Subtyping

The remaining rules are self-explanatory.  $\Sigma \vdash T \leq_a S \rightarrow^* \Sigma' \vdash T' \leq_a S'$  is the reflexive and transitive closure of the transition relation among judgements. We write  $\Sigma \vdash T \leq_a S \rightarrow_{\text{ok}}$  if the judgement  $\Sigma \vdash T \leq_a S$  matches the conclusion of one of the axioms **Asmp** or **End**, and  $\Sigma \vdash T \leq_a S \rightarrow_{\text{err}}$  to mean that no rule can be applied to  $\Sigma \vdash T \leq_a S$ . Due to input branching and output selection, the rules **In** and **Out** could generate branching also in the state space to be explored by the procedure. Namely, given a judgement  $\Sigma \vdash T \leq_a S$ , there are several subsequent judgements  $\Sigma' \vdash T' \leq_a S'$  such that  $\Sigma \vdash T \leq_a S \rightarrow \Sigma' \vdash T' \leq_a S'$ . The procedure could (i) successfully terminate because all the explored branches reach a successful judgement  $\Sigma' \vdash T' \leq_a S' \rightarrow_{\text{ok}}$ , (ii) terminate with an error in case at least one judgement  $\Sigma' \vdash T' \leq_a S' \rightarrow_{\text{err}}$  is reached, or (iii) diverge because no branch terminates with an error and at least one branch never reaches a successful judgement. As we prove in [5] the procedure is sound with respect to asynchronous subtyping  $\leq$  and it can diverge only if the checked types are in the  $\leq$  relation.

If we consider types  $T$  and  $S$  of the example considered after Definition 17 the subtyping procedure in Fig. 3 applied to  $\emptyset \vdash T \leq_a S$  does not terminate. The problem is that the termination rule **Asmp** cannot be applied because the term on the r.h.s. (i.e. the supertype) generates always new terms in the form  $\&\{l : \&\{l : \dots \&\{l : S\} \dots\}\}$ . Notice that, in this particular example, these infinitely many distinct terms are obtained by adding single inputs (i.e. single-choice input branchings) in front of the term in the r.h.s.: we call this *linear input accumulation*. In general, however, input accumulation takes the form of a tree (thus accounting for all possible alternative accumulated input behaviors at the same time).

We now show how to decide asynchronous subtyping over single-out types, i.e. when input accumulation can indeed be in the general form of a tree, but, due



to the absence of output selections with multiple choices, it gets accumulated in a deterministic (i.e. unique) way. This will also allow us to deal with single-in types by exploiting duality. As anticipated, it deals with general input accumulation by representing it as a tree. We need to be able to extract the leaves from these trees: this is done by the *leaf set* function. The leaf set of a session type  $T$  is the set of subterms reachable from its root through a path of inputs. For example, the leaf set of the term  $\&\{l_1 : \mu\mathbf{t}. \oplus \{l_2 : \mathbf{t}\}, l_3 : \&\{l_4 : \oplus \{l_2 : \mu\mathbf{t}. \oplus \{l_2 : \mathbf{t}\}\}\}\}$  is  $\{\mu\mathbf{t}. \oplus \{l_2 : \mathbf{t}\}, \oplus \{l_2 : \mu\mathbf{t}. \oplus \{l_2 : \mathbf{t}\}\}\}$ .

During the check of subtyping, according to Fig. 3 (rule Out), when a term in the r.h.s. having input accumulation has to mimic an output in front of the l.h.s., such output must be present in front of all the leaves of the tree. In this case, the checking continues by anticipating the output from all the leaves. We make use of an auxiliary *output anticipation* function, called **antOut**, that indicates the way a term changes after having anticipated a sequence of outputs.  $\mathbf{antOut}(T, \tilde{l})$  yields the term obtained from  $T$  by anticipating all outputs occurring in the sequence  $\tilde{l}$ . For example, the function applied to the type  $T = \mu\mathbf{t}. \oplus \{l_1 : \&\{l : \oplus \{l_2 : \mathbf{t}\}, l' : \oplus \{l_2 : \mathbf{t}\}\}\}$  and the sequence  $(l_1, l_2)$  returns  $\&\{l : T, l' : T\}$ , while it is undefined with the sequence  $(l_1, l_1)$ . Moreover, we say that  $T$  can infinitely anticipate outputs, written  $\mathbf{antOutInf}(T)$ , if there exists an infinite sequence of labels  $l_{i_1} \cdots l_{i_j} \cdots$  such that  $\mathbf{antOut}(T, l_{i_1} \cdots l_{i_n})$  is defined for every  $n$ . The definition of  $\mathbf{antOutInf}(T)$  is not algorithmic in that it quantifies on every possible natural number  $n$ . Nevertheless, it can be decided by checking whether, for every session type obtained from  $T$  by means of output anticipations, all the terms populating its leaf set can anticipate the same output label. Although the types that can be obtained from  $T$  by means of output anticipations may be infinite, the terms populating the leaf sets are finite and are over-approximated by the function  $\mathbf{reach}(T)$  which is defined as the minimal set of (single-out) session types such that:

1.  $T \in \mathbf{reach}(T)$ ;
2.  $\&\{l_i : T_i\}_{i \in I} \in \mathbf{reach}(T)$  implies  $T_i \in \mathbf{reach}(T)$  for every  $i \in I$ ;
3.  $\mu\mathbf{t}.T' \in \mathbf{reach}(T)$  implies  $T'\{\mu\mathbf{t}.T'/\mathbf{t}\} \in \mathbf{reach}(T)$ ;
4.  $\oplus\{l : T'\} \in \mathbf{reach}(T)$  implies  $T' \in \mathbf{reach}(T)$ .

Notice that  $\mathbf{reach}(T)$  contains the session types obtained by consuming initial inputs and outputs, and by unfolding recursion when it is at the top level.

**Proposition 3.** *Given a single-out session type  $T$ ,  $\mathbf{reach}(T)$  is finite and it is decidable whether  $\mathbf{antOutInf}(T)$ .*

*Subtyping Algorithm for Single-Out Types.* We are now ready to present an additional termination condition that, once included into the subtyping procedure in Fig. 3, makes it a valid algorithm for checking subtyping for single-out types. The termination condition is defined as an additional rule, named **Asmp2**, that complements the already defined **Asmp** rule by detecting those cases in which the subtyping procedure in Fig. 3 does not terminate (**Asmp2**, presented below,

is assumed to have the same priority as rule **Asmp**: both rules have highest priority). The new rule is defined parametrically on the session type  $Z$ , which is the type on the right-hand side of the initial pair of types to be checked (i.e. the algorithm is intended to check  $V \leq_t Z$ , for some type  $Z$ ). We start from the initial judgement  $\emptyset \vdash V \leq_a Z$  and then apply from bottom to top the rules in Fig. 3, where  $\leq_a$  is replaced by  $\leq_t$ , plus the following additional rule:

$$\frac{S \in \text{reach}(Z) \quad \text{antOutInf}(S) \quad |\gamma| < |\beta| \quad \text{leafSet}(\text{antOut}(S, \gamma)) = \text{leafSet}(\text{antOut}(S, \beta))}{\Sigma, (T, \text{antOut}(S, \gamma)) \vdash T \leq_t \text{antOut}(S, \beta)} \text{Asmp2}$$

Intuitively, we have that this additional termination rule guarantees to catch all those cases where the term on the right grows indefinitely, by anticipating outputs and accumulating inputs. These infinitely many distinct types are anyway obtainable starting from the finite set  $\text{reach}(Z)$ , by means of output anticipations. Hence there exists  $S \in \text{reach}(Z)$  that can generate infinitely many of these types: this guarantees  $\text{antOutInf}(S)$  to be true. As observed above, the leaves of such infinitely many terms are themselves taken from the finite set  $\text{reach}(Z)$ . In Sect. 2 we have commented that multiset inclusion is a wqo over multisets defined on a finite domain; here we use a similar result according to which set equality is a wqo over the subsets of a finite given set. In fact, the possible subsets in this case are finite thus in an infinite sequence of such subsets at least one is repeated. The termination of our algorithm follows this wqo: **Asmp2**, besides checking conditions that are guaranteed to hold if the procedure  $\leq_a$  continues indefinitely, checks also for the equality between the set of leaves of the r.h.s. term in the current judgement and in a previously checked one.

**Theorem 7 (Decidability for Single-out Types).** *Asynchronous subtyping  $\leq$  over single-out session types is decidable.*

Exploiting dual closeness of  $\leq$  we can use the algorithm presented for single-out types also for single-in types (it is sufficient to check subtyping on the duals, observing that the dual of a single-in type is single-out).

**Corollary 1 (Decidability for Single-in Types).** *Asynchronous subtyping  $\leq$  over single-in session types is decidable.*

## 5 Conclusion

In this survey paper we have recalled the main results related with two lines of research on the expressiveness of Linda-like coordination models and on the theory of behavioural contracts. In the third part of the paper, we have discussed how the techniques developed for Linda-like coordination have recently contributed to the advancement of the research in the context of session types

for asynchronous communication. Session types, which can be seen as a simplification of contracts, already had a wide application on concurrent programming languages, as, e.g., Haskell [34], Go [39] and Rust [30].

We expect two possible lines for future work: on the one hand, analyse the impact on the theory of contracts of our results for session types (in fact, very few results are present in the literature about contracts for asynchronous communication) and, on the other hand, continue in the context of session types by investigating novel techniques for sound algorithmic characterizations of asynchronous session subtyping.

## References

1. Barbanera, F., de'Liguoro, U.: Two notions of sub-behaviour for session-based client/server systems. In: Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2010, pp. 155–164. ACM (2010)
2. Boreale, M., Bravetti, M.: Advanced mechanisms for service composition, query and discovery. In: Wirsing, M., Hölzl, M. (eds.) Rigorous Software Engineering for Service-Oriented Systems. LNCS, vol. 6582, pp. 282–301. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20401-2\\_13](https://doi.org/10.1007/978-3-642-20401-2_13)
3. Brand, D., Zafropulo, P.: On communicating finite-state machines. *J. ACM* **30**(2), 323–342 (1983)
4. Bravetti, M., Carbone, M., Zavattaro, G.: Undecidability of asynchronous session subtyping. *Inf. Comput.* **256**, 300–320 (2017)
5. Bravetti, M., Carbone, M., Zavattaro, G.: On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.* **722**, 19–51 (2018)
6. Bravetti, M., Zavattaro, G.: Contract based multi-party service composition. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 207–222. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-75698-9\\_14](https://doi.org/10.1007/978-3-540-75698-9_14)
7. Bravetti, M., Zavattaro, G.: A theory for strong service compliance. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 96–112. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-72794-1\\_6](https://doi.org/10.1007/978-3-540-72794-1_6)
8. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-77351-1\\_4](https://doi.org/10.1007/978-3-540-77351-1_4)
9. Bravetti, M., Zavattaro, G.: A foundational theory of contracts for multi-party service composition. *Fundamenta Informaticae* **89**(4), 451–478 (2008)
10. Bravetti, M., Zavattaro, G.: Contract compliance and choreography conformance in the presence of message queues. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 37–54. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-01364-5\\_3](https://doi.org/10.1007/978-3-642-01364-5_3)
11. Bravetti, M., Zavattaro, G.: Choreographies and behavioural contracts on the way to dynamic updates. In: Proceedings of the 1st Workshop on Logics and Model-checking for Self-\* Systems, MOD\* 2014. EPTCS, vol. 168, pp. 12–31 (2014)
12. Brogi, A., Jacquet, J.-M.: Modeling coordination via asynchronous communication. In: Garlan, D., Le Métayer, D. (eds.) COORDINATION 1997. LNCS, vol. 1282, pp. 238–255. Springer, Heidelberg (1997). [https://doi.org/10.1007/3-540-63383-9\\_84](https://doi.org/10.1007/3-540-63383-9_84)

13. Busi, N., Gorrieri, R., Zavattaro, G.: On the Turing equivalence of Linda coordination primitives. In: Proceedings of 4th Workshop on Expressiveness in Concurrency, EXPRESS 1997. Elsevier (1997). *Electr. Notes Theor. Comput. Sci.* **7**
14. Busi, N., Gorrieri, R., Zavattaro, G.: Three semantics of the output operation for generative communication. In: Garlan, D., Le Métayer, D. (eds.) COORDINATION 1997. LNCS, vol. 1282, pp. 205–219. Springer, Heidelberg (1997). [https://doi.org/10.1007/3-540-63383-9\\_82](https://doi.org/10.1007/3-540-63383-9_82)
15. Busi, N., Gorrieri, R., Zavattaro, G.: Temporary data in shared dataspace coordination languages. In: Honsell, F., Miculan, M. (eds.) FoSSaCS 2001. LNCS, vol. 2030, pp. 121–136. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-45315-6\\_8](https://doi.org/10.1007/3-540-45315-6_8)
16. Busi, N., Zavattaro, G.: On the expressiveness of event notification in data-driven coordination languages. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 41–55. Springer, Heidelberg (2000). [https://doi.org/10.1007/3-540-46425-5\\_3](https://doi.org/10.1007/3-540-46425-5_3)
17. Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A formal account of contracts for web services. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 148–162. Springer, Heidelberg (2006). [https://doi.org/10.1007/11841197\\_10](https://doi.org/10.1007/11841197_10)
18. Carriero, N., Gelernter, D.: The S/Net’s Linda kernel (extended abstract). In: Proceedings of 10th ACM Symposium on Operating System Principles, SOSP 1985, p. 160. ACM (1985)
19. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 261–272. ACM (2008)
20. Chen, T., Dezani-Ciancaglini, M., Scalas, A., Yoshida, N.: On the preciseness of subtyping in session types. *Log. Methods Comput. Sci.* **13**(2), 1–61 (2017)
21. De Nicola, R., Ferrari, G.L., Pugliese, R.: Coordinating mobile agents via blackboards and access rights. In: Garlan, D., Le Métayer, D. (eds.) COORDINATION 1997. LNCS, vol. 1282, pp. 220–237. Springer, Heidelberg (1997). [https://doi.org/10.1007/3-540-63383-9\\_83](https://doi.org/10.1007/3-540-63383-9_83)
22. De Nicola, R., Pugliese, R.: A process algebra based on Linda. In: Ciancarini, P., Hankin, C. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 160–178. Springer, Heidelberg (1996). [https://doi.org/10.1007/3-540-61052-9\\_45](https://doi.org/10.1007/3-540-61052-9_45)
23. Dufourd, C., Finkel, A., Schnoebelen, P.: Reset nets between decidability and undecidability. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 103–115. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0055044>
24. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! *Theor. Comput. Sci.* **256**(1–2), 63–92 (2001)
25. Fournet, C., Hoare, T., Rajamani, S.K., Rehof, J.: Stuck-free conformance. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 242–254. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-27813-9\\_19](https://doi.org/10.1007/978-3-540-27813-9_19)
26. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. *Acta Informatica* **42**(2–3), 191–225 (2005)
27. Gelernter, D.: Generative communication in Linda. *ACM Trans. Program. Lang. Syst.* **7**(1), 80–112 (1985)
28. 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). <https://doi.org/10.1007/BFb0053567>

29. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 273–284. ACM (2008)
30. Jespersen, T.B.L. Munksgaard, P., Larsen, K.F.: Session types for Rust. In: Proceedings of 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, pp. 13–22 (2015)
31. Kozen, D.: Automata and Computability. Springer, New York (1997). <https://doi.org/10.1007/978-1-4612-1844-9>
32. Laneve, C., Padovani, L.: The *Must* preorder revisited. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 212–225. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-74407-8\\_15](https://doi.org/10.1007/978-3-540-74407-8_15)
33. Lange, J., Yoshida, N.: On the undecidability of asynchronous session subtyping. In: Esparza, J., Murawski, A.S. (eds.) FoSSaCS 2017. LNCS, vol. 10203, pp. 441–457. Springer, Heidelberg (2017). [https://doi.org/10.1007/978-3-662-54458-7\\_26](https://doi.org/10.1007/978-3-662-54458-7_26)
34. Lindley, S., Morris, J.G.: Embedding session types in Haskell. In: Proceedings of 9th International Symposium on Haskell, Haskell 2016, pp. 133–145 (2016)
35. Milner, R.: Communication and Concurrency. Prentice Hall, Harlow (1989)
36. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall Inc., Englewood Cliffs (1967)
37. Mostrous, D., Yoshida, N.: Session typing and asynchronous subtyping for the higher-order  $\pi$ -calculus. Inf. Comput. **241**, 227–263 (2015)
38. Mostrous, D., Yoshida, N., Honda, K.: Global principal typing in partially commutative asynchronous sessions. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 316–332. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-00590-9\\_23](https://doi.org/10.1007/978-3-642-00590-9_23)
39. Ng, N., Yoshida, N.: Static deadlock detection for concurrent go by global session graph synthesis. In: Proceedings of 25th International Conference on Compiler Construction, CC 2016, pp. 174–184 (2016)
40. OASIS: Web Services Business Process Execution Language Version 2.0 OASIS Standard. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf>
41. Rensink, A., Vogler, W.: Fair testing. Inf. Comput. **205**(2), 125–198 (2007)
42. Shepherdson, J.C., Sturgis, H.E.: Computability of recursive functions. J. ACM **10**(2), 217–255 (1963)



# Twenty Years of Coordination Technologies: State-of-the-Art and Perspectives

Giovanni Ciatto<sup>1</sup> , Stefano Mariani<sup>2</sup> , Maxime Louvel<sup>3</sup> ,  
Andrea Omicini<sup>1</sup> , and Franco Zambonelli<sup>2</sup> 

<sup>1</sup> Alma Mater Studiorum–Università di Bologna, Cesena, Italy  
{giovanni.ciatto, andrea.omicini}@unibo.it

<sup>2</sup> Università di Modena e Reggio Emilia, Reggio Emilia, Italy  
{stefano.mariani, franco.zambonelli}@unimore.it

<sup>3</sup> Bag-Era, Montbonnot-Saint-Martin, France  
maxime.louvel@bag-era.fr

**Abstract.** Since complexity of inter- and intra-systems interactions is steadily increasing in modern application scenarios (e.g., the IoT), coordination technologies are required to take a crucial step towards maturity. In this paper we look back at the history of the COORDINATION conference in order to shed light on the current status of the coordination technologies there proposed throughout the years, in an attempt to understand success stories, limitations, and possibly reveal the gap between actual technologies, theoretical models, and novel application needs.

**Keywords:** Coordination technologies · Middleware · Survey

## 1 Scope, Goal, and Method

Complexity of computational systems, as well as their impact on our everyday life, is constantly increasing, along with the growing complexity of *interaction*—inter- and intra-systems. Accordingly, the role of *coordination models* should expectedly grow, along with the relevance of *coordination technologies* within ICT systems: instead, this is apparently not happening—*yet*.

Then, it is probably the right time – now, after twenty years of the COORDINATION conference – and the right place – the COORDINATION conference itself – to take a step back and reflect on what happened to coordination models, languages, and (above all) *technologies* in the last two decades. That is why in this paper we survey all the technologies presented and discussed at COORDINATION, examine their stories and their current status, and try to provide an overall view of the state-of-the art of coordination technologies as emerging from twenty years of work by the COORDINATION community. The main goal is to

provide a sound basis to answer questions such as: Are coordination technologies ready for the industry? If not, what is currently missing? Which archetypal models lie behind them? Which are the research areas most/least explored? And what about the target application scenarios?

## 1.1 Structure and Contribution of the Paper

Section 2 first provides an overview of the data about papers published in the conference throughout the years (Subsect. 2.1), as collected from the official SpringerLink website and its companion BookMetrix service, with the aim of emphasising trends concerning (i) the number of *papers* published in each volume, (ii) the number of *citations* generated by each volume, (iii) the number of *downloads* generated by each volume, (iv) the *most cited* paper of each volume, and (v) the *most downloaded* paper of each volume.

Then, the scope of our analysis narrows down to those papers bringing a technological contribution, in the sense of describing a *software artefact* offering an API exploitable by other software to coordinate its components. Accordingly, Subsect. 2.2 provides an overview of all the technologies born within the COORDINATION conference series. For each one, the reference model implemented, and the web URL where to retrieve the software – if any – are given.

Then, a brief description of all the software for which no working implementation could be found is reported for the sake of completeness, whereas technologies still available are thoroughly described in Subsect. 2.3. There, each selected technology was downloaded and tested to clearly depict its *health status*:

- date of last update to the source code (or project web page, if the former is not available)
- whether the software appears to be actively developed, in maintenance mode, or discontinued
- whether suitable documentation is available
- whether the source code is publicly available
- whether the build process of the software artefact is reproducible
- whether the software artefact, once built, executes with no errors

For the latter two items, in case of failures, an explanation of the problem and, if needed, the steps undertaken in the attempt to overcome it, are provided too. In particular, the latter test is not meant to measure performance, or, to provide a benchmark for comparisons: its purpose is to assess whether the technology is *usable*, that is, executable on nowadays software platforms and by nowadays programming languages. For instance, a library requiring an obsolete third-party libraries that hinders smooth deployment is considered not usable. Accordingly, each technology is tested either running provided example code, or developing a minimal working example of usage of the API.

Section 3 discusses the data collected so as to deliver insights about: (i) the *evolution* of technologies as they are stemming from a few archetypal models (Fig. 5), (ii) the *relationships* between the selected technologies, as a *comparison* of their features (Fig. 6), and (iii) the *main goal* and *reference scenario* of

each technology (Fig. 7). Also, a general discussion is provided, reporting about success stories, peculiarities, and opportunities.

Finally, Sect. 4 concludes the paper by summarising the results of the survey and providing some perspectives for the future of coordination technologies.

## 1.2 Method

The scope of this survey is indeed the COORDINATION conference series. There, we focus on coordination *technologies* intended as software implementing a given coordination model, language, mechanism, or approach with the goal of providing coordination *services* to other software applications. In other words, our focus is on technologies implementing some form of *coordination middleware or API*—analysed in Subsect. 2.2. We nevertheless include in our overview other technologies produced within COORDINATION (Subsect. 2.1), such as *simulation* frameworks, *model-checking* tools, and proof-of-concept implementations of *process algebras*—which are only described in short.

Starting from the COORDINATION conference proceedings available online from SpringerLink<sup>1</sup>, the survey proceeds as follows:

1. for each conference year, papers describing a coordination-related technology were gathered manually into a Google Spreadsheet
2. for each collected paper, we checked whether the paper was actually proposing some software package—papers failing the test are omitted
3. for each paper passing the test, we verified the health *status* of the technology—as described in Subsect. 1.1
4. then, for each paper featuring at least a *usable* distribution – meaning a downloadable version of the software – the corresponding software was downloaded and tested—i.e., installation & basic usage

## 2 The Survey

Although the focus of this paper are coordination technologies, we believe an overview of the whole conference proceedings is due to give context to the survey itself. Accordingly, Subsect. 2.1 summarises and analyses all the data officially available from Springer—concerning, for instance, citations and downloads of each volume and paper. Then, Subsect. 2.2 accounts for all the coordination technologies mentioned in COORDINATION papers, regardless of their actual availability, while Subsect. 2.3 reports about the core of this survey: the status of the coordination technologies nowadays publicly available.

### 2.1 Overview

The COORDINATION conference series has been held 19 times since its first edition in 1996 in Cesena (Italy), and generated as many conference proceedings volumes—all available online (See footnote 1). Data about the number of

<sup>1</sup> <http://link.springer.com/conference/coordination>.



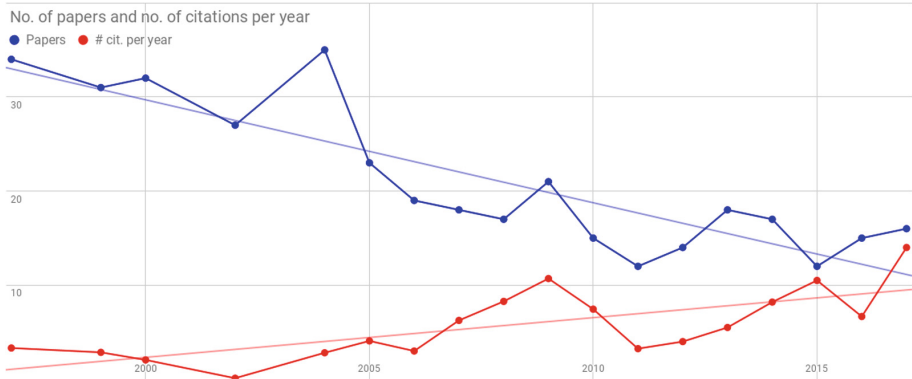
published papers, the number of *citations* and *downloads* per year of each volume, as well as the *most cited* and *most download* paper have been collected from SpringerLink and its companion service BookMetrix<sup>2</sup>—and are reported in Table 1 (last checked February 9th, 2018). Highest values for each column are emphasised in bold.

**Table 1.** Overall data directly available online from Springer regarding the COORDINATION conference series. To compute citations (downloads) per year, the number of citations (downloads) was divided by the number of years the publications is available since. MCP stands for “Most Cited Paper” whereas MDP stands for “Most Downloaded Paper”.

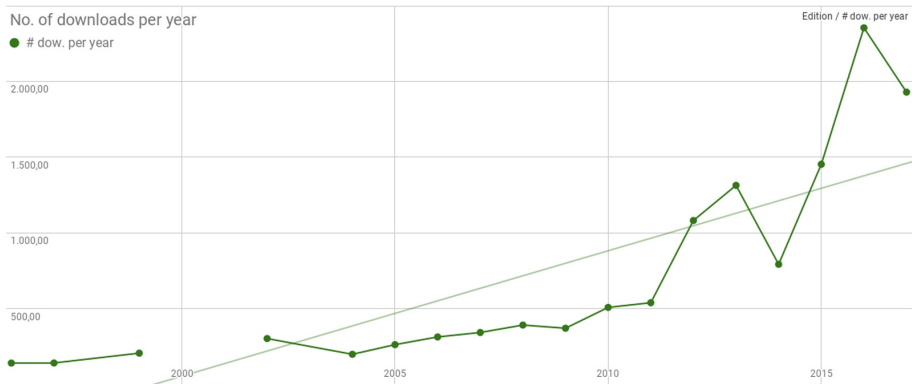
Edition	No. of papers	Citations/year	Downloads/year	MCP	MDP
1996	34	3.32	140.00	16	124
1997	31	2.86	140.48	14	149
1999	32	2.05	205.26	6	154
2000	27	0.11	—	6	158
2002	<b>35</b>	2.81	301.88	7	180
2004	23	4.07	197.86	19	146
2005	19	3.00	261.54	9	214
2006	18	6.25	312.50	<b>22</b>	297
2007	17	8.27	341.82	14	308
2008	21	10.70	391.00	13	227
2009	15	7.44	370.00	13	259
2010	12	3.25	507.50	6	536
2011	14	4.00	538.57	6	<b>675</b>
2012	18	5.50	1081.67	6	523
2013	17	8.20	1314.00	7	547
2014	12	10.50	792.50	10	299
2015	15	6.67	1453.33	11	336
2016	16	<b>14.00</b>	<b>2355.00</b>	4	350
2017	14	2.00	1930.00	1	245
Avg.	20.53	5.53	701.94	10	301.42
Std. Dev.	7.57	3.60	658.47	5.42	160.16

The trend over time of the number of papers, the citations of the volumes, and their downloads, are plotted in Figs. 1 and 2, respectively, along with their trend line. A few significant trends can be spotted in spite of the high variability between different editions of the conference. For the number of published papers,

<sup>2</sup> <http://www.bookmetrix.com/>.



**Fig. 1.** Number of papers in the volume and number of citations per year (computed as described in text) of the volume.



**Fig. 2.** Number of downloads per year (computed as described in text) of the volume.

the trend is clearly *descending*: the first five editions featured an average of 32 papers, whereas the latest five an average of 15. As far as the number of citations per year generated by each volume of the proceedings is concerned, a few oscillations can be observed:

- a first phase (from the 1<sup>st</sup> edition to the 4<sup>th</sup>) shows a *decreasing* number of citations, from 3.32 down to 0.11 (the *all-time-low*)
- then, in a second phase (from the 5<sup>th</sup> to the 10<sup>th</sup> edition) the number of citations *increases*, up to 10.70 in 2008
- finally, after a brief fall in 2009 and 2010, the number of citations per year kept increasing up to the *all-time-high* of 2016 (14.00)

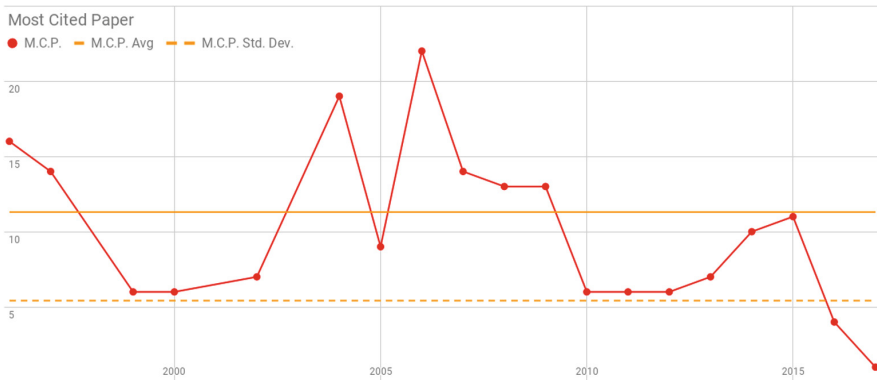
For the number of downloads per year, two phases can be devised out in Fig. 2:

- in the first period, from the 1<sup>st</sup> edition to the 13<sup>th</sup> (2011), the trend is quite *stable*, oscillating between 140 and 538.57

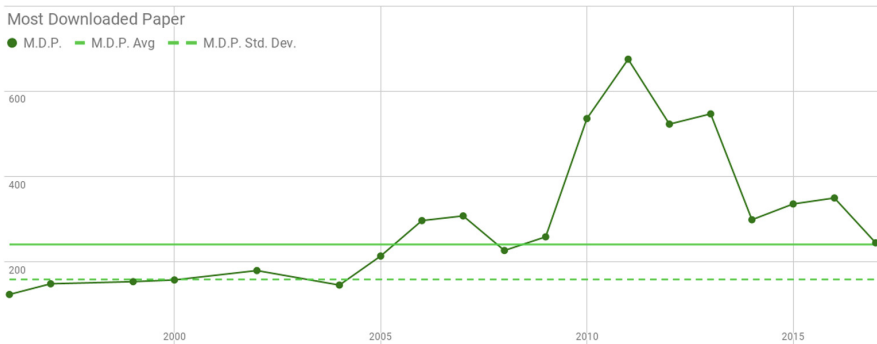
- in the second one instead, from 2012 up to latest edition, there is a *sharp increase* up to the *all-time-high* of 2355.00 in 2016

Finally, Figs. 3 and 4 show the most cited paper and the most downloaded paper per year, respectively. Besides noting (i) the highly *irregular* trend regarding the most cited papers, oscillating from 6 to 22 through approximately three epochs<sup>3</sup> (few citations during 1996–2002, more citations during 2003–2009, few again during 2010–2017), and (ii) the *increasing* number of downloads in recent years: in the four years between 2010 and 2013 the most downloaded papers combined generated more downloads than the most downloaded papers of *all* the previous years combined (2281 vs. 2216), it could be interesting to check how many of such papers are related to technology, if any.

Overall, in the 19 editions of COORDINATION held until now, the most cited/downloaded paper is about technology – in the broadest acceptance of



**Fig. 3.** Most cited paper per year with average values & standard deviation.



**Fig. 4.** Most downloaded paper per year with average values & standard deviation.

<sup>3</sup> Excluding the most recent editions, which had less time to generate citations.

the term – in *slightly less than a half* of them: 7 papers amongst the most cited ones, and 8 amongst the most downloaded ones.

By extending the analysis to all the papers published in the proceedings, instead, out of all the 390 papers published, only 47 (just 12.05%) – based on authors’ inspection of the papers – convey a technological contribution. And, such an estimate is somehow optimistic, since we counted papers just for merely *mentioning* a technology, with no means to access it—see Table 2.

## 2.2 Analysis of Technologies

Table 2 provides an overview of the coordination technologies born within the COORDINATION conference series throughout the years. Only those technologies passing test §2 in Sect. 1.2 are included, that is, those technologies actually delivering some form of *coordination services* to applications—i.e. in the form of a *software library* with suitable API. For each technology, the original paper is referenced, the *model* taken as reference for implementation indicated – if any – and the URL to the technology web page hosting the software given—if any is still reachable. Technologies whose corresponding software is still available – that is, those passing test §3 in Sect. 1.2 – are further discussed in Subsect. 2.3; those with no working software found are briefly described in the following.

*The Early Days.* The first few years of COORDINATION (1996–2000) saw a *flourishing* of successful technologies: some of the ideas introduced back then are still alive and healthy. For instance, *ACLT* [35] adopted *first-order logic terms* as LINDA tuples, an intuition shared by the  $\mu^2$ Log model and its language, MultiBinProlog [31]. Also, *ACLT* allowed agents to *dynamically program* tuple spaces via a specification language, enabling definition of computations to be executed in response to some *events* generated by interacting processes. Both features influenced the TuCSoN model and infrastructure [28], one of the few technologies to be still actively maintained nowadays.

Similarly, the IWIM coordination model and its corresponding language, MANIFOLD [5], were introduced back in 1996 and survived until present days by evolving into Reo [6]. IWIM came by recognising a dichotomy between *exogenous* and *endogenous* coordination, and exploiting *channel composition* as a means to build increasingly complex coordination patterns.

Finally, Moses [4] was presented to the COORDINATION community as an infrastructure reifying the *Low Governed Interaction* (LGI) model. The technology is still alive and inspectable from its homepage, even if apparently no longer maintained. Analogously, the Piccola composition language presented in [2] clearly relies on a coordination technology which reached stability and robustness, even if it seems to be no longer maintained, too.

Besides these success stories, many other papers at that time proposed a technology, but either they only mentioned the technology without actually providing a reference to a publicly available software, or such a reference is no longer reachable (i.e. the link is dead and no reference to the software have been found on the web). For instance:

**Table 2.** Overview of the coordination technologies presented at COORDINATION. “Name” denotes the technology, whereas “Model” makes explicit the model taken as reference for the implementation. The last column points to the web page where the software is available – if any – and provides for additional notes.

Name	Year	Model	(Closest) Web page & notes
Manifold [5]	1996	IWIM [5]	<a href="http://projects.cwi.nl/manifold">http://projects.cwi.nl/manifold</a> <i>no link to implementation</i>
Sonia [12]	1996	LINDA + access control	<i>no implementation found</i>
Laura [92]	1996	service-oriented LINDA	<i>no implementation found</i>
MultiBinProlog [31]	1996	$\mu^2$ Log [31]	<a href="http://cseweb.ucsd.edu/~goguen/courses/230/pl/art.html">http://cseweb.ucsd.edu/~goguen/courses/230/pl/art.html</a> <i>dead links</i>
MESSENGERS [42]	1996	Navigational Programming [42]	<a href="http://www.ics.uci.edu/~bic/messengers">http://www.ics.uci.edu/~bic/messengers</a> <i>dead links</i>
<i>ACLT</i> [35]	1996	LINDA + programmable tuple spaces	<i>evolved into TuCSoN</i>
Blossom [46]	1997	LINDA + coordination patterns	<i>no implementation found</i>
Bonita [84]	1997	asynch LINDA	<i>no implementation found</i>
Berlinda [93]	1997	LINDA	<i>no implementation found</i>
SecOS [22]	1999	LINDA	<i>no implementation found</i>
Messengers [95]	1999	CmPS + mobility [52]	<a href="http://osl.cs.illinois.edu/software/">http://osl.cs.illinois.edu/software/</a> <i>no mention of “Messengers”</i>
MJada [82]	1999	OO LINDA	<a href="http://www.cs.unibo.it/cianca/wwwpages/macondo/">http://www.cs.unibo.it/cianca/wwwpages/macondo/</a> <i>no reference to MJada</i>
STL++ [87]	1999	ECM [87]	<i>no implementation found</i>
Clam [86]	1999	IWIM [5]	<i>no implementation found</i>
TuCSoN [28]	1999	<i>novel</i> (many extensions to LINDA)	<a href="http://tucson.unibo.it/">http://tucson.unibo.it/</a>
Truce [53]	1999	<i>novel</i> (protocols + roles)	<i>no implementation found</i>
CoLaS [29]	1999	<i>novel</i> (protocols + roles)	<i>no implementation found</i>
OpenSpaces [38]	2000	OO LINDA	<i>no implementation found</i>
Piccola [2]	2000	<i>novel</i>	<a href="http://scg.unibe.ch/research/piccola">http://scg.unibe.ch/research/piccola</a>
Moses [4]	2000	LGI [4]	<a href="http://www.moses.rutgers.edu">http://www.moses.rutgers.edu</a>
Scope [63]	2000	LINDA + mobility + space federation	<i>no implementation found</i>
<i>P<math>\epsilon</math>w</i> [6]	2002	IWIM [5]	<a href="http://reo.project.cwi.nl/reo">http://reo.project.cwi.nl/reo</a> <i>evolved into Reo</i>
SpaceTub [94]	2002	LINDA	<i>no implementation found</i>
O’Klaim [15]	2004	OO LINDA + mobility	<a href="http://music.dsi.unifi.it/xklaim">http://music.dsi.unifi.it/xklaim</a> <i>evolved into X-Klaim</i>

(continued)

**Table 2.** (*continued*)

Name	Year	Model	(Closest) Web page & notes
Limone [40]	2004	LINDA + mobility + spaces federation	<a href="http://mobilab.cse.wustl.edu/projects/limone">http://mobilab.cse.wustl.edu/projects/limone</a>
CRIME [65]	2007	LIME [34]	<a href="http://soft.vub.ac.be/amop/crime/introduction">http://soft.vub.ac.be/amop/crime/introduction</a>
TripCom [89]	2007	Triple Space Computing [39]	<a href="http://sourceforge.net/projects/tripcom">http://sourceforge.net/projects/tripcom</a>
CiAN [88]	2008	<i>novel</i>	<a href="http://mobilab.cse.wustl.edu/Projects/CiAN/Home/Home.shtml">http://mobilab.cse.wustl.edu/Projects/CiAN/Home/Home.shtml</a>
Smrl [1]	2008	PEPA [45]	<a href="http://groups.inf.ed.ac.uk/srmc/download.html">http://groups.inf.ed.ac.uk/srmc/download.html</a>
CaSPiS [18]	2008	IMC [17]	<a href="http://sourceforge.net/projects/imc-fi">http://sourceforge.net/projects/imc-fi</a>
LeanProlog [91]	2008	<i>novel</i>	<a href="http://www.cse.unt.edu/~tarau/research/LeanProlog">http://www.cse.unt.edu/~tarau/research/LeanProlog</a>
JErlang [75]	2010	JOIN-CALCULUS [41]	<a href="http://github.com/jerlang/jerlang">http://github.com/jerlang/jerlang</a>
Session Java [68]	2011	Session Types [50]	<a href="http://www.doc.ic.ac.uk/~rhu/sessionj.html">http://www.doc.ic.ac.uk/~rhu/sessionj.html</a>
WikiRecPlay/ InFeed [80]	2012	BPM	<i>no implementation found</i>
Statelets [57]	2012	<i>novel</i>	<a href="http://sourceforge.net/projects/statelets">http://sourceforge.net/projects/statelets</a>
IIC [77]	2012	Reo [6]	<a href="http://github.com/joseproenca/ip-constraints">http://github.com/joseproenca/ip-constraints</a>
LINC [58]	2015	LINDA [44]	<i>implementation not available for commercial reasons</i> <i>see</i> <a href="http://bag-era.fr/index_en.html#about">http://bag-era.fr/index_en.html#about</a>
RepliKlaim [3]	2015	Klaim [19]	<a href="http://sysma.imtlucca.it/wp-content/uploads/2015/03">http://sysma.imtlucca.it/wp-content/uploads/2015/03</a>
Logic Fragments [30]	2014	SAPERE [97]	<i>no implementation found</i>

**Sonia** [12] — a LINDA-like approach supporting *human workflows*, therefore stressing aspects such as understandability of the tuple and template languages, time-awareness and timeouts, and security by means of access control

**Laura** [92] — a language attempting to steer LINDA towards *service-orientation*, where tuples can represent (formal descriptions of) service requests, offers, or results, thus enabling loosely coupled agents to cooperate by means of Linda-like primitives

**MESSENGERS** [42] — following the *Navigational Programming* methodology [42], where strongly-mobile agents – a.k.a. *Messengers* – can migrate between nodes. Here, coordination is seen as “invocation [of distributed computations] and exchange of data” and it “is managed by groups of Messengers propagating autonomously through the computational network”

**Blossom** [46] — a LINDA variant focusing on safety, which is provided by supporting a *type system* for tuples and templates, and a taxonomy of access patterns to tuple spaces, aimed at supporting a sort of “least privilege” principle w.r.t. access rights of client processes

- Bonita** [84] — another LINDA-like technology – as its successor WCL [85] – focusing on *asynchronous* primitives and distribution of tuple spaces, which can also migrate closer to their users
- Berlinda** [93] — providing a *meta-model* – along with a Java implementation – for instantiating different LINDA-like models
- SecOS** [22] — a LINDA variant focusing on *security* and exploring the exploitation of (a)symmetric key encryption
- Messengers** [95] — not to be confused with [42] despite its name, which focusses on *message exchange* by means of migrating actors
- MJada** [82] — an extension of the Jada language [25], focusing on coordinating concurrent (possibly distributed) Java agents by means of LINDA-like tuple spaces with an extended primitive set and *object-oriented tuples*
- Clam** [86] — a coordination language based on the IWIM model [5]
- Truce** [53] — a scripting language aimed at describing *protocols* to which agents must comply by enacting one or more *roles*
- CoLaS** [29] — a model and its corresponding language providing a framework where a number of *participants* can join interaction *groups* and play one or more *roles* within the scope of some coordination *protocol*. In particular, CoLaS focuses on the enforcement of coordination rules by validating and constraining participants behaviour

*The Millennials.* After year 2000, technologies are *less* present amongst COORDINATION papers, but not necessarily less important. For instance, Reo made its first appearance in 2002 [6], its name written in Greek ( $P\epsilon\omega$ ). Reo provides an *exogenous* way of governing interactions between processes in a concurrent and possibly distributed system. Its strength is due to its *sound semantics*, enabling researchers to formally verify system evolution, as well as to the availability of *software tools*. The technology is indeed still alive and actively developed.

Recent implementations are more easily available on the web. Out of 22 coordination technologies, just 6 were not found on the web during the survey:

- OpenSpaces** [38] — focussing on the harmonisation of the LINDA model with the OOP paradigm and, in particular, with the inheritance mechanism
- Scope** [63] — analogously to Lime, it provides multiple distributed tuple spaces cooperating by means of local interactions when some process attempts to access a tuple, thus providing a sort of federated view on the tuple space
- SpaceTub** [94] — successor of Berlinda, it aims at providing a meta-framework where other LINDA-like frameworks can be reproduced
- WikiRecPlay/InFeed** [80] — a pair of tools (browser extensions, no longer available) aimed at extracting and manipulating information from web applications to record them and later *replay*, enabling the definition of sequences of activities that can be *synchronised* with each other. The goal here is to augment *social software* with coordination capabilities.
- LINC** [58] — a coordination environment implementing the basic LINDA primitives – *out*, *in*, *rd* – in a setting in which each tuple space (called bag) could implement the primitives differently (still preserving semantics), a convenient

opportunity when dealing with *physical devices* (i.e. in the case of deployment to IoT scenarios) or *legacy systems* (i.e. databases). It provides *transactions* to alleviate to developers the burden of rolling back actions in the case of failures, and a chemical-reaction model inspired to Gamma [11] for enacting *reaction rules*. Several tools [59] are provided to help developers debug the rules, and to generate rules from high level specifications. The LINC software is nevertheless not publicly available because it is exploited by the Bag-Era company. Accordingly, it is not further analysed in Subsect. 2.3, but it is included in Sect. 3 as an example of industrial success.

**Logic Fragments** [30] — a *chemical-based* and programmable coordination model likewise SAPERE [97] – to which it is inspired – enriched with a *logic-based* one through the notion of Logic Fragments, which are combinations of logic programs defining on-the-fly, ad-hoc chemical reactions – similar to SAPERE eco-laws – that apply on matching tuples to manipulate other tuples or to produce new Logic Fragments. The aim is to guarantee data consistency, favour knowledge representation of partial information, and support constraints satisfaction, thanks to verification of *global properties* enabled by the logic nature of the framework.

All the others are still publicly available, thus further analysed in next section.

For instance, the O’Klaim language presented in [15] evolved into the X-Klaim project [16] which is still alive, even if apparently no longer maintained. Similar considerations can be made for Limone [40] and CRIME [65], which both stem from the idea of *opportunistic federation* of transient tuple spaces introduced by LIME [66], and improve it with additional features such as lightness and orientation to ambient-programming.

Analogously, the CiAN [88] model and middleware, targeting the coordination of distributed workflows over *Mobile Ad-hoc Networks* (MANETs), comes with a mature implementation, although no longer maintained. An extension to Session Java [51] is proposed in [68] to explicitly tackle synchronisation issues such as freedom from deadlock via multi-channel session primitives. Whereas the implementation was discontinued in 2011<sup>4</sup>, the source code is still available from GoogleCode archive. JErLang [75], a Java-based implementation of Erlang extended with constructs borrowed from the JOIN-CALCULUS [41], appears to be no longer maintained too as explicitly stated in its home page<sup>5</sup>, although a couple of implementations are still available and (partially) working.

Also RepliKlaim [3], an implementation of KLAIM [19] aimed at optimising performance and reliability through *replication* of tuples and tuple spaces, received updates until 2015 as far as we know, thus appears to be discontinued. Likewise, 2015 is the year when both Statelets [57] and IIC [77] received their last known update: the former is a programming model and language aimed at integrating *social context* and *network effects*, derived from social networks analysis, as well as *semantic* relationships amongst shared artefacts in, i.e. groupware

<sup>4</sup> Year of latest commit: <https://code.google.com/archive/p/sessionj>.

<sup>5</sup> <http://jerslang.org/>.



applications, into a single and coherent coordination model, while the latter proposes *Interactive Interaction Constraints* (IIC) as a novel framework to ground *channel-based* interaction – *à la* Reo – upon *constraints satisfaction*, interpreting the process of coordinating components as the execution of a constraints solver.

Next section briefly focuses on those technologies—that is, coordination technologies that can be actually installed and used nowadays—step §4 in Sect. 1.2.

### 2.3 Analysis of Selected Technologies

Table 3 overviews the *working technologies* we were able to somewhat successfully test, that is, only those technologies listed in Table 2 which successfully surpassed test §4 described in Sect. 1.2—a software artefact exists and is still working.

It is worth noting that, w.r.t. Table 2, a few technologies are not included in this section despite the corresponding software is available from the reference web page therein referenced. The reason is:

- Smrl requires ancient software to run—that is, an old version of Eclipse requiring in turn an ancient version of the Java runtime (1.4)
- CaSPiS [18] (or better, JCaSPiS, namely the Java-based implementation of CaSPiS) was not found anywhere—neither in the author personal pages, nor in their account profiles on Github, nor in the web pages of the SENSORIA project mentioned in the paper. Nevertheless, the IMC model and framework allegedly grounding its implementation is still accessible<sup>6</sup>. Then we proceeded to download it looking for the CaSPiS code, without success. It is worth to be mentioned, anyway, that the IMC framework code appears to be broken, since compilation fails unless a restricted/deprecated Java API is used<sup>7</sup>, and even in the case of instructing the compiler to allow for it<sup>8</sup> the attempt to run any part of the software failed without informative error messages—just generic Java exceptions.
- LeanProlog is not usable as a coordination technology as defined in Sect. 1.2: it is a Prolog engine with low-level mechanisms for handling multi-threading, and provides no API for general purpose coordination
- Session Java, as explicitly stated in its home page, requires an ancient version of the Java runtime to run, that is, 1.4
- Statelets is explicitly tagged as being in “pre-alpha” development stage, and, upon inspection, revealed to be only partially developed

TuCSoN. Although TuCSoN [28] appeared at COORDINATION in 1999, its roots date back to the first edition of the conference, as the *ACCT* model [35].

TuCSoN is a coordination model adopting LINDA as its core but extending it in several ways, such as by adopting nested tuples (expressed as first-order logic terms), adding primitives (i.e. *bulk* [83] and *uniform* [60]), and replacing tuple

<sup>6</sup> <https://sourceforge.net/projects/imc-fi/>.

<sup>7</sup> A class uses a deprecated API, and another one requires breaking access restrictions.

<sup>8</sup> See <https://goo.gl/pdWCsx>.

**Table 3.** Overview of the working coordination technologies presented at COORDINATION. Column “Health” denotes the status of the software, for instance whether it is still actively developed, only in maintenance mode, or actually discontinued, column “Build” is filled whenever source code is available and denotes whether build steps (i.e. compilation into binaries and dependencies resolution) were successful, column “Deployment” indicates whether the software has been successfully executed. It is worth to emphasise that LINC has been left out since it is part of commercial solutions sold by the Bag-Era company, thus no further inspection of the software was possible.

Name	Last update	Health	Documentation	Source code	Build	Deployment
TuCSoN	2017	Actively developed	Available	Available	Successful	Successful
Moses	2017	Actively developed/maintained	Available	Unavailable	—	Successful
JErlang	2017	Discontinued	Poor	Available	Failed	—
IIC	2015	Discontinued	Poor	Available	Failed	Successful
Reo	2013	Actively developed	Available	Available	Successful	Partially successful
TripCom	2009	Discontinued	Partially available	Available	Successful	Successful
CiAN	2008	Discontinued	Available	Available	Successful	Successful
Piccola	2006	Discontinued	Available	Java only	No Smalltalk	Successful
CRIME	2006	Discontinued	Unavailable	Unavailable	—	Successful
Klava	2004	Discontinued	Poor	Available	Successful	Successful
X-Klaim	2004	Discontinued	Available	Available	Failed	—
Limone	2004	Discontinued	Unavailable	Available	Failed	—
RepliKlaim	— <sup>a</sup>	— <sup>a</sup>	Unavailable	Available	Successful	Successful

<sup>a</sup> There is no publicly available code repository, thus no information about latest commits.

spaces with *tuple centres* [71] programmable in the ReSpecT language [70]. It comes with a Java-based implementation providing *coordination as a service* [96] in the form of a Java library providing API and a middleware runtime, especially targeting distributed Java process but open to rational agents implemented in tuProlog [36]. The TuCSoN middleware is *publicly available* from its home page<sup>9</sup>, which provides both the binaries (a ready-to-use Java jar file) and a link to the *source code* repository. From there, also *documentation* pages are available, in the form of a usage guide and a few tutorials providing insights into specific features. Finally, a few related sub-projects are therein described too, such as TuCSoN4JADE [62] and TuCSoN4Jason [61], which are both Java libraries aimed at integrating TuCSoN with JADE [13] and Jason [21] agent runtimes, respectively, by wrapping TuCSoN services into a more convenient form which best suits those developers accustomed to programming in those platforms.

As far as technology is concerned, TuCSoN is still *actively* developed, being the latest commit in 2017, when also the latest related publication has been produced—an extension to the ReSpecT language and toolchain exploited to program tuple centres in TuCSoN [27]. Also, it is *actively exploited* as the infrastructural backbone for other projects – e.g., the smart home logic-based platform Home Manager [23] – and industrial applications—e.g., the Electronic Health Record solution [37]. Nevertheless, TuCSoN is the results of many years of active development by many different people with many different goals. Thus, despite some success stories, TuCSoN would require some substantial refactoring and refinement before it can become a truly commercially-viable product.

*Moses*. Moses [4] is the technology implementing the *Law Governed Interaction* (LGI) coordination model [64], which aims at controlling the interaction of agents interoperating on the Internet. In LGI, each agent interacts with the system by means of a *controller*, that is, a component exposing a fixed set of primitives allowing agents to exchange messages with other agents. The controller is in charge of intercepting invocations of primitives by interacting agents to check if they are allowed according to the *law* currently adopted by that controller.

Laws are shared declarative specifications dictating how the controller should react when it intercepts events of interest. Laws are expressed either in a Prolog-like language or as Java classes. Each controller has its own state which can be altered by reactions to events and can influence the effect of future reactions. Non-allowed activities are technically prohibited by the controller which takes care of aborting the forbidden operation—for instance, by not forwarding a message to the intended receiver if some conditions are met.

The project home page<sup>10</sup> is well-organised and provides a number of resources focussed on Moses/LGI such as reference papers, manuals, tutorials, JavaDoc, examples. The page also provides an archive with the compiled versions of the *Moses middleware*, the latest one dating back to 2017—suggesting that the

<sup>9</sup> <http://tucson.unibo.it>.

<sup>10</sup> <http://www.moses.rutgers.edu/index.html>.

project is *actively maintained and/or developed*, and representing another success story born within the COORDINATION series. We were able to *successfully* compile and execute the code: however, no source code is provided, and some portion of the web page, such as the JavaDoc, are not updated w.r.t. the current Moses implementation. Finally, Moses still bounds to *deprecated technologies* such as Java Applets, which we believe may hinder its adoption.

*JErlang*. JErlang [75] is an extension of the Erlang language for concurrent and distributed programming featuring *joins* as the basic synchronisation construct—as borrowed from the JOIN-CALCULUS [41]. The web page mentioned in the paper<sup>11</sup> is no longer accessible; by searching JErlang and the authors’ names on the web, a GitHub repository with the same broken reference popped up<sup>12</sup>, apparently tracking the development history of the JErlang technology. There, however, JErlang is described as an implementation of Erlang/OTP on the JVM. Also, another apparently very similar technology is therein referenced: Erjang.

Anyway, JErlang installation and usage instructions are nowhere to be found, and, when trying to build the project through the provided Maven pom.xml file, the build fails due to many errors related to obsolete dependencies—which we were not able to fix. Instead, Erjang GitHub repository – with no clue about its links to the paper – provides installation instructions, however building fails due to a Java compilation failure for a “bad class file” error<sup>13</sup>. We feel then free to declare the implementation as discontinued.

*IIC*. *Interactive Interaction Constraints* (IIC) [77] is a sort of “spin-off” of Reo introduced in 2013 [77]. The original approach of implementing Reo connectors as interaction constraints is extended to allow interaction to take place also *between rounds* of constraints satisfaction. This extends the expressive reach of IIC beyond Reo, and makes the whole process of constraints satisfaction *transactional* w.r.t. observable behaviour.

The IIC software is distributed as a Scala library providing a handy syntax which eases definition of Reo-like connectors. The Scala library *source code* is distributed by means of a GitHub repository<sup>14</sup> where the latest commit dates back to 2015. The library ships with a SBT configuration, allegedly supporting automatic building. Nevertheless, we were not able to reproduce the compilation process since the provided SBT configuration depends on an *ancient SBT version*. Therefore, we consider IIC a *no longer maintained* but *still usable* full-fledged coordination technology.

*Reo*. Reo was firstly introduced to the COORDINATION community in [6], its name in Greek letters ( $P\epsilon\omega$ ). Similarly to the IWIM model, Reo adopts a

<sup>11</sup> <https://www.doc.ic.ac.uk/~susan/jerlang/>.

<sup>12</sup> Second link in “See also” section at <https://github.com/jerlang/jerlang>.

<sup>13</sup> Actual error is: “class file contains malformed variable arity method: [...]”.

<sup>14</sup> <http://github.com/joseproenca/ip-constraints>.

paradigm for exogenous coordination of concurrent and possibly distributed software components. According to the Reo model, *components* are the entities to be coordinated, representing the computations to be performed, while *connectors* are the abstraction reifying coordination rules. The only assumption Reo makes about components is that they have a unique name and a well-defined interface in the form of a set of input ports and output ports. Conversely, connectors are composed by *nodes* and *channels*, or other connectors. A number of coordination schemes can be achieved by combining the different sorts of nodes and channels accordingly. This allows to formally specify *how*, *when*, and upon which conditions data may flow from the input to the output ports of components.

Diverse research activities originated from Reo throughout the years, mostly aimed at (i) analysing the formal properties of both Reo connectors and *constraints automata* [10], which are the computational model behind Reo semantics; and (ii) supporting web services orchestration [54], composition, and verification [55] by means of code generation and verification tools.

Several technologies are available from the Reo tools homepage<sup>15</sup>, collectively branded as the Extensible Coordination Tools (ECT). They consist of various Eclipse IDE plugins, such as a graphical designer for Reo connectors, and a code generator which automatically converts the graphical description into Java sources in which developers may inject applicative logic. Nevertheless, the generated code comes with no explicit support for distribution.

According to their home page, ECT are allegedly compatible with any Eclipse version starting from 3.6; while we were not able to reproduce its installation in that version (due to a dependency requiring an higher version of Eclipse), we succeeded in installing it on Eclipse version 4.7 (the latest available), but the code generator appears *buggy and unstable* – thus hindering further testing – because of several non-informative error messages continuously appearing when trying to use the Reo model designer—which is a required step for code generation.

The ECT source code is available from a Google Code repository<sup>16</sup>—last commit dating back to 2013. In [78] a novel implementation is proposed, named *Dreams*, implemented in Scala and aimed at closing the gap between Reo and distributed systems. Nevertheless, its binary distribution seems *unavailable* and no documentation is provided describing how to compile or use it, thus we were not able to further test this novel *Dreams* framework.

*TripCom*. *TripCom* [89] is essentially a departure from the LINDA model where the tuple space abstraction is brought towards the Semantic Web vision [47] and web-based semantic interoperability in general. The former is achieved by employing the Resource Description Framework (RDF) – that is, a representation of semantic information as a triple “subject-predicate-object” – as the tuple representation language, and by considering tuple spaces as RDF triplets containers. Also, LINDA primitives have been consequently re-thought under a semantics-oriented perspective—that is, by adopting an ad-hoc templating

<sup>15</sup> <http://reo.project.cwi.nl/reo/wiki/Tools>.

<sup>16</sup> <https://code.google.com/archive/p/extensible-coordination-tools/source>.

language enabling expression of semantic relationships. The latter is achieved by making triple spaces accessible on the web as SOAP-based web-services.

The implementation is hosted on a SourceForge repository<sup>17</sup> and it is apparently *discontinued*, provided that the last commit dates back to 2009, and the home page lacks any sort of presentation or reference to publications or documentation. Nevertheless, the available source code appears well engineered and is *well documented*. It can be easily compiled into a `.war` file and then deployed on a Web Server (i.e. Apache Tomcat).

Once deployed, the web service is accessible via HTTP – making it is virtually interoperable with any programming language and platform – and can be tested by means of a common web browser. Additionally, the service exposes a WSDL description of the API needed to use it, which implies that a client library (aka stub) may be automatically generated using standard tools for service-oriented architectures. Nevertheless, this WSDL description is the only form of documentation when it comes to actually interact with the web-service.

*CiAN. Collaboration in Ad hoc Networks* (CiAN) [88] is a Workflow Management System (WfMS) enabling users to schedule and execute their custom workflow over MANETs. It comes with a reference architecture and a middleware. The middleware keeps track of the workflow state in a distributed way, and takes into account routing of tasks' input/output data, on top of a dynamic network topology where nodes communication is likely to be opportunistic.

Workflows in CiAN are modelled as directed graphs whose vertices represent *tasks*, and edges represent the data-flow from a task to its successors: when a task is completed, a result value is transferred through its outgoing edges. Conditions may be specified within task definitions stating, for instance, whether a task should wait for all its inputs or just for one of them.

Users can encode their workflow descriptions via a XML-based language to be endowed to an *initiator* singleton node, distributing the workflow to a number of *coordinator* nodes in charge of allocating tasks to the available *worker* nodes.

While the middleware is implemented in Java, tasks logic can be implemented virtually by means of any language since CiAN only assumes the application logic to interact with the middleware by means of the SOAP protocol, which provides great interoperability. Both the middleware's source code and its compiled version are distributed through CiAN website<sup>18</sup>, together with detailed documentation and some runnable examples. The source code can be easily compiled, and both the obtained binaries and those publicly available can be run smoothly. The code is well documented and engineered. Nevertheless, the source code and documentation both date back to 2008: we therefore consider the project to be mature and usable, but no longer maintained.

*Piccola*. Piccola [2] is in its essence a *composition language*. It provides simple yet powerful abstractions: *forms* as immutable, prototype-like, key-value objects;

<sup>17</sup> <https://sourceforge.net/projects/tripcom>.

<sup>18</sup> <http://mobilab.cse.wustl.edu/Projects/CiAN/Software/Software.shtml>.

*services* as functional forms which can be invoked and executed; *agents* as concurrent services; and *channels* as inter-agent communication facilities. Virtually *any* interaction mechanism can be built by properly composing these abstractions, such as shared variables, push and pull streams, message-passing, publish-subscribe, and so on.

Nevertheless, a limitation is due to the fact that not solely the coordination mechanisms are to be programmed with the Piccola language, but *also* the coordinated entities. There is thus no possibility of integration with mainstream programming languages, which is a severe limitation for adoption. Additionally, even if Piccola comes with networking capabilities *virtually* enabling deployment to a distributed setting, there is no middleware facility available and no opportunity with integration with others is given, which is another factor likely to hinder Piccola adoption within the scope of distributed programming and coordination.

Piccola home page<sup>19</sup> is still available and collects a number of useful resources such as documentation pages and implementation. This comes in two flavours: JPiccola, based on Java, which reached version 3.7, and SPiccola, based on Smalltalk, which reached version 0.7. Source code is provided for the Java implementation only, which *correctly compiles and executes*.

Nevertheless, the project appears to be *discontinued*, given that the last commit on the source repository dates back to 2006.

*CRIME*. CRIME adheres to the *Fact Spaces* model, a variant of LINDA which absorbs transient federation of tuple space from Lime [66] for implementing mobile *fact spaces*—tuple spaces where tuples are logic facts and each tuple space is indeed a logic theory. Federated fact spaces are therefore seen as distributed knowledge bases.

In this sense, CRIME has some similarities with TuCSon, which exploits first-order logic tuples both as the communication items and as the coordination laws. In this context, LINDA *out* and *in* primitives collapse into logic facts assertions and retractions, respectively.

Suspensive semantics is not regarded as being essential within the scope of the *Fact Spaces* model, since the focus is about programming fact spaces to react to information insertion/removal (or appearance/disappearance in case of transient federation). Accordingly, users can register arbitrary logic rules by means of a Prolog-like syntax. The head of such rules represent propositions which may be proved true (activated) or unknown (deactivated) given the current knowledge base by evaluating the body of the rule. Users can then plug arbitrary application logic reacting to (de)activation of these rules.

Implementation of CRIME is available on the project home page<sup>20</sup> and consists of an archive shipping pre-compiled Java classes with no attached source code. The software is apparently *no longer maintained*: the web page has been updated last in 2010, and the archive dates back to 2006. Nevertheless, the archive provides a number of example applications which have been tested and

<sup>19</sup> <http://scg.unibe.ch/research/piccola>.

<sup>20</sup> <http://soft.vub.ac.be/amop/crime/introduction>.

are still *correctly working*. No support is provided to application deployment and *no documentation* has been found describing how to deploy CRIME to an actual production environment.

*Klava-\**. With notation Klava- $\star$  we refer to the family of models and technologies stemming from KLAIM [19] – such as O’Klaim [15] and MoMi [14] – which nowadays evolved into the X-Klaim/Klava framework [20].

X-Klaim consists of a domain-specific language and its compiler, which produces Java code by accepting X-Klaim sources as input. The produced code exploits the Klava library in turn, that is, the Java library implementing the middleware corresponding to the KLAIM model.

The overall framework explicitly targets code mobility, thus allowing both processes and data to migrate across a network. To do so, X-Klaim and Klava provide a first-class abstraction known as *locality*. Localities are of two sorts: either *physical*, such as network *nodes* identifiers, or *logical*, such as symbolic references to network nodes having a local semantics. Each locality hosts its own tuple space, and the processes therein interacting. The LINDA primitives supported by Klava are always explicitly or implicitly related to the tuple space hosted on a specific locality. Furthermore, processes are provided with primitives enabling them to migrate from a locality to another in a *strong* manner, that is, along with their execution state.

Both X-Klaim and Klava are distributed by means of the KLAIM Project home page<sup>21</sup>, providing well detailed *documentation*. For what concerns X-Klaim, its C++ *source code* – dating back to 2004, date of the last edit, visible right below the title – is *publicly available* along with a self-configuring script meant to ease compilation. Nevertheless, we were not able to reproduce the compilation process on modern Linux distributions, seemingly due to some missing (and undocumented) dependency. No clues about how to fix the self-configuration process when it fails is provided, neither we were able to find some sort of documentation explicitly enumerating the compilation dependencies.

Conversely, the Klava library – actually implementing the coordination middleware – is distributed as a single `.jar` file containing both Java sources and the binaries. The `.jar` file dates back to 2004 likewise for X-Klaim, so it is apparently no longer developed, but further testing showed how the Klava library is still *functioning*, since it is self-contained and targets Java versions 1.4+.

*Limone*. Limone [40] is a model and middleware meant to improve scalability and security in Lime [66] through access control, and explicitly targeting distributed mobile systems and, in particular, agents roaming across ad-hoc networks built on top of opportunistically interconnected mobile devices.

Once two or more devices enter within their respective communication range and thus establish a connection, the agents running on top of them are (potentially) enabled to interact by means of transient sharing of their own tuple spaces. But, for some agents to be actually able to communicate, Limone states they should specify their *engagement* policies. An agent *A*’s engagement policy

<sup>21</sup> <http://music.dsi.unifi.it/klaim.html>.



determines which agents are allowed to interact with it and to which extent, that is, which primitives are allowed to be invoked. Agents satisfying the policy are registered within  $A$ 's *acquaintance* list. So, each agent only has to care about its acquaintance list, thus reducing the bandwidth requirements for the middleware.

A reactive programming mechanism completes the picture, enabling agents to inform their peer about their interest in tuples matching a given template, in order to be informed when such tuples becomes available.

The Limone technology is distributed by means of the project web page<sup>22</sup> in the form of a compressed archive containing the Java source code (dated back in 2004) and a `Makefile` for automatic build. Nevertheless, the code strictly requires to be compiled against a Java version *prior to 1.5*, and modern Java compilers do not support such an ancient version<sup>23</sup>. For these reasons, we could not proceed to further test the technology and we consider it to be no longer maintained *nor actually usable*.

**RepliKlaim.** RepliKlaim [3] is a variant of Klaim [19] introducing first-class abstractions and mechanisms to deal with *data locality* and *consistency*, so as to give programmers the ability to explicitly account for and tackle these aspects when developing parallel computing applications. Specifically, the idea is to let the programmer specify and *coordinate replication of data*, and operate on replicas with a configurable level of consistency. This enables the programmer to adapt data distribution and locality to the needs of the application at hand, especially with the goal of improving *performance* in terms of concurrency level and data access speed—in spite of latencies due to distribution.

Most of the abstractions and mechanisms, as well as syntax elements and semantics, of RepliKlaim are exactly as in Klaim, such as data repositories, processes, locations, and many actions. When due, actions are extended to explicitly deal with replication aspects, such as in the case of an `out` primitive putting multiple copies of the same tuple in multiple localities, or an `in` primitive removing all replicas from all locations at once. Also, various *degrees of consistency* among replicas in the same or different locations are achieved depending on whether primitives are synchronous (namely, atomically executed) or asynchronous.

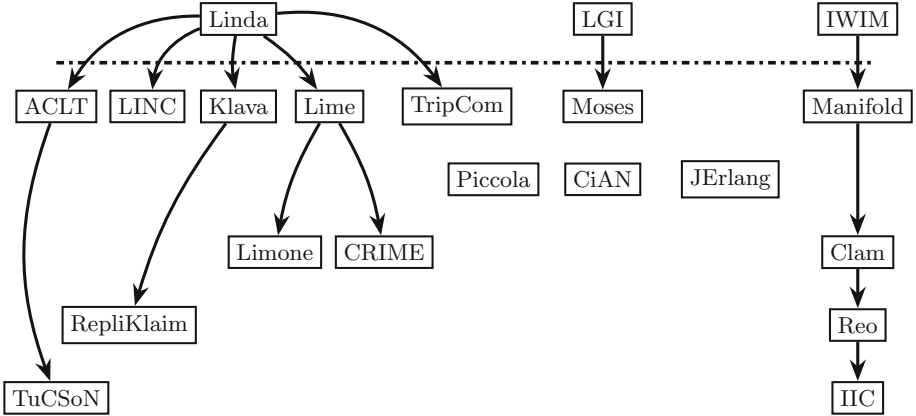
There exists a *prototype* implementation of RepliKlaim on top of Klava, the Java implementation of Klaim, available for direct download from a URL<sup>24</sup> given in its companion paper [3]. From there, a `.rar` archive is provided, containing a version of Klava and the *source* files implementing RepliKlaim, which can be easily compiled and run successfully.

Nevertheless, as stated in the paper describing RepliKlaim, its implementation currently relies on encoding its model in the standard Klaim model, thus, on the practical side the code provided only features examples about how to translate RepliKlaim primitives into Klava. *No higher-level API* directly providing to developers the replica-oriented operations of RepliKlaim is provided.

<sup>22</sup> <http://mobilab.cse.wustl.edu/projects/limone>.

<sup>23</sup> As stated here: <https://docs.oracle.com/javase/9/tools/javac.htm#JSWOR627>.

<sup>24</sup> <http://sysma.imtlucca.it/wp-content/uploads/2015/03/RepliKlaim-test-examples.rar>.



**Fig. 5.** Lines of evolution of selected technologies (below the dashed line), as stemming from a few archetypal coordination model (above the dashed line).

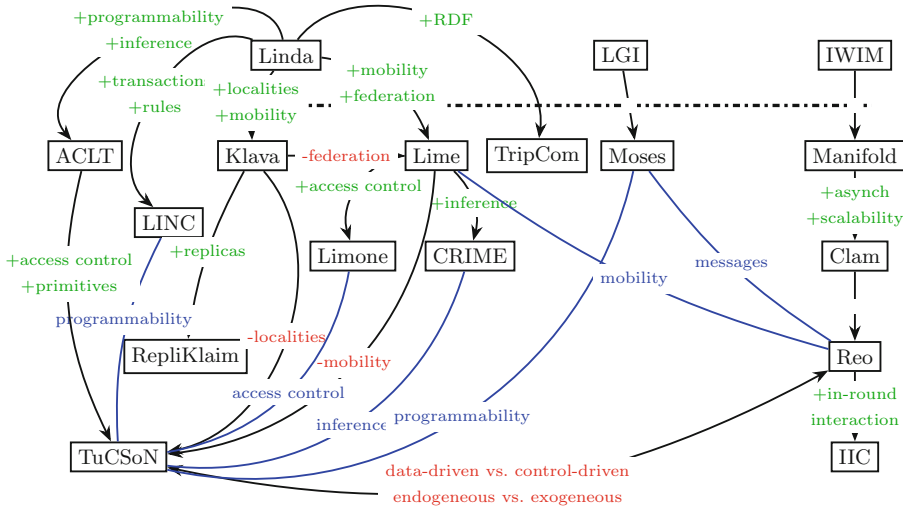
In other words, there exists no RepliKlaim Java library which can be imported to other java projects in order to exploit its provided coordination services.

### 3 Discussion

In this section we aim at providing further insights about the technologies described in Subsect. 2.3, especially to understand (i) whether they stem from a common archetypal coordination framework (Fig. 5), (ii) their relationships in terms of the features they provide (Fig. 6), and (iii) which goal mostly motivated their development and which application scenario they mostly target (Fig. 7).

*A Family Tree.* Figure 5 depicts a sort of “family tree” of the selected coordination technologies, emphasising how they stem from a few archetypal coordination models/languages, and how they are built on each other. It makes thus apparent how most of the technologies still available stem from two archetypal models: LINDA [44] and IWIM [5]. Nevertheless, whereas in the case of LINDA many heterogeneous extensions have been proposed throughout the years, focussing on different features and thus diverging from LINDA in many diverse ways, the evolution of the IWIM model appears much more homogeneous, featuring descendants which “linearly” extend their ancestors’ features. Summing up, from LINDA stem the TuCSoN family, the Klaim [19] family – including Klava and RepliKlaim –, the LIME [74] family – with Limone and CRIME –, besides the lone runners LINC and TripCom, whereas from the IWIM root stems the Reo family—completed by Manifold, Clam, and the latest extension IIC.

Apart from these two big family trees, we have the LGI model, along with its implementation, Moses, and a small group of “lone runners” with unique features: Piccola, CiAN, and JErLang. While the former inspired some features



**Fig. 6.** Main differences (in green and red) and similarities (in blue) amongst selected technologies. Arrows indicate what it takes (in green, add something; in red, remove something) to go from one technology (the source) to another (the destination). (Color figure online)

of technologies stemming from other models – for instance, its *programmable laws* inspired essentially any other technology or model having *reactive rules* of some sort, such as LINC –, the latter remained mostly confined to itself.

It is interesting to notice how “the IWIM family” and “the LINDA family” remained well-isolated one from each other over all these years. Whereas this can be easily attributed to the fundamental difference in the approach to coordination they have – data-driven vs. control-driven, as also emphasised in Fig. 6 – it seems odd that nobody tried to somewhat integrate these two extremely successful coordination models, in an attempt to improve the state of art by cherry-picking a few features from both to create a novel, *hybrid* coordination model [69], with “the best of two worlds”. To some extent, the TuCSoN model, along with its coordination language, ReSpecT, pursues this path: ReSpecT in fact can be regarded as a data-driven model because coordination is based on availability of tuples, as in LINDA, but, at the same time, coordination policies are enforced by declarative specifications which *control* the way in which the coordination medium behaves, thus, ultimately, how the coordinated components interact—as typical for control-driven models like IWIM.

We believe that the path toward integration could be the key in further perfecting and improving coordination models and languages, by complementing data-driven models elegance and flexibility with control-driven models fine-grained control and predictability.

*Families Marriage.* Figure 6 enriches the family tree just described with relationships indicating *differences* (red and green arrows) and *similarities* (blue arrows) in features provided—notice that w.r.t. Fig. 5 Piccola, CiAN, and JERlang have been removed because they are so unique that no clear relationship may be found with other technologies. As already mentioned for Fig. 5, LINDA has been taken as the common ground for many technologies which are instead very heterogeneous in the aim pursued: if *ACLT*, TuCSoN, and LINC have a LINDA core enriched with many other features – such as programmability, transactionality, and novel primitives –, the Klaim family and the LIME one diverge more, by changing the way in which primitives behave – as in the case of localities in Klaim –, or the way in which the interacting processes see each others’ tuple spaces—as for LIME transient federation.

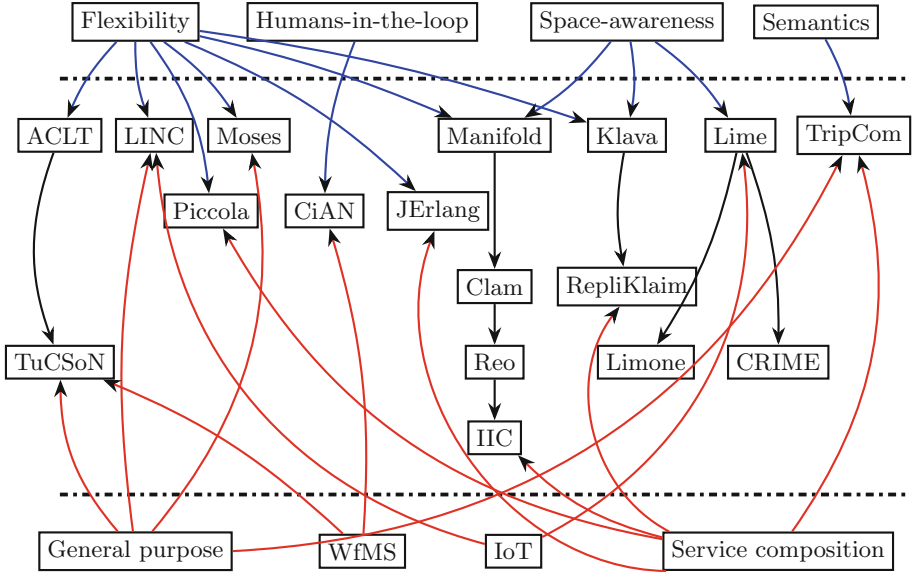
Nevertheless, technologies which may appear as being far apart from each other have interesting similarities, as in the case of the interaction rules of LGI, thus Moses, which strongly resembles *ACLT* and TuCSoN reactions, or the fact that both the Reo family and Moses are based on message passing. Or, the fact that both CRIME and TuCSoN rely on logic tuples so as to leverage on the inference capabilities of interacting agents, while Reo and both LIME and Klaim take into account mobility of processes and coordination abstractions (tuple spaces vs. channels) as a first-class citizen.

It is worth emphasising here that Fig. 6 highlights the features to which more attention has been devoted throughout the years: programmability, access control, and mobility. We believe that these features, possibly extended with scalability and inference capabilities, are crucial for widening applicability of coordination technologies to real-world scenarios. For instance, the Internet of Things (IoT) [9] – along with its variants Web of Things [48] and Internet of Intelligent Things [8] – is a very good fit for testing coordination technologies, and requires precisely the aforementioned features.

*Goals & Preferred Scenarios.* Finally, Fig. 7 relates the selected technologies with the main aim pursued which motivates their extension in a particular direction, along with the applications scenario they best target.

From the description of the selected technologies we gathered, two are the main goals motivating their evolution: (i) providing *flexibility* so as to deal with the majority of heterogeneous application scenarios possible, and (ii) focussing on first-class abstractions for better supporting *space-awareness* of both the coordination abstractions and the interacting processes.

In fact, TuCSoN/*ACLT*, LINC, and Moses all provide means to somewhat *program* the coordinative behaviour of the coordination medium, thus aim at making it configurable, adaptable, malleable, even at run-time, and/or provide additional coordination primitives to expand the expressive reach of the coordination technology. The Klaim family, the Reo family, and the Lime family instead, are geared toward some forms of *space-awareness*, be it by promoting mobility or by providing location-sensitive primitives.



**Fig. 7.** Selected technologies per main *goal* pursued (top, blue arrows) and preferred *application scenario* (bottom, red arrows). (Color figure online)

Besides these, two more main goals can be devised, peculiar to specific technologies: *(iii)* supporting *humans-in-the-loop*, in the case of CiAN, and *(iv)* provide a *semantic* representation of data items, in the case of TripCom.

About the application scenarios explicitly declared as of particular interest for the technology, the most prominent one is *service composition*, which is especially interesting for Piccola, JErLang, the Reo family, the Klaim family, and TripCom—besides being naturally applicable to all other technologies too. Then, whereas technologies such as LINC and the Lime family are mainly tailored to the *IoT landscape*, being meant to cope with the requirements posed by small, possibly portable, possibly embedded devices with low resources, *Workflow Management* (WfMS) is peculiar to CiAN, while also considered by TuCSoN [79]. Besides these application scenarios, there are many technologies without a specific focus, although they have been applied to many different ones, such as TuCSoN itself, LINC, Moses, and TripCom: these have been associated with the generic “General purpose” scenario.

We believe that the goals and application scenarios just highlighted strengthen our previous consideration that the IoT could be the “killer-app” for coordination technologies. In fact, flexibility (there including programmability and configurability), space-awareness (there including mobility and location-awareness), and semantics (there including interoperability of data representation formats) are all necessary ingredients for any non-trivial IoT deployment: the former helps in dealing with *uncertainty* and *unpredictability* typical of the IoT scenarios, the latter is required for building open IoT systems, and some

form of space-awareness is a common feature of many IoT deployments, from retail to industry 4.0. Also, the fact that service composition has been already thoroughly explored is a great advantage and the perfect starting point for tackling IoT challenges: both the IoT and the Web of Things vision foster a world where connected objects provide and consume services, which can be composed in increasingly high-level ones.

## 4 Conclusion

The main aim of this paper is to provide insights about the state-of-the-art of coordination technologies after twenty years of the COORDINATION conference series, and to stimulate informed discussion about future perspectives. Overall, apart from some notable success stories – i.e. the commercial success of LINC along with the active development of TuCSoN and Reo – most coordination technologies have gone through a rapid and effective development at the time they were presented, then lacked further improvements or even maintenance of its usability, thus never reached a wider audience—i.e. outside the COORDINATION community or in the industry. Obviously, something also happens outside the COORDINATION boundaries. For instance, coordination technologies are surveyed in [73], whereas [81] focuses on tuple-based technologies. However, mostly of the technological developments reported here just happened after those survey were published, in 2001 [72].

Although we acknowledge that researchers are usually mostly concerned with providing scientifically-relevant models rather than production-ready software, we also believe that backing up models and languages with more than proof-of-concept software is crucial to promote wider adoption of both the technology itself and the models, which in turn may provide invaluable feedback to researchers for further developing and tuning models. The next decade will probably tell us more about the actual role of coordination technologies in the development of forthcoming application scenarios: the IoT, for instance, is at the “peak of inflated expectations” according to Gartner’s hype cycle for 2017, and is expected to reach the plateau in 2 to 5 years. This means the time is ripe for pushing forward the development of coordination technologies, so as to have them ready when the IoT will be mature enough to actually benefit from their added value.

Besides coordination technologies, we believe the COORDINATION conference is quite healthy: although the number of published papers is decreasing, citations and downloads grows (modulo too recent years), and contributions conveying technological advancements still represent almost a half of all the contributions.

## References

1. Abreu, J., Fiadeiro, J.L.: A coordination model for service-oriented interactions. In: Lea and Zavattaro [56], pp. 1–16
2. Achermann, F., Kneubuehl, S., Nierstrasz, O.: Scripting coordination styles. In: Porto and Roman [76], pp. 19–35
3. Andrić, M., De Nicola, R., Lafuente, A.L.: Replica-based high-performance tuple space computing. In: Holvoet and Viroli [49], pp. 3–18
4. Ao, X., Minsky, N., Nguyen, T.D., Ungureanu, V.: Law-Governed Internet communities. In: Porto and Roman [76], pp. 133–147
5. Arbab, F.: The IWIM model for coordination of concurrent activities. In: Ciancarini and Hankin [24], pp. 34–56
6. Arbab, F., Mavaddat, F.: Coordination through channel composition. In: Arbab and Talcott [7], pp. 22–39
7. Arbab, F., Talcott, C. (eds.): COORDINATION 2002. LNCS, vol. 2315. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-46000-4>
8. Arsénio, A., Serra, H., Francisco, R., Nabais, F., Andrade, J., Serrano, E.: Internet of intelligent things: bringing artificial intelligence into things and communication networks. In: Xhafa, F., Bessis, N. (eds.) Inter-cooperative Collective Intelligence: Techniques and Applications. SCI, vol. 495, pp. 1–37. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-642-35016-0\\_1](https://doi.org/10.1007/978-3-642-35016-0_1)
9. Atzori, L., Iera, A., Morabito, G.: The internet of things: a survey. *Comput. Netw.* **54**(15), 2787–2805 (2010)
10. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.* **61**(2), 75–113 (2006)
11. Banâtre, J.-P., Fradet, P., Le Métayer, D.: Gamma and the chemical reaction model: fifteen years after. In: Calude, C.S., PĂun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2000. LNCS, vol. 2235, pp. 17–44. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-45523-X\\_2](https://doi.org/10.1007/3-540-45523-X_2)
12. Banville, M.: Sonia: an adaptation of Linda for coordination of activities in organizations. In: Ciancarini and Hankin [24], pp. 57–74
13. Bellifemine, F.L., Poggi, A., Rimassa, G.: JADE—a FIPA-compliant agentframework. In: 4th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM-99), pp. 97–108 (1999)
14. Bettini, L., Bono, V., Venneri, B.: Coordinating mobile object-oriented code. In: Arbab and Talcott [7], pp. 56–71
15. Bettini, L., Bono, V., Venneri, B.: O’Klaim: a coordination language with mobile mixins. In: De Nicola et al. [33], pp. 20–37
16. Bettini, L., De Nicola, R.: Mobile distributed programming in X-KLAIM. In: Bernardo, M., Bogliolo, A. (eds.) SFM-Moby 2005. LNCS, vol. 3465, pp. 29–68. Springer, Heidelberg (2005). [https://doi.org/10.1007/11419822\\_2](https://doi.org/10.1007/11419822_2)
17. Bettini, L., De Nicola, R., Falassi, D., Lacoste, M., Lopes, L., Oliveira, L., Paulino, H., Vasconcelos, V.T.: A software framework for rapid prototyping of run-time systems for mobile calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 179–207. Springer, Heidelberg (2005). [https://doi.org/10.1007/978-3-540-31794-4\\_10](https://doi.org/10.1007/978-3-540-31794-4_10)
18. Bettini, L., De Nicola, R., Loreti, M.: Implementing session centered calculi. In: Lea and Zavattaro [56], pp. 17–32
19. Bettini, L., Loreti, M., Pugliese, R.: An infrastructure language for open nets. In: 2002 ACM Symposium on Applied Computing (SAC 2002), pp. 373–377. ACM, New York (2002)

20. Bettini, L., Nicola, R.D., Pugliese, R.: X-Klaim and Klava: programming mobile code. *Electron. Notes Theor. Comput. Sci.* **62**, 24–37 (2002)
21. Bordini, R.H., Hübner, J.F., Wooldridge, M.J.: *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Wiley, Chichester (2007)
22. Bryce, C., Oriola, M., Vitck, J.: A coordination model for agents based on secure spaces. In: Ciancarini and Wolf [26], pp. 4–20
23. Calegari, R., Denti, E.: Building smart spaces on the home manager platform. *ALP Newsllett.* (2016). <https://www.cs.nmsu.edu/ALP/2016/12/building-smart-spaces-on-the-home-manager-platform/>
24. Ciancarini, P., Hankin, C. (eds.): *COORDINATION 1996*. LNCS, vol. 1061. Springer, Heidelberg (1996). <https://doi.org/10.1007/3-540-61052-9>
25. Ciancarini, P., Rossi, D.: Jada: coordination and communication for Java agents. In: Vitek, J., Tschudin, C. (eds.) *MOS 1996*. LNCS, vol. 1222, pp. 213–226. Springer, Heidelberg (1997). [https://doi.org/10.1007/3-540-62852-5\\_16](https://doi.org/10.1007/3-540-62852-5_16)
26. Ciancarini, P., Wolf, A.L. (eds.): *COORDINATION 1999*. LNCS, vol. 1594. Springer, Heidelberg (1999). <https://doi.org/10.1007/3-540-48919-3>
27. Ciatto, G., Mariani, S., Omicini, A.: Programming the interaction space effectively with ReSpecT $\times$ . In: Ivanović, M., Bădică, C., Dix, J., Jovanović, Z., Malgeri, M., Savić, M. (eds.) *IDC 2017*. *SCI*, vol. 737, pp. 89–101. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-66379-1\\_9](https://doi.org/10.1007/978-3-319-66379-1_9)
28. Cremonini, M., Omicini, A., Zambonelli, F.: Coordination in context: authentication, authorisation and topology in mobile agent applications. In: Ciancarini and Wolf [26], p. 416
29. Cruz, J.C., Ducasse, S.: A group based approach for coordinating active objects. In: Ciancarini and Wolf [26], pp. 355–370
30. De Angelis, F.L., Di Marzo Serugendo, G.: Logic Fragments: a coordination model based on logic inference. In: Holvoet and Viroli [49], pp. 35–48
31. De Bosschere, K., Jacquet, J.M.:  $\mu$ 2Log: towards remote coordination. In: Ciancarini and Hankin [24], pp. 142–159
32. De Meuter, W., Roman, G.-C. (eds.): *COORDINATION 2011*. LNCS, vol. 6721. Springer, Heidelberg (2011). <https://doi.org/10.1007/978-3-642-21464-6>
33. De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.): *COORDINATION 2004*. LNCS, vol. 2949. Springer, Heidelberg (2004). <https://doi.org/10.1007/b95570>
34. Dedecker, J., Van Cutsem, T., Mostinckx, S., D’Hondt, T., De Meuter, W.: Ambient-oriented programming. In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, pp. 31–40. ACM, New York (2005)
35. Denti, E., Natali, A., Omicini, A., Venuti, M.: An extensible framework for the development of coordinated applications. In: Ciancarini and Hankin [24], pp. 305–320
36. Denti, E., Omicini, A., Ricci, A.: tuProlog: a light-weight prolog for internet applications and infrastructures. In: Ramakrishnan, I.V. (ed.) *PADL 2001*. LNCS, vol. 1990, pp. 184–198. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-45241-9\\_13](https://doi.org/10.1007/3-540-45241-9_13)
37. Dubovitskaya, A., Urovi, V., Barba, I., Aberer, K., Schumacher, M.I.: A multiagent system for dynamic data aggregation in medical research. *BioMed Res. Int.* **2016** (2016). <https://www.hindawi.com/journals/bmri/2016/9027457/>
38. Ducasse, S., Hofmann, T., Nierstrasz, O.: Openspaces: an object-oriented framework for reconfigurable coordination spaces. In: Porto and Roman [76], pp. 1–18



39. Fensel, D.: Triple-space computing: semantic web services based on persistent publication of information. In: Agesen, F.A., Anutariya, C., Wuwongse, V. (eds.) INTELLCOMM 2004. LNCS, vol. 3283, pp. 43–53. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30179-0\\_4](https://doi.org/10.1007/978-3-540-30179-0_4)
40. Fok, C.L., Roman, G.C., Hackmann, G.: A lightweight coordination middleware for mobile computing. In: De Nicola et al. [33], pp. 135–151
41. Fournet, C., Gonthier, G.: The reflexive CHAM and the join-calculus. In: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 372–385. ACM (1996)
42. Fukuda, M., Bic, L.F., Dillencourt, M.B., Merchant, F.: Intra- and inter-object coordination with MESSENGERS. In: Ciancarini and Hankin [24], pp. 179–196
43. Garlan, D., Le Métayer, D. (eds.): COORDINATION 1997. LNCS, vol. 1282. Springer, Heidelberg (1997). <https://doi.org/10.1007/3-540-63383-9>
44. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. (TOPLAS) **7**(1), 80–112 (1985)
45. Gilmore, S., Hillston, J.: The PEPA workbench: a tool to support a process algebra-based approach to performance modelling. In: Haring, G., Kotsis, G. (eds.) TOOLS 1994. LNCS, vol. 794, pp. 353–368. Springer, Heidelberg (1994). [https://doi.org/10.1007/3-540-58021-2\\_20](https://doi.org/10.1007/3-540-58021-2_20)
46. van der Goot, R., Schaeffer, J., Wilson, G.V.: Safer tuple spaces. In: Garlan and Le Métayer [43], pp. 289–301
47. Hendler, J.A.: Agents and the Semantic web. IEEE Intell. Syst. **16**(2), 30–37 (2001)
48. Heuer, J., Hund, J., Pfaff, O.: Toward the web of things: applying web technologies to the physical world. Computer **48**(5), 34–42 (2015)
49. Holvoet, T., Viroli, M. (eds.): COORDINATION 2015. LNCS, vol. 9037. Springer, Cham (2015). <https://doi.org/10.1007/978-3-319-19282-6>
50. 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). <https://doi.org/10.1007/BFb0053567>
51. Hu, R., Yoshida, N., Honda, K.: Session-based distributed programming in Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 516–541. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-70592-5\\_22](https://doi.org/10.1007/978-3-540-70592-5_22)
52. Jagannathan, S.: Communication-passing style for coordination languages. In: Garlan and Le Métayer [43], pp. 131–149
53. Jamison, W.C., Lea, D.: TRUCE: agent coordination through concurrent interpretation of role-based protocols. In: Ciancarini and Wolf [26], pp. 384–398
54. Jongmans, S.S.T.Q., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Orchestrating web services using Reo: from circuits and behaviors to automatically generated code. SOCA **8**(4), 277–297 (2014)
55. Kokash, N., Krause, C., de Vink, E.: Reo + mCRL2: a framework for model-checking dataflow in service compositions. Formal Aspects Comput. **24**(2), 187–216 (2012)
56. Lea, D., Zavattaro, G. (eds.): COORDINATION 2008. LNCS, vol. 5052. Springer, Heidelberg (2008). <https://doi.org/10.1007/978-3-540-68265-3>
57. Liptchinsky, V., Khazankin, R., Truong, H.L., Dustdar, S.: Statelets: coordination of social collaboration processes. In: Sirjani [90], pp. 1–16
58. Louvel, M., Pacull, F.: LINC: a compact yet powerful coordination environment. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 83–98. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-662-43376-8\\_6](https://doi.org/10.1007/978-3-662-43376-8_6)

59. Louvel, M., Pacull, F., Rutten, E., Sylla, A.N.: Development tools for rule-based coordination programming in LINC. In: Jacquet, J.-M., Massink, M. (eds.) COORDINATION 2017. LNCS, vol. 10319, pp. 78–96. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-59746-1\\_5](https://doi.org/10.1007/978-3-319-59746-1_5)
60. Mariani, S., Omicini, A.: Coordination mechanisms for the modelling and simulation of stochastic systems: the case of uniform primitives. *SCS M&S Mag.* **IV**(3), 6–25 (2014)
61. Mariani, S., Omicini, A.: Multi-paradigm coordination for MAS: integrating heterogeneous coordination approaches in MAS technologies. In: Santoro, C., Messina, F., De Benedetti, M. (eds.) WOA 2016 – 17th Workshop “From Objects to Agents”, CEUR-WS.org, vol. 1664, pp. 91–99. Sun SITE Central Europe, 29–30 July 2016
62. Mariani, S., Omicini, A., Sangiorgi, L.: Models of autonomy and coordination: integrating subjective and objective approaches in agent development frameworks. In: Camacho, D., Braubach, L., Venticinque, S., Badica, C. (eds.) Intelligent Distributed Computing VIII. SCI, vol. 570, pp. 69–79. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-10422-5\\_9](https://doi.org/10.1007/978-3-319-10422-5_9)
63. Merrick, I., Wood, A.: Scoped coordination in open distributed systems. In: Porto and Roman [76], pp. 311–316
64. Minsky, N.H., Leichter, J.: Law-Governed Linda as a coordination model. In: Ciancarini, P., Nierstrasz, O., Yonezawa, A. (eds.) ECOOP 1994. LNCS, vol. 924, pp. 125–146. Springer, Heidelberg (1995). [https://doi.org/10.1007/3-540-59450-7\\_8](https://doi.org/10.1007/3-540-59450-7_8)
65. Mostinckx, S., Scholliers, C., Philips, E., Herzeel, C., De Meuter, W.: Fact spaces: coordination in the face of disconnection. In: Murphy and Vitek [67], pp. 268–285
66. Murphy, A.L., Picco, G.P., Roman, G.C.: LIME: a coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Method. (TOSEM)* **15**(3), 279–328 (2006)
67. Murphy, A.L., Vitek, J. (eds.): COORDINATION 2007. LNCS, vol. 4467. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-3-540-72794-1>
68. Ng, N., Yoshida, N., Pernet, O., Hu, R., Kryftis, Y.: Safe parallel programming with session Java. In: De Meuter and Roman [32], pp. 110–126
69. Omicini, A.: Hybrid coordination models for handling information exchange among internet agents. In: Bonarini, A., Colombetti, M., Lanzi, P.L. (eds.) Workshop “Agenti intelligenti e Internet: teorie, strumenti e applicazioni”, 7th AI\*IA Convention (AI\*IA 2000), Milano, Italy, pp. 1–4, 13 September 2000
70. Omicini, A.: Formal ReSpecT in the A&A perspective. *Electron. Notes Theor. Comput. Sci.* **175**(2), 97–117 (2007)
71. Omicini, A., Denti, E.: From tuple spaces to tuple centres. *Sci. Comput. Program.* **41**(3), 277–294 (2001)
72. Omicini, A., Zambonelli, F., Klusch, M., Tolksdorf, R. (eds.): Coordination of Internet Agents: Models, Technologies, and Applications. Springer, Heidelberg (2001). <https://doi.org/10.1007/978-3-662-04401-8>
73. Papadopoulos, G.A.: Models and technologies for the coordination of Internet agents: a survey. In: Omicini et al. [72], chap. 2, pp. 25–56
74. Picco, G.P., Murphy, A.L., Roman, G.C.: LIME: Linda meets mobility. In: 1999 International Conference on Software Engineering (ICSE 1999), pp. 368–377, May 1999
75. Plociniczak, H., Eisenbach, S.: JErLang: Erlang with joins. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 61–75. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-13414-2\\_5](https://doi.org/10.1007/978-3-642-13414-2_5)
76. Porto, A., Roman, G.-C. (eds.): COORDINATION 2000. LNCS, vol. 1906. Springer, Heidelberg (2000). <https://doi.org/10.1007/3-540-45263-X>

77. Proença, J., Clarke, D.: Interactive interaction constraints. In: De Nicola, R., Julien, C. (eds.) COORDINATION 2013. LNCS, vol. 7890, pp. 211–225. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-38493-6\\_15](https://doi.org/10.1007/978-3-642-38493-6_15)
78. Proença, J., Clarke, D., de Vink, E., Arbab, F.: Dreams: a framework for distributed synchronous coordination. In: 27th Annual ACM Symposium on Applied Computing (SAC 2012), pp. 1510–1515. ACM, New York (2012)
79. Ricci, A., Omicini, A., Denti, E.: Virtual enterprises and workflow management as agent coordination issues. *Int. J. Coop. Inf. Syst.* **11**(3/4), 355–379 (2002)
80. Rossi, D.: A social software-based coordination platform. In: Sirjani [90], pp. 17–28
81. Rossi, D., Cabri, G., Denti, E.: Tuple-based technologies for coordination. In: Omicini et al. [72], chap. 4, pp. 83–109
82. Rossi, D., Vitali, F.: Internet-based coordination environments and document-based applications: a case study. In: Ciancarini and Wolf [26], pp. 259–274
83. Rowstron, A.I.T.: Bulk primitives in Linda run-time systems. Ph.D. thesis, The University of York (1996)
84. Rowstron, A.I.T.: Using asynchronous tuple-space access primitives (bonita primitives) for process co-ordination. In: Garlan and Le Métayer [43], pp. 426–429
85. Rowstron, A.I.T.: WCL: a co-ordination language for geographically distributed agents. *World Wide Web* **1**(3), 167–179 (1998)
86. Sample, N., Beringer, D., Melloul, L., Wiederhold, G.: CLAM: Composition language for autonomous megamodules. In: Ciancarini and Wolf [26], pp. 291–306
87. Schumacher, M., Chantemargue, F., Hirsbrunner, B.: The STL++ coordination language: a base for implementing distributed multi-agent applications. In: Ciancarini and Wolf [26], pp. 399–414
88. Sen, R., Roman, G.C., Gill, C.: CiAN: a workflow engine for MANETs. In: Lea and Zavattaro [56], pp. 280–295
89. Simperl, E., Krummenacher, R., Nixon, L.: A coordination model for triplespace computing. In: Murphy and Vitek [67], pp. 1–18
90. Sirjani, M. (ed.): COORDINATION 2012. LNCS, vol. 7274. Springer, Heidelberg (2012). <https://doi.org/10.1007/978-3-642-30829-1>
91. Tarau, P.: Coordination and concurrency in multi-engine Prolog. In: De Meuter and Roman [32], pp. 157–171
92. Tolksdorf, R.: Coordinating services in open distributed systems with Laura. In: Ciancarini and Hankin [24], pp. 386–402
93. Tolksdorf, R.: Berlinda: An object-oriented platform for implementing coordination languages in Java. In: Garlan and Le Métayer [43], pp. 430–433
94. Tolksdorf, R., Rojec-Goldmann, G.: The SPACETUB models and framework. In: Arbab and Talcott [7], pp. 348–363
95. Varela, C., Agha, G.: A hierarchical model for coordination of concurrent activities. In: Ciancarini and Wolf [26], pp. 166–182
96. Viroli, M., Omicini, A.: Coordination as a service. *Fundamenta Informaticae* **73**(4), 507–534 (2006)
97. Zambonelli, F., Omicini, A., et al.: Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive Mob. Comput.* **17**, 236–252 (2015)



# On Multiplicities in Tuple-Based Coordination Languages: The Bach Family of Languages and Its Expressiveness Study

Denis Darquennes<sup>1</sup> , Jean-Marie Jacquet<sup>1</sup> , and Isabelle Linden<sup>2</sup> 

<sup>1</sup> Faculty of Computer Science, University of Namur,  
Rue Grandgagnage 21, 5000 Namur, Belgium

{denis.darquennes, jean-marie.jacquet}@unamur.be

<sup>2</sup> Business Administration Department, University of Namur,  
Rempart de la Vierge 8, 5000 Namur, Belgium  
isabelle.linden@unamur.be

**Abstract.** Building upon previous work by the authors, this paper reviews and proposes extensions of Linda-like languages aiming at coordinating data-intensive distributed systems. The languages manipulate tokens associated in different ways with a notion of multiplicity. Thanks to De Boer and Palamidessi's notion of modular embedding, we establish expressiveness hierarchies. We also discuss implementation issues and argue that the more expressive the language is the more expensive is its implementation.

## 1 Introduction

Technological evolutions over the last recent years have confirmed the upward trends in pervading our everyday environment by more and more numerical artifacts, mobile or not, injecting or retrieving an endless increasing amount of information. As a result, service-oriented applications have become more and more necessary and indeed have been developed at an increasing speed. Most of them are based on popularity and quality measures, with, as a key feature, the fact that these measures are not determined at specific points in time but rather continuously, as user experiences evolve.

Moreover, with the help of machine learning techniques, data tend to be more and more transformed in knowledge, which leads our daily life to be more and more mediated by knowledge systems. In this context coordinating systems relying on a huge amount of data appears to be a central task. Coordination languages and models have proved to be well suited to program the interaction of conventional distributed systems, and in particular to model service-oriented applications (see e.g. [7, 23, 31]). Recently, it has been shown in [28] how they can be used to code complex socio-technological systems based on the interaction of

knowledge intensive components. This paper aims at addressing a more fundamental issue in exploring how the addition of multiplicity information to tuples increases the expressiveness of Linda [18], the seminal coordination language, while being able to handle the above mentioned requirement of popularity and quality measures.

To do so, we shall start with a dialect of Linda developed at the University of Namur, named Bach. Following Linda, it permits to model in an elegant way the interaction between different components through the deposit and retrieval of tuples in a shared space. As its basic form only allows the manipulation of one tuple at a time and since the selection between several tuples matching a required one is provided in a non-deterministic fashion, a first extension was proposed in [20] in the aim of enriching traditional data-based coordination languages by a notion of multiplicity (historically named density) attached to tuples, thereby yielding a new coordination language, called Dense Bach. In a second extension we have proposed in [16] to consider lists of tuples among which densities are distributed. The resulting language has been named DBD-Bach. It turns out that its presentation can be made more elegant by using a variant, named VD-Bach, in which arguments of coordination primitives are composed of lists of so-called dense tokens.

Introducing variants of languages necessarily calls for a gain of expressiveness. Based on previous work by the authors, among others of [8, 11, 12, 16, 20, 27], we shall employ de Boer and Palamidessi’s modular embedding and show that Bach is less expressive than Dense Bach, which itself is less expressive than VD-Bach. VD-Bach being similar in essence to multiset rewriting, as introduced in Gamma [1, 2], we shall also compare the two languages and prove that Gamma is actually more expressive than VD-Bach.

Since our purposes are essentially of a theoretical nature, for simplicity purposes, we shall consider in this paper simplified versions of the languages where tuples are taken in their simplest form of flat and unstructured tokens. Nevertheless, as we shall argue at the end of the paper, the resulting simplification of the matching process is orthogonal to our purposes and, consequently, our results can be directly extended to more general tuples. Consequently, our languages will subsequently be renamed with a *T* suffix, thus yielding BachT, DBD-BachT and VD-BachT.

Despite this simplification, we shall also discuss implementation issues and, show, without big surprise, that the more expressive a language is the more expensive is its implementation. This highlights from another perspective the expressiveness results: instead of directly using the more expressive language, it is of interest from an efficiency point of view to use the language just expressive enough for coding purposes under consideration.

The rest of the paper is organized as follows. Section 2 presents the languages studied in the paper and defines an operational semantics. Section 3 evidences the interest of these languages through the coding of some examples but also by showing how the newly introduced language VD-BachT can express the language DBD-Bach introduced in [16]. Section 4 provides a short presentation of

modular embedding and, on that basis, proceeds with an exhaustive comparison of the relative expressive power of the languages. Section 5 shows how these expressiveness results can be lifted to tuple-based languages. Section 6 discusses implementation issues. Finally, Sect. 7 compares our work with related work, draws our conclusions and presents expectations for future work.

## 2 Densed Tuple-Based Coordination Languages

### 2.1 Primitives

#### A. BachT and Dense BachT

Let us start by defining the BachT and Dense BachT languages [20] from which the languages under study in this paper are extensions. The following definition formalizes how we attach a multiplicity or density to them.

**Definition 1.** *Let  $\text{Stoken}$  be a enumerable set, the elements of which are subsequently called tokens and are typically represented by the letters  $t$  and  $u$ . Define the association of a token  $t$  and a positive integer  $n \in \mathbb{N}$  as a dense token. Such an association is typically denoted as  $t(n)$ . Define then the set of dense tokens as the set  $\text{SDtoken}$ . Note that since  $\text{Stoken}$  and  $\mathbb{N}$  are both enumerable, the set  $\text{SDtoken}$  is also enumerable.*

*Intuitively, a dense token  $t(m)$  represents the simultaneous presence of  $m$  occurrences of  $t$ . As a result,  $\{t(m)\}$  is subsequently used to represent the multiset  $\{t, \dots, t\}$  composed of these  $m$  occurrences. Moreover, given two multisets of tokens  $\sigma$  and  $\tau$ , we shall use  $\sigma \cup \tau$  to denote the multiset union of elements of  $\sigma$  and  $\tau$ . As a particular case, by slightly abusing the syntax in writing  $\{t(m), t(n)\}$ , we have  $\{t(m)\} \cup \{t(n)\} = \{t(m), t(n)\} = \{t(m+n)\}$ . Finally, we shall use  $\sigma \uplus \{t(m)\}$  to denote, on the one hand, the multiset union of  $\sigma$  and  $\{t(m)\}$ , and, on the other hand, the fact that  $t$  does not belong to  $\sigma$ .*

**Definition 2.** *Define the set  $\mathcal{T}_b$  of the token-based primitives as the set of primitives  $T_b$  generated by the following grammar:*

$$T_b ::= \text{tell}(t) \mid \text{ask}(t) \mid \text{get}(t) \mid \text{nask}(t)$$

*where  $t$  represents a token. Similarly, define the set of dense token-based primitives  $\mathcal{T}_{db}$  as the set of primitives  $T_{db}$  generated by the following grammar:*

$$T_{db} ::= \text{tell}(t(m)) \mid \text{ask}(t(m)) \mid \text{get}(t(m)) \mid \text{nask}(t(m))$$

*where  $t$  represents a token and  $m$  a positive natural number.*

The primitives of the BachT language are essentially the Linda ones rephrased in a constraint-like setting. As a result, by calling *store* a multiset of tokens aiming at representing the current content of the tuple space, the execution of the  $\text{tell}(t)$  primitive amounts to enriching the store by an occurrence of  $t$ . The  $\text{ask}(t)$  and  $\text{get}(t)$  primitives check whether  $t$  is present on the store with

$$\begin{aligned}
(\mathbf{T}) \quad & \langle \text{tell}(t) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \cup \{t\} \rangle \\
(\mathbf{A}) \quad & \langle \text{ask}(t) \mid \sigma \cup \{t\} \rangle \longrightarrow \langle E \mid \sigma \cup \{t\} \rangle \\
(\mathbf{G}) \quad & \langle \text{get}(t) \mid \sigma \cup \{t\} \rangle \longrightarrow \langle E \mid \sigma \rangle \\
(\mathbf{N}) \quad & \frac{t \notin \sigma}{\langle \text{nask}(t) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \rangle}
\end{aligned}$$

**Fig. 1.** Transition rules for token-based primitives (BachT)

$$\begin{aligned}
(\mathbf{T}_d) \quad & \frac{m \in \mathbb{N}_0}{\langle \text{tell}(t(m)) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \cup \{t(m)\} \rangle} \\
(\mathbf{A}_d) \quad & \frac{m \in \mathbb{N}_0}{\langle \text{ask}(t(m)) \mid \sigma \cup \{t(m)\} \rangle \longrightarrow \langle E \mid \sigma \cup \{t(m)\} \rangle} \\
(\mathbf{G}_d) \quad & \frac{m \in \mathbb{N}_0}{\langle \text{get}(t(m)) \mid \sigma \cup \{t(m)\} \rangle \longrightarrow \langle E \mid \sigma \rangle} \\
(\mathbf{N}_d) \quad & \frac{n < m}{\langle \text{nask}(t(m)) \mid \sigma \uplus \{t(n)\} \rangle \longrightarrow \langle E \mid \sigma \uplus \{t(n)\} \rangle}
\end{aligned}$$

**Fig. 2.** Transition rules for dense token-based primitives (Dense BachT)

the latter removing one occurrence. Dually,  $\text{nask}(t)$  tests whether  $t$  is absent from the store.

The primitives of the Dense BachT language extend these primitives by simultaneously handling multiple occurrences. Accordingly,  $\text{tell}(t(m))$  atomically puts  $m$  occurrences of  $t$  on the store and  $\text{ask}(t(m))$  together with  $\text{get}(t(m))$  require the presence of at least  $m$  occurrences of  $t$  with the latter removing  $m$  of them. Moreover,  $\text{nask}(t(m))$  verifies that there are less than  $m$  occurrences of  $t$ .

These executions can be formalized by the transition steps of Figs. 1 and 2, where configurations are pairs of instructions, for the moment reduced to simple primitives, coupled to the contents of a store. Note that  $E$  is used to denote a terminated computation. As can be seen by the above description, the primitives of BachT are those of Dense BachT with a density of 1. Consequently, our explanation starts by the more general rules of Fig. 2. Rule  $(T_d)$  states that for any store  $\sigma$  and any token  $t$  with density  $m$ , the effect of the tell primitive is to enrich the current multiset of tokens by  $m$  occurrences of token  $t$ . Note that  $\cup$  denotes multi-set union. Rules  $(A_d)$  and  $(G_d)$  specify the effect of ask and get primitives, both requiring the presence of at least  $m$  occurrences of  $t$ , but the latter also consuming them. Rule  $(N_d)$  defines the nask primitive, which tests for the absence of  $m$  occurrences of  $t$ . Note that there might be some provided there are less than  $m$ . It is also worth observing that thanks to the notation  $\sigma \uplus \{t(n)\}$  one is sure that  $t$  does not occur in  $\sigma$  and consequently that there

$$\begin{aligned}
 (\mathbf{T}_v) \quad & \frac{m_1, \dots, m_n \in \mathbb{N}_0}{\langle \text{tell}(t_1(m_1), \dots, t_n(m_n)) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \cup \{t_1(m_1), \dots, t_n(m_n)\} \rangle} \\
 (\mathbf{A}_v) \quad & \frac{m_1, \dots, m_n \in \mathbb{N}_0}{\langle \text{ask}(t_1(m_1), \dots, t_n(m_n)) \mid \sigma \cup \{t_1(m_1), \dots, t_n(m_n)\} \rangle \longrightarrow \langle E \mid \sigma \cup \{t_1(m_1), \dots, t_n(m_n)\} \rangle} \\
 (\mathbf{G}_v) \quad & \frac{m_1, \dots, m_n \in \mathbb{N}_0}{\langle \text{get}(t_1(m_1), \dots, t_n(m_n)) \mid \sigma \cup \{t_1(m_1), \dots, t_n(m_n)\} \rangle \longrightarrow \langle E \mid \sigma \rangle} \\
 (\mathbf{N}_v) \quad & \frac{m_1, \dots, m_n \in \mathbb{N}_0, p_1 < m_1, \dots, p_n < m_n}{\langle \text{nask}(t_1(m_1), \dots, t_n(m_n)) \mid \sigma \uplus \{t_1(p_1), \dots, t_n(p_n)\} \rangle \longrightarrow \langle E \mid \sigma \uplus \{t_1(p_1), \dots, t_n(p_n)\} \rangle}
 \end{aligned}$$

**Fig. 3.** Transition rules for vectorized dense token-based primitives (VD-BachT)

are exactly  $n$  occurrences of  $t$ . This does not apply for rules  $(A_d)$  and  $(G_d)$  for which it is sufficient to assume the presence of at least  $m$  occurrences, allowing  $\sigma$  to contain others.

Figure 1 specifies the transition rules for the primitives of the BachT language. As expected, they amount to the rules of Fig. 2 where the density  $m$  is taken to be 1.

## B. Vectorized Dense BachT

A natural extension is to replace a dense token by a set of dense tokens in the primitives. For instance, the primitive  $\text{ask}(t(1), u(2), v(3))$  would succeed on a store containing one occurrence of  $t$ , two of  $u$  and three of  $v$ . Dually, the computation of  $\text{tell}(t(1), u(2), v(3))$  would result in adding one occurrence of  $t$  on the store, two of  $u$  and three of  $v$ .

The following definitions formalize this intuition. As seen above, to avoid using unnecessary brackets, we shall slightly abuse notations and use lists of dense tokens, which we shall subsequently designate as vectors of dense tokens, hence the name Vectorized Dense BachT or VD-BachT for short. The intuition remains however that of sets, with the order of the dense tokens being meaningless.

**Definition 3.** *Define a vector of dense tokens as a list  $\overrightarrow{t_1(m_1), \dots, t_n(m_n)}$  of dense tokens. Such a vector is subsequently denoted as  $\overrightarrow{t(m)}$ . Define SVDtoken as the set of vectors of dense tokens.*

**Definition 4.** *Define the set of vectorized dense token-based primitives  $\mathcal{T}_{vb}$  as the set of primitives  $T_{vb}$  generated by the following grammar:*

$$T_{vb} ::= \text{tell}(\overrightarrow{t(m)}) \mid \text{ask}(\overrightarrow{t(m)}) \mid \text{get}(\overrightarrow{t(m)}) \mid \text{nask}(\overrightarrow{t(m)})$$

where  $\overrightarrow{t(m)}$  represents a vector of dense tokens.

The transition steps for these primitives are defined in Fig. 3. As suggested above, rule  $(T_v)$  asserts that telling a vector of dense tokens amounts to adding



$$(\mathbf{W}_v) \frac{m_1, \dots, m_n \in \mathbb{N}_0, \{t_1(p_1), \dots, t_n(p_n)\} \not\subseteq \sigma}{\langle \overrightarrow{wnask}(t_1(m_1), \dots, t_n(m_n)) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \rangle}$$

**Fig. 4.** Transition rule for the weak nask

each of them with the corresponding density on the store. Similarly, rule  $(A_v)$  requires for an ask primitive to succeed the presence, for each token  $t_i$ , of at least  $m_i$  occurrences on the store. According to rule  $(G_v)$  the behavior of a get primitive performs such a test for presence but also removes  $m_i$  occurrences of  $t_i$  on the store. Finally, rule  $(N_v)$  requires, for each token  $t_i$ , the absence of  $m_i$  occurrences. It is here worth noting that, in contrast to BachT and Dense BachT, the behavior of the nask primitive is not the negation of that of the ask primitive. Indeed, this interpretation would have required for the nask primitive that, for some token  $t_i$ , less than  $m_i$  occurrences are present on the store. It will however be handful to have such a nask primitive. We thus introduce it, name it weak nask and denote it by  $\overrightarrow{wnask}$ . It is formally defined by rule  $(W_w)$  of Fig. 4.

It is worth observing that with such a definition,  $\overrightarrow{wnask}(t(m))$  succeeds whenever  $\overrightarrow{nask}(t(m))$  succeeds. However, the converse is not true. Consider, for instance, the store composed of 2 occurrences of  $t$  and 4 of  $u$ . In that context,  $\overrightarrow{nask}(t(1), u(5))$  does not succeed since, although  $4 < 5$  the inequality  $2 < 1$  does not hold. However,  $\overrightarrow{wnask}(t(1), u(5))$  succeeds since, as multisets,  $\{t(2), u(4)\} \not\subseteq \{t(1), u(5)\}$ . Rephrased using the notation of rule  $(N_v)$ , it is required for  $\overrightarrow{nask}$  that  $p_1 < m_1 \wedge \dots \wedge p_n < m_n$  whereas  $\overrightarrow{wnask}$  only requires that  $p_1 < m_1 \vee \dots \vee p_n < m_n$ . In view of that, it is easy to verify that  $\overrightarrow{wnask}(t_1(m_1), \dots, t_n(m_n))$  can be encoded as follows:  $\overrightarrow{nask}(t_1(m_1)) + \dots + \overrightarrow{nask}(t_n(m_n))$  where  $+$  denotes the non-deterministic choice. As a result, although useful later,  $\overrightarrow{wnask}$  does not bring an increase of expressiveness.

### C. MRT

The last language we shall consider is a Gamma-like language [1, 2], based on the chemical reaction metaphor. It considers communication primitives as the rewriting of pre-condition multi-sets into post-condition multi-sets. Intuitively, the operational effect of a multi-set rewriting ( $pre$ ,  $post$ ) consists in inserting all the positive post-conditions, and in deleting all the negative post-conditions from the current store  $\sigma$ , provided that  $\sigma$  contains all positive pre-conditions and does not meet any of the negative pre-conditions. Formally, these rewritings are specified as follows.

**Definition 5.** Define the set of multi-set rewriting primitives  $\mathcal{T}_{MR}$  as the set of primitives  $T_{MR}$  generated by the following grammar:

$$\begin{aligned} T_{MR} &::= (\{M\}, \{M\}) \\ M &::= \lambda \mid +t \mid -t \mid M, M \end{aligned}$$

where  $\lambda$  indicates an empty multi-set and where  $t$  denotes a token.

$$(CM) \quad \frac{pre^+ \subseteq \sigma, pre^- \perp \sigma, \quad \sigma' = (\sigma \setminus post^-) \cup post^+}{\langle (pre, post) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma' \rangle}$$

**Fig. 5.** Transition rules for multi-set rewriting-based primitives (MRT)

It is worth observing that not all pairs of preconditions and postconditions correspond to reasonable computations. Indeed, as stated above, it is possible to require in a precondition that the same token is present and absent or to require in the postcondition the removal of a token which has not been tested for presence in the precondition. We subsequently define such reasonable pairs of pre- and post-conditions as respectively consistent and valid. To that end, we first introduce some notations.

**Definition 6.** Given a multi-set rewriting pair  $(Pre, Post)$ , denote by  $Pre^+$  the multi-set  $\{t \mid +t \in Pre\}$  of tokens positively appearing in the precondition and by  $Pre^-$  the multi-set  $\{t \mid -t \in Pre\}$  negatively appearing in it. Similarly, we shall denote by  $Post^+$  and  $Post^-$  the multiset of tokens appearing positively and negatively in the postcondition.

A multi-set rewriting pair  $(Pre, Post)$  is said to be consistent if  $Pre^+ \cap Pre^- = \emptyset$ . It is said to be valid if  $Post^- \subseteq Pre^+$ .

A consequence of consistency and validity is that four basic pairs of pre- and post-conditions can be put forward:  $(\{+t\}, \{\})$ ,  $(\{-t\}, \{\})$ ,  $(\{\}, \{+t\})$ ,  $(\{+t\}, \{-t\})$ . They correspond respectively to the *ask*( $t$ ), *nask*( $t$ ), *tell*( $t$ ) and *get*( $t$ ) of the BachT language.

It turns out that it is possible to define it by one rule. To express it, an auxiliary notion is however needed. It extends the notations of Definition 6 to capture the fact that, for each token, the tokens mentioned negatively in the definition are not with their multiplicity on the current store  $\sigma$ .

**Definition 7.** For any token  $t$ , define  $Pre^-[t]$  as the multiset of negatively marked tokens  $t$  in the precondition  $Pre$ :

$$Pre^-[t] = \{t : -t \in Pre^-\}.$$

Given a precondition  $Pre$  and a store  $\sigma$ , we then define the non element-wise inclusion operator  $\perp$  as follows:

$$Pre^- \perp \sigma \text{ iff } Pre^-[t] \not\subseteq \sigma, \text{ for any token } t.$$

With this notation, rule (CM) of Fig. 5 states that a multi-set rewriting  $(Pre, Post)$  can be executed in a store  $\sigma$  if the multi-set  $Pre^+$  is included in  $\sigma$  and if no negative pre-condition occurs with the required multiplicity in  $\sigma$ . Under these conditions, the effect of the rewriting is to delete from  $\sigma$  all the negative post-conditions and to add to  $\sigma$  all the positive post-conditions.

## 2.2 Languages

We are now in a position to formally define the languages we shall consider in the paper. The statements of these languages, also called *agents*, are defined from the tell, ask, get and nask primitives by possibly combining them by the classical non-deterministic choice operator  $+$ , parallel operator (denoted by the  $\parallel$  symbol) and the sequential operator (denoted by the  $;$  symbol). The formal definition is as follows.

**Definition 8.** *Define the BachT language  $\mathcal{L}_B$  as the set of agents  $A$  generated by the following grammar:*

$$A ::= T_b \mid A ; A \mid A \parallel A \mid A + A$$

where  $T_b$  represents a token-based primitive. Define similarly the Dense BachT language  $\mathcal{L}_{DB}$ , the VD-BachT language  $\mathcal{L}_{VB}$ , the MRT language  $\mathcal{L}_{MR}$  by taking instead of the token-based primitive  $T_b$ , respectively the dense token-based primitives  $T_{db}$ , the list of token-based primitive  $T_{vb}$  and the multi-set rewriting primitive  $T_{MR}$ .

Moreover, subsequently, we shall consider sublanguages formed similarly but by considering only subsets of these primitives. In that case, if  $\mathcal{H}$  denotes such a subset, then we shall write the induced sublanguages as  $\mathcal{L}_B(\mathcal{H})$ ,  $\mathcal{L}_{DB}(\mathcal{H})$ ,  $\mathcal{L}_{VB}(\mathcal{H})$  and  $\mathcal{L}_{MR}(\mathcal{H})$  respectively. Note that for the latter sublanguages, the tell, ask, nask and get primitives are associated with the basic pairs described above.

## 2.3 Transition System

To study the expressiveness of the languages, a semantics needs to be defined. As suggested in the previous subsections, we shall use an operational one, based on transition systems. For each transition system, the configuration consists of agents (summarizing the current state of the agents running on the store) and a multi-set of tokens (denoting the current state of the store). In order to express the termination of the computation of an agent, we extend the set of agents by adding a special terminating symbol  $E$  that can be seen as a completely computed agent. For uniformity purpose, we abuse the language by qualifying  $E$  as an agent. To meet the intuition, we shall always rewrite agents of the form  $(E; A)$ ,  $(E \parallel A)$  and  $(A \parallel E)$  as  $A$ . This is technically achieved by defining the extended sets of agents as  $\mathcal{L}_B \cup \{E\}$ ,  $\mathcal{L}_{DB} \cup \{E\}$ ,  $\mathcal{L}_{VB} \cup \{E\}$  or  $\mathcal{L}_{MR} \cup \{E\}$  and by justifying the simplifications by imposing a bimonoid structure.

The rules for the primitives of the languages have been given in Figs. 1, 2, 3, 4 and 5. Figure 6 details the usual rules for sequential composition, parallel composition, interpreted in an interleaving fashion, and non-deterministic choice.

## 2.4 Observables and Operational Semantics

We are now in a position to define what we want to observe from the computations. Following previous work by some of the authors (see e.g. [10, 11, 24–26]),

$$\begin{array}{l}
\text{(S)} \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A ; B \mid \sigma \rangle \longrightarrow \langle A' ; B \mid \sigma' \rangle} \\
\text{(P)} \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A \parallel B \mid \sigma \rangle \longrightarrow \langle A' \parallel B \mid \sigma' \rangle} \\
\qquad \qquad \frac{\langle B \mid \sigma \rangle \longrightarrow \langle B' \mid \sigma' \rangle}{\langle B \parallel A \mid \sigma \rangle \longrightarrow \langle B \parallel A' \mid \sigma' \rangle} \\
\text{(C)} \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A + B \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle} \\
\qquad \qquad \frac{\langle B \mid \sigma \rangle \longrightarrow \langle B' \mid \sigma' \rangle}{\langle B + A \mid \sigma \rangle \longrightarrow \langle B' \mid \sigma' \rangle}
\end{array}$$

**Fig. 6.** Transition rules for the operators

we shall actually take an operational semantics recording the final state of the computations, this being understood as the final store coupled to a mark indicating whether the considered computation is successful or not. Such marks are respectively denoted as  $\delta^+$  (for the successful computations) and  $\delta^-$  (for failed computations).

### Definition 9.

1. Define the set of stores  $Sstore$  as the set of finite multisets with elements from  $Stoken$ .
2. Let  $\delta^+$  and  $\delta^-$  be two fresh symbols denoting respectively success and failure. Define the set of histories  $Shist$  as the cartesian product  $Sstore \times \{\delta^+, \delta^-\}$ .
3. For each language  $\mathcal{L}_I$  of the languages  $\mathcal{L}_B$ ,  $\mathcal{L}_{DB}$ ,  $\mathcal{L}_{VB}$ ,  $\mathcal{L}_{MR}$ , define the operational semantics  $\mathcal{O}_I : \mathcal{L}_I \rightarrow \mathcal{P}(Shist)$  as the following function: for any agent  $A \in \mathcal{L}$

$$\begin{aligned}
\mathcal{O}(A) = & \{(\sigma, \delta^+) : \langle A \mid \emptyset \rangle \rightarrow^* \langle E \mid \sigma \rangle\} \\
& \cup \{(\sigma, \delta^-) : \langle A \mid \emptyset \rangle \rightarrow^* \langle B \mid \sigma \rangle \nrightarrow, B \neq E\}
\end{aligned}$$

## 3 Applications

To evidence the interest of Dense BachT and VD-BachT, let us turn to some applications and see how they easily allow for their encodings.

### 3.1 A Simple Taxi Application

A typical service application inspired by Uber consists in a system allowing to select taxi drivers based on their reputation. To be operational, such a system needs on the one hand to allow users to express their satisfaction with regard to the service provided, and on the other hand, to test that a taxi driver is recognized at a sufficient level of satisfaction. For illustration purposes, we will assume that only positive marks are taken into account and that the service offered by a taxi driver can be evaluated as good or excellent, corresponding to a respective evaluation with numbers 1 and 2. We will then imagine that a level of satisfaction 100 is a minimal satisfaction mark for a reasonable driver.

Using Dense BachT for the first task, the satisfaction of a user can be registered by inserting the token `taxi_driver_id` once if the evaluation mark is good and twice if it is excellent. Technically, with `taxi_driver_id` being the identifier of the taxi driver, this amounts to respectively executing `tell(taxi_driver_id(1))` or `tell(taxi_driver_id(2))`. As regards the second task, making sure that a proposed driver, say identified by `id`, has reached a level of satisfaction of at least 100, can be simulated by executing the primitive `ask(id(100))`. Note that, as the number of matching tuples is only counted, such a satisfaction level may be reached thanks to the contribution of many users. Of course, different policies can be implemented in the application, for instance to forbid a user to mark a taxi driver more than once a day. It is also worth noting that thanks to the space and time decoupling between information producers and information consumers offered by coordination languages, it is very easy to introduce new users and new taxi drivers in the application.

### 3.2 The Dining Philosophers

Formulated by Edsger Dijkstra in 1965, the dining philosophers is a classical concurrency problem addressing the synchronisation of processes sharing resources. It is formulated as follows:  $N$  philosophers spend their time thinking and eating. To eat, they must sit on a round table in front of a dish and take the forks on their left and right sides. There is however only one fork between two dishes, which makes it impossible for all the philosophers to eat simultaneously.

The classical solution is to use semaphores, one for each fork and one to let only enter to the table a number of philosophers. Other solutions have been proposed by the coordination community, for instance, using Respect [29]. The Vector Dense BachT language proposes a very simple solution by associating each fork with a token and by simulating each philosopher taking his two forks by a `get` primitive on these tokens and dually each philosopher realising them by means of a `tell` primitive. For  $N = 5$ , the philosophers may then be coded as follows:

$$\begin{aligned} Phil_0 &= get(f_0, f_1); tell(f_0, f_1); Phil_0 \\ Phil_1 &= get(f_1, f_2); tell(f_1, f_2); Phil_1 \\ &\dots \\ Phil_4 &= get(f_4, f_0); tell(f_4, f_0); Phil_4 \end{aligned}$$

with the whole set of philosophers simulated by

$$Phil_0 \parallel Phil_1 \parallel Phil_2 \parallel Phil_3 \parallel Phil_4$$

### 3.3 An Online Shopping System

Let us now consider an online shopping system related to an European sporting goods store, present in five different European cities: Brussels, Paris, London, Berlin and Rome. All these shops propose the same articles. In order to manage

efficiently the number of orders that arrive through the online system, these are distributed on the different shops present in the five cities. Assume that a group of 50 orders arrive and has to be distributed equally between the different shops. This can be simulated through the execution of the following tell primitive:

```
tell(Brussels(10), Paris(10), London(10), Berlin(10), Rome(10)).
```

Assume now that the following maxima of orders to be processed have been imposed for the shops: 200 orders for Brussels, 75 for Paris, 50 for London, 150 for Berlin and 70 for Rome. A check whether these maxima have not been reached can be simulated by executing the following nask primitive:

```
nask(Brussels(200), Paris(75), London(50), Berlin(150), Rome(70)).
```

### 3.4 Distributed Density

In the online shopping problem, the arrival of 50 orders has been explicitly distributed on the shops. A natural extension is to let the execution of the primitive non-deterministically choose the distribution. We are then lead to consider a list of tokens together with a density and to distribute it on the tokens. The following definition formalizes such an association.

**Definition 10.** *Let  $S_{nlt}$  denote the set of non-empty lists of tokens in which, for simplicity purposes, each token differs from the others. Such a list is typically denoted as  $L = [t_1, \dots, t_p]$  and is thus such that  $t_i \neq t_j$  for  $i \neq j$ . Define a dense list of tokens as a list of  $S_{nlt}$  associated with a strictly positive integer. Such a dense list is typically represented as  $L(m)$ , with  $L$  the list of tokens and  $m$  an integer.*

The distribution of the density over a list of tokens is formalized through the following distribution function.

**Definition 11.** *Define the distribution of tokens from dense lists of tokens to sets of tuples of dense tokens as follows:*

$$Di([t_1, \dots, t_p](m)) = \{(t_1(m_1), \dots, t_p(m_p)) : m_1 + \dots + m_p = m\}$$

*Note that, thanks to the definition of dense tokens, we assume above that the  $m_i$ 's are positive integers. For the sake of simplicity, we shall call the set  $Di([t_1, \dots, t_p](m))$  the distribution of  $m$  over  $[t_1, \dots, t_p]$ .*

The distribution of an integer  $m$  over a list of tokens  $L$  has the potential to express the behavior of the BachT primitives extended with dense lists of tokens as arguments. Indeed, telling a dense list amounts to telling atomically the  $t_i[m_i]$ 's of a tuple defined above. Asking or getting a dense list requires to check that a tuple of  $Di([t_1, \dots, t_p](m))$  is present on the considered store. For the negative ask, the requirement is that none of the tuple is present. For the ease of writing and to make this latter concept clear, we introduce the following concept of intersection.

$$\begin{aligned}
(\mathbf{T}_{dbd}) \quad & \frac{(t_1(m_1), \dots, t_p(m_p)) \in \mathcal{D}i(L(m))}{\langle \text{tell}(L(m)) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \cup \{t_1(m_1), \dots, t_p(m_p)\} \rangle} \\
(\mathbf{A}_{dbd}) \quad & \frac{\mathcal{D}i(L(m)) \cap \sigma \neq \emptyset}{\langle \text{ask}(L(m)) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \rangle} \\
(\mathbf{G}_{dbd}) \quad & \frac{(t_1(m_1), \dots, t_p(m_p)) \in \mathcal{D}i(L(m))}{\langle \text{get}(L(m)) \mid \sigma \cup \{t_1(m_1), \dots, t_p(m_p)\} \rangle \longrightarrow \langle E \mid \sigma \rangle} \\
(\mathbf{N}_{dbd}) \quad & \frac{m > 0 \text{ and } \mathcal{D}i(L(m)) \cap \sigma = \emptyset}{\langle \text{nask}(L(m)) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \rangle}
\end{aligned}$$

**Fig. 7.** Transition rules for list of token-based primitives (Dense BachT with distributed Density)

**Definition 12.** Let  $m$  be a positive integer,  $L = [t_1, \dots, t_p]$  be a list of tokens and  $\sigma$  a store. We define  $\mathcal{D}i(L(m)) \cap \sigma$  as the following set of tuples of dense tokens:

$$\mathcal{D}i(L(m)) \cap \sigma = \{(t_1(m_1), \dots, t_p(m_p)) \in \mathcal{D}i(L(m)) : \{t_i(m_i)\} \subseteq \sigma\}$$

We are now in a position to specify the language extension handling dense lists of tokens.

**Definition 13.** Define the set of dense lists primitives  $\mathcal{T}_{dbd}$  as the set of primitives  $T_{dbd}$  generated by the following grammar:

$$T_{dbd} ::= \text{tell}(L(m)) \mid \text{ask}(L(m)) \mid \text{get}(L(m)) \mid \text{nask}(L(m))$$

where  $L(m)$  represents a dense list of tokens.

The transition steps for these primitives are defined in Fig. 7. As suggested above, rule  $(\mathbf{T}_{dbd})$  specifies that telling a dense list  $L(m)$  of tokens amounts to atomically adding the multiple occurrences  $t_i(m_i)$ 's of the tokens of a tuple of the distribution of  $m$  over  $L$ . Note that the selected tuple is chosen non-deterministically, which gives to a tell primitive a non-deterministic behavior as opposed to the tell primitives of BachT and Vectorized Dense BachT. Rule  $(\mathbf{A}_{dbd})$  states that asking for the dense list  $L(m)$  amounts to testing that a tuple of the distribution of  $m$  over  $L$  is in the store, which is technically stated through the non-emptiness of the intersection of the distribution and the store. Rule  $(\mathbf{G}_{dbd})$  requires that the tokens of the tuples are removed in the considered multiplicity. Finally, rule  $(\mathbf{N}_{dbd})$  specifies that negatively asking  $L(m)$  succeeds if  $m$  is strictly positive and no tuple of the distribution of  $m$  over  $L$  is present on the current store.

We are now in a position to define the language Dense BachT with a Distribution of the density over a list of tokens by considering the statements of this language as defined from the tell, ask, get and nask primitives possibly combined by the non deterministic choice, parallel and sequential operators.

A further extension consists in equipping the tokens of a dense list with minimal and maximal numbers, as follows.

**Definition 14.** Define the association of a token and two positive integers of  $\mathbb{N}$  as a capacity dense token. Such a token is typically denoted as  $t(m, n)$  where  $t$  is the token and  $m, n$  are the integers.

**Definition 15.** Let  $\text{Snlct}$  denote the set of non-empty lists of capacity dense tokens in which, for simplicity purposes, each token differs from the others. Such a list is typically denoted as  $L = [t_1(m_1, n_1), \dots, t_p(m_p, n_p)]$  and is thus such that  $t_i \neq t_j$  for  $i \neq j$ . Define a dense list of capacity dense tokens as a list of  $\text{Snlct}$  associated with a strictly positive integer. Such a list is typically represented as  $L(m)$ , with  $L$  the list of capacity dense tokens and  $m$  an integer.

The expected extended language is simply obtained by slightly modifying the notion of distribution introduced in Definition 11.

**Definition 16.** Define the cardinality based distribution of tokens from dense lists of capacity tokens to sets of tuples of extended dense tokens as follows:

$$\begin{aligned} \mathcal{Dc}([t_1(m_1, n_1), \dots, t_p(m_p, n_p)](q)) \\ = \{(t_1(q_1), \dots, t_p(q_p)) : q_1 + \dots + q_p = q \text{ and } m_i \leq q_i \leq n_i \text{ for } i \in \{1, \dots, p\}\} \end{aligned}$$

Note that nothing guarantees that the above set is non empty. We shall subsequently called *coherent* those dense lists of capacity based tokens such that their cardinality based distribution is non empty and restrict ourselves to such coherent dense lists in the following.

To conclude this section, it is worth observing that it is straightforward to translate the positive version of DBD-BachT and its cardinality extension in terms of VD-BachT.

Indeed, as easily observed, one can code the tell, ask and get primitives of DBD-BachT as follows:

$$\begin{aligned} \text{tell}(L(m)) &= \sum_{\mathbf{v} \in \mathcal{Di}(L(m))} \text{tell}(\mathbf{v}^r) \\ \text{ask}(L(m)) &= \sum_{\mathbf{v} \in \mathcal{Di}(L(m))} \text{ask}(\mathbf{v}^r) \\ \text{get}(L(m)) &= \sum_{\mathbf{v} \in \mathcal{Di}(L(m))} \text{get}(\mathbf{v}^r) \end{aligned}$$

where  $\mathbf{v}^r$  denotes the vector  $\mathbf{v}$  restricted to its strictly positive dense tokens.

Translating the nask primitive is slightly more complicated in requiring the parallel composition of the weak form of nask of vectors:

$$\text{nask}(L(m)) = \parallel_{\mathbf{v} \in \mathcal{Di}(L(m))} \text{wnask}(\mathbf{v}^r)$$

Translating the DBD-BachT language with cardinality proceeds similarly by using  $\mathcal{Dc}(L(m))$  instead of  $\mathcal{Di}(L(m))$ .

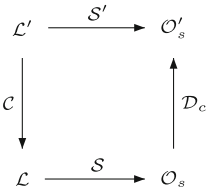


## 4 Expressiveness Study

The translation just introduced evidences the need for a study of the expressiveness power of the languages introduced in Sect. 2, to which we now turn.

### 4.1 On the Expressiveness of Languages

A natural way to compare the expressive power of two languages is to determine whether all programs written in one language can be easily and equivalently translated into the other language, where equivalent is intended in the sense of conserving the same observable behaviors.



**Fig. 8.** Basic embedding.

According to this intuition, Shapiro introduced in [30] a first notion of embedding as follows. Consider two languages  $\mathcal{L}$  and  $\mathcal{L}'$ . Assume given the semantics mappings (*Observation criteria*)  $S : \mathcal{L} \rightarrow \mathcal{O}_s$  and  $S' : \mathcal{L}' \rightarrow \mathcal{O}'_s$ , where  $\mathcal{O}_s$  and  $\mathcal{O}'_s$  are on some suitable domains. Then  $\mathcal{L}$  can *embed*  $\mathcal{L}'$  if there exists a mapping  $\mathcal{C}$  (coder) from the statements of  $\mathcal{L}'$  to the statements of  $\mathcal{L}$ , and a mapping  $\mathcal{D}_c$  (decoder) from  $\mathcal{O}_s$  to  $\mathcal{O}'_s$ , such that the diagram of Fig. 8 commutes,

namely such that for every statement  $A \in \mathcal{L}' : \mathcal{D}_c(S(\mathcal{C}(A))) = S'(A)$ .

This basic notion of embedding turns out however to be too weak since, for instance, the above equation is satisfied by any pair of Turing-complete languages. De Boer and Palamidessi hence proposed in [17] to add three constraints on the coder  $\mathcal{C}$  and on the decoder  $\mathcal{D}_c$  in order to obtain a notion of *modular* embedding usable for concurrent languages:

1.  $\mathcal{D}_c$  should be defined in an element-wise way with respect to  $\mathcal{O}_s$ , namely for some appropriate mapping  $\mathcal{D}_{el}$

$$\forall X \in \mathcal{O}_s : \mathcal{D}_c(X) = \{\mathcal{D}_{el}(x) \mid x \in X\} \quad (P_1)$$

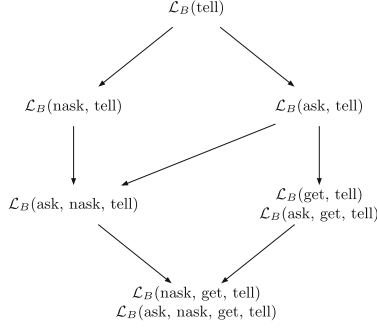
2. the coder  $\mathcal{C}$  should be defined in a compositional way with respect to the sequential, parallel and choice operators:

$$\begin{aligned}
 \mathcal{C}(A ; B) &= \mathcal{C}(A) ; \mathcal{C}(B) \\
 \mathcal{C}(A \parallel B) &= \mathcal{C}(A) \parallel \mathcal{C}(B) \\
 \mathcal{C}(A + B) &= \mathcal{C}(A) + \mathcal{C}(B)
 \end{aligned} \quad (P_2)$$

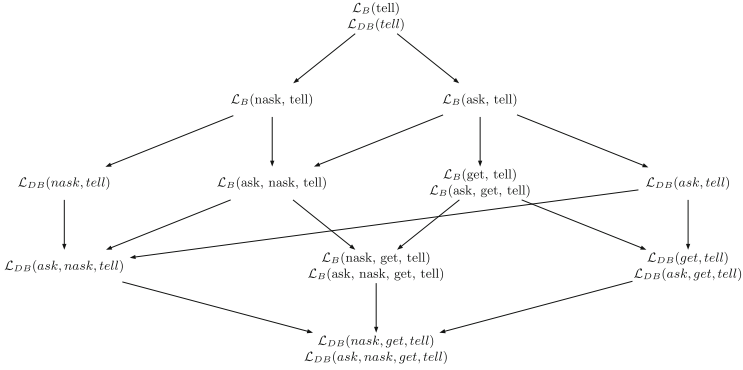
3. the embedding should preserve the behavior of the original processes with respect to deadlock, failure and success (*termination invariance*):

$$\forall X \in \mathcal{O}_s, \forall x \in X : tm'(\mathcal{D}_{el}(x)) = tm(x) \quad (P_3)$$

where  $tm$  and  $tm'$  extract the termination information from the observables of  $\mathcal{L}$  and  $\mathcal{L}'$ , respectively.



**Fig. 9.** Embedding hierarchy of BachT Languages.



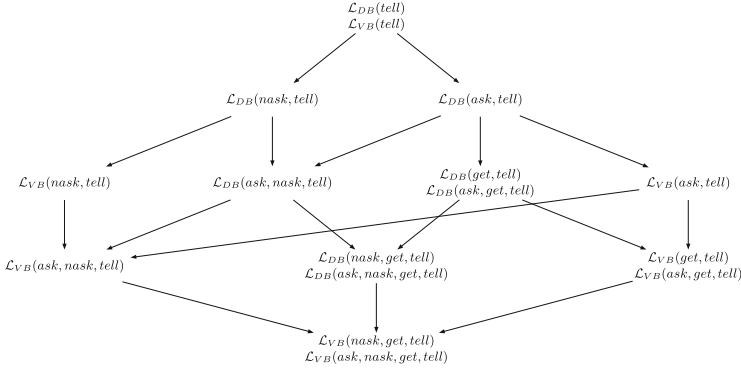
**Fig. 10.** Embedding hierarchy of BachT and Dense BachT

An embedding is then called *modular* if it satisfies properties  $P_1$ ,  $P_2$ , and  $P_3$ . The existence of a modular embedding from  $\mathcal{L}'$  into  $\mathcal{L}$  is subsequently denoted by  $\mathcal{L}' \leq \mathcal{L}$ . It is easy to prove that  $\leq$  is a pre-order relation. Moreover if  $\mathcal{L}' \subseteq \mathcal{L}$  then  $\mathcal{L}' \leq \mathcal{L}$  that is, any language embeds all its sublanguages. This property descends immediately from the definition of embedding, by setting  $\mathcal{C}$  and  $\mathcal{D}_c$  equal to the identity function.

### 4.2 Comparing BachT, Dense BachT and Vectorized Dense BachT

The expressive power of the different sublanguages of BachT has been studied in [9–11] from which the expressiveness hierarchy of Fig. 9 can be established. Building upon these results, the article [22] has established the embedding relations of Fig. 10.

In both figures, an arrow from a language  $\mathcal{L}_1$  to a language  $\mathcal{L}_2$  means that  $\mathcal{L}_2$  embeds  $\mathcal{L}_1$ , that is  $\mathcal{L}_1 \leq \mathcal{L}_2$ . When an arrow from  $\mathcal{L}_1$  to  $\mathcal{L}_2$  has no counterpart from  $\mathcal{L}_2$  to  $\mathcal{L}_1$ , then  $\mathcal{L}_1$  is strictly less expressive than  $\mathcal{L}_2$ , that is  $\mathcal{L}_1 < \mathcal{L}_2$ . If  $\mathcal{L}_1 \leq \mathcal{L}_2$  and  $\mathcal{L}_2 \leq \mathcal{L}_1$  then  $\mathcal{L}_1$  and  $\mathcal{L}_2$  are equivalent, that is  $\mathcal{L}_1 = \mathcal{L}_2$ . In that



**Fig. 11.** Embedding hierarchy of Dense BachT and Vectorized Dense BachT languages.

case, they are depicted together. If  $\mathcal{L}_1 \not\leq \mathcal{L}_2$  and  $\mathcal{L}_2 \not\leq \mathcal{L}_1$  then  $\mathcal{L}_1$  and  $\mathcal{L}_2$  are not comparable with each other. This is subsequently denoted by  $\mathcal{L}_1 \dot{\setminus} \mathcal{L}_2$ . Thanks to the transitivity, both figures contain only a minimal amount of arrows. Apart from these induced relations, no other relation holds.

It is worth noting that the hierarchy relations presented in Fig. 9 appear in the center of Fig. 10. This reflects the fact that BachT is a special case of Dense BachT. Moreover, the hierarchy of the Dense BachT sublanguages resembles that of the BachT sublanguages. This intuitively results from the very nature of the ask, nask and get primitives, which are not altered by the density of tokens. Nevertheless, except for the sublanguage reduced to a tell primitive, it is worth observing that the dense sublanguages are strictly more expressive than their BachT counterparts. This highlights the fact that Dense BachT is an extension of BachT bringing more expressiveness.

Following the same reasonings as those published in [16] it is possible to establish similar embedding relations between Dense BachT and Vectorized BachT. Due to space limits, the proof are not reproduced here but the results are depicted in Fig. 11. To get a complete expressiveness picture, it remains to study the expressiveness of Vectorized Dense BachT and MRT. This is the purpose of the next subsection. Due to space limits, the key points are only given, the interested reader being referred to [15] where all the proofs are conducted in details.

### 4.3 Relating Vectorized Dense BachT and MRT

As a first observation, it is easy to establish that the VD-BachT sublanguages are embedded in the corresponding MRT sublanguages.

**Proposition 1.**  $\mathcal{L}_{VB}(\chi) \leq \mathcal{L}_{MR}(\chi)$ , for any subset of  $\chi$  of primitives.

*Proof.* Immediate by defining the coder as follows:

$$\begin{aligned}
\mathcal{C}(\text{tell}((t_1(m_1), \dots, t_k(m_k)))) &= (\{\}, \underbrace{\{+t_1, \dots, +t_1\}}_{m_1 \text{ times}}, \dots, \underbrace{\{+t_k, \dots, +t_k\}}_{m_k \text{ times}}) \\
\mathcal{C}(\text{ask}((t_1(m_1), \dots, t_k(m_k)))) &= (\underbrace{\{+t_1, \dots, +t_1\}}_{m_1 \text{ times}}, \dots, \underbrace{\{+t_k, \dots, +t_k\}}_{m_k \text{ times}}, \{\}) \\
\mathcal{C}(\text{get}((t_1(m_1), \dots, t_k(m_k)))) &= (\underbrace{\{+t_1, \dots, +t_1\}}_{m_1 \text{ times}}, \dots, \underbrace{\{+t_k, \dots, +t_k\}}_{m_k \text{ times}}, \\
&\quad \underbrace{\{-t_1, \dots, -t_1\}}_{m_1 \text{ times}}, \dots, \underbrace{\{-t_k, \dots, -t_k\}}_{m_k \text{ times}}) \\
\mathcal{C}(\text{nask}((t_1(m_1), \dots, t_k(m_k)))) &= (\underbrace{\{-t_1, \dots, -t_1\}}_{m_1 \text{ times}}, \dots, \underbrace{\{-t_k, \dots, -t_k\}}_{m_k \text{ times}}, \{\})
\end{aligned}$$

and using the identity as decoder.

As for the results mentioned before, we shall only consider non trivial sublanguages, namely sublanguages containing at least the tell primitive. The store being feeded with tokens, the second step is to provide the sublanguage with a possibility to question the store about the presence or the absence of tokens on it. Those two capacities result from the introduction of the ask and nask primitives. A third important property is then to allow the language to retrieve tokens from the store, by using the get primitive. Finally the last step studies the most complete language, combining the get and tell primitives with the nask and/or ask primitives.

### A. Adding Tokens on the Store

When only constituted by the tell primitive the sublanguages are equivalent, namely  $\mathcal{L}_{VB}(\text{tell})$  and  $\mathcal{L}_{MR}(\text{tell})$  are equivalent.

**Proposition 2.**  $\mathcal{L}_{MR}(\text{tell})$  and  $\mathcal{L}_{VB}(\text{tell})$  are equivalent.

*Proof.* We have  $\mathcal{L}_{VB}(\text{tell}) \leq \mathcal{L}_{MR}(\text{tell})$  by Proposition 1. Furthermore,  $\mathcal{L}_{MR}(\text{tell}) \leq \mathcal{L}_{VB}(\text{tell})$  is established by coding any tell primitive of  $\mathcal{L}_{MR}(\text{tell})$  as the composition of their dense versions:  $\mathcal{C}(\{\}, \underbrace{\{+t_1, \dots, +t_1\}}_{m_1 \text{ times}}, \dots, \underbrace{\{+t_k, \dots, +t_k\}}_{m_k \text{ times}}) = \text{tell}((t_1(m_1), \dots, t_k(m_k)))$ .

### B. Checking for Presence and/or Absence When Adding Tokens

In contrast to what is obtained in the comparison of Dense BachT and Vectorized BachT languages,  $\mathcal{L}_{VB}(\text{ask}, \text{tell})$  is as expressive as  $\mathcal{L}_{MR}(\text{ask}, \text{tell})$ .

**Proposition 3.**  $\mathcal{L}_{VB}(\text{ask}, \text{tell}) = \mathcal{L}_{MR}(\text{ask}, \text{tell})$

*Proof.* (i) On the one hand,  $\mathcal{L}_{VB}(\text{ask}, \text{tell}) \leq \mathcal{L}_{MR}(\text{ask}, \text{tell})$ , by Proposition 1. (ii) On the other hand,  $\mathcal{L}_{MR}(\text{ask}, \text{tell}) \leq \mathcal{L}_{VB}(\text{ask}, \text{tell})$  is established by noting that any agent of  $\mathcal{L}_{MR}(\text{ask}, \text{tell})$  can be simulated by an agent of  $\mathcal{L}_{MR}(\text{ask})$  followed by an agent of  $\mathcal{L}_{MR}(\text{tell})$ .

In contrast,  $\mathcal{L}_{VB}(\text{nask}, \text{tell})$  is strictly less expressive than  $\mathcal{L}_{MR}(\text{nask}, \text{tell})$ .

**Proposition 4.**  $\mathcal{L}_{VB}(nask, tell) < \mathcal{L}_{MR}(nask, tell)$ .

*Proof.* (i) On the one hand,  $\mathcal{L}_{VB}(nask, tell) \leq \mathcal{L}_{MR}(nask, tell)$  holds by Proposition 1. (ii) On the other hand,  $\mathcal{L}_{MR}(nask, tell) \not\leq \mathcal{L}_{VB}(nask, tell)$  is proved by considering agent  $AB = (\{-a\}, \{+b\})$  and agent  $BA = (\{-b\}, \{+a\})$ , with  $\mathcal{O}(AB \parallel BA) = \{(\emptyset, \delta^-)\}$ . The proof proceeds by contradiction, by assuming the existence of a coder  $\mathcal{C}$  with  $\mathcal{C}(AB)$  in normal form [9], and thus written as  $tell(\vec{t}_1); A_1 + \dots + tell(\vec{t}_p); A_p + nask(\vec{u}_1); B_1 + \dots + nask(\vec{u}_q); B_q$ . In this expression we will establish that there is no alternative guarded by a  $tell(\vec{t}_i)$  operation and no alternative guarded by a  $nask(\vec{u}_j)$  operation either, which is impossible since  $\mathcal{C}(AB)$  must contain at least one primitive. We notice that the coding of  $\mathcal{C}(AB \parallel BA)$  can be written as  $\mathcal{C}(AB) \parallel \mathcal{C}(BA)$  by  $P_2$ .

Let us first establish that there is no alternative guarded by a  $tell(\vec{t}_i)$  operation. Indeed if there is an alternative guarded, say by  $tell(\vec{t}_i)$ , then  $D = \langle \mathcal{C}(AB \parallel BA) \mid \emptyset \rangle \rightarrow \langle (A_i \parallel \mathcal{C}(BA)) \mid \{\vec{t}_i\} \rangle$  is a valid computation prefix of  $\mathcal{C}(AB \parallel BA)$ . It should deadlock afterwards since  $\mathcal{O}(AB \parallel BA) = (\emptyset, \delta^-)$ . However  $D$  is also a valid computation prefix of  $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$ . Hence,  $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$  admits a failing computation which contradicts the fact that  $\mathcal{O}((AB \parallel BA) + (\{\}, \{+a\})) = (\{a\}, \delta^+)$ .

Secondly we establish that there is no alternative guarded by a  $nask(\vec{u}_j)$  operation. Indeed starting from the empty store, if there is an alternative guarded, say by  $nask(\vec{u}_j)$ , then  $D = \langle \mathcal{C}(AB \parallel BA) \mid \emptyset \rangle \rightarrow \langle (B_j \parallel \mathcal{C}(BA)) \mid \{\vec{t}_i\} \rangle$  is a valid computation prefix of  $\mathcal{C}(AB \parallel BA)$ . It should deadlock afterwards since  $\mathcal{O}(AB \parallel BA) = (\emptyset, \delta^-)$ . However  $D$  is also a valid computation prefix of  $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$ . Hence,  $\mathcal{C}((AB \parallel BA) + (\{\}, \{+a\}))$  admits a failing computation which contradicts the fact that  $\mathcal{O}((AB \parallel BA) + (\{\}, \{+a\})) = (\{a\}, \delta^+)$ .

$\mathcal{L}_{MR}(ask, tell)$  and  $\mathcal{L}_{VB}(nask, tell)$  are not comparable with each other, and so are  $\mathcal{L}_{MR}(nask, tell)$  and  $\mathcal{L}_{VB}(ask, tell)$ .

**Proposition 5.**  $\mathcal{L}_{MR}(ask, tell) \wr \mathcal{L}_{VB}(nask, tell)$

*Proof.* On the one hand, we have that  $\mathcal{L}_{MR}(ask, tell) \not\leq \mathcal{L}_{VB}(nask, tell)$ . Otherwise, by non embedding by transitivity, we have  $\mathcal{L}_{VB}(ask, tell) \leq \mathcal{L}_{VB}(nask, tell)$  which has been proved impossible (see Fig. 11). On the other hand,  $\mathcal{L}_{VB}(nask, tell) \not\leq \mathcal{L}_{MR}(ask, tell)$  is established by contradiction. Indeed, assuming the relation holds, we would then have  $\mathcal{L}_B(nask, tell) \leq \mathcal{L}_{MR}(ask, tell)$ , which has been proved impossible in [11].

**Proposition 6.**  $\mathcal{L}_{MR}(nask, tell) \wr \mathcal{L}_{VB}(ask, tell)$

*Proof.* On the one hand,  $\mathcal{L}_{MR}(nask, tell) \not\leq \mathcal{L}_{VB}(ask, tell)$  holds. Otherwise, by embedding by transitivity, we have  $\mathcal{L}_{VB}(nask, tell) \leq \mathcal{L}_{VB}(ask, tell)$  which is impossible (see Fig. 11). On the other hand,  $\mathcal{L}_{VB}(ask, tell) \not\leq \mathcal{L}_{MR}(nask, tell)$  is established by contradiction by considering  $tell(t(1)); ask(t(1))$  and by noting that  $\mathcal{O}(tell(t(1)); ask(t(1))) = (\{\{t(1)\}, \delta^+\})$ .

We now prove that  $\mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell})$  and  $\mathcal{L}_{MR}(\text{nask}, \text{tell})$  are not comparable with each other.

**Proposition 7.**  $\mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell}) \wr \mathcal{L}_{MR}(\text{nask}, \text{tell})$

*Proof.* (i) We have that  $\mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell}) \not\leq \mathcal{L}_{MR}(\text{nask}, \text{tell})$ . Otherwise, by embedding by transitivity,  $\mathcal{L}_{VB}(\text{ask}, \text{tell}) \leq \mathcal{L}_{MR}(\text{nask}, \text{tell})$ , which has been proved impossible in Proposition 6.

(ii) The proof of  $\mathcal{L}_{MR}(\text{nask}, \text{tell}) \not\leq \mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell})$  is an extension of the proof used in Proposition 4 with normal forms extended with *ask* primitives. It is established by considering agent AB = ( $\{-a\}, \{+b\}$ ) and agent BA = ( $\{-b\}, \{+a\}$ ), with  $\mathcal{O}((\{-a\}, \{+b\}) \parallel (\{-b\}, \{+a\})) = \{(\emptyset, \delta^-)\}$ .

Let us now prove that  $\mathcal{L}_{MR}(\text{ask}, \text{tell})$  is strictly less expressive than  $\mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell})$ .

**Proposition 8.**  $\mathcal{L}_{MR}(\text{ask}, \text{tell}) < \mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell})$

*Proof.* (i) On the one hand, thanks to Proposition 3,  $\mathcal{L}_{MR}(\text{ask}, \text{tell}) = \mathcal{L}_{VB}(\text{ask}, \text{tell}) \leq \mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell})$  and thus  $\mathcal{L}_{MR}(\text{ask}, \text{tell}) \leq \mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell})$ .

(ii) On the other hand,  $\mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell}) \not\leq \mathcal{L}_{MR}(\text{ask}, \text{tell})$ , since otherwise,  $\mathcal{L}_{VB}(\text{nask}, \text{tell}) \leq \mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell}) \leq \mathcal{L}_{MR}(\text{ask}, \text{tell})$ , which has been proved impossible in Proposition 5.

We now prove that  $\mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell})$  is strictly less expressive than  $\mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell})$ .

**Proposition 9.**  $\mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell}) < \mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell})$

*Proof.* (i) On the one hand, the fact that  $\mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell}) \leq \mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell})$  is immediate by Proposition 1. (ii) On the other hand,  $\mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell}) \not\leq \mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell})$ . Otherwise, by embedding by transitivity, from  $\mathcal{L}_{MR}(\text{nask}, \text{tell}) \leq \mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell})$ , one would get  $\mathcal{L}_{MR}(\text{nask}, \text{tell}) \leq \mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell})$ , which has been proved impossible in Proposition 7.

$\mathcal{L}_{VB}(\text{get}, \text{tell})$  is not comparable with  $\mathcal{L}_{MR}(\text{ask}, \text{tell})$  nor with  $\mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell})$ .

**Proposition 10.**  $\mathcal{L}_{VB}(\text{get}, \text{tell}) \wr \mathcal{L}_{MR}(\text{ask}, \text{tell})$

*Proof.* See [15].

**Proposition 11.**  $\mathcal{L}_{VB}(\text{get}, \text{tell}) \wr \mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell})$

*Proof.* On the one hand, for  $\mathcal{L}_{VB}(\text{get}, \text{tell}) \not\leq \mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell})$ , we refer the reader to [15]. On the other hand,  $\mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell}) \not\leq \mathcal{L}_{VB}(\text{get}, \text{tell})$ . Otherwise, by transitivity of the embedding, one would have that  $\mathcal{L}_{MR}(\text{ask}, \text{tell}) \leq \mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell}) \leq \mathcal{L}_{VB}(\text{get}, \text{tell})$  which has been proved impossible in Proposition 10.

$\mathcal{L}_{MR}(\text{ask}, \text{tell})$  can be proved to be not comparable with  $\mathcal{L}_{VB}(\text{nask}, \text{get}, \text{tell})$  nor is  $\mathcal{L}_{MR}(\text{ask}, \text{tell})$ .

**Proposition 12.**  $\mathcal{L}_{MR}(\text{ask}, \text{tell}) \wr \mathcal{L}_{VB}(\text{nask}, \text{get}, \text{tell}) \wr \mathcal{L}_{MR}(\text{nask}, \text{tell})$

*Proof.* See [15].

We are now in a position to establish that  $\mathcal{L}_{VB}(\text{get}, \text{tell})$  is not comparable with  $\mathcal{L}_{MR}(\text{nask}, \text{tell})$ .

**Proposition 13.**  $\mathcal{L}_{VB}(\text{get}, \text{tell}) \wr \mathcal{L}_{MR}(\text{nask}, \text{tell})$

*Proof.* On the one hand,  $\mathcal{L}_{VB}(\text{get}, \text{tell}) \not\leq \mathcal{L}_{MR}(\text{nask}, \text{tell})$ . Otherwise, as  $\mathcal{L}_{VB}(\text{ask}, \text{tell}) < \mathcal{L}_{VB}(\text{get}, \text{tell})$ , one would have  $\mathcal{L}_{VB}(\text{ask}, \text{tell}) \leq \mathcal{L}_{VB}(\text{get}, \text{tell}) \leq \mathcal{L}_{MR}(\text{nask}, \text{tell})$  which has been proved impossible in Proposition 6. On the other hand,  $\mathcal{L}_{MR}(\text{nask}, \text{tell}) \not\leq \mathcal{L}_{VB}(\text{get}, \text{tell})$ . Otherwise, we would have  $\mathcal{L}_{MR}(\text{nask}, \text{tell}) \leq \mathcal{L}_{VB}(\text{get}, \text{tell}) \leq \mathcal{L}_{VB}(\text{nask}, \text{get}, \text{tell})$  which has been proved impossible in Proposition 12.

$\mathcal{L}_{VB}(\text{nask}, \text{get}, \text{tell})$  is not comparable with  $\mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell})$ .

**Proposition 14.**  $\mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell}) \wr \mathcal{L}_{VB}(\text{nask}, \text{get}, \text{tell})$

*Proof.* (i) On the one hand  $\mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell}) \not\leq \mathcal{L}_{VB}(\text{nask}, \text{get}, \text{tell})$ . Otherwise,  $\mathcal{L}_{MR}(\text{ask}, \text{tell}) \leq \mathcal{L}_{VB}(\text{nask}, \text{get}, \text{tell})$  which contradicts Proposition 12. (ii) On the other hand,  $\mathcal{L}_{VB}(\text{nask}, \text{get}, \text{tell}) \not\leq \mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{tell})$ . By contradiction, consider  $\text{tell}(t(1)) ; \text{get}(t(1))$ .  $\mathcal{O}(\text{tell}(t(1)) ; \text{get}(t(1))) = \{(\emptyset, \delta^+)\}$ . Hence any computation of  $\mathcal{C}(\text{tell}(t(1))) ; \mathcal{C}(\text{get}(t(1)))$  is successful. Such a computation is composed of a computation for  $\mathcal{C}(\text{tell}(t(1)))$  followed by a computation for  $\mathcal{C}(\text{get}(t(1)))$ . As  $\mathcal{C}(\text{get}(t(1)))$  is composed of ask, nask, tell primitives which do not destroy elements on the store, the latter computation can be repeated step by step which yields successful computation for  $\mathcal{C}(\text{tell}(t(1))) ; (\mathcal{C}(\text{get}(t(1))) \parallel \mathcal{C}(\text{get}(t(1))))$ . However,  $\mathcal{O}(\text{tell}(t(1)) ; (\text{get}(t(1)) \parallel \text{get}(t(1)))) = \{(\emptyset, \delta^-)\}$ .

## C. Retrieving Tokens from the Store

**Proposition 15.**  $\mathcal{L}_{VB}(\text{get}, \text{tell}) < \mathcal{L}_{MR}(\text{get}, \text{tell})$

*Proof.* See [15].

We can now prove that  $\mathcal{L}_{MR}(\text{get}, \text{tell})$  is not comparable respectively with  $\mathcal{L}_{VB}(\text{nask}, \text{tell})$ ,  $\mathcal{L}_{VB}(\text{nask}, \text{get}, \text{tell})$  and  $\mathcal{L}_{VB}(\text{ask}, \text{nask}, \text{tell})$ .

**Proposition 16.**  $\mathcal{L}_{MR}(\text{get}, \text{tell}) \wr \mathcal{L}_{VB}(\text{nask}, \text{tell})$

*Proof.* On the one hand,  $\mathcal{L}_{MR}(\text{get}, \text{tell}) \not\leq \mathcal{L}_{VB}(\text{nask}, \text{tell})$ . Otherwise,  $\mathcal{L}_{MR}(\text{ask}, \text{tell}) \leq \mathcal{L}_{MR}(\text{nask}, \text{tell})$  which has been proved impossible in [11]. On the other hand,  $\mathcal{L}_{VB}(\text{nask}, \text{tell}) \not\leq \mathcal{L}_{MR}(\text{get}, \text{tell})$  is established by contradiction, by considering  $\text{tell}(t(1)) ; \text{nask}(t(1))$ . Indeed, one has  $\mathcal{O}(\text{tell}(t(1)) ; \text{nask}(t(1))) = \{\{\{t(1)\}, \delta^-\}$  whereas it is possible to establish that  $\mathcal{C}(\text{tell}(t(1))) ; \mathcal{C}(\text{nask}(t(1)))$  has a successful computation.

**Proposition 17.**  $\mathcal{L}_{MR}(get,tell) \wr \mathcal{L}_{VB}(nask,get,tell)$

*Proof.* On the one hand,  $\mathcal{L}_{MR}(get,tell) \not\leq \mathcal{L}_{VB}(nask,get,tell)$ . Otherwise, as  $\mathcal{L}_{MR}(ask,tell) \leq \mathcal{L}_{MR}(get,tell)$ , we then have  $\mathcal{L}_{MR}(ask,tell) \leq \mathcal{L}_{VB}(nask,get,tell)$  which has been proved impossible in Proposition 12. On the other hand,  $\mathcal{L}_{VB}(nask,get,tell) \not\leq \mathcal{L}_{MR}(get,tell)$ . Otherwise, we would have  $\mathcal{L}_{VB}(nask,tell) \leq \mathcal{L}_{MR}(get,tell)$  which has been proved impossible in Proposition 16.

**Proposition 18.**  $\mathcal{L}_{MR}(get,tell) \wr \mathcal{L}_{VB}(ask,nask,tell)$

*Proof.* On the one hand,  $\mathcal{L}_{MR}(get,tell) \not\leq \mathcal{L}_{VB}(ask,nask,tell)$ . Otherwise, one has  $\mathcal{L}_{MR}(ask,tell) \leq \mathcal{L}_{MR}(get,tell) \leq \mathcal{L}_{VB}(ask,nask,tell)$  which has been proved impossible in Proposition 8. On the other hand,  $\mathcal{L}_{VB}(ask,nask,tell) \not\leq \mathcal{L}_{MR}(get,tell)$ . Otherwise, one would have  $\mathcal{L}_{MR}(ask,tell) \leq \mathcal{L}_{MR}(ask,nask,tell) \leq \mathcal{L}_{VB}(get,tell)$  which has been proved impossible in Proposition 10.

#### D. Checking for Presence and/or Absence When Adding and/or Retrieving Tokens

We finally prove that  $\mathcal{L}_{VB}(ask,nask,get,tell)$  is strictly less expressive than  $\mathcal{L}_{MR}(ask,nask,get,tell)$ .

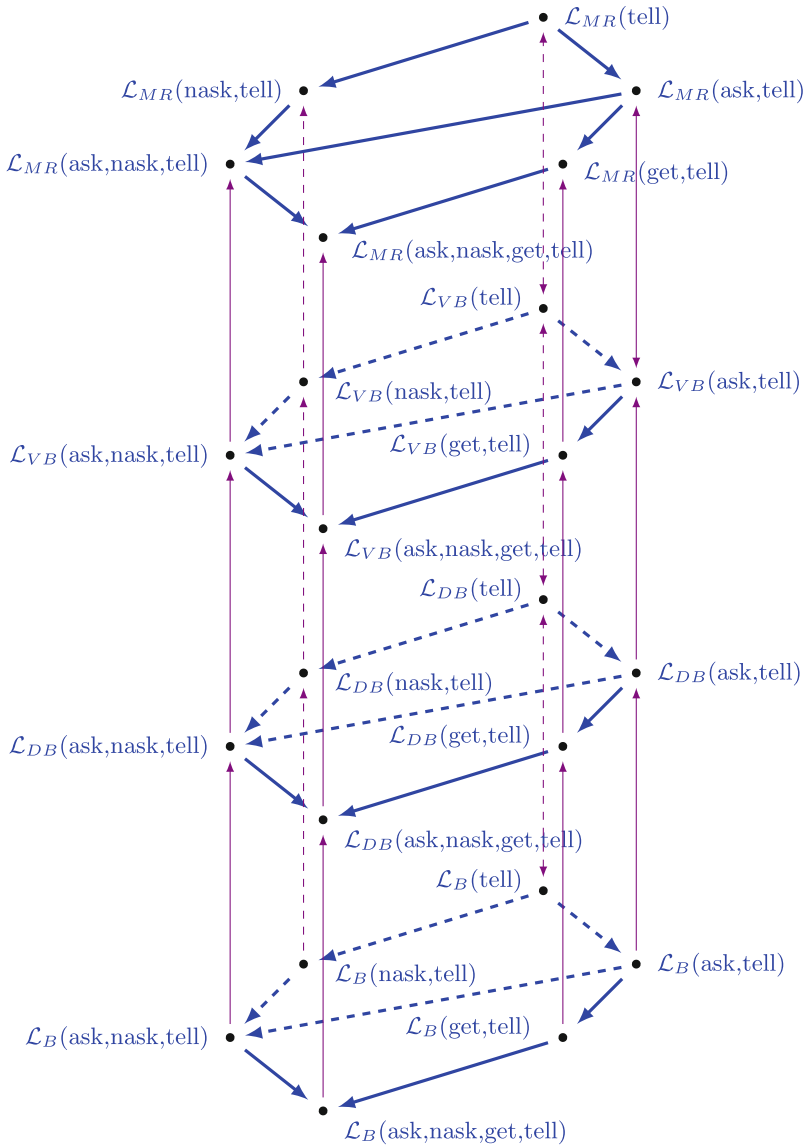
**Proposition 19.**  $\mathcal{L}_{VB}(ask,nask,get,tell) < \mathcal{L}_{MR}(ask,nask,get,tell)$

*Proof.* On the one hand,  $\mathcal{L}_{VB}(ask,nask,get,tell) \leq \mathcal{L}_{MR}(ask,nask,get,tell)$  is immediate by Proposition 1. On the other hand,  $\mathcal{L}_{MR}(ask,nask,get,tell) \not\leq \mathcal{L}_{VB}(ask,nask,get,tell)$  is established by contradiction. Indeed, assuming that  $\mathcal{L}_{MR}(ask,nask,get,tell) \leq \mathcal{L}_{VB}(ask,nask,get,tell)$ , as  $\mathcal{L}_{VB}(ask,nask,get,tell) = \mathcal{L}_{VB}(nask,get,tell)$ , one would have  $\mathcal{L}_{MR}(nask,tell) \leq \mathcal{L}_{MR}(ask,nask,get,tell) \leq \mathcal{L}_{VB}(ask,nask,get,tell) \leq \mathcal{L}_{VB}(nask,get,tell)$  which has been proved impossible in Proposition 12.

#### E. Summary

Figure 12 provides a summary of the expressiveness results developed in this paper in a three dimensional perspective. It is worth observing that the Vectorized Dense BachT language obeys the same hierarchy as the BachT, Dense BachT and MRT languages. This is due to the nature of the tell, ask, nask and get primitives, which is preserved by the extension provided to the tokens. It is also worth noting that the sublanguage reduced to the tell primitive has the same power in all the languages. The other sublanguages obey the expressiveness studies already developed for BachT and Dense BachT. Finally, the Vectorized Dense Bach sublanguages appear to be strictly more expressive than their Dense BachT counterparts but are strictly less expressive than their MRT counterparts.





**Fig. 12.** Three-dimensional representation of the expressiveness relations between the different languages.

The notable exception is provided by the  $\mathcal{L}_{VB}$ (ask,tell) sublanguage, which is as expressive as the  $\mathcal{L}_{MR}$ (ask,tell) sublanguage.

Note that, in the picture, the dash arrows are drawn to suggest the three-dimensional perspective but have the same meaning as the plain arrows.

## 5 From Tokens to Tuples

As announced in the introduction, the paper has so far concentrated on token based versions of the languages. A natural question to ask is whether the expressiveness study performed in Sect. 4 can be extended to tuples and thereby can embrace Linda-like languages in their full version.

To that end, two issues need to be taken into account. On the one hand, structured pieces of information need to be tackled instead of flat tokens. Using our notations, this would lead to consider tuples of the form  $\langle t_1, \dots, t_n \rangle$  instead of  $t$  as arguments of *tell*, *ask*, *get* and *nask* primitives. On the other hand, it is desirable to introduce variables as arguments of the tuples in *ask*, *get* and *nask* primitives to retrieve values from the tuples stored on the tuple space<sup>1</sup>. The resulting tuples are classically called templates or anti-tuples.

It turns out that tackling these issues can be reduced to using flat tokens by assuming a very reasonable hypothesis: variables have to range over enumerable sets of values. If this is the case, then, as exemplified in process algebras like mCRL2 [19], any primitive containing a tuple with variables can be rewritten as a choice of that primitive with the variables instantiated to values. For instance,  $ask(\langle 1, X : Int \rangle)$  can be rewritten as  $\sum_{i \in Int} ask(\langle 1, i \rangle)$ . As a result, assuming a general choice over enumerable sets in the language, we may reduce the language to primitives involving tuples without variables. As a further step, tuples having a finite number of arguments and the choices being on enumerable sets, these tuples range over enumerable unions of enumerable sets, namely over an enumerable set. We may thus associate any of these tuples to a token of *Token* and vice-versa. By doing so, we are back to the token-based languages studied before, provided that an interpretation is given to dense tuples. Two are actually possible. Consider for instance a store containing twice  $t(1)$ , three times  $t(2)$  and one time  $t(3)$ . Then, in a first interpretation, the request  $ask(t(X : Int)(3))$  can be satisfied if one may instantiate  $X$  to an integer, say  $i$ , such that the induced instance  $t(i)$  appears at least three times on the store. In our example, this would be possible by giving to  $X$  the value 2. Under this interpretation, the translation just provided from tuple-based languages to token-based languages applies directly, which thus allows to lift the results obtained in Sect. 4 to tuple-based languages. In another interpretation, one may argue that one should find three occurrences of tuples which matches  $t(X)$  by possibly instantiating  $X$  to different values. Under this interpretation, the request  $ask(t(X : Int)(3))$  not only succeeds but also  $ask(t(X : Int)(5))$  since  $t(1)$  and  $t(2)$  both match  $t(X)$ . However, this interpretation is exactly what is captured by DBD-BachT language. In our example,  $ask(t(X : Int)(5))$  could indeed be reformulated as  $ask([t(1), t(2), t(3), \dots](5))$ . Hence, under that second interpretation too, the results obtained in Sect. 4 can be lifted to tuple-based languages.

<sup>1</sup> In Linda, variables are also allowed in *tell* primitives to denote unknown attributes. However, as argued in [12], we believe that it is better to use  $\psi$ -terms in this case, which allows to keep the idea of structured information without the need for writing unknown arguments.

## 6 Implementation Issues

The expressiveness study has shown that BachT is strictly less expressive than Dense BachT, which itself is strictly less expressive than VD-BachT, which is finally less expressive than MRT. One may thus wonder about the interest of all the languages, except the most expressive one. The aim of this section is to convince the reader that implementing the more expressive languages comes with a higher cost and thus that one better selects the language just expressive enough for its coding purposes.

To start with, let us first detail how the token space (or more generally the tuple space<sup>2</sup>) may be implemented for Bach. Since our first implementation (see [4]), processes have been implemented as threads and the tuple space has been implemented as a token-indexed list. Per list element (token), we keep track of the number of identical tokens (*token counter*), and of the input primitives that are suspended on this token. The token list is stored in shared memory. The list is directly updated by the communication primitives, as one may guess: the tell primitive adding tokens and the get primitive consuming them. In order to guarantee exclusive access, the individual list elements are protected by a lock, which means that operations on different tokens can execute in parallel. This has turned out to generate interesting speed-ups over the naive implementation which would lock the entire tuple space for each primitive execution. More precisely, the following algorithms are employed for the primitives.

Performing an *nask* primitive first checks whether the associated token is known to the tuple space (i.e. has already an element in the list of tokens). If not, the tuple space is locked, and a list element for the token is created. If the token counter equals zero, the *nask*-primitive succeeds. If not, it suspends, and is added to the list of suspended primitives, until the token counter reaches zero.

Asking a token  $t$  first checks whether at least one occurrence of  $t$  is present in the tuple space. If so, the primitive succeeds. Otherwise, the ask primitive is put in the associated list of waiting processes, until the token counter is positive.

Getting a token  $t$  proceeds similarly but decrements the token counter for  $t$ . If the token counter reaches zero, we check whether there are suspended *nask*( $t$ ) primitives. If so, the process associated with the *nask* primitive is resumed and is removed from the list of waiting primitives.

Finally, telling a token  $t$  proceeds dually. The list of waiting processes is first inspected to discover an *ask* or *get* primitive waiting for  $t$ . In case an *ask* primitive is discovered, it is resumed and the search continues. In case a *get* primitive is discovered the token  $t$  is consumed by that primitive and the corresponding process is resumed. If no waiting *get* primitives are encountered, then the token counter for  $t$  is incremented.

---

<sup>2</sup> In Bach, following [12], tuples are actually represented in the form of a functor name followed by a series of pairs, each consisting of an attribute associated with a value. Without entering into details, the functor names play the role of tokens and, in that manner, the implementation sketched in this section can be lifted from the tokens considered in this paper to more general tuples.

As the careful reader will have noticed, lifting the Bach implementation to Dense Bach is quite easy. One basically just needs to count the number of occurrences of the tokens for the ask and get primitives and to upgrade the number of tokens for the tell primitives. The case of general tuples is slightly more complicated in that pattern matching needs to be done in addition but only on the elements of the list associated with the functor taken as a token. However, the key property remains: the tuple space need not be blocked globally, locks are only put on the list associated with the considered token.

Moving to VD-Bach is more subtle since several locks need to be taken and hence the above key property cannot be met. Consider for instance a multi-get primitive which needs to consume tokens  $a$ ,  $b$  and  $c$ . To evaluate it, one needs in principle to lock the lists associated with  $a$ ,  $b$  and  $c$ . However, as other primitives may compete, for instance to get  $b$ ,  $c$  and  $d$ , one actually needs to lock at once the three lists associated with  $a$ ,  $b$  and  $c$  in order to prevent the system from deadlocks. In practice, an easy way to do so is to lock the whole tuple space. However, by using abstract interpretation techniques on a static code or by using declarations on the vectors employed and by employing the activator vectors of [21], one may slightly relax the global lock by adding locks for super sets of vectors, in the example above to  $a$ ,  $b$ ,  $c$  and  $d$ . This still allows for parallel computations, although to a lesser extent than for Dense Bach.

It is worth noting that the more the structure involves tokens the more constraining are the locks used. In particular, a similar technique can be used to implement MRT but, as one may expect, with a greater computation overhead.

As a conclusion, the more expressive the language is the more expensive is its implementation. Hence, there is obviously a trade-off to be made between the programming ease offered by the language expressiveness and the computation costs needed by its implementation.

## 7 Conclusion

This paper is written in the continuity of our previous research on the expressiveness of Linda-like languages. It has presented extensions of our Bach language aiming at handling multiplicities. In particular, as a novel piece of work, we have presented an extension of our Dense BachT language, that has promoted the interest of vectors of dense tokens. The new language, called Vectorized Dense BachT proposes to atomically perform multiple operations on dense tokens by introducing lists of dense tokens in the four classical primitives of our BachT language.

Our work thus builds upon our previous work [10, 11, 16, 20, 22, 24–26]. We have essentially followed the same lines and in particular have used De Boer and Palamidessi’s notion of modular embedding to compare the families of sub-languages of Dense BachT and Vectorized Dense BachT. Accordingly, we have established a gain of expressivity, namely that Vectorized Dense BachT is strictly more expressive than Dense BachT and, consequently, in view of the results of [20], strictly more expressive than the BachT and Linda languages. However the

structure of the hierarchies of the sublanguages of a family is kept, which shows that the very nature of the tell, ask, get and nask primitives is preserved. We have also compared Vectorized Dense BachT with a multiset rewriting language and showed that it is strictly less expressive. However, as shown in the paper, it is expressive enough to code interesting applications as well as Dense Bach with Distributed Density, a language we introduced in [16]. Moreover, the fact that Vectorized Dense BachT only provides atomic tell, ask, nask and get allows for more efficient implementations than MRT.

Our work has similarities but also differences with several work on the expressiveness of Linda-like languages. Compared to [33,34], it is worth observing that a different comparison criteria is used to compare the expressiveness of languages. Indeed, in these pieces of work, the comparison is performed on (i) the compositionality of the encoding with respect to parallel composition, (ii) the preservation of divergence and deadlock, and (iii) a symmetry condition. Moreover, we have taken a more liberal view with respect to the preservation of termination marks in requiring these preservations on the store resulting from the execution from the empty store of the coded versions of the considered agents and not on the same store. In particular, these ending stores are not required to be of the form  $\sigma \cup \sigma$  (where  $\cup$  denotes multi-set union) if this is so for the stores resulting from the agents themselves.

In [3], nine variants of the  $\mathcal{L}_B(\text{ask}, \text{nask}, \text{get}, \text{tell})$  language are studied. They are obtained by varying both the nature of the shared data space and its structure. Rephrased in the setting of [17], this amounts to considering different operational semantics. In contrast, in our work we fix an operational semantics and compare different languages on the basis of this semantics. In [14], a process algebraic treatment of a family of Linda-like concurrent languages is presented. Again, different semantics are considered whereas we have stucked to one semantics and have compared languages on this basis.

In [13], a study of the absolute expressive power of different variants of Linda-like languages has been made, whereas we study the relative expressive power of different variants of such languages (using modular embedding as a yard-stick and the ordered interpretation of tell).

It is worth observing that [3,13,14,33,34] do not deal with a notion of density attached to tuples. In contrast, [5,6] decorate tuples with an extra field in order to investigate how probabilities and priorities can be introduced in the Linda coordination model. Different expressiveness results are established in [5] but on an absolute level with respect to Turing expressiveness and the possibility to encode the Leader Election Problem. Our work contrasts in several aspects. First, we have established relative expressiveness results by comparing the sublanguages of two families. Moreover, some of these sublanguages incorporate the *nask* primitives, which, strictly increases the expressiveness. Finally, the introduction of density resembles but is not identical to the association of weights to tuples. Indeed, in contrast to [5,6] we do not modify the tuples on the store and do not modify the matching function so as to retrieve the tuple with the highest weight. In contrast, we modify the tuple primitives so as to

be able to atomically put several occurrences of a tuple on the store and check for the presence or absence of a number of occurrences. As can be appreciated by the reader through the comparison of BachT, Dense BachT and Vectorized Dense BachT, this facility of handling atomically several occurrences produces a real increase of expressiveness. One may however naturally think of encoding the number of occurrences of a tuple as an additional weight-like parameter. It is nevertheless not clear how our primitives tackling at once several occurrences can be rephrased in Linda-like primitives and how the induced encoding would still fulfill the requirements of modularity. This will be the subject for future research.

In [32], Viroli and Casadei propose a stochastic extension of the Linda framework, with a notion of tuple concentration, similar to the weight of [5, 6] and our notion of density. The syntax of this tuple space is modeled by means of a calculus, with an operational semantics given as an hybrid CTMC/DTMC model. This operational semantics describes the behavior of tell, ask and get like primitives but does not consider a nask like primitive. Moreover, no expressiveness results are established and there is no counterpart for non-determinism arising from the distribution of density on tokens.

These three last pieces of work tackle probabilistic extensions of Linda-like languages. As a further and natural step in our research, we aim at studying how our notions of multiplicity can be the basis of such probabilistic extensions.

**Acknowledgment.** We thank the anonymous reviewers for their comments and suggestions. We also thank A. Brogi and E. de Vink for helpful discussions on the expressiveness of coordination languages.

## References

1. Banâtre, J.-P., Le Métayer, D.: Programming by multiset transformation. *Commun. ACM* **36**(1), 98–111 (1993)
2. Banâtre, J.-P., Le Métayer, D.: Gamma and the chemical reaction model: ten years after. In: *Coordination Programming*, pp. 3–41. Imperial College Press, London (1996)
3. Bonsangue, M.M., Kok, J.N., Zavattaro, G.: Comparing coordination models based on shared distributed replicated data. In: *ACM Symposium on Applied Computing*, pp. 156–165 (1999)
4. De Bosschere, K., Jacquet, J.-M.: Multi-prolog: definition, operational semantics, and implementation. In: Warren, D.S. (ed.) *Proceedings of the International Conference on Logic Programming*, Budapest, Hongrie, pp. 299–314. The MIT Press (1993)
5. Bravetti, M., Gorrieri, R., Lucchi, R., Zavattaro, G.: Quantitative information in the tuple space coordination model. *Theoret. Comput. Sci.* **346**(1), 28–57 (2005)
6. Bravetti, M., Gorrieri, R., Lucchi, R., Zavattaro, G.: Probabilistic and prioritized data retrieval in the Linda coordination model. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) *COORDINATION 2004*. LNCS, vol. 2949, pp. 55–70. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-24634-3\\_7](https://doi.org/10.1007/978-3-540-24634-3_7)
7. Bravetti, M., Zavattaro, G.: Service oriented computing from a process algebraic perspective. *J. Logic Algebr. Program.* **70**(1), 3–14 (2007)

8. Brogi, A., Jacquet, J.-M.: Modeling coordination via asynchronous communication. In: Garlan, D., Le Métayer, D. (eds.) COORDINATION 1997. LNCS, vol. 1282, pp. 238–255. Springer, Heidelberg (1997). [https://doi.org/10.1007/3-540-63383-9\\_84](https://doi.org/10.1007/3-540-63383-9_84)
9. Brogi, A., Jacquet, J.-M.: On the expressiveness of Linda-like concurrent languages. *Electron. Not. Theoret. Comput. Sci.* **16**(2), 61–82 (1998)
10. Brogi, A., Jacquet, J.-M.: On the expressiveness of coordination models. In: Ciancarini, P., Wolf, A.L. (eds.) COORDINATION 1999. LNCS, vol. 1594, pp. 134–149. Springer, Heidelberg (1999). [https://doi.org/10.1007/3-540-48919-3\\_11](https://doi.org/10.1007/3-540-48919-3_11)
11. Brogi, A., Jacquet, J.-M.: On the expressiveness of coordination via shared dataspace. *Sci. Comput. Program.* **46**(1–2), 71–98 (2003)
12. Brogi, A., Jacquet, J.-M., Linden, I.: On modeling coordination via asynchronous communication and enhanced matching. *Electron. Not. Theoret. Comput. Sci.* **68**(3), 284–309 (2003)
13. Busi, N., Gorrieri, R., Zavattaro, G.: On the Turing equivalence of Linda coordination primitives. *Electron. Not. Theoret. Comput. Sci.* **7**, 75–75 (1997)
14. Busi, N., Gorrieri, R., Zavattaro, G.: A process algebraic view of Linda coordination primitives. *Theoret. Comput. Sci.* **192**, 167–199 (1998)
15. Darquennes, D.: On Multiplicities in Coordination Languages. Ph.D. thesis, Faculty of Computer Science, University of Namur, Namur, Belgium (2017)
16. Darquennes, D., Jacquet, J.-M., Linden, I.: On distributed density in tuple-based coordination languages. In: Cámara, J., Proença, J. (eds.) Proceedings of the 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems. EPTCS, vol. 175, pp. 36–53. Springer (2015)
17. de Boer, F.S., Palamidessi, C.: Embedding as a tool for language comparison. *Inf. Comput.* **108**(1), 128–157 (1994)
18. Gelernter, D.: Generative communication in Linda. *ACM Trans. Program. Lang. Syst.* **7**(1), 80–112 (1985)
19. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
20. Jacquet, J.-M., Linden, I., Darquennes, D.: On density in coordination languages. In: Canal, C., Villari, M. (eds.) ESOC 2013. CCIS, vol. 393, pp. 189–203. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-45364-9\\_16](https://doi.org/10.1007/978-3-642-45364-9_16)
21. Jacquet, J.-M., Linden, I., Staicu, M.-O.: Blackboard rules: from a declarative reading to its application for coordinating context-aware applications in mobile ad hoc networks. *Sci. Comput. Program.* **115–116**, 79–99 (2016)
22. Jacquet, J.M., Linden, I., Darquennes, D.: On the introduction of density in tuple-space coordination languages. *Sci. Comput. Program.* **115–116**, 149–176 (2016)
23. 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.) ESOC 2012. LNCS, vol. 7592, pp. 1–16. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33427-6\\_1](https://doi.org/10.1007/978-3-642-33427-6_1)
24. Linden, I., Jacquet, J.-M.: On the expressiveness of absolute-time coordination languages. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 232–247. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-24634-3\\_18](https://doi.org/10.1007/978-3-540-24634-3_18)
25. Linden, I., Jacquet, J.-M.: On the expressiveness of timed coordination via shared dataspace. *Electron. Not. Theoret. Comput. Sci.* **180**(2), 71–89 (2007)
26. Linden, I., Jacquet, J.-M., De Bosschere, K., Brogi, A.: On the expressiveness of relative-timed coordination models. *Electron. Not. Theoret. Comput. Sci.* **97**, 125–153 (2004)

27. Linden, I., Jacquet, J.-M., De Bosschere, K., Brogi, A.: On the expressiveness of timed coordination models. *Sci. Comput. Program.* **61**(2), 152–187 (2006)
28. Mariani, S.: Coordination of Complex Sociotechnical Systems - Self-organisation of Knowledge in MoK. *Artificial Intelligence: Foundations, Theory, and Algorithms*. Springer, Cham (2016). <https://doi.org/10.1007/978-3-319-47109-9>
29. Omicini, A.: Formal ReSpecT in the A&A perspective. *Electron. Not. Theoret. Comput. Sci.* **175**(2), 97–117 (2007)
30. Shapiro, E.: Embeddings among concurrent programming languages. In: Cleaveland, W.R. (ed.) *CONCUR 1992*. LNCS, vol. 630, pp. 486–503. Springer, Heidelberg (1992). <https://doi.org/10.1007/BFb0084811>
31. Tolksdorf, R.: Laura - a service-based coordination language. *Sci. Comput. Program.* **31**(2–3), 359–381 (1998)
32. Viroli, M., Casadei, M.: Biochemical tuple spaces for self-organising coordination. In: Field, J., Vasconcelos, V.T. (eds.) *COORDINATION 2009*. LNCS, vol. 5521, pp. 143–162. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-02053-7\\_8](https://doi.org/10.1007/978-3-642-02053-7_8)
33. Zavattaro, G.: On the incomparability of Gamma and Linda. In: *Electronic Transactions on Numerical Analysis* (1998)
34. Zavattaro, G.: Towards a hierarchy of negative test operators for generative communication. *Electron. Not. Theoret. Comput. Sci.* **16**, 154–170 (1998)





# A Formal Approach to the Engineering of Domain-Specific Distributed Systems

Rocco De Nicola<sup>1</sup>(✉), Gianluigi Ferrari<sup>2</sup>(✉), Rosario Pugliese<sup>3</sup>(✉),  
and Francesco Tiezzi<sup>4</sup>(✉)

<sup>1</sup> IMT Institute for Advanced Studies Lucca, Lucca, Italy

`rocco.denicola@imtlucca.it`

<sup>2</sup> Università di Pisa, Pisa, Italy

`giangi@di.unipi.it`

<sup>3</sup> Università degli Studi di Firenze, Florence, Italy

`rosario.pugliese@unifi.it`

<sup>4</sup> Università di Camerino, Camerino, Italy

`francesco.tiezzi@unicam.it`

**Abstract.** We review some results regarding specification, programming and verification of different classes of distributed systems which stemmed from the research of the Concurrency and Mobility Group at University of Firenze. More specifically, we review distinguishing features of network-aware programming, service-oriented computing, automatic computing, and collective adaptive systems programming. We then present an overview of four different languages, namely KLAIM, COWS, SCEL and AbC. For each language, we discuss design choices, present syntax and informal semantics, show some illustrative examples, and describe programming environments and verification techniques.

## 1 Introduction

Since the mid-90s, we have witnessed an evolution of distributed computing towards increasingly complex systems formed by several software components featuring asynchronous interactions, and operating in open-ended and non-deterministic environments. Such transformation, initially induced by the spreading of internetworking technologies, led to a paradigm shift making software components *aware* of the underlying network infrastructure. Such awareness, on the one hand, constrained the remote access to distributed resources and, on the other hand, enabled computation mobility, to support different kinds of optimisations.

On top of these networked systems, software components have been then deployed to provide *services* accessible by end-users and other system components. This fostered the development of sophisticated applications built by reusing and composing simpler elements. Such service-based compositional approach required to overcome the interaction challenges posed by the heterogeneity

of the involved components; interoperability was then achieved through the definition of standard protocols and suitable run-time support for programming languages.

Later on, the need arose of reducing the maintenance cost of these systems, whose size was becoming bigger and bigger, and of extending their applicability to interact with and control the physical world, possibly in scenarios where human intervention was difficult or even impossible. It was then advocated to rely on *autonomic* components, which are capable of continuously monitoring their internal status and the working environment, and to adapt their behaviour accordingly.

More recently, a growing interest emerged in a new class of computational systems consisting of a large number of interacting components featuring complex behaviour that are usually distributed, heterogeneous, decentralised and interdependent, and operate in dynamic and possibly unpredictable environments. The components form *collectives* by combining their behaviours to achieve specific goals, or to contribute to an emerging behaviour of the global system. Collectives abstract from the identity of the single components to guarantee scalability.

The evolution of distributed computing described above corresponds to the emergence of classes of systems that characterise specific programming domains. Correspondingly, dedicated programming paradigms have been proposed, namely *network-aware programming*, *service-oriented computing*, *autonomic computing*, and *collective adaptive systems programming*. Besides dealing with the distinctive aspects of each of such domains, the main challenge in engineering these classes of distributed systems is to coordinate the overall behaviour resulting from the involved components while ensuring trustworthiness of the whole system. To meet this goal, many researchers have adopted language-based approaches that combine the use of formal methods techniques with model-driven software engineering. The key ingredients of the resulting methodology can be summarised as follows:

1. a specification language equipped with a formal semantics, which associates mathematical models to each term of the language to precisely establish the expected behaviour of systems;
2. a set of techniques and tools, built on top of the models, to express and verify properties of interest for the considered class of systems;
3. a programming framework together with the associated runtime environment to actually execute the specified systems.

When specialising this methodology, a major challenge for (specification or programming) language designers is to devise appropriate abstractions and linguistic primitives to deal with the specificities of the domain under investigation. Indeed, including the distinctive aspects of the domain as first-class elements of the language makes systems design more intuitive and concise, and their analysis more effective. In fact, when the outcome of a verification activity is expressed by considering the high level features of a system, and not its low-level representation, system designers can be provided with more direct feedbacks.

This paper reviews some of the efforts, to which the authors have contributed, in applying the outlined methodology to the classes of distributed systems mentioned above by taking as starting point process algebras and some of the verification techniques and tools developed for them. The approach was initially applied to network-aware programming and the main result was the definition of the KLAIM language [23] (Sect. 2). Afterwards, the approach was applied to service-oriented computing resulting in the design of COWS [57] (Sect. 3), to autonomic computing obtaining as a result SCEL [27] (Sect. 4), and to collective adaptive systems programming to obtain AbC [2] (Sect. 5). For each of these domain-specific languages, we discuss design choices, present syntax and informal semantics, show some simple but illustrative example specifications, and describe programming environments and verification techniques. We want to stress that all languages have been equipped with a formal operational semantics, based on labelled transition systems, that is omitted here for the sake of space; the interested reader is referred to the relevant papers in the bibliography. Moreover, for the sake of readability and understandability, the examples are presented at the level of the specification language; of course they can be refined in order to be implemented by means of the proposed programming environments, but currently they are not. The paper ends with a summary of distinguishing features of the presented languages and with a few considerations about the lessons learnt (Sect. 6).

## 2 KLAIM: A Kernel Language for Agents Interaction and Mobility

Network awareness indicates the ability of the software components of a distributed application to manage directly a sufficient amount of knowledge about the network environment where they are currently deployed. This capability allows components to have a highly dynamic behaviour and manage unpredictable changes of the network environment over time. This is of great importance when programming mobile components capable of disconnecting from one node of the underlying infrastructure and of reconnecting to a different node. Programmers are usually supported with primitive constructs that enable components to communicate, distribute and retrieve data to and from the nodes of the underlying infrastructure.

KLAIM (*Kernel Language for Agents Interaction and Mobility*, [23]) has been specifically devised to design distributed applications consisting of several components (both stationary and mobile) deployed over the nodes of a distributed infrastructure. The KLAIM programming model relies on a unique interface (i.e. set of operations) supporting component communications and data management.

*Localities* are the basic building blocks of KLAIM for guaranteeing network awareness. They are symbolic addresses (i.e. network references) of nodes and are referred by means of identifiers. Localities can be exchanged among the computational components and are subjected to sophisticated scoping rules. They

provide the naming mechanism to identify network resources and to represent the notion of administrative domain: computations at a given locality are under the control of a specific authority. This way, localities naturally support programming spatially distributed applications.

KLAIM builds on Linda's notion of *generative communication* through a single shared tuple space [36] and generalizes it to multiple distributed tuple spaces. A tuple space is a multiset of tuples. Tuples are *anonymous* sequences of data items and are retrieved from tuple spaces by means of an *associative selection*. Interprocess communication occurs through *asynchronous* exchange of tuples via tuple spaces: there is no need for producers (i.e. senders) and consumers (i.e. receivers) of a tuple to synchronise.

The obtained communication model has a number of properties that make it appealing for distributed computing in general (see, e.g., [14, 19, 31, 37]). It supports *time uncoupling* (data life time is independent of the producer process life time), *destination uncoupling* (the producer of a datum does not need to know the future use or the final destination of that datum) and *space uncoupling* (programmers need to know a single interface only to operate over the tuple spaces, regardless of the network node where the action will take place).

## 2.1 Syntax

The syntax of KLAIM is presented in Table 1. We assume existence of two disjoint sets: the set of *localities*, ranged over by  $l$ , and the set of *locality variables*, ranged over by  $u$ . Their union gives the set of *names*, ranged over by  $\ell$ . We also assume three other disjoint sets: a set of *value variables*, ranged over by  $x$ , a set of *process variables*, ranged over by  $X$ , and a set of *process identifiers*, ranged over by  $A$ .

**Table 1.** Klaim syntax

NETS:		TUPLES:	
$N ::= l ::_{\rho} P$	(computational node)	$t ::= f \mid f, t$	
$\mid l :: \langle et \rangle$	(located tuple)	TUPLE FIELDS:	
$\mid N_1 \parallel N_2$	(net composition)	$f ::= e \mid \ell \mid u \mid P$	
PROCESSES:		EVALUATED TUPLES:	
$P ::= \mathbf{nil}$	(null process)	$et ::= ef \mid ef, et$	
$\mid a.P$	(action prefixing)	EVALUATED TUPLE FIELDS:	
$\mid P_1 \mid P_2$	(parallel composition)	$ef ::= V \mid l \mid P$	
$\mid X$	(process variable)	TEMPLATES:	
$\mid A$	(process invocation)	$T ::= F \mid F, T$	
ACTIONS:		TEMPLATE FIELDS:	
$a ::= \mathbf{out}(t)@l$	(output)	$F ::= f \mid !x \mid !u \mid !X$	
$\mid \mathbf{in}(T)@l$	(input)	EXPRESSIONS:	
$\mid \mathbf{read}(T)@l$	(read)	$e ::= V \mid x \mid \dots$	
$\mid \mathbf{eval}(P)@l$	(migration)		
$\mid \mathbf{newloc}(u)$	(creation)		

NETS are finite collections of nodes where processes and data can be placed. A *computational node* takes the form  $l::_{\rho}P$ , where  $\rho$  is an *allocation environment* and  $P$  is a process. Since processes may refer to locality variables, the allocation environment acts as a *name solver* binding locality variables to specific localities.

PROCESSES are the active computational units of KLAIM. Their syntax is standard and specifies the ACTIONS to be executed. Recursive behaviours are modelled via process definitions; it is assumed that each identifier  $A$  has a *single* defining equation  $A \triangleq P$ .

The tuple space of a node consists of all the EVALUATED TUPLES located there. TUPLES are sequences of *actual* fields, i.e. expressions, localities or locality variables, or processes. The precise syntax of EXPRESSIONS is deliberately not specified; it is just assumed that they contain, at least, basic values, ranged over by  $V$ , and variables, ranged over by  $x$ . TEMPLATES are sequences of actual and formal fields, and are used as patterns to select tuples in a tuple space. *Formal* fields are identified by the !-tag (e.g.  $!x$ ) and are used to bind variables to values.

## 2.2 Informal Semantics

NETS aggregate nodes through the *composition* operator  $- \parallel -$ , which is both commutative and associative. PROCESSES are concurrently executed in an *inter-leaving* fashion, either at the same computational node or at different nodes. They can perform operations borrowed from a unique interface which provides two categories of actions. The first one consists of the programming abstractions supporting data management. Three primitive behaviours are provided: adding (**out**), withdrawing (**in**) and reading (**read**) a tuple to/from a tuple space. Input and output actions are *mutators*: their execution modifies the tuple space. The read action is an *observer*: it checks the availability and takes note of the content of a certain tuple without removing it from the tuple space. The second category of actions refers to network awareness: the migration action (**eval**) activates a new process over a network node, while the creation action (**newloc**) generates a new network node. The latter action is the only one not indexed by a locality because it acts locally; all the other actions are tagged with the (possibly remote) locality where they will take place. Note that, in principle, each network node can provide its own implementation of the action interface. This feature can be suitably exploited to sustain different policies for data handling as done, e.g., in META KLAIM [34].

Only evaluated tuples can be added to a tuple space and templates must be evaluated before they can be used for retrieving tuples. Tuple and template evaluation amounts to computing the values of their expressions. Localities and formal fields are left unchanged by such evaluation.

A *pattern-matching* mechanism is used for associatively selecting (evaluated) tuples from tuple spaces according to (evaluated) templates. Intuitively, an evaluated template matches against an evaluated tuple if both have the same number of fields and corresponding fields do match; two values (localities) match only if they are identical, while formal fields match any value of the same type. A successful matching returns a substitution associating the variables contained in

the formal fields of the template with the values contained in the corresponding actual fields of the accessed tuple.

Process variables support *higher-order* communication, namely the capability to exchange (the code of) a process and possibly execute it. This is realised by first adding a tuple containing the process to a tuple space and then retrieving/withdrawing this tuple while binding the process to a process variable.

Finally, KLAIM offers two forms of process mobility. One is based on *static scoping*: by exploiting higher-order communication, a process moves along the nodes of a net with a fixed binding of resources determined by the allocation environments of the nodes from where, from time to time, it is going to move. The other form of mobility relies on *dynamic scoping*: when migrating, a process breaks the local links to resources and inherits those of the destination node.

### 2.3 Example: A Street Light Controller

We outline here the main features of the design of a (simplified) *Street Light Controller* working on a one-way street, inside a restricted traffic zone. It consists of several integrated components. Smart lamp post components are cyber-physical entities (battery powered). They can sense their surrounding environment and can communicate with their neighbours to share information. For instance if (a sensor of) the lamp post perceives a pedestrian and there is not enough light in the street it turns its light on and communicates the presence of the pedestrian to the lamp posts nearby. A further component of the street light controller uses the information provided by the electronic access point to the street. When a car crosses the checkpoint, a message is sent to the supervisor of the street accesses, that in turn notifies the presence of the car to a further component of the system: the supervisor of the street. A notice is also sent to the node that hosts the cloud service of the police department. This service checks whether the car is enabled to enter that restricted zone, through automatic number plate recognition. The street supervisor, as a result of this coordinated behaviour, is in charge of sending the authorisation message to the lamp post closest to the checkpoint that starts a forward chain till the end of the street, thus completing the overall cooperative behaviour. For simplicity, here we assume that each sensor has a unique name and the sensed values are modelled as tuples containing the name of the sensor and the detected value. Since every cyber-physical node has a fixed number of sensors, the tuple space of the node is designated to store the values read by sensors.

The process running at checkpoint node is the driver of the visual sensor  $S_{cp}$ , defined below. The driver takes a picture of the car detected in the street and stores it in the tuple space:

$$S_{cp} \triangleq \mathbf{in}(probe, !v)@\mathbf{self}.\mathbf{out}(picture, v)@\mathbf{self}.S_{cp}$$

where *probe* is the unique identifier of the sensor and the tuple tagged by *picture* identifies the collected picture of the car. Then, the picture is enhanced (by using

the function *noiseRed* for reducing noise) by the process  $P_{cp}$  and sent to the supervisor:

$$P_{cp} \triangleq \mathbf{in}(picture, !z)@\mathbf{self.out}(enPicture, noiseRed(z))@\mathbf{controller}.P_{cp}$$

The checkpoint node  $N_{cp}$  is defined as  $l_{cp}::\rho_{cp}(P_{cp} \mid S_{cp} \mid B_{cp})$ , where  $\rho_{cp}$  is the allocation environment binding the locality variable *controller* to the locality  $l_a$  where the access controller node is deployed, and  $B_{cp}$  abstracts other components we are not interested in, among which the tuple space at  $l_{cp}$ . The access controller node  $N_a$  receives the picture and communicates the presence of the car to the lamp posts supervisor and to the police department. The behaviour of the driver process running at node  $N_a$  is as follows

$$P \triangleq \mathbf{in}(enPicture, !x)@\mathbf{self.out}(car, x)@\mathbf{supervisor.out}(car, x)@\mathbf{pdept}.P$$

and the node is defined as  $l_a::\rho_a(P \mid B_a)$ , where  $\rho_a$  binds the locality variable *supervisor* and *pdept* to the localities where the street supervisor and the police department are deployed. The process  $B_a$  abstracts other components we are not interested in, among which the tuple space at  $l_a$ . The supervisor node  $N_s$  contains the process  $P_s$  that receives the picture from  $N_a$  and sends a message to the lamp node closest to the checkpoint; its behaviour is straightforward.

In our smart street light control system there is a node  $N_p$  for each lamp post. Each lamp post is equipped with four sensors to sense (1) the environment light, (2) the solar light, (3) the battery level and (4) the presence of a pedestrian. These sensors are the interface towards the cyber-physical world and their asynchronous behaviour simply inserts the acquired information in the tuple space of the node. The drivers of sensors share the same structure; hence we only show that for the battery level:

$$S_{battery} \triangleq \mathbf{in}(probeBatteryLevel, !v)@\mathbf{self.out}(batteryLevel, v)@\mathbf{self}.S_{battery}$$

The control process reads the current values from the sensors and stores the resulting values in a local tuple consisting of four terms, i.e. environment light, solar light, battery level and presence of pedestrian, by means of the action  $\mathbf{out}(el, sl, bl, p)@\mathbf{self}$ . Action  $\mathbf{read}(!el, !sl, !bl, !p)@\mathbf{self}$  is used to access such information in order to detect the actual state of affair: (i) a pedestrian is in the street ( $p = true$ ), (ii) the intensity of environment and solar lights are greater than, or equal to, the given thresholds,  $el \geq th_1$  and  $sl \geq th_2$ , and (iii) there is enough battery (at least  $bl \geq th_3$ ). The presence of the pedestrian is communicated to the lamp posts nearby, whose locality is obtained from the allocation environment ( $\mathbf{out}(pedestrian, p)@\mathbf{next}$ ). In case the battery level is insufficient, an error message is sent to the supervisor node ( $\mathbf{out}(failure)@\mathbf{supervisor}$ ).

The overall intelligent controller of the street lights is then described as the parallel composition of the checkpoint node  $N_{cp}$ , the supervisor nodes  $N_a$  and  $N_s$ , the nodes of lamp posts  $N_p$ , with  $p \in [1, k]$ , and the police department node  $N_{pd}$ :

$$N_{cp} \parallel N_a \parallel N_s \parallel N_1 \parallel \dots \parallel N_k \parallel N_{pd}$$

## 2.4 Programming Environment

X-KLAIM (*eXtended* KLAIM, [11]) is an experimental programming language that extends KLAIM with a high level syntax for processes. It provides variable declarations, enriched operations, assignments, conditionals, sequential and iterative process composition. The implementation of X-KLAIM is based on KLAVA<sup>1</sup> (KLAIM *in* Java, [12]), a Java package that provides the run-time system for X-KLAIM operations, and on a compiler, which translates X-KLAIM programs into Java programs that use KLAVA. X-KLAIM can be used to write the higher layer of distributed applications while KLAVA can be seen both as a middleware for X-KLAIM programs and as a Java framework for programming according to the KLAIM paradigm. By using KLAVA directly, the programmer is able to exchange, through tuples, any kind of Java object, and to implement a finer grained type of mobility.

## 2.5 Verification Techniques

Many verification techniques have been defined for KLAIM and variants thereof. Due to lack of space, here we only mention a few of them. In [26] a temporal logics is proposed for specifying and verifying dynamic properties of mobile processes specified in KLAIM. The inspiration for the proposal was the Hennessy-Milner Logics, but it needed significant adaptations due to the richer operating context of components. The resulting logic provides tools for establishing not only deadlock freedom, liveness and correctness with respect to given specifications (which are crucial properties for process calculi and similar formalisms), but also properties concerned with resource allocation, resource access and information disclosure (which are important issues for processes involving different actors and authorities).

An important topic deeply investigated for KLAIM is the use of type systems for security [24, 25, 40], devoted to control accesses to tuple spaces and mobility of processes. In these type systems, traditional types are generalised to *behavioural types*. These are abstractions of process behaviours that provide information about *processes capabilities*, namely the operations that processes can execute at a specific locality (downloading/consuming a tuple, producing a tuple, activating a process, and creating a new node). When using behavioural types, each KLAIM node is equipped with a security policy, determined by a net coordinator, that specifies the execution privileges; the policy of a node describes the actions processes there located can execute. By exploiting static and dynamic checks, type checking guarantees that only processes whose intentions match the rights granted to them by coordinators are allowed to proceed.

An alternative approach to control accesses to tuple spaces and mobility of processes is introduced in [28]. It is based on Flow Logic and permits statically checking absence of violations. Starting from an existing type system for KLAIM with some dynamic checks, the insights from the Flow Logic approach are

---

<sup>1</sup> X-KLAIM and KLAVA are available on line at <http://music.dsi.unifi.it>.



exploited to construct a type system for statically guaranteeing secure access to tuple spaces and safe process migration for a smooth extension of KLAIM. This is the first completely static type system for controlling accesses devised for a tuple space-based coordination language.

Finally, an expressive language extension, called METAKLAIM, and a powerful type system are described in [34]. METAKLAIM is a higher order distributed process calculus equipped with staging mechanisms. It integrates METAML (an extension of SML for multi-stage programming) and KLAIM, to permit interleaving of meta-programming activities (such as assembly and linking of code fragments), dynamic checking of security policies at administrative boundaries, and traditional computational activities on a wide area network (such as remote communication and code mobility). METAKLAIM exploits a powerful type system (including polymorphic types *à la* system F) to deal with highly parameterised mobile components and to enforce security policies dynamically: types are meta-data that are extracted from code at run-time and are used to express trustiness guarantees. The dynamic type checking ensures that the trustiness guarantees of wide area network applications are maintained also when computations inter-operate with potentially untrusted components.

### 3 Cows: Calculus for Orchestration of Web Services

Since the early 2000s, the increasing success of e-business, e-learning, e-government, and other similar systems, has led the World Wide Web, initially thought of as a system for human use, to evolve towards an architecture for Service-Oriented Computing (SOC) supporting automated use. The SOC paradigm, that finds its origin in object-oriented and component-based software development, aims at enabling developers to build networks of distributed, interoperable and collaborative applications, regardless of the platform where the applications run and of the programming language used to develop them. The paradigm is based on the use of independent computational units, called *services*. They are loosely coupled reusable components, that are built with little or no knowledge about clients and about other services involved in their operating environment.

One successful instantiation of the general SOC paradigm is given by the Web Service technology [62], which exploits the pervasiveness of the Internet and related standards. Traditional software engineering technologies, however, do not neatly fit with SOC, thus hindering its full realisation in practice. The challenges come from the necessity of dealing at once with such issues as asynchronous interactions, concurrent activities, workflow coordination, business transactions, resource usage, and security, in a setting where demands and guarantees can be very different for the many involved components.

Cows (*Calculus for Orchestration of Web Services*, [44,57]) is a process calculus whose design has been influenced by the OASIS standard WS-BPEL [54] for orchestration of web services. In Cows, *services* are computational entities capable of generating multiple instances to concurrently handle different client

**Table 2.** COWS syntax

SERVICES:	RECEIVE-GUARDED CHOICE:
$s ::= u \bullet u' ! \bar{e}$ (invoke)	$g ::= \mathbf{0}$ (nil)
$\mathbf{kill}(k)$ (kill)	$p \bullet o ? \bar{w}.s$ (request processing)
$g$ (receive-guarded choice)	$g + g$ (choice)
$s \mid s$ (parallel composition)	
$\{s\}$ (protection)	
$[e]s$ (delimitation)	
$*s$ (replication)	

requests. Inter-service communication occurs through *communication endpoints* and relies on pattern-matching for logically correlating messages to form an interaction session by means of their identical contents. Differently from most process calculi, receive activities in COWS bind neither names nor variables, and this is crucial for allowing concurrent service instances to share (part of) the state. The calculus also supports service fault and termination handling by providing activities to force termination of labelled service instances and to protect service activities from a forced termination.

### 3.1 Syntax

The syntax of COWS is presented in Table 2. We use three countable disjoint sets: the set of *values* (ranged over by  $v$ ), the set of ‘write once’ *variables* (ranged over by  $x$ ), and set of *killer labels* (ranged over by  $k$ ). The set of values is left unspecified; however, we assume that it includes the set of partner and operation *names* (ranged over by  $n, p, o$ ) mainly used to represent communication endpoints. We also use a set of *expressions* (ranged over by  $\epsilon$ ), whose exact syntax is deliberately omitted; we just assume that expressions contain values and variables, and do not contain killer labels. As a matter of notation,  $w$  ranges over values and variables,  $u$  ranges over names and variables, and  $e$  ranges over *elements*, i.e. killer labels, names and variables. Notation  $\bar{x}$  stands for tuples, e.g.  $\bar{x}$  means  $\langle x_1, \dots, x_n \rangle$  (with  $n \geq 0$ ), where variables in the same tuple are all distinct.

*Services* are structured activities built from basic activities, i.e. the empty activity  $\mathbf{0}$ , the invoke activity  $\_ \bullet \_!$ , the receive activity  $\_ \bullet \_?$ , and the kill activity  $\mathbf{kill}(\_)$ , by means of prefixing  $\_ \_$ , choice  $\_ + \_$ , parallel composition  $\_ \mid \_$ , protection  $\{ \_ \}$ , delimitation  $[ \_ ]$  and replication  $* \_$ . We write  $I \triangleq s$  to assign a name  $I$  to the term  $s$ .

### 3.2 Informal Semantics

*Invoke* and *receive* are the communication activities. The former permits invoking an operation (i.e., a functionality like a method in object-oriented programming) offered by a service, while the latter permits waiting for an invocation to

arrive. Besides output and input parameters, both activities indicate an endpoint through which communication should occur.

An *endpoint*  $p \bullet o$  can be interpreted as a specific implementation of operation  $o$  provided by the service identified by the logic name  $p$ . The names composing an endpoint can be dealt with separately, as in an asynchronous request-response interaction, where usually the service provider statically knows the name of the operation for sending the response, but not the partner name of the requesting service it has to reply to. Partner and operation names can be exchanged in communication, thus enabling many different interaction patterns among service instances. However, dynamically received names cannot form the endpoints used to receive further invocations (as in *localised*  $\pi$ -calculus [51]). In other words, endpoints of receive activities are identified statically because the syntax only allows using names and not variables for them. This design choice reflects the current (web) service technologies that require endpoints of receive activities be statically determined.

An invoke  $p \bullet o!(\epsilon_1, \dots, \epsilon_n)$  can proceed as soon as all expression arguments are successfully evaluated. A receive  $p \bullet o?(w_1, \dots, w_n).s$  offers an invocable operation  $o$  along with a given partner name  $p$ , thereafter the service continues as  $s$ . An inter-service communication between these two activities takes place when the tuple of values  $\langle v_1, \dots, v_n \rangle$ , resulting from the evaluation of the invoke argument, matches the template  $\langle w_1, \dots, w_n \rangle$  argument of the receive. This causes a substitution of the variables in the receive template (within the scope of variables declarations) with the corresponding values produced by the invoke.

Communication is asynchronous, as in KLAIM. This results from the syntactic constraints that invoke activities cannot be used as prefixes and choice can only be guarded by receive activities (as in *asynchronous*  $\pi$ -calculus [6]). Indeed, in service-oriented systems, communication is usually asynchronous, in the sense that (i) there may be an arbitrary delay between the sending and the receiving of a message, (ii) the order in which messages are received may differ from that in which they were sent (iii) a sender cannot determine if and when a sent message will be received.

The *empty* activity does nothing, while *choice* permits selecting for execution one between two alternative receives.

Execution of *parallel* services is interleaved. However, if more matching receives are ready to process a given invoke, only one of the receives that generate a substitution with smallest size (in terms of number of variable-value replacements) is allowed to progress (namely, execution of this receive takes precedence over that of the others). This mechanism permits to model the precedence of a service instance over the corresponding service specification when both of them can process the same request (see [57] for detailed examples), and enables a sort of blind-date conversation joining strategy [16].

*Delimitation* is the only binding construct:  $[e]s$  binds the element  $e$  in the scope  $s$ . According to its first argument, delimitation is used for three different purposes: (i) to regulate the range of application of substitutions produced by communication, when the delimited element is a variable; (ii) to generate fresh

names, when the delimited element is a name; (iii) to confine the effect of a kill activity, when the delimited element is a killer label. The scope of names can be dynamically extended, in order to model the communication of private names, as done with the restriction operator in  $\pi$ -calculus [52]. Instead, killer labels cannot be dynamically extended, because the activities whose termination would be forced by the execution of a kill need to be statically determined.

The *kill* activity forces immediate termination of all the concurrent activities not enclosed within the *protection* operator. To faithfully model fault and termination handling of SOC applications, kill activities are executed eagerly with respect to the communication activities enclosed within the delimitation of the corresponding killer label.

Finally, the *replication* construct  $*s$  permits to spawn in parallel as many copies of  $s$  as necessary. This, for example, is exploited to implement recursive behaviours and to model business process definitions, which can create multiple instances to serve several requests simultaneously.

### 3.3 Example: A Travel Agency Scenario

We report here a few examples aimed at illustrating the main COWS features. We consider a typical SOC scenario, where a travel agency exposes a service to automatically book a hotel and a flight according to customers' requests.

At a high level of abstraction, the travel agency service is rendered in COWS as:

$$\text{TravelAgency} \triangleq * [x_{cust}, x_{dates}, x_{dest}] p_{ta} \bullet o_{req} ? \langle x_{cust}, x_{dates}, x_{dest} \rangle . \\ x_{cust} \bullet o_{resp} ! \langle \text{book}(x_{dates}, x_{dest}) \rangle$$

The replication operator  $*$  is used here to specify that the service is *persistent*, i.e. capable of creating multiple instances to serve several requests simultaneously. The delimitation operator specifies the scope of the variables arguments of the subsequent receive activity on operation  $o_{req}$ , used to receive a request message from a customer. Besides dates and destination of the travel, this message contains the partner name that the customer will use to receive the response, which will be sent by the service by means of the invoke activity on operation  $o_{resp}$ . Booking of hotel and flight is here abstracted by the (unspecified) expression  $\text{book}(x_{dates}, x_{dest})$ .

A customer of the travel agency is specified as follows:

$$\text{Customer} \triangleq p_{ta} \bullet o_{req} ! \langle p_c, v_{dates}, v_{dest} \rangle \mid [x_{travel}] p_c \bullet o_{resp} ? \langle x_{travel} \rangle . s$$

The customer behaviour is specular to that of the travel agency: it starts with an invoke and then waits for a response message containing the travel data.

The overall specification of the scenario is simply the parallel composition of the two components:  $(\text{Customer} \mid \text{TravelAgency})$ . Whenever prompted by a client request, the travel agency service creates an instance to serve that specific request, and is immediately ready to concurrently serve other possible requests.

Therefore, the resulting COWS term after such a computational step is the following:

$$[x_{travel}] p_c \bullet o_{resp} ? \langle x_{travel} \rangle . s \mid TravelAgency \mid p_c \bullet o_{resp} ! \langle book(v_{dates}, v_{dest}) \rangle$$

The created service instance (highlighted by a grey background) is represented as a service running in parallel with the other terms. Notably, the variables of the invoke activity are instantiated (i.e., replaced) by the corresponding values exchanged in the communication. This invoke activity can now synchronise with the receive activity of the customer, whose execution will then continue as  $s$  with  $x_{travel}$  replaced by the value resulting from the evaluation of the  $book$  expression.

Let us now consider a more refined specification, where the role of the  $book$  expression is played by the interactions with services for flights and hotels searching:

$$TravelAgency' \triangleq * [x_{cust}, x_{dates}, x_{dest}] p_{ta} \bullet o_{req} ? \langle x_{cust}, x_{dates}, x_{dest} \rangle . \\ \begin{aligned} & [p, o, x_{flight}, x_{hotel}] \\ & ( (p_{flight} \bullet o_{book} ! \langle p_{ta}, x_{cust}, x_{dates}, x_{dest} \rangle \\ & \quad | p_{ta} \bullet o_{fRes} ? \langle x_{cust}, x_{dates}, x_{dest}, x_{flight} \rangle . (p \bullet o ! \langle end \rangle \mid s_f) ) \\ & \quad | (p_{hotel} \bullet o_{book} ! \langle p_{ta}, x_{cust}, x_{dates}, x_{dest} \rangle \\ & \quad \quad | p_{ta} \bullet o_{hRes} ? \langle x_{cust}, x_{dates}, x_{dest}, x_{hotel} \rangle . (p \bullet o ! \langle end \rangle \mid s_h) ) \\ & \quad | p \bullet o ? \langle end \rangle . p \bullet o ? \langle end \rangle . x_{cust} \bullet o_{resp} ! \langle x_{flight}, x_{hotel} \rangle ) \end{aligned}$$

After the reception of a customer request, the service contacts in parallel the two searching services (by invoking the operation  $o_{book}$ ). When the responses from both services are available, the travel agency service combines them and replies to the customer. To this aim, a private endpoint  $p \bullet o$  is exploited: the reception of a message from a searching service triggers an  $end$  signal (i.e., an internal message) along the private endpoint, and two of such signals are necessary to trigger the invoke the activity for replying to the customer. Notice that the scope of variable  $x_{flight}$  (resp.  $x_{hotel}$ ) includes not only the continuation  $s_f$  (resp.  $s_h$ ) of the service performing the receive, but also the activity for sending the response to the customer. This is different from most process calculi and accounts for easily expressing variables shared among parallel activities within the same service instance, which is a feature typically supported in SOC.

The behaviour of the above service is of particular interest when it is included in a scenario with multiple customers (the specifications of customers and searching services are omitted, we just assume that they follow the communication protocol established by the travel agency specification):

$$Customer_1 \mid Customer_2 \mid TravelAgency' \mid FlightBooking \mid HotelBooking$$

After a certain number of computational steps have taken place, we can obtain a system configuration where one instance of the travel agency service is created per each customer, and both instances have sent their requests to the searching services and are waiting for replies. Now, to send the values resulting from the processing of the request of the first customer, the flight searching service has to

perform an invoke activity of the form  $p_{ta} \bullet o_{fRes}! \langle p_{c1}, v_{dates}, v_{dest}, v_{flight} \rangle$ . However, the travel agency service has two instances waiting for such message along the endpoint  $p_{ta} \bullet o_{fRes}$ . In order to deliver the message to the proper instance, i.e. the one serving the request of the first customer, the *message correlation* mechanism is used. In fact, in SOC, it is up to each single message to provide a form of context that enables services to associate the message with the appropriate instance. This is achieved by embedding values, called *correlation data*, in the message itself. Pattern-matching is the mechanism used by the COWs's semantics for locating correlation data. In our example, these data are the customer's partner name, the travel dates and the destination, which have instantiated the corresponding variables in the receive activity  $p_{ta} \bullet o_{fRes}? \langle p_{c1}, v_{dates}, v_{dest}, x_{flight} \rangle$  within  $Customer_1$ . While the receive of the first customer is enabled, the one within the second customer instance is not, as it has been instantiated with unmatchable values.

Finally, let us provide further details of the travel agency specification, in order to add fault and compensation handling activities (highlighted by a grey background):

$$\begin{aligned}
 TravelAgency'' \triangleq & * [x_{cust}, x_{dates}, x_{dest}] p_{ta} \bullet o_{req}? \langle x_{cust}, x_{dates}, x_{dest} \rangle. \\
 & [p, o, x_{flight}, x_{hotel}, k] \\
 & ( (p_{flight} \bullet o_{book}! \langle p_{ta}, x_{cust}, x_{dates}, x_{dest} \rangle \\
 & \quad | p_{ta} \bullet o_{fRes}? \langle x_{cust}, x_{dates}, x_{dest}, x_{flight} \rangle. \\
 & \quad (p \bullet o! \langle end \rangle | s_f \\
 & \quad \quad | \{ \{ p \bullet o? \langle comp \rangle. p_{flight} \bullet o_{cancel}! \langle x_{cust}, x_{dates}, x_{dest} \rangle \} \} ) \\
 & \quad + p_{ta} \bullet o_{fFault}? \langle x_{cust}, x_{dates}, x_{dest} \rangle. \\
 & \quad (kill(k) | \{ \{ p \bullet o! \langle comp \rangle | p \bullet o! \langle fault \rangle \} \} ) \\
 & \quad | (p_{hotel} \bullet o_{book}! \langle p_{ta}, x_{cust}, x_{dates}, x_{dest} \rangle \\
 & \quad \quad | p_{ta} \bullet o_{hRes}? \langle x_{cust}, x_{dates}, x_{dest}, x_{hotel} \rangle. \\
 & \quad \quad (p \bullet o! \langle end \rangle | s_h \\
 & \quad \quad \quad | \{ \{ p \bullet o? \langle comp \rangle. p_{hotel} \bullet o_{cancel}! \langle x_{cust}, x_{dates}, x_{dest} \rangle \} \} ) \\
 & \quad + p_{ta} \bullet o_{hFault}? \langle x_{cust}, x_{dates}, x_{dest} \rangle. \\
 & \quad (kill(k) | \{ \{ p \bullet o! \langle comp \rangle | p \bullet o! \langle fault \rangle \} \} ) \\
 & \quad | p \bullet o? \langle end \rangle. p \bullet o? \langle end \rangle. x_{cust} \bullet o_{resp}! \langle x_{flight}, x_{hotel} \rangle \\
 & \quad | \{ \{ p \bullet o? \langle fault \rangle. x_{cust} \bullet o_{fault}! \langle \rangle \} \} ) )
 \end{aligned}$$

Now, when a positive response from a searching service is received, a compensation handler is installed. This consists of an invoke activity on operation  $o_{cancel}$ , triggered by a *comp* signal, devoted to cancel the booking. If a negative response on  $o_{fFault}$  (resp.  $o_{hFault}$ ) is received, the normal execution of the service is immediately terminated (by means of the **kill** activity), the activity compensating the hotel (resp. flight) booking is activated, if installed, and a *fault* signal is emitted. This last signal triggers the execution of the fault handler, consisting of an invoke activity for notifying the customer that the request booking is failed. Notably, fault and compensation activities are enclosed within protection blocks, in order to protect them from the killing effect of the **kill** activities.

### 3.4 Programming Environment

To effectively program SOC applications, COWS, originally conceived as a process calculus, has been extended with high-level features, such as standard control flow constructs (i.e., sequentialization, assignment, conditional choice, iteration) and a scope activity explicitly defining fault and compensation handlers. The implementation of the resulting orchestration language, called *Blite* [46], is based on a software tool [15] supporting a rapid and easy development of SOC applications via the translation of service orchestrations written in *Blite* into executable WS-BPEL programs. More specifically, a *Blite* program given as input to this tool also includes a declarative part, containing the variable types and the physical service bindings, necessary for generating the corresponding WSDL document and the process deployment descriptor. These files, together with the one containing the WS-BPEL code, are organised in a package that can be deployed and executed in a WS-BPEL engine.

### 3.5 Verification Techniques

The main verification techniques devised for COWS specifications are the following: (i) a type system for checking confidentiality properties [45], which uses types to express and enforce policies for regulating the exchange of data among services; (ii) a bisimulation-based observational semantics [58], which permits to check interchangeability of services and conformance against service specifications; (iii) a verification methodology for checking functional properties specific of SOC systems [33].

Concerning the third technique, the properties are described by means of SoCL, a logic specifically designed to express in a convenient way distinctive aspects of services, such as, e.g., acceptance of a request, provision of a response, and correlation among service requests and responses. The verification of SoCL formulae over COWS specifications is assisted by the on-the-fly model checker CMC. This approach has been used in [33,38,49] to verify some properties of interest of an automotive scenario, an e-Health authentication protocol, and a finance case study, respectively.

## 4 SCEL: Software Component Ensemble Language

Developing massively distributed and highly dynamic computing systems which interact with and control the physical world is a major challenge in today's software engineering. Difficulties arise from the open-ended and dynamic nature of large-scale systems, the non-deterministic and unpredictably changing external environment, the often limited or even impossible human intervention, and the need of ensembles of components to interact and collaborate for achieving specific goals, while hiding the complexity to end-users. A possible answer to the problems posed by such systems is to make them *self-aware*, by continuously monitoring their behaviour and their working environment, and able to

*self-adapt* their behaviour or structure, by selecting the actions to perform for dealing with the current status of affairs. These and other *self-management* capabilities, like self-configuration, self-healing, self-optimisation, and self-protection, characterise *autonomic computing* [43] systems.

SCEL (*Software Component Ensemble Language*, [22, 27]) is a formal language providing a set of linguistic abstractions for specifying the behaviour of (autonomic) components, the interaction among them, and the formation of their ensembles. In SCEL, components are computational entities that have assigned dedicated knowledge repositories and behavioural policies. They also have an interface exposing characterising attributes. Ensembles, in turn, are aggregations of interacting partner components dynamically determined by means of predicates validated by each component on the basis of its attributes.

SCEL linguistic abstractions support programming self- and context-awareness, adaptation and autonomicity. Indeed, through the knowledge repositories, components can gain information on their status (self-awareness) and environment (context awareness). By exploiting awareness and higher-order features (i.e. the capability to store/retrieve processes in/from components knowledge repositories and to dynamically activate new processes), components can trigger self-adaptation and/or initiate self-healing actions for reacting to faults or activate optimization strategies by, e.g., including or replacing processes and other components. By integrating SCEL with suitable policy languages, it is possible to guarantee self-protection against, e.g., unauthorized accesses or denial-of-service attacks.

## 4.1 Syntax

SCEL syntax is reported in Table 3. Five countable disjoint sets are used: the set of *names* (ranged over by  $n, n', \dots$ ), the set of *predicate names* (ranged over by  $p, \dots$ ), the set of *variables for names* (ranged over by  $x, x', \dots$ ), the set of *variables for processes* (ranged over by  $X, Y, \dots$ ), and the set of parameterised *process identifiers* (ranged over by  $A, \dots$ ). *self* is a distinguished variable standing for the name of a component.

SYSTEMS result from the aggregation of COMPONENTS which, in turn, result from the aggregation of KNOWLEDGE and PROCESSES, according to some POLICIES. PROCESSES specify the flow of the ACTIONS that can be performed. ACTIONS can have a TARGET to determine the other components, in addition to the subject one, that are involved in that action.

SCEL is parametric with respect to some syntactic categories, namely POLICIES, KNOWLEDGE, TEMPLATES and ITEMS (with the last two determining the part of KNOWLEDGE to be retrieved/removed or added, respectively). This choice permits integrating different approaches to policy specification and knowledge handling within SCEL, like, e.g., the *access control policies* of [48] and the *constraint stores* of [53]. A simple, yet expressive, instance of SCEL, named SCELIGHT, has been introduced in [29] where policies are absent (equivalently, any process action is authorised) and knowledge repositories are implemented as tuple spaces *à la* KLAIM.



**Table 3.** SCEL syntax (POLICIES  $\Pi$ , KNOWLEDGE  $\mathcal{K}$ , TEMPLATES  $T$ , and ITEMS  $t$  are parameters of the language)

SYSTEMS:		ACTIONS:	
$S ::= C$	(component)	$a ::= \mathbf{get}(T)@c$	(withdraw)
$S_1 \parallel S_2$	(composition)	$\mathbf{qry}(T)@c$	(retrieve)
$(\nu n)S$	(name restriction)	$\mathbf{put}(t)@c$	(addition)
COMPONENTS:		$\mathbf{fresh}(n)$	(scope)
$C ::= \mathcal{I}[\mathcal{K}, \Pi, P]$	(single component)	$\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$	(new)
PROCESSES:		TARGETS:	
$P ::= \mathbf{nil}$	(inert)	$c ::= n$	(name)
$a.P$	(action prefixing)	$x$	(variable)
$P_1 + P_2$	(choice)	$\mathbf{self}$	(self)
$P_1 \mid P_2$	(composition)	$\mathcal{P}$	(predicate)
$X$	(process variable)	$p$	(pred. name)
$A(\bar{p})$	(invocation)		

## 4.2 Informal Semantics

SYSTEMS aggregate COMPONENTS through the *composition* operator  $\_ \parallel \_$ , which is both commutative and associative. It is also possible to restrict the scope of a name, say  $n$ , by using the name *restriction* operator  $(\nu n)\_$ . In a system of the form  $S_1 \parallel (\nu n)S_2$ , the effect of the operator is to make name  $n$  invisible from  $S_1$ .

A COMPONENT  $\mathcal{I}[\mathcal{K}, \Pi, P]$  consists of:

- An *interface*  $\mathcal{I}$  publishing and making available information about the component itself in the form of *attributes*, i.e. names acting as references to information stored in the component’s knowledge repository. Among them, attribute *id* is mandatory and is bound to the name of the component.
- A *knowledge repository*  $\mathcal{K}$  managing both *application data* and *awareness data*, together with the specific handling mechanism. Application data are used for enabling the progress of components’ computations, while awareness data provide information about the environment in which the components are running (e.g. monitored data from sensors) or about the status of a component (e.g. its current location).
- A set of *policies*  $\Pi$  regulating the interaction between the different parts of a single component and the interaction between components.
- A *process*  $P$ , together with a set of process definitions that can be dynamically activated.

PROCESSES are the active computational units. Each process is built up from the *inert* process  $\mathbf{nil}$  via *action prefixing* ( $a.P$ ), nondeterministic *choice* ( $P_1 + P_2$ ), controlled *composition* ( $P_1 \mid P_2$ ), *process variable* ( $X$ ), and parameterised process *invocation* ( $A(\bar{p})$ ). The semantics of the construct  $P_1 \mid P_2$  is another parameter of SCEL. It can be instantiated so as to capture various forms of *parallel composition* commonly used in process calculi. For example, in SCELIGHT, it corresponds to the standard *interleaving* execution of the two involved processes.

Communication can be *higher-order*, as in KLAIM. We assume that  $A$  ranges over a set of parameterised *process identifiers* that are used in (possibly recursive) process definitions. We also assume that each process identifier  $A$  has a *single* definition of the form  $A(\bar{f}) \triangleq P$ . Lists of actual and formal parameters are denoted by  $\bar{p}$  and  $\bar{f}$ , respectively.

Processes can perform five different kinds of ACTIONS. Actions **get**( $T$ )@ $c$ , **qry**( $T$ )@ $c$  and **put**( $t$ )@ $c$  are used to manage shared knowledge repositories by *withdrawing/retrieving/adding* information items from/to the knowledge repository identified by target  $c$ . These actions exploit templates  $T$  as patterns to select knowledge items  $t$  in the repositories. They heavily depend on the chosen kind of knowledge repository (a parameter of SCEL, as we have already noticed) and are implemented by invoking the knowledge handlers it provides. Action **fresh**( $n$ ) introduces a *scope* restriction for the name  $n$  so that this name is guaranteed to be *fresh*, i.e., different from any other name previously used. Action **new**( $\mathcal{I}, \mathcal{K}, \Pi, P$ ) creates a *new* component  $\mathcal{I}[\mathcal{K}, \Pi, P]$ .

Action **get** may cause the process executing it to wait for the expected element, in case it is not (yet) available in the knowledge repository. Action **qry**, exactly like **get**, may suspend the process executing it if the knowledge repository does not (yet) contain or cannot “produce” the expected element. The two actions differ for the fact that **get** removes the found item from the knowledge repository while **qry** leaves the target repository unchanged. Actions **put**, **fresh** and **new** are instead immediately executed (provided that their execution is allowed by the policies in force).

Different entities may be used as the target  $c$  of an action. In addition to names and variables for names, the distinguished variable **self** can be used by processes to refer to the name of the component hosting them. The possible targets could be also singled out via a *predicate*  $\mathcal{P}$  (or the name  $p$  of a predicate). Predicates are boolean-valued expressions obtained by logically combining relations between attributes and value expressions. When the target of a communication action is a predicate, this predicate acts as a “guard” specifying the *ensemble* of all those components with which the process performing the action intends to interact. Thus, e.g., actions **put**( $t$ )@ $n$  and **put**( $t$ )@ $\mathcal{P}$  give rise to two different primitive forms of communication: the former is a *point-to-point* communication, while the latter is a sort of *group-oriented* communication.

It is worth noticing that the group-oriented variant of action **put** is used to insert a knowledge item in the repositories of *all* components belonging to the ensemble identified by the target predicate. Differently, the group-oriented variants of actions **get** and **qry** withdraw and retrieve, respectively, an item from a *single* component non-deterministically selected among those satisfying the target predicate.

### 4.3 Example: A Collection of Service Components

SCEL has proved to be suitable for modelling autonomic systems from different application scenarios such as, e.g., collective robotic systems [17, 27], cooperative e-vehicles [13], service provision and cloud-computing [22, 48, 50]. Here, we

consider a scenario, borrowed from [29] and modelled in SCELIGHT, consisting of  $m$  provider components  $\mathcal{I}_{p_j}[\mathcal{K}_{p_j}, A_{p_j}]$ , offering a variety of services, and  $n$  client components  $\mathcal{I}_{c_h}[\mathcal{K}_{c_h}, P_{c_h}]$ :

$$\mathcal{I}_{p_1}[\mathcal{K}_{p_1}, A_{p_1}] \parallel \dots \parallel \mathcal{I}_{p_m}[\mathcal{K}_{p_m}, A_{p_m}] \parallel \mathcal{I}_{c_1}[\mathcal{K}_{c_1}, P_{c_1}] \parallel \dots \parallel \mathcal{I}_{c_n}[\mathcal{K}_{c_n}, P_{c_n}].$$

Each service component manages and elaborates service requests with different requirements, roughly summarised by the following three quality levels: *gold*, *silver* and *base*. These levels are defined via a combination of predicates on the hardware configuration and the runtime state of the provider components. To this aim, we assume that attributes named *hw* and *load* are provided by each service component. The former can take an integer value from 0 to 10 that gives an indication of the capacity of the hardware configuration of the component, while the latter can take an integer value from 0 to 100 that estimates the actual computational load of the component. The three quality of service levels are then characterised by following predicates:

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

identifying, respectively, three ensembles of service components that

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

Each service component also stores in its knowledge repository a collection of items indicating the provided services, together with the component identifier. For example, the provider  $p_j$  offering the *factorial* service stores in its local repository the item  $\langle service, factorial, i_{p_j} \rangle$ . Note that including the identifier in the tuple publishing the service is fundamental as the group-oriented communication primitives are completely anonymous, i.e., the actual targets of a group-oriented communication action are not known to the subject.

Finally, each service component  $p_j$  runs the process  $A_{p_j}$  defined as:

$$\begin{aligned} A_{p_j} &\triangleq \mathbf{get}(invoke, factorial, ?x, ?y)@self. \\ &\quad \mathbf{get}(load, ?z)@self. \\ &\quad \mathbf{put}(load, (z + 20))@self. \\ &\quad (A_{p_j} \mid Q(x, y)) \end{aligned}$$

The process is triggered by a client request. Whenever this happens, the computational load is updated; we assume that each service instance uses 20% of the

server's capacity. Then, the *factorial* service becomes again ready to serve other client requests, and the process  $Q$ , which actually computes the result of the invoked service for the current request, is executed. We assume that, before its termination, process  $Q$  updates the value of attribute *load*, and puts the result of the computation in the repository of the client.

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

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

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

for some service level  $k$  in  $\{b, s, g\}$  and some argument  $v$  for the factorial function the client would like the server to execute. Intuitively, such process first searches among the components belonging to the ensemble identified by predicate  $P_k$ , via a **qry** action, an item matching the template  $(service, factorial, ?x)$ . In this way, by taking advantage of group-oriented communication, the client is able to dynamically identify a component  $x$  that provides the *factorial* service at the desired service level  $k$ . If more than one provider component meets these requirements, one of them will be non-deterministically selected. Then, via a **put** action, the process invokes the selected service, in a point-to-point fashion, by providing the actual parameter  $v$  of the request. After issuing the invocation, the process waits for the result (recall that action **get** is blocking). Whenever the result of the service invocation is made available, the process can withdraw it from the local repository and continue as process  $P'_{c_h}$ .

#### 4.4 Programming Environment

SCEL systems can be executed and simulated in jRESP<sup>2</sup> (Java Runtime Environment for SCEL Programs), which offers specific software tools to develop and support SCEL systems. In particular, jRESP provides an API that permits enriching Java programs with the SCEL's linguistic constructs. The API is instrumental to assist programmers in the implementation of autonomic systems, which thus turns out to be simplified with respect to using "pure" Java. Moreover, jRESP provides a set of classes enabling execution of *virtual components* on top of a simulation environment that can control component interactions and collect relevant simulation data.

#### 4.5 Verification Techniques

A prototype framework for statistical model-checking has been developed [30] by relying on the jRESP simulation environment. The tool is parameterised with

<sup>2</sup> jRESP website: <http://jresp.sourceforge.net/>.

respect to a given *tolerance*  $\varepsilon$  and *error probability*  $p$ , thus allowing one to verify whether the implementation of a system satisfies a given property with a certain degree of confidence. The underlying randomised algorithm guarantees that the difference between the computed value and the exact one is greater than  $\varepsilon$  with a probability that is less than  $p$ .

Qualitative properties of SCELIGHT specifications have been verified through the Spin model checker [42]. The verification relies on a preliminary translation from SCELIGHT into Promela, i.e., the input language of Spin. This approach has been used in, e.g., [29] to verify some properties of interest of the application scenario illustrated in Sect. 4.3, like absence of deadlock, server overload and responsiveness, and in [30] to verify similar properties for a swarm robotics scenario.

SCEL's operational semantics has also been implemented by using the Maude framework [18]. The outcome, named MISSCEL (Maude Interpreter and Simulator for SCEL), focuses on SCELIGHT and exploits the rich Maude toolset to perform, among other things, qualitative analysis via Maude's invariant and LTL model checkers, and statistical model checking via MULTIVESTA [59] (as done in [10] for a robotic collision avoidance scenario). A further advantage of MISSCEL is that SCEL specifications can be intertwined with (very expressive) raw Maude code. This permits to obtain sophisticated specifications in which SCEL is used to model behaviours, aggregations, and knowledge handling, while scenario-specific details are specified with Maude.

## 5 AbC: Attribute-Based communication

Collective-Adaptive Systems (CAS) [35] are new emerging computational systems, consisting of a massive number of components, featuring complex interaction mechanisms. These systems are usually distributed, heterogeneous, decentralised and interdependent, and are operating in dynamic and often unpredictable environments. CAS components combine their behaviours, by forming collectives, to achieve specific goals depending on their attributes, objectives, and functionalities. CAS are inherently scalable and their boundaries are fluid in the sense that components may enter or leave the collective at any time; so they need to dynamically adapt to their environmental conditions and contextual data.

AbC (Attribute-based Communication calculus, [2,4]) is a process calculus specifically designed to deal with CAS. It has been heavily inspired by SCEL, but has been designed to reduce complexity and keep the set of linguistic primitives to a minimum. Indeed, it was originally designed as a trimmed version of SCEL that was obtained by ignoring the parts relative to policies and knowledge and concentrating only on behaviours and interfaces. In this respect, AbC has similar aims to SCELIGHT, but the underlying communication paradigm is very different; explicit message passing for the former and shared memory *à la* KLAIM for the latter.

Indeed, the original aim of AbC was to assess the impact of the new message passing paradigm based on attributes and compare it with more classical

ones that handle the interaction between distributed components by relying on identities (Actors [5]), or channels ( $\pi$ -calculus), or broadcast (B- $\pi$ -calculus [55]). In all these formalisms, messages exchanges rely on names or addresses of the involved components and are independent of their status and capabilities. This makes it hard to program, coordinate, and adapt complex behaviours that highly depend on run-time changes of components.

In AbC, the attribute-based system is however more than just the parallel composition of interacting partners; it is also parametrised with respect to the environment or the space where system components are executed. The environment has a great impact on how components behave and provides a new means of indirect communication, that allows components to mutually influence each other, possibly unintentionally.

## 5.1 Syntax

Table 4 contains the syntax of AbC. The top-level entities of the calculus are COMPONENTS. A component,  $\Gamma;_I P$ , is a process  $P$  associated with an attribute environment  $\Gamma$ , and an interface  $I$ . The *attribute environment* provides a collection of attributes whose values represent the status of the component and influence its run-time behaviour. Formally,  $\Gamma: \mathcal{A} \rightarrow \mathcal{V}$  is a partial map from attribute identifiers ( $a \in \mathcal{A}$ ) to values ( $v \in \mathcal{V}$ ), i.e., to numbers, strings, tuples, ... The *interface*  $I \subseteq \mathcal{A}$  contains the *public attributes* of a component (the attributes in  $\text{dom}(\Gamma) - I$  being *private*). Composed components  $C_1 \parallel C_2$  are built by using the parallel operator.

**Table 4.** The syntax of the AbC calculus

COMPONENTS:		PREDICATES:	
$C ::= \Gamma;_I P$	(component)	$\Pi ::= \text{true}$	(true)
$C_1 \parallel C_2$	(composition)	$\text{false}$	(false)
PROCESSES:		$p(\vec{E})$	(atomic predicate)
$P ::= 0$	(inaction)	$\Pi_1 \wedge \Pi_2$	(conjunction)
$\Pi(\vec{x}).U$	(attribute-based input)	$\Pi_1 \vee \Pi_2$	(disjunction)
$(\vec{E})@ \Pi.U$	(attribute-based output)	$\neg \Pi$	(negation)
$\langle \Pi \rangle P$	(context awareness)	EXPRESSIONS:	
$P_1 + P_2$	(choice)	$E ::= v$	(value)
$P_1   P_2$	(parallel composition)	$x$	(variable)
$K$	(process identifier)	$a$	(attribute identifier)
UPDATES:		$\text{this}.a$	(local reference)
$U ::= [a := E]U$	(attribute update)	$op(\vec{E})$	(operator)
$P$	(process)		

A PROCESS  $P$  can be: the *inactive* process 0; an *action-prefixed* process,  $\text{act}.U$ , where  $\text{act}$  is a communication action and the UPDATE  $U$  is a process

possibly preceded by *attribute updates*; a *context aware* process,  $\langle II \rangle P$ , where  $II$  is a PREDICATE built from boolean constants and from atomic predicates, based on EXPRESSIONS over attributes, by using standard boolean operators; a *nondeterministic choice* between two processes,  $P_1 + P_2$ ; a *parallel composition* of two processes,  $P_1 | P_2$ ; or a process call with an identifier  $K$  used in a unique process definition  $K \triangleq P$ .

## 5.2 Informal Semantics

Attribute-based actions for sending and receiving messages permit to establish communication links between different components according to specific predicates over their attributes.

Specifically, *attribute-based output*  $(\tilde{E})@II$  sends the result of the evaluation of the sequence of expressions  $\tilde{E}$  to the components whose attributes satisfy the predicate  $II$ . Notably, together with the computed values, also the portion of the attribute environment of the sending component that can be perceived by the context is sent; this is obtained from the local environment by limiting its domain to the attributes in the component interface. This information is needed to allow receivers to determine whether they are interested in the sent message.

Instead, *attribute-based input*  $II(\tilde{x})$  specifies receipt of messages from a component satisfying predicate  $II$ ; the sequence  $\tilde{x}$  acts as a placeholder for received values. A message can be received when two *communication constraints* are satisfied: the public local attribute environment satisfies the predicate used by the sender to identify potential receivers, and the sender environment satisfies the receiving predicate. In this case, attribute updates are performed under the generated substitution. An *attribute update*  $[a := E]$  assigns the value of  $E$  to the attribute identifier  $a$ . This action is used to change the values of the attributes according to contextual conditions and to adapt component's behaviour. Notice that the execution of a communication action and the following update(s) is atomic.

The *awareness construct*  $\langle II \rangle P$  blocks the execution of  $P$  until predicate  $II$  is satisfied when using the local attribute environment, possibly after a change of state by a component. This construct permits to collect awareness data and take decisions based on the changes in the attribute environment.

## 5.3 Example: A TV Broadcaster Scenario

We now illustrate the features of AbC by considering a simple scenario borrowed from the paper where AbC was originally introduced [4]. In this scenario, we consider a TV broadcaster (e.g., CNN) represented by the process CNN, and two receivers represented by the processes RcvA and RcvB:

$$\begin{aligned}
\text{CNN} &\triangleq (v_s)@II_{\text{sport}}.\text{CNN} + (v_n)@II_{\text{news}}.\text{CNN} \\
&\quad + ()@false.[\text{Qbrd} := \text{LD}]\text{CNN} + ()@false.[\text{Qbrd} := \text{HD}]\text{CNN} \\
\text{RcvA} &\triangleq (\text{Qbrd} = \text{HD})(x).\text{RcvA} \\
&\quad + ()@false.[\text{Genre} := \text{Sport}]\text{RcvA} + ()@false.[\text{Genre} := \text{News}]\text{RcvA} \\
\text{RcvB} &\triangleq (\text{true})(x).\text{RcvB} + \dots
\end{aligned}$$

where

$$\begin{aligned}
II_{\text{sport}} &= (\text{Genre} = \text{Sport}) \wedge (\text{CNN-Sub} = \text{tt}) \\
II_{\text{news}} &= (\text{Genre} = \text{News})
\end{aligned}$$

The overall system is expressed as the parallel composition below, where the dots refer to other possible broadcasters or receivers:

$$\Gamma_{\text{cnn}}:C \text{ CNN} \mid \Gamma_a:A \text{ RcvA} \mid \dots \mid \Gamma_b:B \text{ RcvB}.$$

CNN periodically broadcasts Sport or News and targets different groups of receivers based on the predicates  $II_{\text{sport}}$  and  $II_{\text{news}}$ .  $II_{\text{sport}}$  targets the group of receivers who want to watch Sport ( $\text{Genre} = \text{Sport}$ ) provided that those receivers have subscribed to CNN ( $\text{CNN-Sub} = \text{tt}$ ). On the other hand,  $II_{\text{news}}$  targets the group of receivers who want to watch News ( $\text{Genre} = \text{News}$ ).

The quality of the broadcasted multimedia varies according to different factors (e.g., low bandwidth). CNN channel non-deterministically chooses to broadcast low-definition ( $[\text{Qbrd} := \text{LD}]$ ) or high-definition ( $[\text{Qbrd} := \text{HD}]$ ) multimedia. The receiving processes RcvA and RcvB almost have the same behaviour except that RcvA is only interested in high quality broadcasts while RcvB is willing to accept broadcasts of any quality. So they either accept the broadcast that their attributes in  $\Gamma_a$  and  $\Gamma_b$  satisfy, or change the genre.

A fragment of the possible interactions in this scenario is reported below; we use  $\longrightarrow_b$  and  $\longrightarrow_\tau$  to denote the computational steps induced by a broadcast action and by an attribute update action, respectively, and also use the grey-shaded box to indicate the components involved in the evolution.

$$\begin{aligned}
&\Gamma_{\text{cnn}}:C \text{ CNN} \mid \Gamma_a:A \text{ RcvA} \mid \dots \mid \Gamma_b:B \text{ RcvB} \\
\longrightarrow_b &\quad \boxed{\Gamma_{\text{cnn}}:C \text{ CNN}} \mid \boxed{\Gamma_a:A \text{ RcvA}} \mid \dots \mid \boxed{\Gamma_b:B \text{ RcvB}} \\
&\quad \vdots \\
\longrightarrow_\tau &\quad \boxed{\Gamma_{\text{cnn}}[\text{Qbrd} \mapsto \text{LD}]:C \text{ CNN}} \mid \Gamma_a:A \text{ RcvA} \mid \dots \mid \Gamma_b:B \text{ RcvB} \\
\longrightarrow_b &\quad \boxed{\Gamma'_{\text{cnn}}:C \text{ CNN}} \mid \Gamma_a:A \text{ RcvA} \mid \dots \mid \Gamma_b:B \text{ RcvB}
\end{aligned}$$

We assume that the initial attribute environments  $\Gamma_{\text{cnn}}$ ,  $\Gamma_a$  and  $\Gamma_b$  are:  $\Gamma_{\text{cnn}} = \{(\text{Qbrd}, \text{HD}), \dots\}$ ,  $\Gamma_a = \{(\text{Genre}, \text{News}), \dots\}$  and  $\Gamma_b = \{(\text{Genre}, \text{News}), \dots\}$ .

The interfaces of CNN, RcvA, and RcvB are defined as follows:  $C = \{\text{Qbrd}\}$  and  $A = B = \{\text{Genre}\}$ . Assume also that CNN initiates the interaction by broadcasting high quality News. As shown above, both RcvA and RcvB can join the collective and receive the broadcast because their attributes satisfy the condition of the broadcast (based on predicate  $II_{\text{news}}$ ). After a while CNN chooses



to lower the quality of multimedia (indeed, its environment is updated with  $Q_{brd} \mapsto LD$ ) to cope with some situations, such as low bandwidth, and **CNN** can evolve independently. Finally, **CNN** continues broadcasting News and in this case **RcvA** chooses to leave the collective because the quality of the broadcast does not satisfy its receiving predicate, while **RcvB** stays because it has no requirement for the input quality.

## 5.4 Programming Environment

Basing the interaction on the values of run-time attributes is indeed a nice idea, but it needs to be supported by a middleware that provides efficient ways for distributing messages, checking attribute values, and updating them. A typical approach is to rely on a centralised broker that keeps track of all components, intercepts every message and forwards it to registered components. It is then the responsibility of each component to decide whether to receive or discard the message. This is the solution proposed in [3] where a Java implementation of **AbC** is provided, that however suffers of serious performance problems. Two additional implementations of **AbC** have thus been considered, which are built on the top of two well-established programming languages largely used for concurrent programming, namely Erlang and Go, guaranteeing better scalability. The two implementations are called *AErlang*, for Attribute based Erlang, and *GoAt*, for Go with attributes.

*AErlang* [21] is a middleware enabling attribute-based communication among programs in Erlang [32], a concurrent functional programming language originally designed for building telecommunication systems and recently successfully adapted to broader contexts, such as large-scale distributed messaging platforms like Facebook and WhatsApp. *AErlang* lifts Erlang's send and receive communication primitives to attribute-based reasoning. In Erlang, the send primitive requires an explicit destination address while in *AErlang* processes are not aware of the presence and identity of each other, and communicate using predicates over attributes. *AErlang* has two main components: (i) a process registry that keeps track of process details, such as the process identifier and the current status, and (ii) a message broker that undertakes the delivery of outgoing messages. The *Process registry* is a generic server that accepts requests regarding process (un)registration and internal updates. It stores process identifiers and all the information used by the message broker to deliver messages. The *Message broker* is responsible for delivering messages between processes. It is implemented as an Erlang server process listening for interactions from attribute-based send. To address potential bottlenecks arising in the presence of a very large number of processes, the message broker can be set up to run in multiple parallel threads. Like the Java implementation for **AbC** presented in [3], the message broker is still centralised, however, to avoid broadcasts, the broker has an attribute registry where components register their attribute values and the broker is now

responsible for message filtering. Different distribution policies have been implemented that can be used by taking into account dynamicity of attributes and of predicates.

*GoAt*<sup>3</sup> extends Go [39], the language introduced by Google to handle massive computation clusters, and to make working in these environments more productive. Go has an intuitive and lightweight concurrency model with a well-understood semantics and extends the CSP model [41] with channel mobility, like in  $\pi$ -calculus. It also supports buffered channels, to provide mailboxes *à la* Erlang. The *Attribute-based API* for Go offers the possibility of using the AbC primitives to program the interaction of CAS applications directly in Go. The actual implementation faithfully models the formal semantics of AbC and it is parametric with respect to the infrastructure that mediates interactions. The *GoAt* API offers the possibility of using three different distributed coordination infrastructures for message exchange, namely cluster, ring, and tree. For all three infrastructures, it has been proved that the message delivery ordering is the same as the one required by the original formal semantics of AbC [1]. An Eclipse plugin permits programming in a high-level syntax, which can be analysed via formal methods by relying on the operational semantics of AbC. Once the code has been analysed, the *GoAt* plugin will generate formally verifiable Go code. Examples available from *GoAt*'s site permit to appreciate how intuitive it is to program a complex variant of the well-known problem of Stable Allocation in Content Delivery Network [47].

## 5.5 Verification Techniques

Some work has now started to verify properties of AbC programs. On the one hand, it is under investigation the use of the generic tools that have been designed for verifying properties of Erlang and Go programs. On the other hand, tools are under development to prove directly properties of the AbC specifications. The second alternative is under consideration because in some cases the correspondence between the actual AbC specifications and the running programs may not be immediate, and the difference would reduce the effectiveness of the effort.

A novel approach to the analysis of concurrent systems modelled as AbC terms has been introduced in [20]. It relies on the UMC model checker, a tool based on modelling concurrent systems as communicating UML-like state machines [61]. A structural translation from AbC specifications to the UMC internal format is used as the basis for program analysis. This permits identifying emerging properties of systems and unwanted behaviours.

Recent work considers a variant of AbC and proposes a technique to prove properties of the system by translating the specifications into symbolic C programs to be analysed with SAT-based approaches.

---

<sup>3</sup> *GoAt* codes and examples can be retrieved from <https://giulio-garbi.github.io/goat/>.

## 6 Concluding Remarks

This paper surveyed four domain-specific coordination languages supporting the engineering of different classes of modern distributed systems. These languages have been developed in the last twenty years by the authors (three of which have been working for quite a while in the Concurrency and Mobility Group at University of Florence) and other collaborators. Within the coordination community other research groups have followed a similar methodology, however relying on different specification models, e.g. coalgebras [7], actors [60] or automata [8], rather than process algebras.

Below, we summarise the programming abstractions introduced with the different formalisms and the lessons learned when designing and using languages for

1. Network-Aware Programming,
2. Service-Oriented Computing,
3. Autonomic Computing,
4. Collective Adaptive Systems Programming.

The design of KLAIM has shown that network awareness in distributed systems can be achieved by the explicit use of localities as first-order citizens of the language. Localities, indeed, identify network nodes, where computation takes place and data is stored. Network awareness relies on the notion of (multiple) tuple spaces, which can be accessed via a unique interface to insert and retrieve data. Communication is thus asynchronous, anonymous and associative, pattern matching plays a crucial role and guarantees high expressive power. Network awareness also supports computation mobility, thus paving the way for different kinds of optimisations.

From COWS, we learnt that SOC applications typically abstract from the structure of the underlying network and from distribution of data, which become transparent to the programmer. Pattern-matching still plays a key role in supporting communication, as it is at the basis of the message correlation mechanism. Novel distinguishing features are service persistence, state sharing among concurrent service instances, and service fault and termination handling. We have shown that the modelling of the first one can rely on the standard process replication operator. Instead, the modelling of the second one relies on the combined use of suitable binder operators, and non-standard receive activities binding neither names nor variables. Similarly, the modelling of the third feature requires a combination of some ingenious constructs to either force termination or protect activities in case of termination of other processes.

In SCAL the central notion is that of ensemble of components, which can be dynamically created in an opportunistic and transparent way. Indeed, the formation of an ensemble and the establishment of interactions among its members rely on the information exposed as attributes in the interface of the involved components. This enables an effective group-oriented communication model. Ensemble components are equipped with knowledge repositories that generalise KLAIM's tuple spaces by supporting different knowledge representations and handling

mechanisms. Self- and context-awareness make these components capable to adapt their behaviour to evolving needs and environmental changes.

Finally, AbC refines the group-oriented communication model of SCEL, in order to convey in a distilled form the attribute-based communication paradigm exploited to model and program Collective Adaptive Systems. The result of this synthesis effort is a compact calculus, suitable for studying the theoretical impact of the novel communication paradigm and for obtaining new programming frameworks by the new paradigm in different well-established programming languages, such as Java, Erlang and Go.

To recap, we think that the engineering methodology we presented, as witnessed by the four instantiations we have illustrated, provides a uniform linguistic approach, based on formal methods techniques, for ensuring the trustworthiness of the considered classes of systems and possibly of the other ones that will emerge in the near future. In this respect, we plan to consider the Aggregate Programming [9] domain, where the abstraction level in designing distributed systems further increases. In such an engineering approach, data and devices are aggregated via ‘under-the-hood’ coordination mechanisms. Although these aggregations resemble the notions of ensemble and collectives discussed in this paper, they mainly focus on distributed computation rather than on communication mechanisms.

As a final disclaimer we would like to say that obviously a section dedicated to related work is missing. Given the time span and the different programming domains covered by the development of our four languages, it would have needed a paper on his own. Thus the only thing we can do is to refer the interested reader to the bibliography sections of the papers that have introduced and developed KLAIM, COWS, SCEL and AbC. Moreover, for references on network-aware programming and relation with KLAIM we refer to [56], for service-oriented computing and COWS to [63] and for autonomic computing and SCEL to [64].

**Acknowledgements.** This work would not have been possible without the contribution of our collaborators that have helped us in shaping the four languages we have introduced. They are too many to be listed, but their names could be inferred from the bibliography below. However, we would like to make an exception and explicitly thank Michele Loreti. Michele has been a driving force for most of the results we have presented, he is not among the authors only because he is one of the PC chairs of the conference to which the work was submitted.

## References

1. Abd Alrahman, Y., De Nicola, R., Garbi, G., Loreti, M.: A distributed coordination infrastructure for attribute-based interaction. In: FORTE, LNCS. Springer (2018, to appear)
2. Abd Alrahman, Y., De Nicola, R., Loreti, M.: On the power of attribute-based communication. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 1–18. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-39570-8\\_1](https://doi.org/10.1007/978-3-319-39570-8_1). Full technical report can be found on <http://arxiv.org/abs/1602.05635>

3. Abd Alrahman, Y., De Nicola, R., Loreti, M.: Programming of CAS systems by relying on attribute-based communication. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 539–553. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-47166-2\\_38](https://doi.org/10.1007/978-3-319-47166-2_38)
4. Abd Alrahman, Y., et al.: A calculus for attribute-based communication. In: SAC 2015, pp. 1840–1845. ACM (2015)
5. Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge (1986)
6. Amadio, R., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-Calculus. *Theor. Comput. Sci.* **195**(2), 291–324 (1998)
7. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2002. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003). [https://doi.org/10.1007/978-3-540-40020-2\\_2](https://doi.org/10.1007/978-3-540-40020-2_2)
8. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.* **61**(2), 75–113 (2006)
9. Beal, J., Viroli, M.: Aggregate programming: from foundations to applications. In: Bernardo, M., De Nicola, R., Hillston, J. (eds.) SFM 2016. LNCS, vol. 9700, pp. 233–260. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-34096-8\\_8](https://doi.org/10.1007/978-3-319-34096-8_8)
10. Belzner, L., De Nicola, R., Vandin, A., Wirsing, M.: Reasoning (on) service component ensembles in rewriting logic. In: Iida, S., Meseguer, J., Ogata, K. (eds.) *Specification, Algebra, and Software*. LNCS, vol. 8373, pp. 188–211. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-642-54624-2\\_10](https://doi.org/10.1007/978-3-642-54624-2_10)
11. Bettini, L., De Nicola, R., Ferrari, G., Pugliese, R.: Interactive mobile agents in X-Klaim. In: WETICE, pp. 110–115. IEEE Computer Society Press (1998)
12. Bettini, L., De Nicola, R., Pugliese, R.: KLAVA: a Java package for distributed and mobile applications. *Softw. Pract. Experience* **32**(14), 1365–1394 (2002)
13. Bures, T., et al.: A life cycle for the development of autonomic systems: the e-mobility showcase. In: SASOW, pp. 71–76. IEEE (2013)
14. Castellani, S., Ciancarini, P., Rossi, D.: The ShaPE of ShaDE: a coordination system. Technical report UBLCS 96–5, Dip. di Scienze dell’Informazione, Univ. Bologna (1996)
15. Cesari, L., Pugliese, R., Tiezzi, F.: A tool for rapid development of WS-BPEL applications. *SIGAPP Appl. Comput. Rev.* **11**(1), 27–40 (2010)
16. Cesari, L., Pugliese, R., Tiezzi, F.: Blind-date conversation joining. *Serv. Oriented Comput. Appl.* **11**(3), 265–283 (2017)
17. Cesari, L., De Nicola, R., Pugliese, R., Puviani, M., Tiezzi, F., Zambonelli, F.: Formalising adaptation patterns for autonomic ensembles. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) FACS 2013. LNCS, vol. 8348, pp. 100–118. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-07602-7\\_8](https://doi.org/10.1007/978-3-319-07602-7_8)
18. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: *All About Maude - A High-Performance Logical Framework*. LNCS, vol. 4350. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-3-540-71999-1>
19. Davies, N., Wade, S.P., Friday, A., Blair, G.S.: L<sup>2</sup>imbo: a tuple space based platform for adaptive mobile applications. In: Rolia, J., Slonim, J., Botsford, J. (eds.) ICODP/ICDP. IFIPAICT, pp. 291–302. Springer, Boston (1997). [https://doi.org/10.1007/978-0-387-35188-9\\_22](https://doi.org/10.1007/978-0-387-35188-9_22)
20. De Nicola, R., Duong, T., Inverso, O., Mazzanti, F.: Verifying properties of systems relying on attribute-based communication. In: Katoen, J.-P., Langerak, R., Rensink, A. (eds.) *ModelEd, TestEd, TrustEd*. LNCS, vol. 10500, pp. 169–190. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-68270-9\\_9](https://doi.org/10.1007/978-3-319-68270-9_9)

21. De Nicola, R., Duong, T., Inverso, O., Trubiani, C.: A Erlang: empowering erlang with attribute-based communication. In: Jacquet, J.-M., Massink, M. (eds.) COORDINATION 2017. LNCS, vol. 10319, pp. 21–39. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-59746-1\\_2](https://doi.org/10.1007/978-3-319-59746-1_2)
22. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: A language-based approach to autonomic computing. In: Beckert, B., Damiani, F., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 25–48. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-35887-6\\_2](https://doi.org/10.1007/978-3-642-35887-6_2)
23. De Nicola, R., Ferrari, G.L., Pugliese, R.: Klaim: a kernel language for agents interaction and mobility. *IEEE Trans. Software Eng.* **24**(5), 315–330 (1998)
24. De Nicola, R., Ferrari, G.L., Pugliese, R., Venneri, B.: Types for access control. *Theor. Comput. Sci.* **240**(1), 215–254 (2000)
25. De Nicola, R., Gorla, D., Pugliese, R.: Confining data and processes in global computing applications. *Sci. Comput. Program.* **63**(1), 57–87 (2006)
26. De Nicola, R., Loreti, M.: A modal logic for mobile agents. *ACM Trans. Comput. Log.* **5**(1), 79–128 (2004)
27. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: the SCEL language. *TAAS* **9**(2), 7 (2014)
28. De Nicola, R., et al.: From flow logic to static type systems for coordination languages. *Sci. Comput. Program.* **75**(6), 376–397 (2010)
29. De Nicola, R., Lluch Lafuente, A., Loreti, M., Morichetta, A., Pugliese, R., Senni, V., Tiezzi, F.: Programming and verifying component ensembles. In: Bensalem, S., Lakhneq, Y., Legay, A. (eds.) ETAPS 2014. LNCS, vol. 8415, pp. 69–83. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-642-54848-2\\_5](https://doi.org/10.1007/978-3-642-54848-2_5)
30. De Nicola, R., Latella, D., Lafuente, A.L., Loreti, M., Margheri, A., Massink, M., Morichetta, A., Pugliese, R., Tiezzi, F., Vandin, A.: The SCEL language: design, implementation, verification. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) *Software Engineering for Collective Autonomic Systems*. LNCS, vol. 8998, pp. 3–71. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-16310-9\\_1](https://doi.org/10.1007/978-3-319-16310-9_1)
31. Deugo, D.: Choosing a mobile agent messaging model. In: ISADS, pp. 278–286. IEEE (2001)
32. Ericsson Computer Science Laboratory. The Erlang programming language. <https://www.erlang.org/>. Accessed 12 Apr 2018
33. Fantechi, A., et al.: A logical verification methodology for service-oriented computing. *ACM Trans. Softw. Eng. Methodol.* **21**(3), 16:1–16:46 (2012)
34. Ferrari, G.L., Moggi, E., Pugliese, R.: Metaklaim: a type safe multi-stage language for global computing. *Math. Struct. Comput. Sci.* **14**(3), 367–395 (2004)
35. Ferscha, A.: Collective adaptive systems. In: *UbiComp/ISWC*, pp. 893–895. ACM (2015)
36. Gelernter, D.: Generative communication in linda. *ACM Trans. Program. Lang. Syst.* **7**(1), 80–112 (1985)
37. Gelernter, D.: Multiple tuple spaces in Linda. In: Odijk, E., Rem, M., Syre, J.-C. (eds.) *PARLE 1989*. LNCS, vol. 366, pp. 20–27. Springer, Heidelberg (1989). [https://doi.org/10.1007/3-540-51285-3\\_30](https://doi.org/10.1007/3-540-51285-3_30)
38. Gnesi, S., Pugliese, R., Tiezzi, F.: The SENSORIA approach applied to the finance case study. In: Wirsing, M., Hölzl, M. (eds.) *Rigorous Software Engineering for Service-Oriented Systems*. LNCS, vol. 6582, pp. 698–718. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20401-2\\_34](https://doi.org/10.1007/978-3-642-20401-2_34)
39. Google. The Go programming language. <https://golang.org/doc/>. Accessed 20 Feb 2018

40. Gorla, D., Pugliese, R.: Dynamic management of capabilities in a network aware coordination language. *J. Log. Algebr. Program.* **78**(8), 665–689 (2009)
41. Hoare, C.A.R.: Communicating sequential processes. *Commun. ACM* **21**(8), 666–677 (1978)
42. Holzmann, G.J.: The model checker SPIN. *IEEE Trans. Softw. Eng.* **23**(5), 279–295 (1997)
43. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *Computer* **36**, 41–50 (2003)
44. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola, R. (ed.) *ESOP 2007*. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-71316-6\\_4](https://doi.org/10.1007/978-3-540-71316-6_4)
45. Lapadula, A., Pugliese, R., Tiezzi, F.: Regulating data exchange in service oriented applications. In: Arbab, F., Sirjani, M. (eds.) *FSEN 2007*. LNCS, vol. 4767, pp. 223–239. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-75698-9\\_15](https://doi.org/10.1007/978-3-540-75698-9_15)
46. Lapadula, A., Pugliese, R., Tiezzi, F.: Using formal methods to develop WS-BPEL applications. *Sci. Comput. Program.* **77**(3), 189–213 (2012)
47. Maggs, B.M., Sitaraman, R.K.: Algorithmic nuggets in content delivery. *SIGCOMM Comput. Commun. Rev.* **45**(3), 52–66 (2015)
48. Margheri, A., Pugliese, R., Tiezzi, F.: Linguistic abstractions for programming and policing autonomic computing systems. In: *UIC/ATC*, pp. 404–409. IEEE (2013)
49. Masi, M., Pugliese, R., Tiezzi, F.: On secure implementation of an IHE XUA-based protocol for authenticating healthcare professionals. In: Prakash, A., Sen Gupta, I. (eds.) *ICISS 2009*. LNCS, vol. 5905, pp. 55–70. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-10772-6\\_6](https://doi.org/10.1007/978-3-642-10772-6_6)
50. Mayer, P., et al.: The autonomic cloud: a vision of voluntary, peer-2-peer cloud computing. In: *SASOW*, pp. 89–94. IEEE (2013)
51. Merro, M., Sangiorgi, D.: On asynchrony in name-passing calculi. *Math. Struct. Comput. Sci.* **14**(5), 715–767 (2004)
52. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes I and II. *Inf. Comput.* **100**(1), 1–40, 41–77 (1992)
53. Montanari, U., Pugliese, R., Tiezzi, F.: Programming autonomic systems with multiple constraint stores. In: De Nicola, R., Hennicker, R. (eds.) *Software, Services, and Systems*. LNCS, vol. 8950, pp. 641–661. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-15545-6\\_36](https://doi.org/10.1007/978-3-319-15545-6_36)
54. OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007
55. Prasad, K.V.S.: A calculus of broadcasting systems. *Sci. Comput. Program.* **25**(2–3), 285–327 (1995)
56. Priami, C., Quaglia, P. (eds.): *GC 2004*. LNCS, vol. 3267. Springer, Heidelberg (2005). <https://doi.org/10.1007/b103251>
57. Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. *J. Appl. Logic* **10**(1), 2–31 (2012)
58. Pugliese, R., Tiezzi, F., Yoshida, N.: On observing dynamic prioritised actions in SOC. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikolettseas, S., Thomas, W. (eds.) *ICALP 2009*. LNCS, vol. 5556, pp. 558–570. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-02930-1\\_46](https://doi.org/10.1007/978-3-642-02930-1_46)
59. Sebastio, S., Vandin, A.: MultiVeStA: statistical model checking for discrete event simulators. In: *ValueTools*, pp. 310–315. ICST/ACM (2013)
60. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using rebecca. *Fundamenta Informaticae* **63**(4), 385–410 (2004)

61. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. *Sci. Comput. Program.* **76**(2), 119–135 (2011)
62. W3C. Web services activity. <https://www.w3.org/2002/ws/>. Accessed 20 Feb 2018
63. Wirsing, M., Hölzl, M. (eds.): *Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*. LNCS, vol. 6582. Springer, Heidelberg (2011). <https://doi.org/10.1007/978-3-642-20401-2>
64. Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.): *Software Engineering for Collective Autonomic Systems - The ASCENS Approach*. LNCS, vol. 8998. Springer, Cham (2015). <https://doi.org/10.1007/978-3-319-16310-9>





# Rule-Based Form for Stream Constraints

Kasper Dokter<sup>(✉)</sup> and Farhad Arbab

Centrum Wiskunde & Informatica, Amsterdam, Netherlands

K.P.C.Dokter@cwi.nl

**Abstract.** Constraint automata specify protocols as labeled transition systems that preserve synchronization under composition. They have been used as a basis for tools, such as compilers and model checkers. Unfortunately, composition of transition systems suffers from state space and transition space explosions, which limits scalability of the tools based on constraint automata. In this work, we propose stream constraints as an alternative to constraint automata that avoids state space explosions. We introduce a rule-based form for stream constraints that can avoid transition space explosions. We provide sufficient conditions under which our approach avoids transition space explosions.

## 1 Introduction

Over a decade ago, Baier et al. introduced *constraint automata* for the specification of interaction protocols [6]. Constraint automata feature a powerful composition operator that *preserves synchrony*: composite constructions not only yield intuitively meaningful asynchronous protocols but also synchronous protocols. Constraint automata have been used as basis for tools, like compilers and model checkers. Jongmans developed Lykos: a compiler that translates constraint automata into reasonably efficient executable Java code [13]. Baier, Blechmann, Klein, and Klüppelholz developed Vereofy, a model checker for constraint automata [4, 19]. Unfortunately, like every automaton model, composition of constraint automata suffers from state space and transition space explosions. These explosions limit the scalability of the tools based on constraint automata.

To improve scalability, Clarke et al. developed a compiler that translates a constraint automaton to a first-order formula [9]. The transitions of the constraint automaton correspond to the solutions of this formula. At run time, a generic constraint solver finds these solutions and simulates the automaton. Since composition and abstraction for constraint automata respectively correspond to conjunction and existential quantification, the first-order specification does not suffer from state space or transition space explosion. However, the approach proposed by Clarke et al. only delays the complexity until run time: calling a generic constraint solver at run time imposes a significant overhead.

Jongmans realized that the overhead of this constraint solver is not always necessary. He developed a commandification algorithm that accepts constraints without disjunctions (i.e., conjunctions of literals) and translates them into a

small imperative program [14]. The resulting program is a light-weight, tailor-made constraint solver with minimal run time overhead. Since commandification accepts only constraints without disjunction, Jongmans applied this technique to data constraints on individual transitions in a constraint automaton. Relying on constraint automata, his approach still suffers from scalability issues [17].

We aim to prevent state space and transition space explosions by combining the ideas of Clarke et al. and Jongmans. To this end, we present the language of *stream constraints*: a generalization of constraint automata based on temporal logic. A stream constraint is an expression that relates streams of observed data at different locations (Sect. 2). We identify a subclass of stream constraints, called *regular (stream) constraints*, which is closed under composition and abstraction (Sect. 3). Regular constraints can be viewed as a constraint automata, and conjunction of reflexive regular constraints is similar to composition of constraint automata (Sect. 4).

A straightforward application of the commandification algorithm of Jongmans to regular stream constraints entails transforming a stream constraint into disjunctive normal form and applying the algorithm to each clause separately. However, the number of clauses in the disjunctive normal form may grow exponentially in the size of the composition. To prevent such exponential blowups of the size of the formula, we recognize and exploit symmetries in the disjunctive normal form. Each clause in the disjunctive normal form can be constructed from a set of basic stream constraints, which we call *rules*. This idea allows us to represent a single large constraint as certain combination of a set of smaller constraints, called the *rule-based form* (Sect. 5). We express the composition of stream constraints in terms of the rule-based normal form (Sect. 6), and show that, for *simple* sets of rules, the number of rules to describe the composition is only linear in the size of the composition (Sect. 7). The class of stream constraints defined by a simple set of rules contains constraints for which the size of the disjunctive normal form explodes, which shows that our approach improves upon existing approaches by Clarke et al. and Jongmans. We express abstraction on stream constraints in terms of the rule-based normal form and provide a sufficient condition under which the number of rules remains constant (Sect. 8). Finally, we conclude and point out future work (Sect. 10).

*Related work.* Representation of stream constraints in rule-based form is part of a larger line of research on symbolic approaches, such a symbolic model checking [5, 8, 20] and symbolic execution [10]. These approaches not only use logic (cf., SAT solving techniques [12, 18] for verification), but also other implicit representations, like binary decision diagrams [7] and Petri nets [21]. Petri nets offer a small representation of protocols with an exponentially large state space. While our focus is more on compilation, Petri nets have been studied in the context of verification. As inspiration for future work, it is interesting to study the similarities between Petri nets and stream constraints.

Since regular stream constraints correspond to constraint automata, we can view regular stream constraints as a restricted temporal logic for which distributed synthesis is easy. In general, distributed (finite state) synthesis of

protocols is undecidable [22,23]. Pushing the boundary from regular to a larger class of stream constraints can be useful for more effective synthesis methods.

## 2 Syntax and Semantics

The semantics of constraint automata is defined as a relation over *timed data streams* [3], which are pairs, each consisting of a non-decreasing stream of time stamps and a stream of observed (exchanged) data items. The primary significance of time streams is the proper alignment of their respective data streams, by allowing “temporal gaps” during which no data is observed. For convenience, we drop the time stream and model protocols as relations over streams of data, augmented by a special symbol that designates “no-data” item.

We first define the abstract behavior of a protocol  $C$ . Fix an infinite set  $X$  of variables, and fix a non-empty set of user-data  $Data \supseteq \{0\}$  that contains a datum 0. Consider the *data domain*  $D = Data \cup \{*\}$  of data stream items, where we use the “no-data” symbol  $* \in D \setminus Data$  to denote the absence of data. We model the a single execution of protocol  $C$  as a function

$$\theta : X \longrightarrow D^{\mathbb{N}} \quad (1)$$

that maps every variable  $x \in X$  to a function  $\theta(x) : \mathbb{N} \longrightarrow D$  that represents a stream of data at location  $x$ . We call  $\theta$  a *data stream tuple* (over  $X$  and  $D$ ). For all  $n \in \mathbb{N}$  and all  $x \in X$ , the value  $\theta(x)(n) \in D$  is the data that we observe at location  $x$  and time step  $n$ . If  $\theta(x)(n) = *$ , we say that no data is observed at  $x$  in step  $n$  (i.e., we may view  $\theta$  as a partial map  $\mathbb{N} \times X \rightarrow Data$ ). The behavior of protocol  $C$  consists of the set

$$\mathcal{L}(C) \subseteq (D^{\mathbb{N}})^X \quad (2)$$

of all possible executions of  $C$ , called the *accepted language* of  $C$ . We can think of accepted language  $\mathcal{L}(C)$  as a relation over data streams. In this paper, we study protocols that are defined as a *stream constraint*:

**Definition 1 (Stream constraints).** *A stream constraint  $\phi$  is an expression generated by the following grammar*

$$\begin{aligned} \phi & ::= \perp \mid t_0 \doteq t_1 \mid \phi_0 \wedge \phi_1 \mid \neg\phi \mid \exists x\phi \mid \Box\phi \\ t & ::= x \mid \mathbf{d} \mid t' \end{aligned}$$

where  $x \in X$  is a variable,  $d \in D$  is a datum, and  $t$  is a stream term.

We use the following standard syntactic sugar:  $\top = \neg\perp$ ,  $\phi_0 \vee \phi_1 = \neg(\neg\phi_0 \wedge \neg\phi_1)$ ,  $\Diamond\phi = \neg\Box\neg\phi$ ,  $\neg(t_1 \doteq t_2) = (t_1 \not\doteq t_2)$ ,  $(t_1 \doteq t_2 \wedge \dots \wedge t_{n-1} \doteq t_n) = (t_1 \doteq \dots \doteq t_n)$ ,  $t^{(0)} = t$ , and  $t^{(k+1)} = (t^{(k)})'$ , for all  $k \geq 0$ . Following Rutten [25], we call  $t^{(k)}$ ,  $k \geq 0$ , the  $k$ -th *derivative* of term  $t$ .

We interpret a stream constraint as a constraint over streams of data in  $D^{\mathbb{N}}$ . For a datum  $d \in D$ ,  $\mathbf{d}$  is the constant stream defined as  $\mathbf{d}(n) = d$ , for all  $n \in \mathbb{N}$ . The operator  $(-)'$ , called *stream derivative*, drops the head of the stream and

is defined as  $\sigma'(n) = \sigma(n + 1)$ , for all  $n \in \mathbb{N}$  and  $\sigma \in D^{\mathbb{N}}$ . Streams can be related by  $\doteq$  that expresses equality of their heads:  $x \doteq y$  iff  $x(0) = y(0)$ , for all  $x, y \in D^{\mathbb{N}}$ . The modal operator  $\Box$  allows us to express that a stream constraint holds after applying any number of derivatives to all variables. For example,  $\Box(x \doteq y)$  iff  $x^{(k)}(0) = y^{(k)}(0)$ , for all  $k \in \mathbb{N}$  and  $x, y \in D^{\mathbb{N}}$ . Stream constraints can be composed via conjunction  $\wedge$ , or negated via negation  $\neg$ . Streams can be hidden via existential quantification  $\exists$ .

Each stream term  $t$  evaluates to a data stream in  $D^{\mathbb{N}}$ . Let  $\theta : X \rightarrow D^{\mathbb{N}}$  be a data stream tuple. We extend the domain of  $\theta$  from the set of variables  $X$  to the set of terms  $T \supseteq X$  as follows: we define  $\theta : T \rightarrow D^{\mathbb{N}}$  via  $\theta(\mathbf{d}) = \mathbf{d}$  and  $\theta(t') = \theta(t)'$ , for all  $d \in D$  and terms  $t \in T$ .

Next, we interpret a stream constraint  $\phi$  as a relation over streams.

**Definition 2 (Semantics).** *The language  $\mathcal{L}(\phi) \subseteq (D^{\mathbb{N}})^X$  of a stream constraint  $\phi$  over variables  $X$  and data domain  $D$  is defined as*

1.  $\mathcal{L}(\perp) = \emptyset$ ;
2.  $\mathcal{L}(t_0 \doteq t_1) = \{\theta : X \rightarrow D^{\mathbb{N}} \mid \theta(t_0)(0) = \theta(t_1)(0)\}$ ;
3.  $\mathcal{L}(\phi_0 \wedge \phi_1) = \mathcal{L}(\phi_0) \cap \mathcal{L}(\phi_1)$ ;
4.  $\mathcal{L}(\neg\phi) = (D^{\mathbb{N}})^X \setminus \mathcal{L}(\phi)$ ;
5.  $\mathcal{L}(\exists x\phi) = \{\theta : X \rightarrow D^{\mathbb{N}} \mid \theta[x \mapsto \sigma] \in \mathcal{L}(\phi), \text{ for some } \sigma \in D^{\mathbb{N}}\}$ ;
6.  $\mathcal{L}(\Box\phi) = \{\theta : X \rightarrow D^{\mathbb{N}} \mid \theta^{(k)} \in \mathcal{L}(\phi), \text{ for all } k \geq 0\}$ ,

where  $\theta[x \mapsto \sigma] : X \rightarrow D^{\mathbb{N}}$  is defined as  $\theta[x \mapsto \sigma](x) = \sigma$  and  $\theta[x \mapsto \sigma](y) = \theta(y)$ , for all  $y \in X \setminus \{x\}$ ; and  $\theta^{(k)} : X \rightarrow D^{\mathbb{N}}$  is defined as  $\theta^{(k)}(x) = \theta(x^{(k)})$ , for all  $x \in X$ .

Let  $\phi$  and  $\psi$  be two stream constraints and  $\theta : X \rightarrow D^{\mathbb{N}}$  a data stream tuple. We say that  $\theta$  satisfies  $\phi$  (and write  $\theta \models \phi$ ), whenever  $\theta \in \mathcal{L}(\phi)$ . We say that  $\phi$  implies  $\psi$  (and write  $\phi \models \psi$ ), whenever  $\mathcal{L}(\phi) \subseteq \mathcal{L}(\psi)$ . We call  $\phi$  and  $\psi$  equivalent (and write  $\phi \equiv \psi$ ), whenever  $\mathcal{L}(\phi) = \mathcal{L}(\psi)$ .

*Example 1.* One of the simplest stream constraints is  $\text{sync}(a, b)$ , which is defined as  $\Box(a \doteq b)$ . Constraint  $\text{sync}(a, b)$  encodes that the data streams at  $a$  and  $b$  are equal:  $\theta(a)(k) = \theta(b)(k)$ , for all  $k \in \mathbb{N}$  and all  $\theta \in (D^{\mathbb{N}})^X$ . Therefore,  $\text{sync}(a, b)$  synchronizes the data flow observed at ports  $a$  and  $b$ .

Conjunction  $\wedge$  and existential quantification  $\exists$  provide natural operators for composition and abstraction for stream constraints. For example, the composition  $\text{sync}(a, b) \wedge \text{sync}(b, c)$  synchronizes ports  $a$ ,  $b$ , and  $c$ . Hiding port  $b$  yields  $\exists b(\text{sync}(a, b) \wedge \text{sync}(b, c))$ , which is equivalent to  $\text{sync}(a, c)$ .  $\triangle$

*Example 2.* Recall that  $x^{(k)}$ , for  $k \geq 0$ , is the  $k$ -th derivative of  $x$ . We can express that a stream  $x$  is periodic via the stream constraint  $\Box(x^{(k)} \doteq x)$ , for some  $k \geq 1$ . For  $k = 1$ , stream  $x$  is constant, like  $\mathbf{0}$  and  $*$ .  $\triangle$

*Example 3.* The stream constraint  $\text{fifo}(a, b, m)$  defined as  $m \doteq * \wedge \Box((a \doteq m' \doteq \mathbf{0} \wedge b \doteq m \doteq *) \vee (a \doteq m' \doteq * \wedge b \doteq m \doteq \mathbf{0}) \vee (a \doteq b \doteq * \wedge m' \doteq m))$  models a 1-place buffer with input location  $a$ , output location  $b$ , and memory location  $m$  that can be full ( $m \doteq \mathbf{0}$ ) or empty ( $m \doteq *$ ).  $\triangle$

*Example 4.* Recall that  $*$  models absence of data. Stream constraint  $\Box\Diamond(a \neq *)$  expresses that always eventually we observe some datum at  $a$ . A constraint of such form can be used to define fairness.  $\triangle$

### 3 Regular Constraints

We identify a subclass of stream constraints that naturally correspond to constraint automata. We first introduce some notation.

To denote that a string  $s$  occurs as a substring in a stream constraint  $\phi$  or a stream term  $t$ , we write  $s \in \phi$  or  $s \in t$ , respectively.

Every stream constraint  $\phi$  admits a set  $\text{free}(\phi) \subseteq X$  of *free variables*, defined inductively via  $\text{free}(\perp) = \emptyset$ ,  $\text{free}(t_0 \dot{\equiv} t_1) = \{x \in X \mid x \in t_0 \text{ or } x \in t_1\}$ ,  $\text{free}(\phi_0 \wedge \phi_1) = \text{free}(\phi_0) \cup \text{free}(\phi_1)$ ,  $\text{free}(\neg\phi) = \text{free}(\Box\phi) = \text{free}(\phi)$ , and  $\text{free}(\exists x\phi) = \text{free}(\phi) \setminus \{x\}$ .

For every variable  $x \in X$ , we define the *degree of  $x$  in  $\phi$*  as

$$\text{deg}_x(\phi) = \max(\{-1\} \cup \{k \geq 0 \mid x^{(k)} \in \phi\}),$$

and the *degree of  $\phi$*  as  $\text{deg}(\phi) = \max_{x \in X} \text{deg}_x(\phi)$ . Note that for  $x \notin \phi$  we have  $\text{deg}_x(\phi) = -1$ . For  $k \geq 0$ , we write  $\text{free}^k(\phi) = \{x \in \text{free}(\phi) \mid \text{deg}_x(\phi) = k\}$  for the set of all free variables of  $\phi$  of degree  $k$ .

We call a variable  $x$  of degree zero in  $\phi$  a *port variable* and write  $P(\phi) = \text{free}^0(\phi)$  for the set of port variables of  $\phi$ . We call a variable  $x$  of degree one or higher in  $\phi$  a *memory variable* and write  $M(\phi) = \bigcup_{k \geq 1} \text{free}^k(\phi)$  for the set of memory variables of  $\phi$ .

**Definition 3 (Regular).** *A stream constraint  $\phi$  is regular if and only if  $\phi = \psi_0 \wedge \Box\psi$ , such that  $\Box \notin \psi_0 \wedge \psi$  and  $\text{deg}_x(\psi_0) < \text{deg}_x(\psi) \leq 1$ , for all  $x \in X$ .*

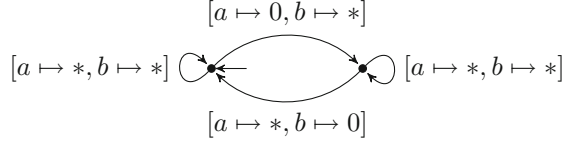
For a regular stream constraint  $\phi = \psi_0 \wedge \Box\psi$ , we refer to  $\psi_0$  as the *initial condition* of  $\phi$  and we refer to  $\psi$  as the *invariant* of  $\phi$ . Stream constraints  $\text{sync}(a, b)$  and  $\text{fifo}(a, b, m)$  in Examples 1 and 3 are regular stream constraints.

A regular stream constraint  $\phi$  has an operational interpretation in terms of a labeled transition system  $\llbracket \phi \rrbracket$ . States of the transition system consist of maps  $q : M(\phi) \rightarrow D$  that assign data to memory locations, and its labels consist of maps  $\alpha : P(\phi) \rightarrow D$  that assign data to ports. We write  $Q(\phi)$  for the set of states of  $\phi$  and  $A(\phi)$  for the set of labels of  $\phi$ .

**Definition 4 (Operational semantics).** *The operational semantics  $\llbracket \phi \rrbracket$  of a regular stream constraint  $\phi = \psi_0 \wedge \Box\psi$  consists of a labeled transition system  $(Q(\phi), A(\phi), \rightarrow, Q_0)$ , with set of states  $Q(\phi)$ , set of labels  $A(\phi)$ , set of transitions  $\rightarrow = \{(q_\phi(\theta), q_\phi(\theta'), \alpha_\phi(\theta)) \mid \theta \in \mathcal{L}(\psi)\}$ , and set of initial states  $Q_0 = \{q_\phi(\theta) \mid \theta \in \mathcal{L}(\psi_0 \wedge \psi)\}$ , where*

1.  $q_\phi(\theta) : M(\phi) \rightarrow D$  is defined as  $q_\phi(\theta)(x) = \theta(x)(0)$ , for  $x \in M(\phi)$ ; and
2.  $\alpha_\phi(\theta) : P(\phi) \rightarrow D$  is defined as  $\alpha_\phi(\theta)(x) = \theta(x)(0)$ , for  $x \in P(\phi)$ .

and  $\theta'$  is defined as  $\theta'(x)(n) = \theta(x)(n+1)$ , for all  $x \in X$  and  $n \in \mathbb{N}$ .



**Fig. 1.** Semantics of  $\text{fifo}(a, b, m)$  over the trivial data domain  $\{0, *\}$ .

*Example 5.* Consider the regular stream constraint  $\text{fifo}(a, b, m)$  from Example 3. Note that in this example, the set of ports equals  $\text{free}^0(\text{fifo}) = \{a, b\}$  and the set of memory locations equals  $\text{free}^1(\text{fifo}) = \{m\}$ . The semantics of  $\text{fifo}(a, b, m)$  over the trivial data domain  $D = \{0, *\}$  consists of 4 transitions:

1.  $([m \mapsto *], [m \mapsto 0], [a \mapsto 0, b \mapsto *])$ ;
2.  $([m \mapsto 0], [m \mapsto *], [a \mapsto *, b \mapsto 0])$ ; and
3.  $([m \mapsto d], [m \mapsto d], [a \mapsto *, b \mapsto *])$ , for every  $d \in \{*, 0\}$ .

Figure 1 shows the semantics of  $\text{fifo}$  over the trivial data domain. △

Equivalent stream constraints do not necessarily have the same operational semantics. We are, therefore, interested in operational equivalence of constraints:

**Definition 5 (Operational equivalence).** *Stream constraints  $\phi$  and  $\psi$  are operationally equivalent ( $\phi \simeq \psi$ ) iff  $\phi \equiv \psi$  and  $\text{free}^k(\phi) = \text{free}^k(\psi)$ , for  $k \geq 0$ .*

*Example 6.* Let  $\phi$  be a stream constraint, let  $t$  be a term and let  $x \notin t$  be a variable that does not occur in  $t$ . Then, we have  $\exists x(x \doteq t \wedge \phi) \equiv \phi[t/x]$ , where  $\phi[t/x]$  is obtained from  $\phi$  by substituting  $t$  for every free occurrence of  $x$ . Observe that  $\exists x(x \doteq t \wedge \phi)$  and  $\phi[t/x]$  may admit different sets of free variables: if  $\phi$  is just  $\top$  and  $t$  is a variable  $y$ , the equivalence amounts to  $\exists x(x \doteq y) \equiv \top$ . To ensure that the free variables coincide, we can add the equality  $t \doteq t$  and obtain the operational equivalence  $\exists x(x \doteq t \wedge \phi) \simeq \phi[t/x] \wedge t \doteq t$ . △

Operational equivalence of stream constraints  $\phi$  and  $\psi$  implies that their operational semantics are identical, i.e.,  $\llbracket \phi \rrbracket = \llbracket \psi \rrbracket$ . It is possible to introduce weaker equivalences by, for example, demanding that  $\llbracket \phi \rrbracket$  and  $\llbracket \psi \rrbracket$  are only weakly bisimilar. Such weaker equivalence offer more room for simplification of stream constraints than operational equivalence does. As our work does not need this generality, we leave the study of such weaker equivalences as future work.

The most important operations on stream constraints are composition ( $\wedge$ ) and hiding ( $\exists$ ). The following result shows that regular stream constraints are closed under conjunction and existential quantification of degree zero variables.

**Theorem 1.** *For all stream constraints  $\phi$  and  $\psi$  and variables  $x$ , we have*

1.  $\Box\phi \wedge \Box\psi \equiv \Box(\phi \wedge \psi)$ ; and
2.  $\exists x\Box\phi \equiv \Box\exists x\phi$ , whenever  $\text{deg}_x(\phi) \leq 0$  and  $\Box \notin \phi$ .

*Proof.* For assertion 1,  $\mathcal{L}(\Box\phi \wedge \Box\psi) = \{\theta \in (D^{\mathbb{N}})^X \mid \forall k \geq 0 : \theta^{(k)} \models \phi \wedge \psi\} = \mathcal{L}(\Box(\phi \wedge \psi))$  shows that  $\Box\phi \wedge \Box\psi \equiv \Box(\phi \wedge \psi)$ .

For assertion 2, suppose that  $\deg_x(\phi) \leq 0$  and  $\Box \not\models \phi$ . We show that  $\theta \in \mathcal{L}(\Box\exists x\phi)$  if and only if  $\theta \in \mathcal{L}(\exists x\Box\phi)$ , for all  $\theta \in (D^{\mathbb{N}})^X$ . By Definition 2, this equivalence can be written as

$$\theta^{(k)}[x \mapsto \mu_k] \models \phi \iff (\theta[x \mapsto \sigma])^{(k)} \models \phi, \quad (3)$$

for all  $k \geq 0$ ,  $\sigma \in D^{\mathbb{N}}$ , and  $\mu_k \in D^{\mathbb{N}}$  such that  $\mu_k(0) = \sigma^{(k)}(0)$ .

To prove Eq. (3), we proceed by induction on the length of  $\phi$ :

*Case 1* ( $\phi := \perp$ ): Since  $\mathcal{L}(\perp) = \emptyset$ , Eq. (3) holds trivially.

*Case 2* ( $\phi := t_0 \doteq t_1$ ): Observe that, since  $\deg_x(\phi) \leq 0$ , for all terms  $t$ , we have  $x \in t$  iff  $t = x$ . We conclude Eq. (3) from  $\mu_k(0) = \sigma^{(k)}(0)$  and

$$\theta^{(k)}[x \mapsto \mu_k](t)(0) = \begin{cases} \mu_k(0) & \text{if } t = x \\ \theta^{(k)}(t)(0) & \text{if } t \neq x \end{cases} = (\theta[x \mapsto \sigma])^{(k)}(t)(0).$$

*Case 3* ( $\phi := \psi_0 \wedge \psi_1$ ): By the induction hypothesis, Eq. (3) holds for  $\psi_0$  and  $\psi_1$ . By conjunction of Eq. (3), we conclude Eq. (3) for  $\phi$ .

*Case 4* ( $\phi := \neg\psi$ ): By the induction hypothesis, Eq. (3) holds for  $\psi$ . By contraposition of Eq. (3), we conclude Eq. (3) for  $\phi$ .

*Case 5* ( $\phi := \exists y\psi$ ): If  $y = x$ , then  $x \notin \text{free}(\phi)$  and both sides in Eq. (3) are equivalent to  $\theta^{(k)} \models \phi$ . Hence, Eq. (3) holds for  $y = x$ . Suppose  $y \neq x$ . Then,  $\theta^{(k)}[x \mapsto \mu_k] \models \phi$  is equivalent to  $(\theta[y \mapsto \tau])^{(k)}[x \mapsto \mu_k] \models \psi$ , for some  $\tau \in D^{\mathbb{N}}$ . Applying the induction hypothesis for  $\theta$  equal to  $\theta[y \mapsto \tau]$ , we conclude that  $\theta^{(k)}[x \mapsto \mu_k] \models \phi$  is equivalent to  $(\theta[y \mapsto \tau][x \mapsto \sigma])^{(k)} \models \psi$ , for some  $\tau \in D^{\mathbb{N}}$ . Since  $y \neq x$ , we conclude that Eq. (3) holds.

We conclude that the claim holds for all  $\phi$  with  $\deg_x(\phi) \leq 0$  and  $\Box \not\models \phi$ .  $\square$

## 4 Reflexive Constraints

Conjunction of stream constraints is a simple syntactic composition operator with clear semantics: a data stream tuple  $\theta$  satisfies a conjunction  $\phi_0 \wedge \phi_1$  if and only if  $\theta$  satisfies both  $\phi_0$  and  $\phi_1$ . In view of the semantics of regular stream constraints in Definition 2, it is less obvious how  $\llbracket \phi_0 \wedge \phi_1 \rrbracket$  relates to  $\llbracket \phi_0 \rrbracket$  and  $\llbracket \phi_1 \rrbracket$ . The following result characterizes their relation when no memory is shared.

**Theorem 2.** *Let  $\phi_0$  and  $\phi_1$  be regular stream constraints such that  $\text{free}(\phi_0) \cap \text{free}(\phi_1) \subseteq P(\phi_0 \wedge \phi_1)$ , and let  $(q_i, q'_i, \alpha_i) \in Q(\phi_i)^2 \times A(\phi_i)$ , for  $i \in \{0, 1\}$ . The following are equivalent:*

1.  $q_0 \xrightarrow{\alpha_0} q'_0$  in  $\llbracket \phi_0 \rrbracket$ ,  $q_1 \xrightarrow{\alpha_1} q'_1$  in  $\llbracket \phi_1 \rrbracket$ , and  $\alpha_0|_{P(\phi_1)} = \alpha_1|_{P(\phi_0)}$ ;
2.  $q_0 \cup q_1 \xrightarrow{\alpha_0 \cup \alpha_1} q'_0 \cup q'_1$  in  $\llbracket \phi_0 \wedge \phi_1 \rrbracket$ ,

where  $|$  is restriction of maps, and  $\cup$  is union of maps.

*Proof.* Write  $\phi_i = \psi_{i0} \wedge \square \psi_i$ , with  $\square \notin \psi_{i0} \wedge \psi_i$  and  $\deg_x(\psi_{i0}) < \deg_x(\psi_i) \leq 1$ , for all  $x \in X$ . Then,  $\text{free}^k(\phi_i) = \text{free}^k(\psi_i)$ , for all  $i, k \in \{0, 1\}$ .

Suppose that assertion 1 holds. By Definition 2, we find, for all  $i \in \{0, 1\}$ , some  $\theta_i \in \mathcal{L}(\psi_i)$  such that  $q_i = q_{\phi_i}(\theta_i)$ ,  $q'_i = q_{\phi_i}(\theta'_i)$ , and  $\alpha_i = \alpha_{\phi_i}(\theta_i)$ . Define  $\theta : X \rightarrow D^{\mathbb{N}}$  by  $\theta(x) = \theta_i(x)$ , if  $x \in \text{free}(\phi_i)$ , and  $\theta(x) = *$ , otherwise. Since  $\text{free}(\phi_0) \cap \text{free}(\phi_1) \subseteq P(\phi_0 \wedge \phi_1)$  and  $\alpha_0|_{P(\phi_1)} = \alpha_1|_{P(\phi_0)}$ , we have that  $\theta_0(x) = \theta_1(x)$ , for all  $x \in \text{free}(\phi_0) \cap \text{free}(\phi_1)$ . Hence,  $\theta$  is well-defined. By construction,  $\theta \models \psi_0$  and  $\theta \models \psi_1$ . By Definition 2, we have  $\theta \models \psi_0 \wedge \psi_1$ . By Theorem 1, we have  $\phi_0 \wedge \phi_1 = \psi_{00} \wedge \psi_{10} \wedge \square(\psi_0 \wedge \psi_1)$ . Since  $q_0 \cup q_1 = q_{\phi_0 \wedge \phi_1}(\theta)$ ,  $q'_0 \cup q'_1 = q_{\phi_0 \wedge \phi_1}(\theta')$ , and  $\alpha_0 \cup \alpha_1 = \alpha_{\phi_0 \wedge \phi_1}(\theta)$ , we conclude assertion 2.

Suppose that assertion 2 holds. We find some  $\theta \in \mathcal{L}(\psi_0 \wedge \psi_1)$ , such that  $q_0 \cup q_1 = q_\theta$ ,  $q'_0 \cup q'_1 = q_{\theta'}$ , and  $\alpha_0 \cup \alpha_1 = \alpha_\theta$ . Then, we conclude assertion 1, for  $q_i = q_{\phi_i}(\theta)$ ,  $q'_i = q_{\phi_i}(\theta')$ , and  $\alpha_i = \alpha_{\phi_i}(\theta)$ .  $\square$

Stream constraints  $\phi_0$  and  $\phi_1$  without shared variables ( $\text{free}(\phi_0) \cap \text{free}(\phi_1) = \emptyset$ ) seem completely independent. However, Theorem 2 shows that their composition  $\phi_0 \wedge \phi_1$  admits a transition only if  $\phi_0$  and  $\phi_1$  admit respective local transitions  $(q_0, q'_0, \alpha_0)$  and  $(q_1, q'_1, \alpha_1)$ , such that  $\alpha_0|_{P(\phi_1)} = \alpha_1|_{P(\phi_0)}$ . Since  $\phi_0$  and  $\phi_1$  do not share variables, the latter condition on  $\alpha_0$  and  $\alpha_1$  is trivially satisfied. Still, for one protocol  $\phi_i$ , with  $i \in \{0, 1\}$ , to make progress in the composition  $\phi_0 \wedge \phi_1$ , constraint  $\phi_{1-i}$  must admit an idling transition.

To allow such independent progress, we assume that  $\phi_{1-i}$  admits an *idling* transition  $(q, q, \tau)$ , where  $\tau$  is the silent label over  $P(\phi_{1-i})$ . The *silent label* over a set of ports  $P \subseteq X$  is the map  $\tau : P \rightarrow D$  that maps  $x \in P$  to  $*$  in  $D$ . If such idling transitions are available in every state of  $\phi_1$ , we say that  $\phi_1$  is *reflexive*:

**Definition 6 (Reflexive).** *A stream constraint  $\phi$  is reflexive if and only if  $q \xrightarrow{\tau} q$  in  $\llbracket \phi \rrbracket$ , for all  $q \in Q(\phi)$ .*

For regular constraints, we can define reflexiveness also syntactically, for which we need some notation. For a variable  $x \in X$  and an integer  $k \in \mathbb{N} \cup \{-1\}$ , we define the predicate  $x \dagger_k$  (pronounced: “ $x$  is blocked at step  $k$ ”) as follows:

$$x \dagger_k := (x^{(k)} \doteq x^{(k-1)}), \quad \text{with } x^{(k)} \doteq *, \text{ for all } k < 0.$$

Predicate  $x \dagger_{-1} \equiv \top$  is trivially true. Predicate  $x \dagger_0 \equiv (x \doteq *)$  means that we observe no data flow at port  $x$ . Predicate  $x \dagger_1 \equiv (x' \doteq x)$  means that the data in memory variable  $x$  remains the same.

We now provide a syntactic equivalent of Definition 6 for regular constraints.

**Lemma 1.** *A regular stream constraint  $\phi = \psi_0 \wedge \square \psi$  is reflexive if and only if  $\bigwedge_{x \in X} x \dagger_{d(x)} \models \psi$ , where  $d(x) = \deg_x(\phi)$ , for all  $x \in X$ .*

*Proof.* Since  $d(x) = -1$ , for all but finitely many  $x \in X$ , the stream constraint  $\bigwedge_{x \in X} x \dagger_{d(x)}$  is well-defined. By definition,  $\bigwedge_{x \in X} x \dagger_{d(x)} \models \psi$  if and only if, for all  $q \in Q(\phi)$ , there exists some  $\theta \in \mathcal{L}(\psi)$ , such that  $q_\theta = q_{\theta'} = q$  and  $\alpha_\theta = \tau$ .  $\square$



*Example 7.* The stream constraint  $\text{sync}(a, b) := \Box(a \dot{=} b)$  from Example 1 is reflexive, because  $\bigwedge_{x \in X} x \dagger_{d(x)} = a \dot{=} * \wedge b \dot{=} *$  implies  $a \dot{=} b$ . The stream constraint  $\text{fifo}$  from Example 3 is reflexive, because  $\bigwedge_{x \in X} x \dagger_{d(x)} = a \dot{=} * \wedge b \dot{=} * \wedge m' \dot{=} m$  is one of the clauses of  $\text{fifo}$ .  $\triangle$

Theorem 2 suggests a composition operator  $\times$  on labeled transition systems, satisfying  $\llbracket \phi_0 \rrbracket \times \llbracket \phi_1 \rrbracket = \llbracket \phi_0 \wedge \phi_1 \rrbracket$ . For reflexive constraints  $\phi_0$  and  $\phi_1$ , composition  $\times$  simulates composition of constraint automata [6]. Constraint automata also feature a hiding operator that naturally corresponds to existential quantification  $\exists$  for stream constraints. We leave a full formal comparison between stream constraints and constraint automata as future work.

## 5 Rule-Based Form

The commandification algorithm developed by Jongmans accepts only conjunctions of literals [14]. To apply commandification to the invariant  $\psi$  of an arbitrary regular stream constraint  $\psi_0 \wedge \Box \psi$ , we can first transform  $\psi$  into disjunctive normal form (DNF). However, the number of clauses in the disjunctive normal form may be exponential in the length of the constraint. In this section, we introduce an alternative to the disjunctive normal form that prevents such exponential blow up, for a strictly larger class of stream constraints. Our main observation is that the clauses of the disjunctive normal form may contain many symmetries, in the sense that we may generate all clauses from a set of stream constraints  $R$ , called a *set of rules*. A *rule* is a stream constraint  $\rho$ , such that  $\text{deg}(\rho) \leq 1$  and  $\Box \notin \rho$ .

**Definition 7 (Rule-based form).** *A reflexive stream constraint  $\phi$  is in rule-based form iff  $\phi$  equals*

$$\text{rbf}(R) = \bigwedge_{x \in \text{free}(R)} x \dagger_{d(x)} \vee \bigvee_{\rho \in R: x \in \text{free}(\rho)} \rho \quad (4)$$

with  $R$  a finite set of rules,  $\text{free}(R) = \bigcup_{\rho \in R} \text{free}(\rho)$ , and  $d(x) = \max_{\rho \in R} \text{deg}_x(\rho)$ . A stream constraint  $\phi$  is defined by  $R$  iff  $\phi \simeq \text{rbf}(R)$ .

We apply the rule-based form to the invariant of regular constraints, via  $\psi_0 \wedge \Box \text{rbf}(R)$ , for some degree zero stream constraint  $\psi_0$  and set of rules  $R$ . Intuitively,  $R$  remains smaller than the DNF of  $\text{rbf}(R)$  under composition.

*Example 8.*  $\psi \simeq \text{rbf}(\{\psi\})$ , for all reflexive stream constraints  $\psi$ , with  $\text{deg}(\psi) \leq 1$  and  $\Box \notin \psi$ . Hence, Example 7 shows  $\text{sync}(a, b) = \Box(a \dot{=} b) \simeq \Box \text{rbf}(\{a \dot{=} b\})$ .  $\triangle$

*Example 9.* The stream constraint  $\text{lossy}(a, b) := \Box \text{rbf}(\{a \dot{=} a, a \dot{=} b\})$  is equivalent to  $\Box(b \dot{=} * \vee a \dot{=} b)$ . Note that  $\Box \text{rbf}(\{\top, a \dot{=} b\}) \simeq \Box \text{rbf}(\{a \dot{=} b\}) \simeq \text{sync}(a, b)$ . Hence, rules  $a \dot{=} a$  and  $\top$  are very different.  $\triangle$

*Example 10.* The set of rules that define a stream constraint is not unique. Consider the stream constraint  $\text{fifo}$  from Example 3. On the one hand, we have  $\text{fifo}(a, b, m) \simeq m \dot{=} * \wedge \square \text{rbf}(\{\varphi, \psi\})$ , where  $\varphi \simeq a \dot{=} m' \dot{=} \mathbf{0} \wedge m \dot{=} *$  models the action that puts data in the buffer and  $\psi \simeq m' \dot{=} * \wedge b \dot{=} m \dot{=} \mathbf{0}$  models the action that takes data out of the buffer. On the other hand, we have  $\text{fifo}(a, b, m) \simeq m \dot{=} * \wedge \square \text{rbf}(\{a \dot{=} m' \dot{=} \mathbf{0} \wedge b \dot{=} m \dot{=} *, a \dot{=} m' \dot{=} * \wedge b \dot{=} m \dot{=} \mathbf{0}\})$ .  $\triangle$

*Example 11.* Rule-based forms are an alternative to disjunctive normal forms. Consider the reflexive constraint  $\phi := \bigvee_{i=1}^n \rho_i$  in DNF for which the first conjunctive clause  $\rho_1$  is equivalent to  $\bigwedge_{x \in \text{free}(\phi)} x \dagger_{d(x)}$ , with  $d(x) = \text{deg}_x(\phi)$ . By adding equalities of the form  $x \dot{=} x$ , we assume without loss of generality that  $\text{free}(\rho_i) = \text{free}(\phi)$ , for all  $2 \leq i \leq n$ . For  $R = \{\rho_i \mid 2 \leq i \leq n\}$ , it follows from

$$\text{rbf}(R) \equiv \bigwedge_{x \in \text{free}(R)} \left( x \dagger_{d(x)} \vee \bigvee_{\rho \in R} \rho \right) \equiv \left( \bigwedge_{x \in \text{free}(\phi)} x \dagger_{d(x)} \right) \vee \bigvee_{\rho \in R} \rho \equiv \phi \quad (5)$$

that  $\phi$  is defined by the set  $R$ .  $\triangle$

Definition 7 presents the rule-based form as a conjunctive normal form. The following result computes the disjunctive normal form of  $\text{rbf}(R)$ .

**Lemma 2.** *For every set of rules  $R$ , we have*

$$\text{rbf}(R) \simeq \text{dnf}(R) := \bigvee_{T \subseteq R} \bigwedge_{\rho \in T} \rho \wedge \bigwedge_{x \in \text{free}(R) \setminus \text{free}(T)} x \dagger_{d(x)}.$$

*Proof.* Let  $x \in X$  be arbitrary. By construction, we have  $\text{deg}_x(\text{dnf}(R)) \leq \max_{\rho \in R} \text{deg}_x(\rho)$ . Since  $d(x) = \max_{\rho \in R} \text{deg}_x(\rho)$ , the clause for  $T = \emptyset$  shows that  $\text{deg}_x(\text{dnf}(R)) \geq d(x)$ . By Lemma 4,  $\text{deg}_x(\text{rbf}(R)) = \text{deg}_x(\text{dnf}(R))$ , for all  $x \in X$ . Hence,  $\text{free}^k(\text{rbf}(R)) = \text{free}^k(\text{dnf}(R))$ , for all  $k \geq 0$ .

Next, we show that  $\text{rbf}(R) \models \text{dnf}(R)$ . Let  $\theta \in \mathcal{L}(\text{rbf}(R))$ . We find, for every  $x \in \text{free}(R)$ , some rule  $\rho_x \in R$ , such that  $\theta \models \rho$  and  $x \in \text{free}(\rho)$ . Now, define  $T_\theta := \{\rho_x \mid x \in \text{free}(R) \text{ and } \theta \notin \mathcal{L}(x \dagger_{d(x)})\}$ . By construction,  $\theta \models \rho_x$ , for every  $\rho_x \in T_\theta$ . If  $x \in \text{free}(R)$  and  $\theta \notin \mathcal{L}(x \dagger_{d(x)})$ , then  $\rho_x \in T_\theta$  and  $x \in \text{free}(\rho_x) \subseteq \text{free}(T_\theta)$ . By contraposition, we conclude that  $\theta \models x \dagger_{d(x)}$ , for all  $x \in \text{free}(R) \setminus \text{free}(T_\theta)$ . Hence,  $\theta \models \text{dnf}(R)$ , and  $\mathcal{L}(\text{rbf}(R)) \subseteq \mathcal{L}(\text{dnf}(R))$ .

Finally, we show that  $\text{dnf}(R) \models \text{rbf}(R)$ . Let  $\theta \in \mathcal{L}(\text{dnf}(R))$ . By definition of  $\text{dnf}(R)$ , we find some  $T \subseteq R$  with  $\theta \models \rho$ , for all  $\rho \in T$ , and  $\theta \models x \dagger_{d(x)}$ , for all  $x \in \text{free}(R) \setminus \text{free}(T)$ . Suppose that  $x \in \text{free}(R)$  and  $\theta \not\models x \dagger_{d(x)}$ . Since  $\theta \models x \dagger_{d(x)}$ , for all  $x \in \text{free}(R) \setminus \text{free}(T)$ , we find by contraposition that  $x \in \text{free}(T)$ . Hence, we find some  $\psi \in T$  with  $x \in \text{free}(\psi)$ . Since  $\theta \models \rho$ , for all  $\rho \in T$ , we find that  $\theta \models \psi$ . Hence,  $\theta \models \text{rbf}(R)$  and we conclude that  $\text{rbf}(R) \simeq \text{dnf}(R)$ .  $\square$

## 6 Composition

We express conjunction of stream constraints in terms of their defining sets of rules. That is, for two sets of rules  $R_0$  and  $R_1$ , we define the composition  $R_0 \wedge R_1$

of  $R_0$  and  $R_1$ , such that  $\text{rbf}(R_0 \wedge R_1) \simeq \text{rbf}(R_0) \wedge \text{rbf}(R_1)$ . If  $R_0$  and  $R_1$  do not share any variable (i.e.,  $\text{free}(R_0) \cap \text{free}(R_1) = \emptyset$ ), composition  $R_0 \wedge R_1$  is given by the union  $R_0 \cup R_1$ . In this section, we define the composition  $R_0 \wedge R_1$  of  $R_0$  and  $R_1$  for  $\text{free}(R_0) \cap \text{free}(R_1) \neq \emptyset$ .

In view of Example 11, consider the normal form  $\text{dnf}(R_0 \wedge R_1)$ . Since  $\text{dnf}(R_0 \wedge R_1)$  equals  $\text{dnf}(R_0) \wedge \text{dnf}(R_1)$ , it suffices to characterize the set of clauses of  $\text{dnf}(R_0) \wedge \text{dnf}(R_1)$ . Every such clause is a conjunction of a clause in  $\text{dnf}(R_0)$  and a clause in  $\text{dnf}(R_1)$ . Lemma 2 shows that the clauses of  $\text{dnf}(R_i)$  correspond to subsets  $T_i$  of  $R_i$ , for all  $i \in \{0, 1\}$ . Not every pair of subsets  $T_0 \subseteq R_0$  and  $T_1 \subseteq R_1$  yields a clause of  $\text{dnf}(R_0) \wedge \text{dnf}(R_1)$ , but only if  $S = T_0 \cup T_1$  is *synchronous*:

**Definition 8 (Synchronous).** *A synchronous set over sets of rules  $R_0$  and  $R_1$  is a subset  $S \subseteq R_0 \cup R_1$ , with  $\text{free}(S) \cap \text{free}(R_i) \subseteq \text{free}(S \cap R_i)$ , for all  $i \in \{0, 1\}$ .*

*Example 12.* For any integer  $i \geq 1$ , let  $\varphi_i := a_i \doteq m'_i \doteq \mathbf{0} \wedge m_i \doteq *$  and  $\psi_i := m'_i \doteq * \wedge a_{i+1} \doteq m_i \doteq \mathbf{0}$  be the two rules that define  $\text{fifo}(a_i, a_{i+1}, m_i)$ , from Example 10. The synchronous sets consist of exactly those sets  $S \subseteq \{\varphi_1, \psi_1\} \cup \{\varphi_2, \psi_2\}$  that satisfy  $\psi_1 \in S$  iff  $\varphi_2 \in S$ . That is, the synchronous sets are given by  $\emptyset$ ,  $\{\varphi_1\}$ ,  $\{\psi_2\}$ ,  $\{\psi_1, \varphi_2\}$ ,  $\{\varphi_1, \psi_1, \varphi_2\}$ ,  $\{\psi_1, \varphi_2, \psi_2\}$ ,  $\{\varphi_1, \psi_1, \varphi_2, \psi_2\}$ .  $\triangle$

Next, we recognize symmetries in the collection of synchronous sets. We can construct every synchronous set as a union of *irreducible* synchronous subsets:

**Definition 9 (Irreducibility).** *A non-empty synchronous set  $\emptyset \neq S \subseteq R_0 \cup R_1$  is irreducible if and only if  $S = S_0 \cup S_1$  implies  $S = S_0$  or  $S = S_1$ , for all synchronous subsets  $S_0, S_1 \subseteq R_0 \cup R_1$ .*

*Example 13.* Let  $R_0$  and  $R_1$  be sets of rules, and let  $\rho \in R_0$  be a rule, such that  $\text{free}(\rho) \cap \text{free}(R_1) = \emptyset$ . We show that  $\{\rho\}$  is irreducible synchronous. Since  $\text{free}(\{\rho\}) \cap \text{free}(R_0) = \text{free}(\rho) = \text{free}(\{\rho\} \cap R_0)$  and  $\text{free}(\{\rho\}) \cap \text{free}(R_1) = \emptyset \subseteq \text{free}(\{\rho\} \cap R_1)$ , we conclude that  $\{\rho\}$  is synchronous. Suppose  $\{\rho\} = S_0 \cup S_1$ . Then,  $\rho \in S_i$ , for some  $i \in \{0, 1\}$ . Hence,  $\{\rho\} \subseteq S_i \subseteq \{\rho\}$ , which shows that  $S_i = \{\rho\}$ . We conclude that  $\{\rho\}$  is irreducible synchronous in  $R_0 \cup R_1$ .  $\triangle$

*Example 14.* Consider  $\varphi_i$  and  $\psi_i$ , for  $i \in \{1, 2\}$ , from Example 12. The irreducible synchronous sets of  $\{\varphi_1, \psi_1\} \cup \{\varphi_2, \psi_2\}$  are  $\{\varphi_1\}$ ,  $\{\psi_2\}$ , and  $\{\psi_1, \varphi_2\}$ .  $\triangle$

**Definition 10 (Composition).** *The composition of sets of rules  $R_0$  and  $R_1$  is  $R_0 \wedge R_1 := \{\bigwedge_{\rho \in S} \rho \mid S \subseteq R_0 \cup R_1 \text{ irreducible synchronous}\}$ .*

*Example 15.* Let  $R_0$  and  $R_1$  be sets of rules, with  $\text{free}(R_0) \cap \text{free}(R_1) = \emptyset$ . By Example 13, we find that  $\{\rho\} \subseteq R_0 \cup R_1$ , for all  $\rho \in R_0 \cup R_1$ , is irreducible synchronous. Hence, every synchronous set  $S \subseteq R_0 \cup R_1$ , with  $|S| \geq 2$ , is reducible. Therefore,  $S \subseteq R_0 \cup R_1$  is irreducible synchronous if and only if  $S = \{\rho\}$ , for some  $\rho \in R_0 \cup R_1$ . We conclude that  $R_0 \wedge R_1 = R_0 \cup R_1$ . Consequently,  $\emptyset$  is a (unique) identity element with respect to composition  $\wedge$  of sets of rules.  $\triangle$

To show that the composition of sets of rules coincides with conjunction of stream constraints, we need the following result that shows that every non-empty synchronous set can be covered by irreducible synchronous sets.

**Lemma 3.** *Let  $R_0$  and  $R_1$  be sets of rules, and let  $S \subseteq R_0 \cup R_1$  be a non-empty synchronous set. Then,  $S = \bigcup_{i=1}^n S_i$ , where  $S_i \subseteq R_0 \cup R_1$ , for  $1 \leq i \leq n$ , is irreducible synchronous.*

*Proof.* We prove the lemma by induction on the size  $|S|$  of  $S$ . For the base case, suppose that  $|S| = 1$ . We show that  $S$  is irreducible synchronous, which provides a trivial covering. Suppose that  $S = S_0 \cup S_1$ , for some synchronous sets  $S_0, S_1 \subseteq R_0 \cup R_1$ . Since,  $|S| = 1$ , we have  $S \subseteq S_i \subseteq S$ , for some  $i \in \{0, 1\}$ . Hence,  $S = S_i$ , and  $S$  is irreducible. We conclude that the lemma holds, for  $|S| = 1$ .

For the induction step, suppose that  $|S| = k > 1$ , and suppose that the lemma holds, for  $|S| < k$ . If  $S$  is irreducible, we find a trivial covering of  $S$ . If  $S$  is reducible, we find  $S = S_0 \cup S_1$ , where  $S_0 \neq S \neq S_1$  are synchronous sets in  $R_0 \cup R_1$ . Since  $|S_i| < |S|$ , for  $i \in \{0, 1\}$ , we find by the hypothesis that  $S_i = \bigcup_{j=1}^{n_i} S_{ij}$ . Hence,  $S = S_0 \cup S_1 = \bigcup_{i=0}^1 \bigcup_{j=1}^{n_i} S_{ij}$ . We conclude that the lemma holds, for  $|S| = k$ . By induction on  $|S|$ , we conclude the lemma.  $\square$

**Lemma 4.**  $\deg_x(\text{rbf}(R)) = \max_{\rho \in R} \deg_x(\rho)$ , for all sets of rules  $R$  and  $x \in X$ .

*Proof.* For any set of rules  $R$  and  $y \in X$ , we have

$$\deg_y(\text{rbf}(R)) = \max_{x \in \text{free}(R)} \max(\deg_y(x \uparrow_{d(x)}), \max_{\rho \in R: x \in \text{free}(\rho)} \deg_y(\rho)).$$

Note that  $\deg_y(x \uparrow_{d(x)}) = d(y)$ , if  $y = x$ , and  $\deg_y(x \uparrow_{d(x)}) = -1$ , otherwise. Since  $d(y) = \max_{\rho \in R} \deg_y(\rho)$ , we have  $\deg_y(\text{rbf}(R)) = \max_{\rho \in R} \deg_y(\rho)$ .  $\square$

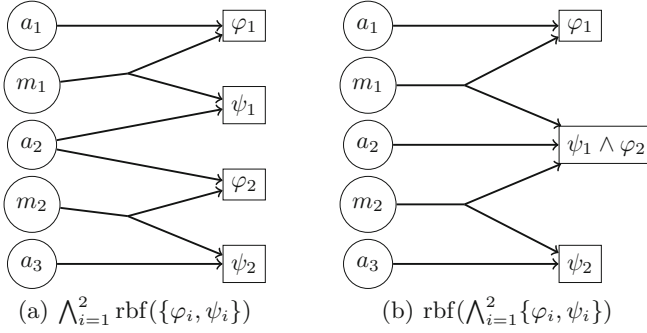
**Theorem 3.**  $\text{rbf}(R_0 \wedge R_1) \simeq \text{rbf}(R_0) \wedge \text{rbf}(R_1)$ , for all sets of rules  $R_0$  and  $R_1$ .

*Proof.* By Lemma 4 and Definition 10,  $\deg_x(\text{rbf}(R_0 \wedge R_1)) = \deg_x(\text{rbf}(R_0) \wedge \text{rbf}(R_1))$ , for all  $x \in X$ . Hence,  $\text{free}^k(\text{rbf}(R_0 \wedge R_1)) = \text{free}^k(\text{rbf}(R_0) \wedge \text{rbf}(R_1))$ , for all  $k \geq 0$ .

Next, we show  $\text{rbf}(R_0) \wedge \text{rbf}(R_1) \models \text{rbf}(R_0 \wedge R_1)$ . Let  $\theta \in \mathcal{L}(\text{rbf}(R_0) \wedge \text{rbf}(R_1))$ . By Definition 7, we must show that for every  $x \in \text{free}(R_0 \wedge R_1)$  there exists some  $\rho_x \in R_0 \wedge R_1$  such that  $x \in \text{free}(\rho_x)$  and either  $\theta \models x \uparrow_{d(x)}$  or  $\theta \models \rho_x$ . Hence, suppose that  $\theta \notin \mathcal{L}(x \uparrow_{d(x)})$ , for some variable  $x \in \text{free}(R_0 \wedge R_1)$ . Since  $\text{free}(R_0 \wedge R_1) = \text{free}(R_0) \cup \text{free}(R_1)$  and  $\theta \models \text{free}(R_0) \wedge \text{free}(R_1)$ , we find from Definition 7 some  $\psi \in R_0 \cup R_1$ , with  $\theta \models \psi$  and  $x \in \text{free}(\psi)$ . We now show that there exists an irreducible synchronous set  $S \subseteq R_0 \cup R_1$ , such that, for  $\rho_x := \bigwedge_{\rho \in S} \rho$ , we have  $\theta \models \rho_x$  and  $x \in \text{free}(\rho_x)$ . By repeated application of Definition 8, we construct a finite sequence

$$\{\psi\} = S_0 \subsetneq \cdots \subsetneq S_n,$$

such that  $S_n \subseteq R_0 \cup R_1$  is synchronous, and  $\theta \models \bigwedge_{\rho \in S_n} \rho$ . Suppose  $S_k \subseteq R_0 \cup R_1$ , for  $k \geq 1$ , is not synchronous. By Definition 8, there exists some  $i \in \{0, 1\}$  and a variable  $x \in \text{free}(S_k) \cap \text{free}(R_i)$ , such that  $x \notin \text{free}(S_k \cap R_i)$ . Since  $x \in \text{free}(R_i)$ , we have  $R_i^x := \{\rho \in R_i \mid x \in \text{free}(\rho)\} \neq \emptyset$ . Since  $\theta \models \text{rbf}(R_i)$ , there exists some  $\psi_k \in R_i^x$  such that  $\theta \models \psi_k$ . Now define  $S_{k+1} := S_k \cup \{\psi_k\}$ . Since  $x \notin \text{free}(S_k \cap R_i)$



**Fig. 2.** Hypergraph representations of  $\bigwedge_{i=1}^2 \text{fifo}(a_i, a_{i+1}, m_i)$ .

and  $x \in \text{free}(S_{k+1} \cap R_i)$ , we have a strict inclusion  $S_k \subsetneq S_{k+1}$ . Due to these strict inclusions, we have, for  $k \geq |R_0 \cup R_1|$ , that  $S_k = R_0 \cup R_1$ , which is trivially synchronous in  $R_0 \cup R_1$ . Therefore, our sequence  $S_0 \subsetneq \dots$  of inclusions terminates, from which we conclude the existence of  $S_n$ . By Lemma 3, we find some irreducible synchronous set  $S \subseteq S_n$ , such that  $\psi \in S$ . We conclude that  $\rho_x := \bigwedge_{\rho \in S} \rho \in R_0 \wedge R_1$  satisfies  $\theta \models \rho_x$  and  $x \in \text{free}(\psi) \subseteq \text{free}(S) = \text{free}(\rho_x)$ . By Definition 7, we have  $\theta \models \text{rbf}(R_0 \wedge R_1)$ , and  $\text{rbf}(R_0) \wedge \text{rbf}(R_1) \models \text{rbf}(R_0 \wedge R_1)$ .

Finally, we prove that  $\text{rbf}(R_0 \wedge R_1) \models \text{rbf}(R_0) \wedge \text{rbf}(R_1)$ . Let  $\theta \in \mathcal{L}(\text{rbf}(R_0 \wedge R_1))$ . We show that  $\theta \models \text{rbf}(R_i)$ , for all  $i \in \{0, 1\}$ . By Definition 7, we must show that for every  $i \in \{0, 1\}$  and every  $x \in \text{free}(R_i)$  there exists some  $\rho \in R_i$  such that  $x \in \text{free}(\rho)$  and either  $\theta \models x \uparrow_{d(x)}$  or  $\theta \models \rho$ . Hence, let  $i \in \{0, 1\}$  and  $x \in \text{free}(R_i)$  be arbitrary, and suppose that  $\theta \notin \mathcal{L}(x \uparrow_{d(x)})$ . Since  $\text{free}(R_i) \subseteq \text{free}(R_0 \wedge R_1)$ , it follows from our assumption  $\theta \models \text{rbf}(R_0 \wedge R_1)$  that  $\theta \models \bigwedge_{\rho \in S} \rho$ , for some irreducible synchronous set  $S \subseteq R_0 \cup R_1$  satisfying  $x \in \text{free}(S)$ . Since  $S \subseteq R_0 \cup R_1$  synchronous, we find that  $x \in \text{free}(S) \cap \text{free}(R_i) = \text{free}(S \cap R_i)$ . Hence, we find some  $\rho \in S \cap R_i$ , such that  $\theta \models \rho$  and  $x \in \text{free}(\rho)$ . By Definition 7, we conclude that  $\theta \models \text{rbf}(R_i)$ , for all  $i \in \{0, 1\}$ . Therefore,  $\text{rbf}(R_0 \wedge R_1) \simeq \text{rbf}(R_0) \wedge \text{rbf}(R_1)$ .  $\square$

*Example 16.* Let  $\varphi_i$  and  $\psi_i$ , for  $i \geq 1$ , be the rules from Example 12. By Example 14, the composition  $\text{fifo}_2 := \bigwedge_{i=1}^2 \text{fifo}(a_i, a_{i+1}, m_i)$  is defined by the set of rules  $\{\varphi_1, \psi_1 \wedge \varphi_2, \psi_2\}$ .<sup>1</sup> To compute a set of rules that defines the composition, it is not efficient to enumerate all (exponentially many) synchronous subsets of  $R_0 \cup R_1$  and remove all reducible sets. Our tools use an algorithm based on

<sup>1</sup> The rules for the composition of two fifo stream constraints has striking similarities with *synchronous region* decomposition developed by Proença et al. [24]. Indeed,  $\varphi_1$ ,  $\psi_1 \wedge \varphi_2$ , and  $\psi_2$  correspond to the synchronous regions in the composition of two buffers. Therefore, rule-based composition generalizes synchronous region decomposition that has been used as a basis for generation of parallel code [15].

hypergraph transformations to compute the irreducible synchronous sets. The details of this algorithm fall outside the scope of this paper. Figure 2 shows a graphical representation of composition  $\text{fifo}_2$ , using hypergraphs. These hypergraphs consist of sets of hyperedges  $(x, F)$ , where  $x$  is a variable and  $F$  is a set of rules. Each hyperedge  $(x, F)$  in a hypergraph corresponds to a disjunction  $x \dagger_{d(x)} \vee \bigvee_{\rho \in F} \rho$  of the rule-based form in Definition 7.  $\triangle$

## 7 Complexity

In the worst case, composition  $R_0 \wedge R_1$  of arbitrary sets of rules  $R_0$  and  $R_1$  may consist of  $|R_0| \times |R_1|$  rules. However, if  $R_0$  and  $R_1$  are simple, the size of the composition is bounded by  $|R_0| + |R_1|$ .

**Definition 11 (Simple).** *A set  $R$  of rules is simple if and only if  $\text{free}(\rho) \cap \text{free}(\rho') \cap P(\text{rbf}(R)) \neq \emptyset$  implies  $\rho = \rho'$ , for every  $\rho, \rho' \in R$ .*

*Example 17.* By Example 10, the invariant of  $\text{fifo}(a, b, m)$  is defined by  $R := \{a \doteq m' \doteq \mathbf{0} \wedge m \doteq *, m' \doteq * \wedge b \doteq m \doteq \mathbf{0}\}$  as well as  $R' := \{a \doteq m' \doteq \mathbf{0} \wedge b \doteq m \doteq *, a \doteq m' \doteq * \wedge b \doteq m \doteq \mathbf{0}\}$ . The set  $R$  is simple, while  $R'$  is not.  $\triangle$

**Lemma 5.** *Let  $R_0$  and  $R_1$  be sets of rules, such that  $\text{free}(R_0) \cap \text{free}(R_1) \subseteq P(\text{rbf}(R_0 \cup R_1))$ , and let  $S \subseteq R_0 \cup R_1$  be synchronous. Let  $G_S$  be a graph with vertices  $S$  and edges  $E_S = \{(\rho, \rho') \in S^2 \mid \text{free}(\rho) \cap \text{free}(\rho') \cap P(\text{rbf}(R_0 \cup R_1)) \neq \emptyset\}$ . If  $S$  is irreducible, then  $G_S$  is connected.*

*Proof.* Suppose that  $G_S$  is disconnected. We find  $\emptyset \neq S_0, S_1 \subseteq S$ , with  $S_0 \cup S_1 = S$ ,  $S_0 \cap S_1 = \emptyset$  and  $\text{free}(S_0) \cap \text{free}(S_1) \cap P(\text{rbf}(R_0 \cup R_1)) = \emptyset$ . We show that  $S_0$  and  $S_1$  are synchronous. Let  $i, j \in \{0, 1\}$  and  $x \in \text{free}(S_i) \cap \text{free}(R_j)$ . We distinguish two cases:

*Case 1* ( $x \in \text{free}(R_{1-j})$ ): Then,  $x \in \text{free}(R_0) \cap \text{free}(R_1) \subseteq P(\text{rbf}(R_0 \cup R_1))$ . Since  $\text{free}(S_0) \cap \text{free}(S_1) \cap P(\text{rbf}(R_0 \cup R_1)) = \emptyset$ , we have  $x \notin \text{free}(S_{1-i})$ . Since  $S$  is synchronous, we have  $x \in \text{free}(S_i) \cap \text{free}(R_j) \subseteq \text{free}(S) \cap \text{free}(R_j) \subseteq \text{free}(S \cap R_j)$ . Hence, we find some  $\rho \in S \cap R_j$ , with  $x \in \text{free}(\rho)$ . Since  $x \notin \text{free}(S_{1-i})$ , we conclude that  $\rho \in S_i \cap R_j$ . Thus,  $x \in \text{free}(S_i \cap R_j)$ , if  $x \in \text{free}(R_{1-j})$ .

*Case 2* ( $x \notin \text{free}(R_{1-j})$ ): Since  $x \in \text{free}(S_i)$ , we find some  $\rho \in S_i$ , with  $x \in \text{free}(\rho)$ . Since  $x \notin \text{free}(R_{1-j})$ , we conclude that  $\rho \in R_j$ . Hence,  $x \in \text{free}(\rho) \subseteq \text{free}(S_i \cap R_j)$ , if  $x \notin \text{free}(R_{1-j})$ .

We conclude in both cases that  $x \in \text{free}(\rho) \subseteq \text{free}(S_i \cap R_j)$ . Hence,  $\text{free}(S_i) \cap \text{free}(R_j) \subseteq \text{free}(S_i \cap R_j)$ , for all  $i, j \in \{0, 1\}$ , and we conclude that  $S_0$  and  $S_1$  are synchronous. Since  $S_0 \neq S \neq S_1$ , we conclude that  $S$  is reducible. By contraposition, we conclude that  $G_S$  is connected, whenever  $S$  is irreducible.  $\square$

**Lemma 6.** *Let  $R_0$  and  $R_1$  be simple sets of rules, with  $\text{free}(R_0) \cap \text{free}(R_1) \subseteq P(\text{rbf}(R_0 \cup R_1))$ , and let  $S_0, S_1 \subseteq R_0 \cup R_1$  be irreducible synchronous. If  $S_0 \cap S_1 \neq \emptyset$ , then  $S_0 = S_1$ .*

*Proof.* Suppose that  $S_0 \cap S_1 \neq \emptyset$ . Then, there exists some  $\rho_0 \in S_0 \cap S_1$ . We show that  $S_i \subseteq S_{1-i}$ , for all  $i \in \{0, 1\}$ . Let  $i \in \{0, 1\}$ , and  $\rho \in S_i$ . By Lemma 5, we find an undirected path in  $G_{S_i}$  from  $\rho_0$  to  $\rho$ . That is, we find a sequence  $\rho_0 \rho_1 \cdots \rho_n \in S^*$ , such that  $\rho_n = \rho$  and  $(\rho_i, \rho_{i+1}) \in E_{S_i}$ , for all  $0 \leq i < n$ . We show by induction on  $n \geq 0$ , that  $\rho_n \in S_{1-i}$ . For the base case ( $n = 0$ ), observe that  $\rho_n = \rho_0 \in S_0 \cap S_1 \subseteq S_{1-i}$ . For the induction step, suppose that  $\rho_n \in S_{1-i}$ . By construction of  $G_{S_i}$ , we find that  $\text{free}(\rho_n) \cap \text{free}(\rho_{n+1}) \cap P_{01} \neq \emptyset$ , where  $P_{01} = P(\text{rbf}(R_0 \cup R_1))$ . Let  $j \in \{0, 1\}$ , such that  $\rho_{n+1} \in R_j$ . Since  $\rho_n \in S_{1-i}$  and  $S_{1-i}$  is synchronous, we have  $\emptyset \neq \text{free}(S_{1-i}) \cap \text{free}(R_j) \cap P_{01} = \text{free}(S_{1-i} \cap R_j) \cap P_{01}$ . We find some  $\rho' \in S_{1-j} \cap R_j$ , with  $\text{free}(\rho_{n+1}) \cap \text{free}(\rho') \cap P_{01} \neq \emptyset$ . Since  $R_j$  is simple, we have  $\rho_{n+1} = \rho' \in S_{1-i}$ , which concludes the proof by induction. It follows from  $\rho_n \in S_{1-i}$  that  $S_i \subseteq S_{1-i}$ , for all  $i \in \{0, 1\}$ , that is,  $S_0 = S_1$ .  $\square$

As seen in Lemma 2, the number of clauses in the disjunctive normal form  $\text{dnf}(R_0 \wedge R_1)$  can be exponential in the number of rules  $|R_0 \wedge R_1|$  of the composition of  $R_0$  and  $R_1$ . However, the following (main) theorem shows the number of rules required to define  $\bigwedge_i \phi_i$  is only linear in  $k$ .

**Theorem 4.** *If  $R_0$  and  $R_1$  are simple sets of rules, and  $\text{free}(R_0) \cap \text{free}(R_1) \subseteq P(\text{rbf}(R_0 \cup R_1))$ , then  $R_0 \wedge R_1$  is simple and  $|R_0 \wedge R_1| \leq |R_0| + |R_1|$ .*

*Proof.* From Lemmas 3 and 6, we find that the irreducible synchronous subsets partition  $R_0 \cup R_1$ . We conclude that  $|R_0 \wedge R_1| \leq |R_0| + |R_1|$ . We now show that  $R_0 \wedge R_1$  is simple. Let  $\rho_0$  and  $\rho_1$  be rules in  $R_0 \wedge R_1$ , with  $\text{free}(\rho_0) \cap \text{free}(\rho_1) \cap P_{01} \neq \emptyset$ , where  $P_{01} = P(\text{rbf}(R_0 \cup R_1))$ . By Definition 10, we find, for all  $i \in \{0, 1\}$ , an irreducible synchronous set  $S_i$ , such that  $\rho_i = \bigwedge_{\psi \in S_i} \psi$ . Since  $\text{free}(\rho_0) \cap \text{free}(\rho_1) \cap P_{01} \neq \emptyset$  and  $\text{free}(\rho_i) = \text{free}(S_i)$ , for all  $i \in \{0, 1\}$ , we find some  $x \in \text{free}(S_0) \cap \text{free}(S_1) \cap P_{01}$ . Suppose that  $x \in \text{free}(R_j)$ , for some  $j \in \{0, 1\}$ . Since  $S_0$  and  $S_1$  are synchronous sets, we have  $x \in \text{free}(S_i) \cap \text{free}(R_j) \subseteq \text{free}(S_i \cap R_j)$ , for all  $i \in \{0, 1\}$ . We find, for all  $i \in \{0, 1\}$ , some  $\psi_i \in S_i \cap R_j$ , such that  $x \in \text{free}(\psi_i)$ . Hence,  $\text{free}(\psi_0) \cap \text{free}(\psi_1) \cap P_{01} \neq \emptyset$ , and since  $R_j$  is simple, we conclude that  $\psi_0 = \psi_1$ . Therefore,  $S_0 \cap S_1 \neq \emptyset$ , and Lemma 6 shows that  $S_0 = S_1$  and  $\rho_0 = \rho_1$ . We conclude that  $R_0 \wedge R_1$  is simple.  $\square$

The number of clauses in the disjunctive normal form of direct compositions of  $k$  fifo constraints grows exponentially in  $k$ . This typical pattern of a sequence of queues manifests itself in many other constructions, which causes serious scalability problems (cf., the benchmarks for ‘Alternator $_k$ ’ in [17, Sect. 7.2]). However, Theorem 4 shows that rule-based composition of  $k$  fifo constraints does not suffer from scalability issues: by Example 17, the fifo constraint can be defined by a simple set of rules. The result in Theorem 4, therefore, promises (exponential) improvement over the classical constraint automaton representation.

Unfortunately, it seems impossible to define any arbitrary stream constraint by a simple set of rules. Therefore, the rule-based form may still blow up for certain stream constraints. It seems, however, possible to recognize even more symmetries (cf., the queue-optimization in [16]) to avoid explosion and obtain comparable compilation and execution performance for these stream constraints.

## 8 Abstraction

We now study how existential quantification of stream constraints operates on its defining set of rules.

**Definition 12 (Abstraction).** *Hiding a variable  $x$  in a set of rules  $R$  yields  $\exists xR := \{\exists x\rho \mid \rho \in R\}$ .*

Unfortunately,  $\exists xR$  does not always define  $\exists x\phi$ , for a stream constraint  $\phi$  defined by a set of rules  $R$ . The following result shows that  $\exists xR$  defines  $\exists x\phi$  if and only if  $\text{rbf}(\exists xR) \models \exists x \text{rbf}(R)$ . In this case, we call variable  $x$  *hidable* in  $R$ .

It is non-trivial to find a defining set of rules for  $\exists x\phi$ , if  $x$  is not hidable in  $R$ , and we leave this as future work.

**Theorem 5.** *Let  $R$  be a set of rules, and let  $x \in X$  be a variable. Then,  $\exists x \text{rbf}(R) \simeq \text{rbf}(\exists xR)$  if and only if  $\text{rbf}(\exists xR) \models \exists x \text{rbf}(R)$ .*

*Proof.* Trivially,  $\exists x \text{rbf}(R) \simeq \text{rbf}(\exists xR)$  implies  $\text{rbf}(\exists xR) \models \exists x \text{rbf}(R)$ . Conversely, suppose that  $\text{rbf}(\exists xR) \models \exists x \text{rbf}(R)$ . From Lemma 2, it follows that  $\exists x \text{rbf}(R) \equiv \exists x \text{dnf}(R)$ . Since existential quantification distributes over disjunction and  $\exists x\phi \wedge \psi \models \exists x\phi \wedge \exists x\psi$ , for all stream constraints  $\phi$  and  $\psi$ , we find

$$\exists x \text{dnf}(R) \models \bigvee_{S \subseteq R} \bigwedge_{\rho \in S} \exists x\rho \wedge \bigwedge_{x \neq y \in \text{free}(R) \setminus \text{free}(S)} y \dagger_{d(y)} \equiv \text{dnf}(\exists xR).$$

By Lemma 2, we have  $\exists x \text{rbf}(R) \models \text{rbf}(\exists xR)$ , and by assumption  $\exists x \text{rbf}(R) \equiv \text{rbf}(\exists xR)$ . Using Lemma 4, we have  $\text{deg}_y(\exists x \text{rbf}(R)) = \max_{\rho \in R} \text{deg}_y(\exists x\rho) = \text{deg}_y(\text{rbf}(\exists xR))$ , for every variable  $y$ . We conclude  $\exists x \text{rbf}(R) \simeq \text{rbf}(\exists xR)$ .  $\square$

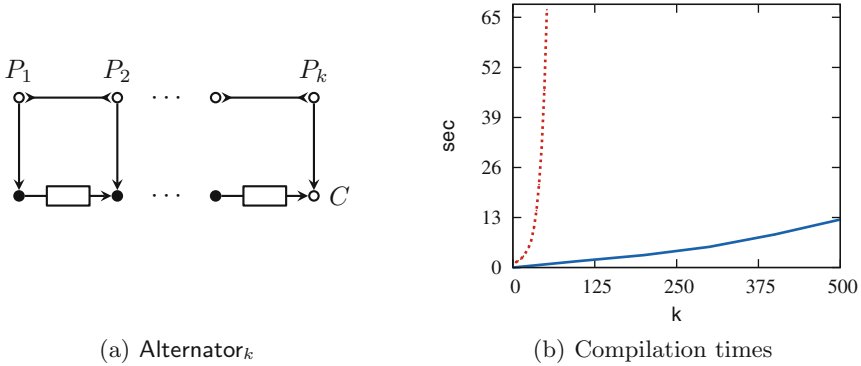
*Example 18.* Suppose  $\text{Data} = \{0, 1\}$ , which means that the data domain equals  $D = \{0, 1, *\}$ . Let  $\mathbf{1}$  be the constant stream defined as  $\mathbf{1}(n) = 1$ , for all  $n \in \mathbb{N}$ . For  $i \in \{0, 1\}$ , consider the set of rules  $R_i = \{x = x, x = y_i = \mathbf{i}\}$ . Observe that  $\{x = x, x = y_i = \mathbf{i}\} \subseteq R_0 \cup R_1$  is synchronous, for all  $i \in \{0, 1\}$ . Hence,  $x = y_i = \mathbf{i} \in R_0 \wedge R_1$ , for all  $i \in \{0, 1\}$ . However, for  $\theta = [y_0 \mapsto \mathbf{0}, y_1 \mapsto \mathbf{1}]$ , we have  $\theta \models \bigwedge_{i \in \{0, 1\}} \exists x(x = y_i = i)$ , while  $\exists x \bigwedge_{i \in \{0, 1\}} x = y_i = i \equiv \perp$ . Thus, variable  $x$  is not hidable from  $R_0 \wedge R_1$ .  $\triangle$

## 9 Application

In on-going work, we applied the rule-based form to compile protocols (in the form of Reo connectors) into executable code. Reo is an exogenous coordination language that models protocols as graph-like structures [1, 2]. We recently developed a textual version of Reo, which we use to design non-trivial protocols [11]. An example of such non-trivial protocol is the *Alternator<sub>k</sub>*, where  $k \geq 2$  is an integer. Figure 3(a) shows a graphical representation of the *Alternator<sub>k</sub>* protocol.

Intuitively, the behavior of the alternator protocol is as follows: The *nodes*  $P_1, \dots, P_k$  accept data from the environment. Node  $C$  offer data to the environment. All other nodes are internal and do not interact with the environment.





**Fig. 3.** Graphical representation (a) of the  $\text{Alternator}_k$  protocol in [17], for  $2 \leq k \leq 500$ , and its compilation time (b). The dotted red line is produced by the Jongmans' compiler (and corresponds to [17, Fig.11(a)]), and the solid blue line is our compiler. (Color figure online)

In the first step of the protocol, the  $\text{Alternator}_k$  waits until the environment is ready to offer data at all nodes  $P_1, \dots, P_k$  and is ready to accept data from node  $C$ . Only then, the  $\text{Alternator}_k$  transfers the data from  $P_k$  to  $C$  via a synchronous channel, and puts the data from  $P_i$  in the  $i$ -th fifo channel, for all  $i < k$ . The behavior of a synchronous channel is defined by the sync stream constraint in Example 1. Each fifo channel has buffer capacity of one, and its behavior is defined by the fifo stream constraint from Example 3. In subsequent steps, the environment can one-by-one retrieve the data from the fifo channel buffers, until they are all empty. Then, the protocol cycles back to its initial configuration, and repeats its behavior. For more details on the Reo language and its semantics, we refer to [1, 2].

As mentioned in the introduction, Jongmans developed a compiler based on constraint automata [17]. The otherwise stimulating benchmarks presented in [17] show that Jongmans' compiler still suffers from state-space explosion. Figure 3(b) shows the compilation time of the  $\text{Alternator}_k$  protocol for Jongmans' compiler and ours. Clearly, the compilation time improved drastically and went from exponential in  $k$  to almost linear in  $k$ .

Every fifo channel in the  $\text{Alternator}_k$ , except the first, either accepts data from the environment or accepts data from the previous fifo channel. This choice is made by the internal node at the input of each fifo channel. Unfortunately, the behavior of such nodes is not defined in terms of a simple set of rules. Consequently, we cannot readily apply Theorem 4 to conclude that the number of rules depends only linearly on  $k$ . However, it turns out that  $\text{Alternator}_k$  can be defined using only  $k$  rules: one rule for filling the buffers of all fifo channels, plus  $k - 1$  rules, one for taking data out of the buffer of each of the  $k - 1$  fifo channels. This observation explains why our compiler drastically improves upon Jongmans' compiler.

## 10 Conclusion

We introduce (regular) stream constraints as an alternative to constraint automata that does not suffer from state space explosions. We define the rule-based form for stream constraints, and we express composition and abstraction of constraints in terms of their rule-based forms. For simple sets of rules, composition of rule-based forms does not suffer from ‘transition space explosions’ either.

We have experimented with a new compiler for protocols using our rule-based form, which avoids the scalability problems of state- and transition-space explosions of previous automata-based tools. Our approach still leaves the possibility for transition space explosion for non-simple sets of rules. In the future, we intend to study symmetries in stream constraints that are not defined by simple sets of rules. The queue-optimization of Jongmans serves as a good source of inspiration for exploiting symmetries [16].

The results in this paper are purely theoretical. In on-going work, we show practical implications of our results by developing a compiler based on stream constraints. Such a compiler requires an extension to the current theory on stream constraints: we did not compute the abstraction  $\exists xR$  on sets of rules  $R$  wherein variable  $x$  is not hidable. Example 11 indicates the existence of situations where we can compute  $\exists xR$  even if  $x$  is not hidable, a topic which we leave as future work.

**Acknowledgements.** The authors thank Benjamin Lion for his help in developing a rule-based compiler and for generating Fig. 3(b).

## References



1. Arbab, F.: Reo: a channel-based coordination model for component composition. *Math. Struct. Comput. Sci.* **14**(3), 329–366 (2004). <https://doi.org/10.1017/S0960129504004153>
2. Arbab, F.: Puff, the magic protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) *Formal Modeling: Actors, Open Systems, Biological Systems*. LNCS, vol. 7000, pp. 169–206. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-24933-4\\_9](https://doi.org/10.1007/978-3-642-24933-4_9)
3. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) *WADT 2002*. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003). [https://doi.org/10.1007/978-3-540-40020-2\\_2](https://doi.org/10.1007/978-3-540-40020-2_2)
4. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A uniform framework for modeling and verifying components and connectors. In: Field, J., Vasconcelos, V.T. (eds.) *COORDINATION 2009*. LNCS, vol. 5521, pp. 247–267. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-02053-7\\_13](https://doi.org/10.1007/978-3-642-02053-7_13)
5. Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M., Ryan, M.: Symbolic model checking for probabilistic processes. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) *ICALP 1997*. LNCS, vol. 1256, pp. 430–440. Springer, Heidelberg (1997). [https://doi.org/10.1007/3-540-63165-8\\_199](https://doi.org/10.1007/3-540-63165-8_199)

6. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. *Sci. Comput. Program.* **61**(2), 75–113 (2006). <https://doi.org/10.1016/j.scico.2005.10.008>
7. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.* **35**(8), 677–691 (1986). <https://doi.org/10.1109/TC.1986.1676819>
8. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking:  $10^{20}$  states and beyond. *Inf. Comput.* **98**(2), 142–170 (1992). [https://doi.org/10.1016/0890-5401\(92\)90017-A](https://doi.org/10.1016/0890-5401(92)90017-A)
9. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Channel-based coordination via constraint satisfaction. *Sci. Comput. Program.* **76**(8), 681–710 (2011). <https://doi.org/10.1016/j.scico.2010.05.004>
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-3-540-71999-1>
11. Dokter, K., Arbab, F.: Treo: textual syntax of reo connectors. In: Proceedings of MeTRiD 2018 (2018, to appear)
12. Ehlers, R.: Minimising deterministic Büchi automata precisely using SAT solving. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 326–332. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-14186-7\\_28](https://doi.org/10.1007/978-3-642-14186-7_28)
13. Jongmans, S.T.Q.: Automata-theoretic protocol programming. Ph.D. thesis, Centrum Wiskunde & Informatica (CWI), Faculty of Science, Leiden University (2016). <http://hdl.handle.net/1887/38223>
14. Jongmans, S.-S.T.Q., Arbab, F.: Take Command of Your Constraints! In: Proceedings of Coordination Models and Languages - 17th IFIP WG 6.1 International Conference, COORDINATION 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, pp. 117–132 (2015). [https://doi.org/10.1007/978-3-319-19282-6\\_8](https://doi.org/10.1007/978-3-319-19282-6_8)
15. Jongmans, S.T.Q., Arbab, F.: Centralized coordination vs. partially-distributed coordination with Reo and constraint automata. *Sci. Comput. Program.* (2017). <http://www.sciencedirect.com/science/article/pii/S0167642317301259>
16. Jongmans, S.-S.T.Q., Halle, S., Arbab, F.: Automata-based optimization of interaction protocols for scalable multicore platforms. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 65–82. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-662-43376-8\\_5](https://doi.org/10.1007/978-3-662-43376-8_5)
17. Jongmans, S.T.Q., Kappé, T., Arbab, F.: Constraint automata with memory cells and their composition. *Sci. Comput. Program.* **146**, 50–86 (2017). <https://doi.org/10.1016/j.scico.2017.03.006>
18. Kemper, S.: SAT-based verification for timed component connectors. *Sci. Comput. Program.* **77**(7–8), 779–798 (2012). <https://doi.org/10.1016/j.scico.2011.02.003>
19. Klüppelholz, S.: Verification of Branching-Time and Alternating-Time Properties for Exogenous Coordination Models. Ph.D. thesis, Dresden University of Technology (2012). [http://www.qucosa.de/recherche/frontdoor/?tx\\_slubopus4frontend\[id\]=8621](http://www.qucosa.de/recherche/frontdoor/?tx_slubopus4frontend[id]=8621)
20. Kwiatkowska, M.Z., Norman, G., Sproston, J., Wang, F.: Symbolic model checking for probabilistic timed automata. *Inf. Comput.* **205**(7), 1027–1077 (2007). <https://doi.org/10.1016/j.ic.2007.01.004>
21. Murata, T.: Petri nets: properties, analysis and applications. *Proc. IEEE* **77**(4), 541–580 (1989)

22. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, 11–13 January 1989, pp. 179–190. ACM Press (1989). <http://doi.acm.org/10.1145/75277.75293>
23. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: 31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, 22–24 October 1990, vol. II, pp. 746–757. IEEE Computer Society (1990). <https://doi.org/10.1109/FSCS.1990.89597>
24. Proença, J., Clarke, D., de Vink, E.P., Arbab, F.: Dreams: a framework for distributed synchronous coordination. In: Ossowski, S., Lecca, P. (eds.) Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, 26–30 March 2012, pp. 1510–1515. ACM (2012). <http://doi.acm.org/10.1145/2245276.2232017>
25. Rutten, J.J.M.M.: Elements of stream calculus (an extensive exercise in coinduction). *Electr. Notes Theor. Comput. Sci.* **45**, 358–423 (2001). [https://doi.org/10.1016/S1571-0661\(04\)80972-1](https://doi.org/10.1016/S1571-0661(04)80972-1)



# Forward to a Promising Future

Kiko Fernandez-Reyes , Dave Clarke  , Elias Castegren ,  
and Huu-Phuc Vo

Department of Information Technology,  
Uppsala University, Uppsala, Sweden  
`dave.clarke@it.uu.se`

**Abstract.** In many actor-based programming models, asynchronous method calls communicate their results using futures, where the fulfilment occurs under-the-hood. Promises play a similar role to futures, except that they must be explicitly created and explicitly fulfilled; this makes promises more flexible than futures, though promises lack fulfilment guarantees: they can be fulfilled once, multiple times or not at all. Unfortunately, futures are too rigid to exploit many available concurrent and parallel patterns. For instance, many computations block on a future to get its result only to return that result immediately (to fulfil their own future). To make futures more flexible, we explore a construct, *forward*, that delegates the responsibility for fulfilling the current implicit future to another computation. *Forward* reduces synchronisation and gives futures promise-like capabilities. This paper presents a formalisation of the *forward* construct, defined in a high-level source language, and a compilation strategy from the high-level language to a low-level, promised-based target language. The translation is shown to preserve semantics. Based on this foundation, we describe the implementation of *forward* in the parallel, actor-based language Encore, which compiles to C.

## 1 Introduction

Futures extend the actor programming model (This paper focuses on futures. From this perspective we consider the actor-, task-, and active object-based models as synonymous.) to express call-return synchronisation of message sends [1]. Each actor is single-threaded, but different actors execute concurrently. Communication between actors happens via asynchronous method calls (messages), which immediately return a future; futures are placeholders for the eventual result of these asynchronous method calls. An actor processes one message at a time and each message has associated a future that will be fulfilled with the returned value of the method. Futures are first-class values, and operations on them may be blocking, such as getting the result out of the future (`get`), or asynchronous, such as attaching a callback to a future. This last operation, known as

future chaining ( $f \rightsquigarrow^x e$ ), attaches a closure  $\lambda x.e$  to the future  $f$  and immediately returns a new future that will contain the result of applying the closure to the value eventually stored in future  $f$ .

Consider the following code (in the actor-based language Encore (<https://github.com/paraplou/encore>) [2]) that implements the broker delegation pattern: the developer's intention is to connect clients (the callers of the `Broker` actor) to a pool of actors that will actually process a job (lines 6–7):

```

1  active class Broker
2    val workers: Buffered[Worker]
3    var current: uint
4
5    def run(job: Job): int
6      val worker = this.workers[++this.current % workers.size()]
7      val future : Fut[int] = worker!start(job)
8      return get(future)
9    end
10 end

```

The problem with this code is that the connection to the `Broker` cannot be completed immediately without blocking the `Broker`'s thread of execution: returning the result of the worker running the computation requires that the `Broker` blocks until the future is fulfilled (line 8). This implementation makes the `Broker` the bottleneck of the application.

One obvious way to avoid this bottleneck is by returning the future, instead of blocking on it, as in the following code:

```

1  def run(job: Job): Fut[int]
2    val worker = this.workers[++this.current % workers.size()]
3    return worker!start(job)
4  end

```

This solution removes the blocking from `Broker`, but returns a future, which results in the client receiving a future containing a future `Fut (Fut int)`, cluttering client code and making the typing more complex.

Another way to avoid the bottleneck is to not block but yield the current thread until the future is fulfilled. This can be done using the `await` command [2,3], which frees up the `Broker` to do other work:<sup>1</sup>

```

1  def run(job: Job): int
2    val worker = this.workers[++this.current % workers.size()]
3    val future = worker!start(job)
4    await(future)
5    return get(future)
6  end

```

This solution frees up the `Broker`, but can result in a lot of memory being consumed to hold the waiting instances of calls `Broker.run()`.

Another alternative is to use promises [4]. A promise can be passed around and fulfilled explicitly at the point where the corresponding result is known.

<sup>1</sup> The essential difference between `get` and `await` is that `get` blocks an actor, whereas `await` blocks only the current method invocation and frees up the actor.

Passing a promise around is akin to passing the responsibility to provide a particular result, thereby fulfilling the promise.

```

1  def run(job: Job, promise: Promise[int]): unit
2    val worker = this.workers[++this.current % workers.size()]
3    worker!start(job, promise)
4  end
5
6  class Worker
7    def start(job: Job, promise: Promise[int]) : unit
8      // actually do job
9      promise.fulfil(result)
10   end
11 end

```

Promises are problematic because they diverge from the commonplace call-return control flow, there is no explicit requirement to actually fulfil a promise, and care is required to avoid fulfilling multiple times. This latter issue, fulfilling a promise multiple times, can be solved by a substructural type system, which guarantees a single writer to the promise [5,6]. Substructural type systems are more complex and not mainstream, which rules out adoption in languages such as Java and C#. Our solution relies on futures and is suitable for mainstream languages.

The main difference between promises and futures are that developers *explicitly* create and fulfil promises, whereas futures are *implicitly* created and fulfilled. Promises are thus more flexible at the expense of any fulfilment guarantees.

This paper explores a construct called *forward* that retains the guarantees of using futures, while allowing some degree of delegation of responsibility to fulfil a future, as in promises. This construct was first proposed a while ago [7], but only recently has been implemented in the language Encore [2].

With *forward*, the `run` of `Broker` method now becomes:

```

1  def run(job: Job): int
2    val worker = this.workers[++this.current % workers.size()]
3    forward(worker!start(job))
4  end

```

*Forward* delegates the fulfilment of the future that `run` will put its result in, to the call `worker!start(job)`. Using *forward* frees up the `Broker` object, as `run` completes immediately, though the future is fulfilled only when `worker!start(job)` produces a result.

The paper makes the following contributions:

- a formalisation and soundness proof of the *forward* construct in a concise, high-level language (Sect. 2);
- a formalisation of a low-level, promise-based language (Sect. 3),
- a translation from the high-level language to the low-level language, a proof of program equivalence, between the high-level language and its translation to the low-level language (Sect. 4); and
- microbenchmarks that compare the `get-and-return` and `await-and-get` pattern versus the `forward` construct (Sect. 5).

## 2 A Core Calculus of Futures and Forward

This section presents a core calculus that includes tasks, futures and operations on them, and forward. The calculus consists of two levels: expressions and configurations. Expressions correspond to programs and what tasks evaluate. Configurations capture the run-time configuration; they are collections of tasks (running expressions), futures, and chains. This calculus is much more concise than the previous formalisation of forward [7].

The syntax of the core calculus is as follows:

$$\begin{aligned} e &::= v \mid e e \mid \mathbf{async} e \mid e \overset{x}{\rightsquigarrow} e \mid \mathbf{if} e \mathbf{then} e \mathbf{else} e \mid \mathbf{forward} e \mid \mathbf{get} e \\ v &::= c \mid f \mid x \mid \lambda x.e \end{aligned}$$

Expressions include values ( $v$ ), function application ( $e e$ ), spawning asynchronous computations ( $\mathbf{async} e$ ), future chaining ( $e \overset{x}{\rightsquigarrow} e'$ ), which attaches  $\lambda x.e'$  onto a future to run as soon as the future produced by  $e$  is fulfilled, *if-then-else* expressions,  $\mathbf{forward}$ , and  $\mathbf{get}$ , which extracts the value from a future. Values are constants ( $c$ ), futures ( $f$ ), variables ( $x$ ) and lambda abstractions ( $\lambda x.e$ ). The calculus has neither actors nor message sends/method calls. For our purposes, tasks play the role of actors and spawning asynchronous computations is analogous to message sends.

Configurations, *config*, give a partial view on the system and are (non-empty) multisets of tasks, futures and chains. They have the following syntax:

$$\mathit{config} ::= (\mathit{fut}_f) \mid (\mathit{fut}_f v) \mid (\mathit{task}_f e) \mid (\mathit{chain}_f f e) \mid \mathit{config} \mathit{config}$$

Future configurations are  $(\mathit{fut}_f)$  and  $(\mathit{fut}_f v)$ , representing an unfulfilled future  $f$  and a fulfilled future  $f$  with value  $v$ . Configuration  $(\mathit{task}_f e)$  is a task running expression  $e$  that will write the result of  $e$  in future  $f$ .<sup>2</sup> Configuration  $(\mathit{chain}_f f e)$  denotes a computation that waits until future  $g$  is fulfilled, applies expression  $e$  to the value stored in  $g$  in a new task whose result will be stored in future  $f$ .

The initial configuration for program  $e$  is  $(\mathit{task}_f e) (\mathit{fut}_f)$ , where the result of  $e$  will be written into future  $f$  at the end result of the program's execution.

### 2.1 Operational Semantics

The operational semantics use a small-step semantics with reduction-based, contextual rules for evaluation within tasks. Evaluation contexts  $E$  contains a hole  $\bullet$  that denotes where the next reduction step happens [8]:

$$E ::= \bullet \mid E e \mid v E \mid E \overset{x}{\rightsquigarrow} e \mid \mathbf{forward} E \mid \mathbf{get} E \mid \mathbf{if} E \mathbf{then} e \mathbf{else} e$$

<sup>2</sup> A reviewer suggested that  $(\mathit{fut}_f)$ ,  $(\mathit{fut}_f v)$ , and  $(\mathit{task}_f e)$  could be combined into a single configuration component. We have considered this conflation in the past. While it would reduce the complexity of the calculus, it would also make compilation into the target calculus and the proofs of correctness more complex.



---


$$\begin{array}{c}
\text{(RED-IF-TRUE)} \qquad \qquad \qquad \text{(RED-}\beta\text{)} \\
(task_f E[\text{if true then } e \text{ else } e']) \rightarrow (task_f E[e]) \qquad (task_f E[\lambda x.e v]) \rightarrow (task_f E[e[v/x]]) \\
\\
\text{(RED-IF-FALSE)} \qquad \qquad \qquad \text{(RED-FWD-FUT)} \\
(task_f E[\text{if false then } e \text{ else } e']) \rightarrow (task_f E[e']) \qquad (task_f E[\text{forward } h]) \rightarrow (chain_f h \lambda x.x) \\
\\
\text{(RED-CHAIN-RUN)} \qquad \qquad \qquad \text{(RED-GET)} \\
(chain_g f e) (fut_f v) \rightarrow (task_g (e v)) (fut_f v) \qquad (task_f E[\text{get } h]) (fut_h v) \rightarrow (task_f E[v]) (fut_h v) \\
\\
\text{(RED-FUT-FULFIL)} \qquad \qquad \qquad \text{(RED-ASYNC)} \\
(task_f v) (fut_f) \rightarrow (fut_f v) \qquad \frac{fresh f}{(task_g E[\text{async } e]) \rightarrow (fut_f) (task_f e) (task_g E[f])} \\
\\
\text{(RED-CHAIN-CREATE)} \\
\frac{fresh g}{(task_f E[h \overset{x}{\rightarrow} e]) \rightarrow (fut_g) (chain_g h \lambda x.e) (task_f E[g])}
\end{array}$$


---

**Fig. 1.** Reduction rules.  $f, g, h$  range over futures.

---


$$\frac{config \rightarrow config''}{config \ config' \rightarrow config'' \ config'} \qquad \frac{config \equiv config' \quad config' \rightarrow config'' \quad config'' \equiv config'''}{config \rightarrow config''}$$


---

**Fig. 2.** Configuration evaluation rules. Equivalence  $\equiv$  (omitted) captures the fact that configurations are a multiset of basic configurations.

The evaluation rules are given in Fig. 1. The evaluation of *if-then-else* expressions and functions applications proceed in the standard fashion (RED-IF-TRUE, RED-IF-FALSE, and RED- $\beta$ ). The *async* construct spawns a new task to execute the given expression, and creates a new future to store its result (RED-ASYNC). When the spawned task finishes its execution, it places the value in the designated future (RED-FUT-FULFIL). To obtain the contents of a future, the blocking construct *get* stops the execution of the task until the future is fulfilled (RED-GET). Chaining an expression on a future results immediately in a new future that will eventually contain the result of evaluating the expression, and a chain configuration storing the expression is connected with the original future (RED-CHAIN-CREATE). When the future is fulfilled, any chain configurations become task configurations and start evaluating the stored expression on the value stored in the future (RED-CHAIN-RUN). Forward applies to a future where the result of the future computation will be the result of the current computation, stored in the future associated with the current task. Forwarding to future  $h$  throws away the remainder of the body of the current task and chains the identity function on the future, the effect of which is to copy the eventual result stored in  $h$  into the current future (RED-FWD-FUT).

The configuration evaluation rules (Fig. 2) describe how configurations make progress, which is either by some subconfiguration making progress, or by

rewriting a configuration to one that will make progress using the equations of multisets.

**Example and Optimisations.** The following example illustrates some aspects of the calculus.

$$\begin{aligned}
& (task_f E[async (forward h)]) (fut_h 42) \\
& \xrightarrow{\text{RED-ASYNC}} (task_f E[g]) (fut_h 42) (fut_g) (task_g forward h) \\
& \xrightarrow{\text{RED-FWD-FUT}} (task_f E[g]) (fut_h 42) (fut_g) (chain_g h \lambda x.x) \\
& \xrightarrow{\text{RED-CHAIN-RUN}} (task_f E[g]) (fut_h 42) (fut_g) (task_g (\lambda x.x) 42) \\
& \xrightarrow{\text{RED-}\beta} (task_f E[g]) (fut_h 42) (fut_g) (task_g 42) \\
& \xrightarrow{\text{RED-FUT-FULFIL}} (task_f E[g]) (fut_h 42) (fut_g 42)
\end{aligned}$$

Firstly, a new task is spawned with the use of `async`. This task forwards the responsibility to fulfil its future to (the task fulfilling) future  $h$ , i.e. future  $g$  gets fulfilled with the value contained in future  $h$ .

Two special cases of `forward` can be given more direct reduction sequences, which correspond to optimisations performed in the Encore compiler. The first case corresponds to forwarding directly to another method call, which is the primary use case for `forward`, namely, forwarding to another method `forward(elm())`. The optimised reduction rule is

$$(task_f E[forward(async e)]) \rightarrow (task_f e)$$

For comparison, the standard reduction sequence<sup>3</sup> is

$$\begin{aligned}
& (task_f E[forward (async e)]) \rightarrow (task_f E[forward g]) (task_g e) (fut_g) \\
& \rightarrow (chain_f g \lambda x.x) (task_g e) (fut_g) \rightarrow^* (chain_f g \lambda x.x) (task_g v) (fut_g) \\
& \rightarrow (chain_f g \lambda x.x) (fut_g v) \rightarrow (task_f (\lambda x.x) v) (fut_g v) \rightarrow (task_f v) (fut_g v)
\end{aligned}$$

This can be seen as equivalent to the reduction sequence

$$(task_f E[forward (async e)]) \rightarrow (task_f e) \rightarrow^* (task_f v)$$

because the future  $g$  will no longer be accessible.

Similarly, forwarding a future chain can be reduced directly to a chain configuration:

$$(task_f E[forward (h \overset{x}{\rightsquigarrow} e)]) \rightarrow (chain_f h \lambda x.e)$$

In both cases, `forward` can be seen as making a call-with-current-future.

<sup>3</sup>  $\rightarrow^*$  is the reflexive, transitive closure of the reduction relation  $\rightarrow$ .

## 2.2 Static Semantics

The type system has basic types,  $K$ , and future types:

$$\tau ::= K \mid Fut \tau$$

(T-CONSTANT) $\frac{c \text{ is a constant of type } \tau}{\Gamma \vdash_{\rho} c : \tau}$	(T-FUTURE) $\frac{f : Fut \tau \in \Gamma}{\Gamma \vdash_{\rho} f : Fut \tau}$	(T-VARIABLE) $\frac{x : \tau \in \Gamma}{\Gamma \vdash_{\rho} x : \tau}$	(T-ABSTRACTION) $\frac{\Gamma, x : \tau \vdash_{\bullet} e : \tau'}{\Gamma \vdash_{\rho} \lambda x. e : \tau \rightarrow \tau'}$
(T-APPLICATION) $\frac{\Gamma \vdash_{\rho} e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash_{\rho} e_2 : \tau}{\Gamma \vdash_{\rho} e_1 e_2 : \tau'}$	(T-IF-THEN-ELSE) $\frac{\Gamma \vdash_{\rho} e : bool \quad \Gamma \vdash_{\rho} e' : \tau \quad \Gamma \vdash_{\rho} e'' : \tau}{\Gamma \vdash_{\rho} \text{if } e \text{ then } e' \text{ else } e'' : \tau}$		(T-GET) $\frac{\Gamma \vdash_{\rho} e : Fut \tau}{\Gamma \vdash_{\rho} \text{get } e : \tau}$
(T-ASYNC) $\frac{\Gamma \vdash_{\tau} e : \tau}{\Gamma \vdash_{\rho} \text{async } e : Fut \tau}$	(T-CHAIN) $\frac{\Gamma \vdash_{\rho} e : Fut \tau \quad \Gamma, x : \tau \vdash_{\tau'} e' : \tau'}{\Gamma \vdash_{\rho} e \overset{x}{\rightsquigarrow} e' : Fut \tau'}$		(T-FORWARD) $\frac{\Gamma \vdash_{\rho} e : Fut \rho \quad \rho \neq \bullet}{\Gamma \vdash_{\rho} \text{forward } e : \tau}$

**Fig. 3.** Typing rules

The typing rules (Fig. 3) define the judgement  $\Gamma \vdash_{\rho} e : \tau$ , which states that in the typing environment  $\Gamma$ , which gives the types of futures and free variables, expression  $e$  has type  $\tau$ , where  $\rho$  is the *expected task type*, the result type of the task in which the expression appears.  $\rho$  ranges over both types  $\tau$  and symbol  $\bullet$  which is not a type.  $\bullet$  is used to prevent the use of **forward** in contexts where the expected task type is not clear, specifically within closures, as a closure can be passed between tasks and run in a context different from their defining contexts. The types of constants are assumed to be provided (Rule T-CONSTANT). Variables and futures types are defined in the typing environment (Rules T-VARIABLE and T-FUTURE). Function application and abstraction have the standard typing rules (Rules T-APPLICATION and T-ABSTRACTION), except that within the body of a closure the expected task type is not known. When **async** is applied to an expression  $e$ , a new task is created and the expected task type changes to the type of the expression. The result type of the **async** call is a future type of the expression's type (Rule T-ASYNC). Chaining is essentially mapping for the *Fut* type constructor, and rule T-CHAIN reflects this fact. In addition, because chaining ultimately creates a new task to run the expression, the expected task type  $\rho$  changes to the return type of the expression. Getting the value from a future of some type results in a value of that type (Rule T-GET). Forwarding requires the argument to **forward** to be a future of the same type as the expected task type (Rule T-FORWARD). As **forward** does not return locally, the result type is arbitrary.

Well-formed configurations,  $\Gamma \vdash \text{config ok}$ , are typed against environment,  $\Gamma$ , that gives the types of futures (Fig. 4). The type rules depend on the following definitions.

---

$\frac{\text{(FUT)}}{f \in \text{dom}(\Gamma)} \quad \frac{\text{(F-FUT)}}{f : \text{Fut } \tau \in \Gamma \quad \Gamma \vdash_{\bullet} v : \tau} \quad \frac{\text{(TASK)}}{f : \text{Fut } \tau \in \Gamma \quad \Gamma \vdash_{\tau} e : \tau}$
$\Gamma \vdash (\text{fut}_f) \text{ ok} \quad \Gamma \vdash (\text{fut}_f v) \text{ ok} \quad \Gamma \vdash (\text{task}_f e) \text{ ok}$
$\frac{\text{(CHAIN)}}{f : \text{Fut } \tau \in \Gamma \quad g : \text{Fut } \tau' \in \Gamma \quad \Gamma \vdash_{\tau} e : \tau' \rightarrow \tau} \quad \frac{\text{(CONFIG)}}{\Gamma \vdash \text{config}_1 \text{ ok} \quad \Gamma \vdash \text{config}_2 \text{ ok} \quad \text{defs}(\text{config}_1) \cap \text{defs}(\text{config}_2) = \emptyset \quad \text{writers}(\text{config}_1) \cap \text{writers}(\text{config}_2) = \emptyset}$
$\Gamma \vdash (\text{chain}_f g e) \text{ ok} \quad \Gamma \vdash \text{config}_1 \text{ config}_2 \text{ ok}$

---

Fig. 4. Configuration typing

**Definition 1.** *The function  $\text{defs}(\text{config})$  extracts the set of futures present in a configuration  $\text{config}$ .*

$$\begin{aligned} \text{defs}((\text{fut}_f)) &= \text{defs}((\text{fut}_f v)) = \{f\} \\ \text{defs}((\text{config}_1 \text{ config}_2)) &= \text{defs}(\text{config}_1) \cup \text{defs}(\text{config}_2) \\ \text{defs}(-) &= \emptyset \end{aligned}$$

**Definition 2.** *The function  $\text{writers}(\text{config})$  extracts the set of writers to futures in configuration  $\text{config}$ .*

$$\begin{aligned} \text{writers}((\text{chain}_f g e)) &= \text{writers}((\text{task}_f e)) = \{f\} \\ \text{writers}(\text{config}_1 \text{ config}_2) &= \text{writers}(\text{config}_1) \cup \text{writers}(\text{config}_2) \\ \text{writers}(-) &= \emptyset \end{aligned}$$

Rules FUT and F-FUT define well-formed future configurations. Rules TASK and CHAIN define well-formed task and future chaining configurations and set the expected task types. Rule CONFIG defines how to build larger configurations from smaller ones. Each future may be defined at most once and there is at most one writer to each future.

The rules for well-formed configurations apply to partial configurations. Complete configurations can be typed by adding extra conditions to ensure that all futures in  $\Gamma$  have a future configuration, there is a one-to-one correspondence between tasks/chains and unfulfilled futures, and dependencies between tasks are acyclic. These definitions have been omitted and are similar to those found in our earlier work [9].

**Formal Properties.** The proof of soundness of the type system follows standard techniques [8]. The proof of progress requires that there is no deadlock, which follows as there is no cyclic dependency between tasks [9].

**Lemma 1 (Type preservation).** *If  $\Gamma \vdash \text{config} \text{ ok}$  and  $\text{config} \rightarrow \text{config}'$ , then there exists a  $\Gamma'$  such that  $\Gamma' \supset \Gamma$  and  $\Gamma' \vdash \text{config}' \text{ ok}$ .*

*Proof.* By induction on the derivation of  $config \rightarrow config'$ .  $\square$

**Definition 3 (Terminal Configuration).** *A complete configuration  $config$  is terminal iff every element of the configuration has the shape:  $(fut_f v)$ .*

**Lemma 2 (Progress).** *For a complete configuration  $config$ , if  $\Gamma \vdash config\ ok$ , then  $config$  is a terminal configuration or there exists a  $config'$  such that  $config \rightarrow config'$ .*

*Proof.* By induction on a derivation of  $config \rightarrow config'$ , relying on the invariance of the acyclicity of task dependencies.  $\square$

### 3 A Promising Implementation Calculus

The implementation of *forward* in the Encore programming language is via compilation into C, linking with Pony's actor-based run-time [10]. At this level, Encore's futures are treated like promises in that they are passed around to the place where the result of a method call is known in order to be fulfilled. To model this implementation approach, we introduce a low-level target calculus based on tasks and promises. This section presents the formalised target calculus, and the next section presents the compilation strategy from the source to the target language.

The syntax of the target language is as follows:

$$\begin{aligned} e ::= & v \mid ee \mid \mathbf{Task}(e, e) \mid \mathbf{stop} \mid e;e \mid \mathbf{Prom} \mid \mathbf{fulfil}(e, e) \mid \mathbf{get}\ e \\ & \mid \mathbf{Chain}(e, e, e) \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \\ v ::= & c \mid f \mid x \mid \lambda x.e \mid () \end{aligned}$$

Expressions consist of values, function application ( $e\ e$ ), sequential composition of expressions ( $e;e$ ), the spawning and stopping of tasks ( $\mathbf{Task}(e, e)$  and  $\mathbf{stop}$ ), the creation, fulfilment, reading, and chaining of promises ( $\mathbf{Prom}$ ,  $\mathbf{fulfil}(e, e)$ ,  $\mathbf{get}\ e$ , and  $\mathbf{Chain}(e, e, e)$ ) and the standard *if-then-else* expression. Values are constants, futures, variables, abstractions and unit  $()$ . The main differences with the source language are that tasks have to be explicitly stopped, which captures non-local exit, and promises must be explicitly created and fulfilled.

#### 3.1 Operational Semantics

The semantics of the target calculus is analogous to the source calculus. The evaluation contexts are:

$$\begin{aligned} E ::= & \bullet \mid Ee \mid vE \mid E;e \mid \mathbf{get}\ E \mid \mathbf{fulfil}(E, e) \mid \mathbf{fulfil}(v, E) \\ & \mid \mathbf{Task}(E, e) \mid \mathbf{Chain}(e, E, e) \mid \mathbf{Chain}(E, v, e) \mid \mathbf{Chain}(v, v, E) \\ & \mid \mathbf{if}\ E\ \mathbf{then}\ e\ \mathbf{else}\ e \end{aligned}$$

$$\begin{array}{c}
\text{(RI-IF-TRUE)} \qquad \qquad \qquad \text{(RI-ERROR)} \\
(\mathbf{task} E[\mathbf{if} \mathit{true} \mathbf{then} e \mathbf{else} e']) \rightarrow (\mathbf{task} E[e]) \quad (\mathit{prm}_f v) (\mathbf{task} E[\mathbf{fulfil}(f, v')]) \rightarrow \mathbf{ERROR} \\
\\
\text{(RI-IF-FALSE)} \qquad \qquad \qquad \text{(RI-PROMISE)} \\
(\mathbf{task} E[\mathbf{if} \mathit{false} \mathbf{then} e \mathbf{else} e']) \rightarrow (\mathbf{task} E[e']) \quad \frac{\mathit{fresh} f}{(\mathbf{task} E[\mathbf{Prom}]) \rightarrow (\mathit{prm}_f) (\mathbf{task} E[f])} \\
\\
\text{(RI-STATEMENT)} \qquad \qquad \qquad \text{(RI-CHAIN)} \\
(\mathbf{task} E[v; e]) \rightarrow (\mathbf{task} E[e]) \quad (\mathbf{task} E[\mathbf{Chain}(f, g, (\lambda x.e))]) \rightarrow (\mathbf{chain} g e[f/x]) (\mathbf{task} E[f]) \\
\\
\text{(RI-}\beta\text{)} \qquad \qquad \qquad \text{(RI-FULFIL)} \\
(\mathbf{task} E[(\lambda x.e) v]) \rightarrow (\mathbf{task} E[e[v/x]]) \quad (\mathit{prm}_f) (\mathbf{task} E[\mathbf{fulfil}(f, v)]) \rightarrow (\mathit{prm}_f v) (\mathbf{task} E[()]) \\
\\
\text{(RI-STOP)} \qquad \qquad \qquad \text{(RI-TASK)} \\
(\mathbf{task} E[\mathbf{stop}]) \rightarrow \epsilon \quad (\mathbf{task} E[\mathbf{Task}(f, (\lambda x.e))]) \rightarrow (\mathbf{task} E[f]) (\mathbf{task} e[f/x]) \\
\\
\text{(RI-CONFIG-CHAIN)} \qquad \qquad \qquad \text{(RI-GET)} \\
(\mathbf{chain} g e) (\mathit{prm}_g v) \rightarrow (\mathbf{task} (e v)) (\mathit{prm}_g v) \quad (\mathbf{task} E[\mathbf{get} h]) (\mathit{prm}_h v) \rightarrow (\mathbf{task} E[v]) (\mathit{prm}_h v)
\end{array}$$


---

Fig. 5. Target reduction rules

Configurations are multisets of promises, tasks, and chains:

$$\mathit{config} ::= \epsilon \mid (\mathit{prm}_f) \mid (\mathit{prm}_f v) \mid (\mathbf{task} e) \mid (\mathbf{chain} f e) \mid \mathit{config} \mathit{config}$$

The empty configuration is represented by  $\epsilon$ , an unfulfilled promise is written as  $(\mathit{prm}_f)$  and a fulfilled promise holding value  $v$  is written as  $(\mathit{prm}_f v)$ .

Tasks and chains work in the same way as in the source language, except that they work now on promises (Fig. 5). Promises are handled much more explicitly than futures are, and need to be passed around like regular values. The creation of a task needs a promise and a function to run; the spawned task runs the function, has access to the passed promise and leaves the promise reference in the spawning task (RI-TASK). Stopping a task just finishes the task (RI-STOP). The construct **Prom** creates an empty promise (RI-PROMISE). Fulfilling a promise results in the value being stored if the promise was empty (RI-FULFIL), or an error otherwise (RI-ERROR). Promises are chained in a similar fashion to futures: the construct **Chain**( $f, g, e$ ) immediately passes the promise  $f$  to expression  $e$  — the intention being that  $f$  will hold the eventual result; the chain then waits on promise  $g$ , and passes the value it receives into expression  $(e f)$  (RI-CHAIN and RI-CONFIG-CHAIN). The target language borrows the configuration evaluation rules from the source language (Fig. 2).

**Example.** For illustration purposes we translate the example from the high-level language,  $(\mathit{fut}_f) (\mathit{task}_f E[\mathbf{forward}(\mathbf{async} e)])$  shown in Sect. 2, and show the reduction steps of the low-level language:

$$\begin{aligned}
& (prm_f) (\text{task } E[\text{Chain}(f, \text{Task}(\text{Prom}, (\lambda d'. \text{fulfil}(d', e); \text{stop})), \lambda d'. \lambda x. \text{fulfil}(d', x); \text{stop}); \text{stop}]) \\
& \longrightarrow (prm_f) (prm_g) (\text{task } E[\text{Chain}(f, \text{Task}(g, (\lambda d'. \text{fulfil}(d', e); \text{stop})), \lambda d'. \lambda x. \text{fulfil}(d', x); \text{stop}); \text{stop}]) \\
& \longrightarrow (prm_f) (prm_g) (\text{task } E[\text{Chain}(f, g, \lambda d'. \lambda x. \text{fulfil}(d', x); \text{stop}); \text{stop}]) (\text{task } \text{fulfil}(g, e); \text{stop}) \\
& \longrightarrow (prm_f) (prm_g) (\text{task } E[f; \text{stop}]) (\text{chain } g (\lambda x. \text{fulfil}(f, x); \text{stop})) (\text{task } \text{fulfil}(g, e); \text{stop}) \\
& \longrightarrow (prm_f) (prm_g) (\text{task } E[\text{stop}]) (\text{chain } g (\lambda x. \text{fulfil}(f, x); \text{stop})) (\text{task } \text{fulfil}(g, e); \text{stop}) \\
& \longrightarrow (prm_f) (prm_g) (\text{chain } g (\lambda x. \text{fulfil}(f, x); \text{stop})) (\text{task } \text{fulfil}(g, e); \text{stop}) \\
& \longrightarrow^* (prm_f) (prm_g) (\text{chain } g (\lambda x. \text{fulfil}(f, x); \text{stop})) (\text{task } \text{fulfil}(g, v); \text{stop}) \\
& \longrightarrow (prm_f) (prm_g) v (\text{chain } g (\lambda x. \text{fulfil}(f, x); \text{stop})) (\text{task } (); \text{stop}) \\
& \longrightarrow (prm_f) (prm_g) v (\text{chain } g (\lambda x. \text{fulfil}(f, x); \text{stop})) (\text{task } \text{stop}) \\
& \longrightarrow (prm_f) (prm_g) v (\text{chain } g (\lambda x. \text{fulfil}(f, x); \text{stop})) \\
& \longrightarrow (prm_f) (prm_g) v (\text{task } (\lambda x. \text{fulfil}(f, x); \text{stop}) v) \\
& \longrightarrow (prm_f) (prm_g) v (\text{task } \text{fulfil}(f, v); \text{stop}) \\
& \longrightarrow (prm_f) v (prm_g) v (\text{task } (); \text{stop}) \\
& \longrightarrow (prm_f) v (prm_g) v (\text{task } \text{stop}) \\
& \longrightarrow (prm_f) v (prm_g) v
\end{aligned}$$

We show how the compilation strategy proceeds in Sect. 4.

### 3.2 Static Semantics

The type system has basic types,  $K$ , and promise types defined below:

$$\tau ::= K \mid \text{Prom } \tau$$

The type rules define the judgment  $\Gamma \vdash e : \tau$  which states that, in the environment  $\Gamma$ , which records the types of promises and free variables, expression  $e$  has type  $\tau$ . The rules for constants, promises, and variables, *if-then-else*, abstraction and function application are analogous to the source calculus, except no expected task type is recorded. The unit value has type **unit** (TI-UNIT); the **stop** expression finishes a task and has any type (TI-STOP). The creation of a promise has type  $\text{Prom } \tau$  (TI-PROMISE-NEW); the fulfilment of a promise  $\text{fulfil}(e, e')$  has type **unit** and requires the first parameter to be a promise and the second to be an expression that matches the type of the promise (TI-FULFIL). To spawn a task ( $\text{Task}(e, e)$ ), the first argument of the task must be a promise and the second a function that takes a promise having the same type as the first argument (TI-TASK); promises can be chained on with functions that run if the promise is fulfilled:  $\text{Chain}(e, e', e'')$  has type  $\text{Prom } \tau$  and  $e$  and  $e'$  are promises and  $e''$  is an abstraction that takes arguments of the first and second promise types. Both task and chain constructors return the promise that is passed to them, for convenience in the compilation scheme.

Soundness of the type system is proven using standard techniques.

$\frac{\text{(TI-CONSTANT)}}{c \text{ is a constant of type } \tau} \quad \Gamma \vdash c : \tau$	$\frac{\text{(TI-PROMISE)}}{f : \text{Prom } \tau \in \Gamma} \quad \Gamma \vdash f : \text{Prom } \tau$	$\frac{\text{(TI-VARIABLE)}}{x : \tau \in \Gamma} \quad \Gamma \vdash x : \tau$	$\frac{\text{(TI-UNIT)}}{\Gamma \vdash () : \text{unit}}$
$\frac{\text{(TI-STOP)}}{\Gamma \vdash \text{stop} : \tau}$	$\frac{\text{(TI-PROMISE-NEW)}}{\Gamma \vdash \text{Prom} : \text{Prom } \tau}$	$\frac{\text{(TI-IF)}}{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e' : \tau \quad \Gamma \vdash e'' : \tau} \quad \Gamma \vdash \text{if } e \text{ then } e' \text{ else } e'' : \tau$	
$\frac{\text{(TI-STATEMENT)}}{\Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau} \quad \Gamma \vdash e_1; e_2 : \tau$	$\frac{\text{(TI-ABSTRACTION)}}{\Gamma, x : \tau \vdash e : \tau'} \quad \Gamma \vdash \lambda x. e : \tau \rightarrow \tau'$	$\frac{\text{(TI-APP)}}{\Gamma \vdash e : \tau' \rightarrow \tau \quad \Gamma \vdash e' : \tau'} \quad \Gamma \vdash e e' : \tau$	
$\frac{\text{(TI-FULFIL)}}{\Gamma \vdash e : \text{Prom } \tau \quad \Gamma \vdash e' : \tau} \quad \Gamma \vdash \text{fulfil}(e, e') : \text{unit}$	$\frac{\text{(TI-TASK)}}{\Gamma \vdash e : \text{Prom } \tau \quad \Gamma \vdash e' : \text{Prom } \tau \rightarrow \tau'} \quad \Gamma \vdash \text{Task}(e, e') : \text{Prom } \tau$		
$\frac{\text{(TI-GET)}}{\Gamma \vdash e : \text{Prom } \tau} \quad \Gamma \vdash \text{get } e : \tau$	$\frac{\text{(TI-CHAIN)}}{\Gamma \vdash e : \text{Prom } \tau \quad \Gamma \vdash e' : \text{Prom } \tau' \quad \Gamma \vdash e'' : \text{Prom } \tau \rightarrow \tau' \rightarrow \tau''} \quad \Gamma \vdash \text{Chain}(e, e', e'') : \text{Prom } \tau$		

---

$\frac{\text{(PROM)}}{f \in \text{dom}(\Gamma)} \quad \Gamma \vdash (\text{prm}_f) \text{ ok}$	$\frac{\text{(F-PROM)}}{f : \text{Prom } \tau \in \Gamma} \quad \Gamma \vdash (\text{prm}_f v) \text{ ok}$	$\frac{\text{(CHAIN-TARGET)}}{\Gamma \vdash f : \text{Prom } \tau \quad \Gamma \vdash e : \tau \rightarrow \tau''} \quad \Gamma \vdash (\text{chain } f e) \text{ ok}$
$\frac{\text{(TASK-TARGET)}}{\Gamma \vdash e : \tau} \quad \Gamma \vdash (\text{task } e) \text{ ok}$		$\frac{\text{(CONFIG-TARGET)}}{\Gamma \vdash \text{config}_1 \text{ ok} \quad \Gamma \vdash \text{config}_2 \text{ ok}} \quad \Gamma \vdash \text{config}_1 \text{ config}_2 \text{ ok}$

**Fig. 6.** Typing rules for expressions and configurations in the target language

## 4 Compilation: From Futures and Forward to Promises

This section presents the compilation function from the source to the target language and outlines a proof that it preserves semantics. The compilation strategy is defined inductively (Fig. 7); the compilation of expressions, denoted  $\mathcal{C}[[e]]_{\text{destiny}}$ , takes an expression  $e$  and a meta-variable **destiny** which holds the promise that the current task should fulfil, and produces an expression in the target language.

Futures are translated to promises, and most other expressions are translated homomorphically. The constructs where something interesting happens are **async**, **forward** and future chaining; these constructs adopt a common pattern implemented using a two parameter lambda abstraction: the first parameter, variable **destiny'**, is the promise to be fulfilled and the second parameter is the value that fulfils the promise. The best illustration of how **forward** behaves differently from a regular asynchronous call is the difference in the rules



---

$\mathcal{C}[\![e]\!]_{\text{destiny}}$	Compilation Strategy
$\begin{aligned} \mathcal{C}[\![f]\!]_{\text{destiny}} &= f & \mathcal{C}[\![x]\!]_{\text{destiny}} &= x & \mathcal{C}[\![c]\!]_{\text{destiny}} &= c \\ \mathcal{C}[\![\lambda x. e]\!]_{\text{destiny}} &= \lambda x. \mathcal{C}[\![e]\!]_{\text{destiny}} \\ \mathcal{C}[\![e_1 e_2]\!]_{\text{destiny}} &= \mathcal{C}[\![e_1]\!]_{\text{destiny}} \mathcal{C}[\![e_2]\!]_{\text{destiny}} \\ \mathcal{C}[\![\text{get } e]\!]_{\text{destiny}} &= \text{get } \mathcal{C}[\![e]\!]_{\text{destiny}} \\ \mathcal{C}[\![\text{async } e]\!]_{\text{destiny}} &= \text{Task}(\text{Prom}, (\lambda \text{destiny}'. \text{fulfil}(\text{destiny}', \mathcal{C}[\![e]\!]_{\text{destiny}'}); \text{stop})) \\ \mathcal{C}[\![\text{forward } e]\!]_{\text{destiny}} &= \text{Chain}(\text{destiny}, \mathcal{C}[\![e]\!]_{\text{destiny}}, \lambda \text{destiny}'. \lambda x. \text{fulfil}(\text{destiny}', x); \text{stop}); \text{stop} \\ \mathcal{C}[\![e \overset{x}{\rightsquigarrow} e']]\!]_{\text{destiny}} &= \text{Chain}(\text{Prom}, \mathcal{C}[\![e]\!]_{\text{destiny}}, (\lambda \text{destiny}'. \lambda x. \text{fulfil}(\text{destiny}', \mathcal{C}[\![e']]\!]_{\text{destiny}'}); \text{stop})) \end{aligned}$	
<hr/>	
$\mathcal{C}[\![e]\!]_{\text{destiny}}$	Optimised Compilation Strategy
$\begin{aligned} \mathcal{C}[\![\text{forward}(\text{async}(e))]\!]_{\text{destiny}} &= \\ & \text{Task}(\text{destiny}, (\lambda \text{destiny}'. \text{fulfil}(\text{destiny}', \mathcal{C}[\![e]\!]_{\text{destiny}'}); \text{stop})); \text{stop} \\ \mathcal{C}[\![\text{forward}(e \overset{x}{\rightsquigarrow} e')]\!]_{\text{destiny}} &= \\ & \text{Chain}(\text{destiny}, \mathcal{C}[\![e]\!]_{\text{destiny}}, \lambda \text{destiny}'. \lambda x. \text{fulfil}(\text{destiny}', \mathcal{C}[\![e']]\!]_{\text{destiny}'}); \text{stop}); \text{stop} \end{aligned}$	
<hr/>	
$\mathcal{T}[\![\text{config}]\!]$	Configuration Compilation Strategy
$\begin{aligned} \mathcal{T}[\![\text{fut}_f]\!] &= (\text{prm}_f) & \mathcal{T}[\![\text{task}_f e]\!] &= (\text{task fulfil}(f, \mathcal{C}[\![e]\!]_f); \text{stop}) \\ \mathcal{T}[\![\text{fut}_f v]\!] &= (\text{prm}_f \mathcal{C}[\![v]\!]_f) & \mathcal{T}[\![\text{config config}']]\!] &= \mathcal{T}[\![\text{config}]\!] \mathcal{T}[\![\text{config}']]\!] \\ & & \mathcal{T}[\![\text{chain}_f g e]\!] &= (\text{chain } g (\lambda x. \text{fulfil}(f, \mathcal{C}[\![e]\!]_f x); \text{stop})) \\ & & & \text{where } x \text{ is fresh} \\ & & \mathcal{T}[\![\text{task}_f e]\!] &= (\text{task} (\text{fulfil}(f, \mathcal{C}[\![e]\!]_f); \text{stop})) \end{aligned}$	
<hr/>	
$\mathcal{C}[\![\tau]\!]$	Type translation
$\begin{aligned} \mathcal{C}[\![ok]\!] &= ok & \mathcal{C}[\![K]\!] &= K \\ \mathcal{C}[\![\text{Fut } \tau]\!] &= \text{Prom } \mathcal{C}[\![\tau]\!] & \mathcal{C}[\![\tau \rightarrow \tau']]\!] &= \mathcal{C}[\![\tau]\!] \rightarrow \mathcal{C}[\![\tau']]\!] \end{aligned}$	
<hr/>	
$\mathcal{T}[\![\Gamma \vdash f : \tau]\!]$	Environment Translation
$\begin{aligned} \mathcal{T}[\![\Gamma \vdash_\rho \text{config}]\!] &= \mathcal{C}[\![\Gamma]\!] \vdash \mathcal{T}[\![\text{config}]\!] & \mathcal{C}[\![\emptyset]\!] &= \epsilon \\ \mathcal{C}[\![\Gamma, f : \text{Fut } \tau]\!] &= \mathcal{C}[\![\Gamma]\!], \mathcal{C}[\![f : \text{Fut } \tau]\!] & \mathcal{C}[\![x : \tau]\!] &= x : \mathcal{C}[\![\tau]\!] \\ \mathcal{C}[\![\Gamma, x : \tau]\!] &= \mathcal{C}[\![\Gamma]\!], \mathcal{C}[\![x : \tau]\!] & \mathcal{C}[\![f : \text{Fut } \tau]\!] &= f : \mathcal{C}[\![\text{Fut } \tau]\!] \end{aligned}$	

---

**Fig. 7.** Compilation strategy of terms, configurations, types and typing rules

for `async`  $e$  and the optimised rule for `forward` (`async`  $e$ ). The translation of `async`  $e$  creates a new promise to store  $e$ 's result value, whereas the translation of `forward` (`async`  $e$ ) reuses the promise from the context, namely the one passed in via the `destiny` variable.

The compilation of configurations, denoted  $\mathcal{T}[\![\text{config}]\!]$ , translates configurations from the source language to the target language. For example, the compilation of the source configuration (`taskf forward` (`async`  $e$ )) compiles into:

$$\begin{aligned} \mathcal{T}[(fut_f)(task_f \text{ forward } (async e))] &= \\ \mathcal{T}[(fut_f)]\mathcal{T}[(task_f \text{ forward } (async e))] &= \\ (prm_f)(\text{task fulfil}(f, \mathcal{C}[\text{forward } (async e)]_f)) & \end{aligned}$$

The optimised compilation of  $\mathcal{C}[\text{forward } (async e)]_f$  is:

$$(prm_f) (\text{task } E[\text{Task}(f, (\lambda d'. \text{fulfil}(d', \mathcal{C}[e]_{d'}); \text{stop}); \text{stop}]); \text{stop}]$$

For comparison, the base compilation gives:

$$(prm_f) (\text{task } E[\text{Chain}(f, \text{Task}(\text{Prom}, (\lambda d'. \text{fulfil}(d', \mathcal{C}[e]_{d'}); \text{stop})), \lambda d'. \lambda x. \text{fulfil}(d', x); \text{stop}); \text{stop}])$$

Types and typing rules are compiled inductively (Fig. 7). The following lemmas guarantee that the compilation strategy does not produce broken target code and state the correctness of the translation.

#### 4.1 Correctness

The correctness of the translation is proven in a number of steps.

The first step involves converting the reduction rules to a labelled transition system where communication via futures is made explicit. This involves splitting several rules involving multiple primitive configurations on the left-hand side to involve single configurations, and labelling the values going into and out of futures. For example,  $(task_f v) (fut_f) \rightarrow (fut_f v)$  is replaced by the two rules:

$$(task_f v) \xrightarrow{\overline{f \downarrow v}} \epsilon \quad (fut_f) \xrightarrow{f \downarrow v} (fut_f v)$$

The other rules introduced are:

$$\begin{aligned} (fut_f v) \xrightarrow{f \uparrow v} (fut_f v) \quad (task_f E[\text{get } h]) \xrightarrow{\overline{h \uparrow v}} (task_f E[v]) \\ (chain_g f e) \xrightarrow{\overline{f \uparrow v}} (task_g e[v/x]) \end{aligned}$$

Label  $f \downarrow v$  captures a value being written to a future, and label  $f \uparrow v$  captures a value being read from a future, both from the future's perspective. Labels  $\overline{f \downarrow v}$  and  $\overline{f \uparrow v}$  are the duals from the perspective of the remainder of the configuration. The remainder of the rules are labelled with  $\tau$  to indicate that no observable behaviour occurs. The same pattern is applied to the target language.

It is important to note that the values in the labels of the source language are the compiled values, while the values in the labels of the target language remain the same.<sup>4</sup> This is needed so that labelled values such as lambda abstraction match during the bisimulation game.

<sup>4</sup> We have omitted the notation from the translation to keep it simple to read.

The composition rules are adapted to propagate or match labels in the standard way. For instance, the rule for matching labels in parallel configurations is:

$$\frac{\text{config} \xrightarrow{l} \text{config}'' \quad \text{config}' \xrightarrow{\bar{l}} \text{config}'''}{\text{config config}' \xrightarrow{\tau} \text{config}'' \text{config}'''}$$

The following theorems capture correctness of the translation.

**Theorem 1.** *If  $\Gamma \vdash \text{config ok}$ , then  $\mathcal{C}[\Gamma] \vdash \mathcal{T}[\text{config}] \text{ok}$ .*

**Theorem 2.** *If  $\Gamma \vdash \text{config ok}$ , then  $\text{config} \sim \mathcal{T}[\text{config}]$ .*

The first theorem states that translating well-typed configurations results in well-typed configurations. The second theorem states that any well-typed configuration in the source language is bisimilar to its translation. The precise notion of bisimilarity used is bisimilarity up-to expansion [11]. This notion of bisimilarity compresses the administrative, unobservable transitions introduced by the translation.

The proof involves taking each derivation rule in the adapted semantics for the source calculus (described above) and showing that each source configuration is bisimilar to its translation. This is straightforward for the base cases, because tasks are deterministic in both source and target languages, and at most two unobservable transitions are introduced by the translation. To handle the parallel composition of configurations, bisimulation is shown to be compositional, meaning that if  $\text{config} \sim \mathcal{T}[\text{config}]$  and  $\text{config}' \sim \mathcal{T}[\text{config}']$ , then  $\text{config config}' \sim \mathcal{T}[\text{config config}']$ ; now by definition  $\mathcal{T}[\text{config config}'] = \mathcal{T}[\text{config}] \mathcal{T}[\text{config}']$ , hence  $\text{config config}' \sim \mathcal{T}[\text{config}] \mathcal{T}[\text{config}']$ .

## 5 Experiments

We benchmarked the implementation of `forward` by comparing it against the blocking pattern `get-and-return` and an implementation that uses the `await-and-get` (both described in Sect. 1). The micro-benchmark used is a variant of the broker pattern with 4 workers, compiled with aggressive optimisations (`-O3`). We report the average time (wall clock) and memory consumption of 5 runs of this micro-benchmark under different workloads (Fig. 8). The processing of each message sent involves complex nested loops with quadratic complexity (in the Workload value) written in such a way to avoid the compiler optimising them away — the higher the workload, the higher the probability that the `Broker` actor blocks or awaits in the non-`forward` implementations.

The performance results (Fig. 8) show that the `forward` version is always faster than the `get-and-return` and `await-and-get` version. In the first case, this is expected as blocking prevents the `Broker` actor from processing messages, while the `forward` version does not block. In the second case, we also expected the `forward` version to be faster than the `await-and-get`: this is due to the overhead of the context switching operation performed on each `await` statement.

Performance (in seconds)			
Workload	Get	Await+Get	Forward
100	0.03	0.03	0.00
500	0.47	0.25	0.02
1000	1.85	0.94	0.06
3000	16.55	8.29	0.39
5000	45.77	23.01	1.03
7500	103.43	51.62	2.26
10000	183.04	91.86	4.02

Memory consumption (in kilobytes)			
Workload	Get	Await+Get	Forward
100	12697	49446	7334
500	12292	49676	6608
1000	12451	49927	6832
3000	12222	49070	7793
5000	12427	48584	7269
7500	12337	48016	7853
10000	12484	48316	8475

**Fig. 8.** Elapsed time (left) and memory consumed (right) by the Broker microbenchmark (the lower the better).

The `forward` version consumes the least amount of memory, while the `await-and-get` version consumes the most (Fig. 8). This is expected: `forward` creates one fewer future per message sent than the other two versions; the `await-and-get` version has around 5 times more overhead than the `forward` implementation, as it needs to save the context (stack) whenever a future cannot immediately be fulfilled.

**Threats to Validity.** The experiments use a microbenchmark, which provides useful information but is not as comprehensive as a case study would be.

## 6 Related Work

Baker discovered futures in 1977 [12]; later Liskov introduced promises to Argus [4]. Around the same time, Halstead introduced implicit futures in Multilisp [13]. Implicit futures do not appear as a first-class construct in the programming language at either the term or type level, as they do in our work.

The `forward` construct was introduced in earlier work [7], in the formalisation of an extension to the active object-based language Creol [14]. The main differences with our work are: our core calculus is much smaller, based on tasks rather than active objects; our calculus includes closures, which complicate the type system, and future chaining; we defined a compilation strategy for `forward`, and benchmark its implementation.

Caromel *et al.* [15] formalise an active object language that transparently handles futures, prove determinism of the language using concepts similar to weak bisimulation, and provide an implementation [16]. In contrast, our work uses a task-based formalism built on top of the lambda calculus and uses futures explicitly. It is not clear whether `forward` can be used in conjunction with transparent futures.

Proving semantics preservation of whole programs is not a new idea [17–23]. We highlight the work from Lochbihler, who added a new phase to the verified, machine-checked Jinja compiler [19] that proves that the translation from multi-threaded Java programs to Java bytecode is semantics preserving, using a delay bisimulation. In contrast, our work uses an on-paper proof using weak

bisimilarity up-to expansion, proving that the compilation strategy preserves the semantics of the high-level language.

Abrahám et al. [5] present an extension of the Creol language with promises. The type system uses linear types to track the use of the write capability (fulfilment) of promises to ensure that they are fulfilled precisely once. In contrast to the present work, their type system is significantly more complex, and no **forward** operation is present. Curiously, Encore supports linear types, though lacks promises and hence does not use linear types to keep promises under control.

Niehren et al. [6] present a lambda calculus extended with futures (which are really promises). Their calculus explores the expressiveness of programming with promises, by using them to express channels, semaphores, and ports. They also present a linear type system that ensures that promises are assigned only once.

## 7 Conclusion

One key difference between futures, futures with forward and promises is that the responsibility to fulfil a future cannot be delegated. The **forward** construct allows such delegation, although only of the implicit future receiving the result of some method call, while promises allow arbitrary delegation of responsibility. This paper presented a formal calculus capturing the **forward** construct, which retains the static fulfilment guarantees of futures. A translation of the source calculus into a target calculus based on promises was provided and proven to be semantics preserving. This translation models how **forward** is implemented in the Encore compiler. Microbenchmarks demonstrated that **forward** improves performance in terms of speed and memory overhead compared to two alternative implementations in the Encore language.

**Acknowledgement.** We are grateful to Joachim Parrow and Johannes Borgström for their comments regarding the bisimulation relation. We also thank the anonymous referees for their useful comments. The underlying research was funded by the Swedish VR project: SCADA.

## References

1. De Boer, F., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. *ACM Comput. Surv.* **50**(5), 76:1–76:39 (2017)
2. Brandauer, S., Castegren, E., Clarke, D., Fernandez-Reyes, K., Johnsen, E.B., Pun, K.I., Tarifa, S.L.T., Wrigstad, T., Yang, A.M.: Parallel objects for multicores: a glimpse at the parallel language ENCORE. In: Bernardo, M., Johnsen, E.B. (eds.) *SFM 2015*. LNCS, vol. 9104, pp. 1–56. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-18941-3\\_1](https://doi.org/10.1007/978-3-319-18941-3_1)
3. 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). [https://doi.org/10.1007/978-3-642-25271-6\\_8](https://doi.org/10.1007/978-3-642-25271-6_8)

4. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Wexelblat, R.L. (eds.) Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, 22–24 June 1988, pp. 260–267. ACM (1988)
5. Ábrahám, E., Grabe, I., Grüner, A., Steffen, M.: Behavioral interface description of an object-oriented language with futures and promises. *J. Log. Algebr. Program.* **78**(7), 491–518 (2009)
6. Niehren, J., Schwinghammer, J., Smolka, G.: A concurrent lambda calculus with futures. *Theor. Comput. Sci.* **364**(3), 338–356 (2006)
7. Clarke, D., Johnsen, E.B., Owe, O.: Concurrent objects à la carte. In: Dams, D., Hannemann, U., Steffen, M. (eds.) *Concurrency, Compositionality, and Correctness. LNCS*, vol. 5930, pp. 185–206. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-11512-7\\_12](https://doi.org/10.1007/978-3-642-11512-7_12)
8. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. *Inf. Comput.* **115**(1), 38–94 (1994)
9. Fernandez-Reyes, K., Clarke, D., McCain, D.S.: ParT: an asynchronous parallel abstraction for speculative pipeline computations. In: Lluch Lafuente, A., Proença, J. (eds.) *COORDINATION 2016. LNCS*, vol. 9686, pp. 101–120. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-39519-7\\_7](https://doi.org/10.1007/978-3-319-39519-7_7)
10. Clebsch, S., Drossopoulou, S.: Fully concurrent garbage collection of actors on many-core machines. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, Part of SPLASH 2013, Indianapolis, IN, USA, 26–31 October 2013*, pp. 553–570. ACM (2013)
11. Pous, D., Sangiorgi, D.: Enhancements of the bisimulation proof method. In: Sangiorgi, D., Rutten, J. (eds.) *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, Cambridge (2012)
12. Baker, H.G., Hewitt, C.: The incremental garbage collection of processes. *SIGART Newsl.* **64**, 55–59 (1977)
13. Halstead Jr., R.H.: Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* **7**(4), 501–538 (1985)
14. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: a type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.* **365**(1–2), 23–66 (2006)
15. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous sequential processes. *Inf. Comput.* **207**(4), 459–495 (2009)
16. Caromel, D., Delbe, C., Di Costanzo, A., Leyton, M.: ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Comput. Methods Sci. Technol.* **12**(1), 69–77 (2006)
17. Leroy, X.: A formally verified compiler back-end. *J. Autom. Reason.* **43**(4), 363–446 (2009)
18. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Gregory Morrisett, J., Peyton Jones, S.L. (eds.) *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, 11–13 January 2006*, pp. 42–54. ACM (2006)
19. Lochbihler, A.: Verifying a compiler for Java threads. In: Gordon, A.D. (ed.) *ESOP 2010. LNCS*, vol. 6012, pp. 427–447. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-11957-6\\_23](https://doi.org/10.1007/978-3-642-11957-6_23)

20. Chlipala, A.: A certified type-preserving compiler from lambda calculus to assembly language. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, 10–13 June 2007, pp. 54–65. ACM (2007)
21. Wand, M.: Compiler correctness for parallel languages. In: Williams, J. (ed.) Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA 1995, La Jolla, California, USA, 25–28 June 1995, pp. 120–134. ACM (1995)
22. Liu, X., Walker, D.: Confluence of processes and systems of objects. In: Mosses, P.D., Nielsen, M., Schwartzbach, M.I. (eds.) CAAP 1995. LNCS, vol. 915, pp. 217–231. Springer, Heidelberg (1995). [https://doi.org/10.1007/3-540-59293-8\\_197](https://doi.org/10.1007/3-540-59293-8_197)
23. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.* **28**(4), 619–695 (2006)



# Aggregation Policies for Tuple Spaces

Linus Kaminskas and Alberto Lluch Lafuente<sup>(✉)</sup>

DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark  
linaskmnsks@gmail.com, albl@dtu.dk

**Abstract.** Security policies are important for protecting digitalized information, control resource access and maintain secure data storage. This work presents the development of a policy language to transparently incorporate aggregate programming and privacy models for distributed data. We use tuple spaces as a convenient abstraction for storage and coordination. The language has been designed to accommodate well-known models such as  $k$ -anonymity and  $(\epsilon, \delta)$ -differential privacy, as well as to provide generic user-defined policies. The formal semantics of the policy language and its enforcement mechanism is presented in a manner that abstracts away from a specific tuple space coordination language. To showcase our approach, an open-source software library has been developed in the Go programming language and applied to a typical coordination pattern used in aggregate programming applications.

**Keywords:** Secure coordination · Policy languages · Privacy models  
Tuple spaces · Aggregate programming

## 1 Introduction

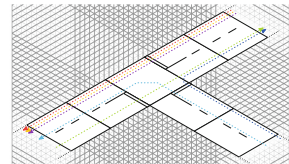
Privacy is an essential part of society. With increasing digitalization the attack surface of IT-based infrastructures and the possibilities for abuse is growing. It is therefore necessary to include privacy models that can scale with the complexity of those infrastructures and their software components, in order to protect information stored and exchanged, while still ensuring information quality and availability. With EU GDPR regulation [19] being implemented in all EU countries, regulation on how data acquisition processes handle and distribute personal information becomes enforced. This affects software development processes and life cycles as security-by-design choices will need to be incorporated. Legacy systems will also be affected by GDPR compliance. With time, these legacy systems will need to be replaced, not only because of technological advancements, but also due to political and social demands for higher quality infrastructure. No matter the perspective, the importance of privacy-preserving data migration, mining and publication will remain relevant as society advances.



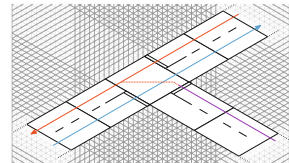
**Aggregation, Privacy and Coordination.** Aggregate programming methods are used for providing privacy guarantees (e.g. by reducing the ability to distinguish individual data items), improving performance (e.g. by reducing storage size and communications) and even as the basis of emergent coordination paradigms (e.g. computational field and aggregate programming based on the field calculus [1, 23] or the SMuC calculus [14]). Basic aggregation functions (e.g. sums, averages, etc.), do not offer enough privacy guarantees (e.g. against statistical attacks) to support the construction of trustworthy coordination systems. The risk is that less users will be willing to share their data. As a consequence, the quality of different infrastructures and services based on data aggregations may degrade. More powerful privacy protection systems are needed to reassure users and foster their participation with useful data. Fortunately, aggregation-based methods can be enhanced by using well-studied privacy models that allow policy makers to trade between privacy and data utility. We investigate in this work how such methods can be easily integrated in a coordination model such as tuple spaces, that in turn can be used as the basis of aggregation-based systems.

**Motivational Examples.** One of our main motivations is to address systems where users provide data in order to improve some services that they themselves may use. In such systems it is often the case that: (i) A user decides how much privacy is to be sacrificed when providing data. Data aggregation is performed according to a policy on their device and transmitted to a data collector. (ii) A data collector partitions data by some quality criterion. Aggregation is then performed on each partition and results are stored, while the received data may be discarded. (iii) A process uses the aggregated data, and shares results back to the users in order to provide a service.

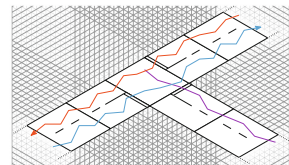
A typical example of such systems are Intelligent Transport System (ITS), which exploit Geographic Information Systems (GIS) data from vehicles to provide better transportation services, e.g. increased green times at intersections, reduction of queue and congestion or exploration of infrastructure quality. As a real world example, bicycle GIS data is exploited by ITS systems to reduce congestion on bicycle paths, while maintaining individuals privacy. Figure 1 shows user positional data in different stages: (a) raw data as collected, (b) data after aggregating multiple trips, and (c) aggregated data with addition of noise to protect privacy. This aggregated data can then be delivered back to the users, in order to support their decision making before more



(a) Raw data



(b) Aggregated data



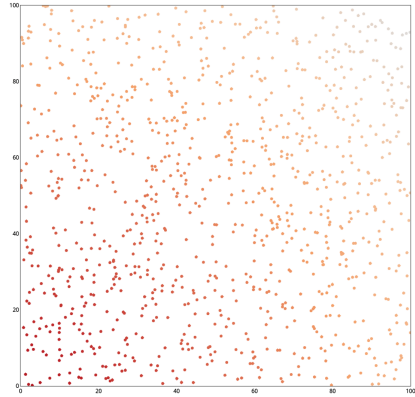
(c) Perturbed data

**Fig. 1.** Different stages of GIS data.

congestion occurs. Depending on the background knowledge and insights in a service, an adversary can partially or fully undo bare aggregation. By using privacy models and controlling aggregate functions, one can remove sensitive fields such as unique identifiers and device names, and add noise to give approximations of aggregation results. This gives a way to trade data accuracy in favor of privacy.

Another typical example are self-organizing systems. Consider, for instance, the archetypal example of the construction of a distance field, identified in [2] as one of the basic self-organization building blocks for aggregate programming. The typical scenario in such systems is as follows. A number of devices and points-of-interest (PoI) are spread over a geographical area. The main aim of each device is to estimate the distance to the closest PoI. The resulting distributed mapping of devices into distances and possibly the next device on the shortest path, forms a computational field. This provides a basic layer for aggregate programming applications and coordinated systems, as

in e.g. providing directions to PoIs. Figure 2 shows an example with the result of 1000 devices in an area with a unique PoI located at (0,0), where each device is represented by a dot and whose color intensity is proportional to the computed distance. The computation of the field needs to be done in a decentralized way, since the range of communication of devices needs to be kept localized. The algorithm that the devices use to compute the field is based on data aggregations: a device iteratively queries the neighbouring devices for their information and updates its own information to keep track of the closest distance to a PoI. Initially, the distance  $d_i$  of each device  $i$  is set to the detected distance to the closest PoI, or to  $\infty$  if no device is detected. At each iteration, a device  $i$  updates its computed distance  $d_i$  as follows. It gets from each neighbour  $j$  its distance  $d_j$ , and then updates  $d_i$  to be the minimum between  $d_i$  or  $d_j$  plus the distance from  $i$  to  $j$ . In this algorithm, the key operation performed by the devices is an aggregation of neighbouring data, which may not offer sufficient privacy guarantees. For instance, the exact location of devices or their exact distance to a PoI could be inferred by a malicious agent. A simple case where this could be done is when one device and a PoI are in isolation. A more complex case could be if the devices are allowed to move and change their distances to a PoI gradually. By observing isolated devices and their interactions with neighbours, one could start to infer more about the behaviour of a device group.



**Fig. 2.** A distance gradient field (Color figure online)

```

1 privacy_policy:
2     noisy_average:
3         aquery avg, data
4             altered by result func add_noise
5     ...

```

**Listing 1.1.** An aggregation policy that adds noise to average-based queries

```

1 program:
2     ...
3     x := aquery avg, data
4     ...

```

**Listing 1.2.** An aggregated query subject to the policy in Listing 1.1.

**Challenges.** Engineering of privacy mechanisms and embedding of these directly into large software systems is not a trivial task, and may be error prone. Therefore, it is crucial to separate privacy mechanisms from an application, in such a way that the privacy mechanisms can be altered without having to change the application logic. For example, Listing 1.1 shows a policy that a data hosting service will provide, and Listing 1.1 shows a program willing to use the data. The policy controls the aggregate query (`aquery`) of the program. It only allows to average (`avg`) some `data` and, in addition, it uses a `add_noise` function to the result before an assignment in the program occurs. In this manner, a clear separation of logic can be achieved, and multiple queries of the same kind can be altered by the same policy. Furthermore, it allows policies to be changed at run-time, in order to adapt to changes in regulations or to optimize the implementation of a policy. Separation of concerns provides convenience for both developers and policy makers alike.

**Contribution.** Our goal is to develop a tool-supported policy language providing access control for which well-studied privacy models, aggregate programming constructs, and coordination primitives could be used to provide non-intrusive data access in distributed applications. We wanted to focus on an interactive setting where data is dynamically produced, consumed and queried, instead of the traditional static data warehousing that privacy models implementations tend to address.

Our first contribution is a novel policy language in Sect. 2 to specify aggregation policies for tuple spaces. The choice of tuple spaces has been motivated by the need to abstract away from concrete data storage models and to address data-oriented coordination models. Our approach to the language provides a clean separation between policies that need to be enforced, and application logic that needs to be executed. The presentation abstracts away from any concrete tuple-based coordination language and we focus on aggregated versions of the traditional operations to add, retrieve and remove tuples.

Our second contribution (Sect. 3) is a detailed description of how two well-studied privacy models such as  $k$ -anonymity and  $(\epsilon, \delta)$ -differential privacy can

be expressed in our language. For this purpose, those models (which are usually presented in a database setting) have been redefined in the setting of tuple spaces. To the authors knowledge, this is the first time that the definition of those models has been adapted to tuple spaces.

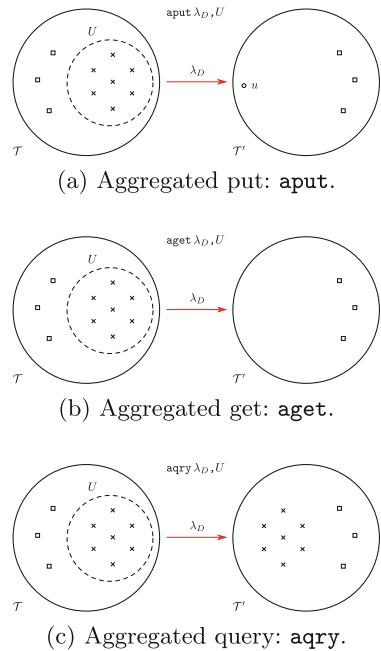
Our third and last contribution (Sect. 4) is an open-source, publicly available implementation of the policy language and its enforcement mechanism in a tuple space library for the Go programming language, illustrated with an archetypal example of a self-organizing pattern used as basic block in aggregate programming approaches [2], namely the above presented computation of a distance gradient.

## 2 A Policy Language for Aggregations

We start the presentation of our policy language by motivating the need of supporting and controlling aggregate programming primitives, and present a set of such primitives. We then move into the description of our policy language, illustrate the language through examples, and conclude the section with formal semantics.

### Aggregate Programming Primitives.

The main computations we focus in this paper are aggregations of multiset of data items. As we have discussed in Sect. 1, such computations are central to aggregate programming approaches. The main motivation is to control how such aggregations are performed: a data provider could want, for instance, to provide access to the average of a data set, but not to the data set or any of its derived forms. Traditional tuple spaces (e.g. those following the Linda model) do not support aggregations as first-class primitives: a user would need to extract all the data first and perform the aggregation during or after the extraction. Such a solution does not allow to control how aggregations are performed, and the user is in any case given access to an entire set of data items that needs to be protected. However, in databases, aggregate operators can be used in queries, providing thus a first-class primitive to perform aggregated queries, more amenable for access control. A similar situation can be found in aggregate programming languages that provide functions to aggregate data from neighbouring components: the field calculus offers a `nbr`



**Fig. 3.** Aggregation primitives.

primitive to retrieve information about neighbouring devices and aggregation is to be performed on top of that, whereas the SMUC calculus is based on atomic aggregation primitives.

We adapt such ideas to tackle the necessity of controlling aggregations in tuple spaces by proposing variants of the classical single-data operations `put/out`, `get/in` and `qry/read`. In particular, we extend them with an additional argument: an aggregation function that is intended to be applied to the multiset of all matched tuples. Typical examples of such functions would be averages, sums, minimum, concatenation, counting functions and so forth. While standard tuple space primitives allow to retrieve *some* or *all* tuples matching some template, the primitives we promote would allow to retrieve the aggregated version of all the matched tuples. More in detail, we introduce the following aggregate programming primitives (Fig. 3):

- `aqry` $\lambda_D, U$ : This operation works similarly to an aggregated query in a database and provides an aggregated view of the data. In particular, it returns the result of applying the aggregation function  $\lambda_D$  to all tuples that match the template  $U$ .
- `aget` $\lambda_D, U$ : This operation is like `aqry`, but removes the matched data with template  $U$ .
- `aput` $\lambda_D, U$ : This operation is like `aget`, but the result of the aggregation is introduced in the tuple space. It provides a common pattern used to atomically compress data.

It is worth to remark that such operations allow to replicate many of the common operations on tuple spaces. Indeed, the aggregation function could be, for instance, the multiset union (providing all the matched tuples) or a function that provides just one of the matched tuples (according to some deterministic or random function).

**Syntax of the Language.** The main concepts of the language are knowledge bases in the form of tuple spaces, policies for expressing how operations should be altered, and aggregate programming operators. The language itself can be embedded in any host coordination language, but the primary focus will be in expressing policies. The language is defined in a way that is reminiscent of a concrete syntax for a programming language. Although, the point is not to force a particular syntax, but to have a convenient abstraction for describing the policies themselves. Further, the language and aggregation policies do not force a traditional access control based model by only permitting or denying access to data: policies allow transformations thus giving different views on the same data. This gives a choice to a policy maker to control the accuracy of the information released to a data consumer. Subjects (e.g. users) and contextual (e.g. location) attributes are not part of our language, in order to keep the presentation focused on the key aspects of aggregate programming. Yet, the language could be easily extended to include these attributes.

**Table 1.** Syntax for policies and aggregate programming operators.

$$\begin{aligned}
 \text{TUPLE SPACE : } \mathcal{T} &::= \emptyset \mid V : u \mid V : u ; \mathcal{T} \\
 \text{POLICY LABEL : } v & \\
 \text{POLICY LABELS : } V &::= \emptyset \mid v \mid v, V \\
 \text{TUPLE : } u &::= \varepsilon \mid c \mid u_1, u_2 \\
 \text{TEMPLATE : } U &::= \varepsilon \mid c \mid \Omega \mid U_1, U_2 \\
 \text{COMPOSABLE POLICY : } \Pi &::= \mathbf{0} \mid \pi \mid \pi ; \Pi \\
 \text{AGGREGATION POLICY : } \pi &::= v : H \\
 \text{AGGREGATION RULE : } H &::= \text{none} \mid \text{put } U \text{ altered by } D_a \\
 &\quad \mid A_D \text{ altered by } D_U D_u D_a \\
 \text{ACTION : } A &::= A_S \mid A_D \\
 \text{SIMPLE ACTION : } A_S &::= \text{put } V : u \\
 \text{AGGREGATE ACTION : } A_D &::= \text{aput } \lambda_D, U \mid \text{aget } \lambda_D, U \mid \text{aqry } \lambda_D, U \\
 \text{TEMPLATE TRANSFORMATION : } D_U &::= \text{template func } \lambda_U \\
 \text{TUPLE TRANSFORMATION : } D_u &::= \text{tuple func } \lambda_u \\
 \text{RESULT TRANSFORMATION : } D_a &::= \text{result func } \lambda_a \\
 \text{AGGREGATE OPERATOR : } \lambda_D &::= \text{sum} \mid \text{avg} \mid \text{min} \mid \text{max} \mid \dots \\
 \text{TEMPLATE OPERATOR : } \lambda_U &::= \text{id} \mid \text{pseudo } i \mid \text{collapse } i \mid \dots \\
 \text{TUPLE OPERATOR : } \lambda_u &::= \text{id} \mid \text{collapse } i \mid \text{noise } X \mid \dots
 \end{aligned}$$

The syntax of our aggregation policy language can be found in Table 1. Let  $\Omega$  denote the types which are exposed by the host language and a type be  $\tau \in \Omega$ . For the sake of exposition we consider the simple case  $\{\text{int}, \text{float}, \text{string}\} \subseteq \Omega$ .  $\mathcal{T}$  is a knowledge base represented by a tuple space with a multiset interpretation, where the order of tuples is irrelevant and multiple copies of the identical tuples are allowed. For  $\mathcal{T}$ , the language operator  $;$  denotes the multiset union,  $\setminus$  is the multiset difference and  $\ominus$  is the multiset symmetric difference.  $\mathcal{T}$  contains labelled tuples, i.e. tuples attached a set of labels, with each label identifying a policy. A *tuple* is denoted and generated by  $u$ , and an empty tuple is denoted by  $\varepsilon$ . Tuples may be primed  $u'$  or stared  $u_*$  to distinguish between different types of tuples. The type of a tuple  $u$  is denoted by  $\tau_u = \tau_{u_1} \times \tau_{u_2} \times \dots \times \tau_{u_n}$ . In Sect. 3, individual tuple fields will be needed, and hence we will be more explicit and use  $u = (u_1, \dots, u_i, \dots, u_n)$ , where  $u_i$  denotes the  $i^{\text{th}}$  tuple field with type  $\tau_{u_i}$ . When dealing with a multiset of tuples of type  $\tau_u$  (e.g. a tuple space), the type  $\tau_u^*$  will be used. For a *label set*  $V$ , a *labelled tuple* is denoted by  $V : u$ . Similarly as for a tuple, the empty labelled tuple is denoted by  $\varepsilon$ . A label serves as a unique attribute when performing policy selection based on an *action*  $A$ . A template  $U$  can contain constants  $c$  and types  $\tau \in \Omega$  and is used for pattern matching against a  $u \in \mathcal{T}$ . As with tuples, we shall be explicit with template fields when necessary and use  $U = (U_1, \dots, U_i, \dots, U_n)$ , where  $U_i$  denotes the  $i^{\text{th}}$  template field with type  $\tau_{U_i}$ . There are three main aggregation actions derived from the classical tuple space operations (**put**, **get**, **qry**), namely: **aput**, **aget** and **aqry**. All operate by applying an aggregate operator  $\lambda_D$  on tuples  $u \in \mathcal{T}$  that matches  $U$ . Aggregate functions  $\lambda_D$  have a functional type  $\lambda_D : \tau_u^* \rightarrow \tau_{u'}$  and

are used to aggregate tuples of type  $\tau_u$  into a tuple of type  $\tau_{u'}$ . The *composable policy*  $\Pi$  is a list of policies that contain *aggregation policies*  $\pi$ . An aggregation  $\pi$  is defined by a *policy label*  $v$  and an *aggregation rule*  $H$ , where  $v$  is used as an identifier for  $H$ . An aggregation rule  $H$  describes how an action  $A$  is altered either by a *template transformation*  $D_U$ , a *tuple transformation*  $D_u$ , and a *result transformation*  $D_a$ , or not at all by **none**. A template transformation  $D_U$  is defined by a *template operator*  $\lambda_U : \tau_U \rightarrow \tau_{U'}$ , and can be used for e.g. hiding sensitive attributes or to adapt the template from the public format of tuples to the internal format of tuples. A tuple transformation  $D_u$  is defined by a *tuple operator*  $\lambda_u : \tau_u \rightarrow \tau_{u'}$ . This allows to apply additional functions on a matched tuple  $u$ , and can be used e.g. for doing sanitization, addition of noise or approximating values, before performing the aggregate operation  $\lambda_D$  on the matched tuples. A *result transformation*  $D_a$  is defined by a tuple operator  $\lambda_a : \tau_{u'} \rightarrow \tau_{u''}$ . The arguments of  $\lambda_a$  are the same as for tuple transformations  $\lambda_u$ , except the transformation is applied on an aggregated tuple. This allows for coarser control, say, in case a transformation on all the matched tuples is computationally expensive or if simpler policies are enough.

**Examples and Comparison with a Database.** Observe that  $\lambda_D$  and any of the aggregation actions in  $A$  can provide all of the aggregate functions found in commercial databases, but with the flexibility of exactly defining how this is performed in the host language itself. The motivation for doing this comes from the fact that: (i) there is tendency for database implementations to provide non-standardized functionalities, introducing software fragmentation when swapping technologies, (ii) user-defined aggregate functions are often defined in a different language from the host language. In our approach, by allowing to directly express both the template for the data needed and aggregate functionality in the host language, helps reducing the programming complexity and improves readability, as the intended aggregation is expressed explicitly and in one place. Moreover, the usage of templates allows to specify the view of data at different granularity levels. For instance, in our motivational example on GIS data, one could be interested in expressing:

1. Field granularity where  $U$  contains concrete values only, but access is provided to some fields only. Listing 1.3 shows how to allow access to a specific data source by using  $U = (\text{"devices"}, \text{"gps"}, \text{"accelerometer"}, \text{"gyroscope"})$  as a template of concrete devices. Here, `id` is the identity function, `first` is an aggregation function which returns the first matched tuple, and `nth 2` selects the second field of the tuple.

```

1 accelerometer-data-only:
2   aqry first, "devices", "gps", "accelerometer", "gyroscope"
3   altered by
4   template func id
5   tuple func id
6   result func (fun x -> nth 2 x)

```

**Listing 1.3.** Example of field granularity policy.

2. Tuple granularity where  $U$  contains concrete values and all fields are provided. Listing 1.4 shows how a policy can provide access to a specific trip. In this case, it is specified by a trip type and trip identifier  $U = (\text{"bike-ride"}, 1)$ .

```

1 single-ride:
2   aqry first, "bike-ride", 1 altered by
3     template func id
4     tuple func id
5     result func id

```

**Listing 1.4.** Example of tuple granularity policy.

3. Mixed granularity where  $U$  contains a mix of concrete values and types. Listing 1.5 shows how this could be used to protect user coordinates expressed as a triplet of `float`'s encoding latitude, longitude, elevation while allowing a certain area. In this case, the `copenhagen` area is exposed, and computation of the average elevation with `avg` is permitted.

```

1 alices-trips:
2   aqry avg, "copenhagen", float, float, float altered by
3     template func id
4     tuple func (fun x -> nth 4 x)
5     result func id

```

**Listing 1.5.** Example of mixed granularity policy.

4. Tuple-type granularity where  $U$  contains only types. Listing 1.6 shows how this could be used to count how many points there are in each discretized part of a map, where `area` maps coordinates into areas.

```

1 map-partition:
2   aqry count, float, float altered by
3     template func id
4     tuple func (fun (x y) -> area(x,y))
5     result func id

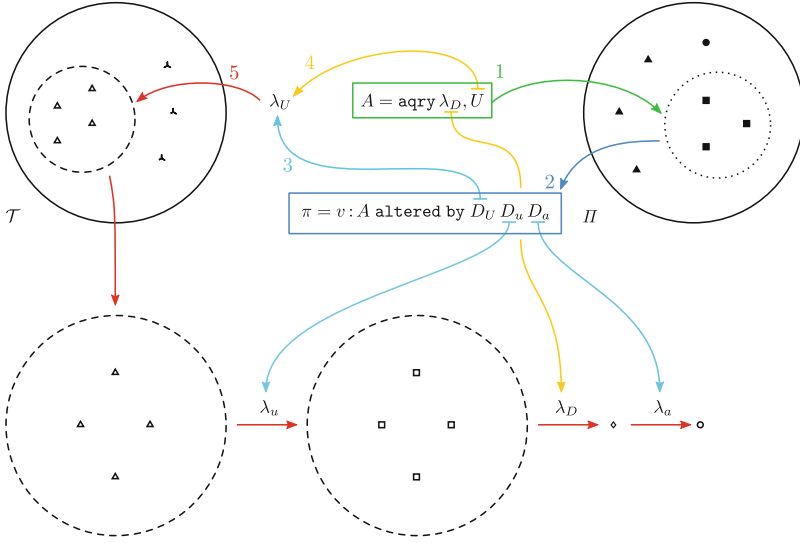
```

**Listing 1.6.** Example of tuple-type granularity policy.

With respect to databases, the aforementioned granularities correspond to: 1. cell level, 2. single row level, 3. multiple row level, and 4. table level. Combined with a user-defined aggregate function  $\lambda_D$  and transformations  $D_U$ ,  $D_u$  and  $D_a$ , one can provide many different views of a tuple space in a concise manner.

**Formal Semantics.** Before presenting the formal semantics, we provide a graphical and intuitive presentation using  $A = \text{aqry} \lambda_D U$  and some  $\mathcal{T}$  and  $\Pi$  as an example shown in Fig. 4. The key idea is: 1. Given some action  $A$ , determine the applicable policy  $\pi$ . There can be multiple matches in  $\Pi$ ; 2. extract the first-matching policy  $\pi$  with some label  $v$ ; 3. extract template, tuple and result operators from transformations  $D_U$ ,  $D_u$ , and  $D_a$  respectively; 4. extract the aggregate operator  $\lambda_D$  and apply  $U' = \lambda_U(U)$ ; 5. based on the tuples  $V:u$





**Fig. 4.** Semantics of an aggregate action  $A$  given an applicable policy  $\pi$ .

from  $\mathcal{T}$  that match  $U'$  and have  $v \in V$ : perform tuple transformation with  $\lambda_u$ , aggregation with  $\lambda_D$ , and result transformation with  $\lambda_a$

The formal operational semantics of our policy enforcement mechanism is described by the set of inference rules in Table 2, whose format is

$$\frac{P_1 \quad \dots \quad P_i \quad \dots \quad P_n}{\mathcal{T}, \Pi \vdash A \rightarrow \mathcal{T}', \Pi \triangleright R}$$

where  $P_1, \dots, P_n$  are premises,  $\mathcal{T}$  is a tuple space subject to  $\Pi$ ,  $A$  is the action subject to control, and the return value (if any) is modelled by  $\triangleright R$ . The return value may then be consumed by the host language. The absence of a return value denotes that no policy was applicable.

The semantics for applying a policy that matches an aggregate action  $\text{aput } \lambda_D, U$ ,  $\text{aget } \lambda_D, U$  and  $\text{aqry } \lambda_D, U$  is respectively defined by rules AGG-PUT-APPLY, AGG-GET-APPLY and AGG-QUERY-APPLY. For performing  $\text{put } V:u$ , PUT-APPLY is used. All three rules apply such transformation and differ only in that  $\text{aget}$  and  $\text{aput}$  modify the tuple space. A visual representation of the semantics of  $\text{aput } \lambda_D, U$ ,  $\text{aget } \lambda_D, U$  and  $\text{aqry } \lambda_D, U$  can be seen in Fig. 4. The premises of the rules include conditions to ensure that the right operation is being captured and a decomposition of how the operation is transformed by the policy. In particular, the set  $\mathcal{T}_1$  represents the actually matched tuples (after transforming the template) and  $\mathcal{T}_2$  is the actual view of the tuple space being considered (after applying the tuple transformations to  $\mathcal{T}_1$ ). It is on  $\mathcal{T}_2$  that the user-defined aggregation  $\lambda_D$  is applied, and then the result transformation  $\lambda_a$  is applied to provide the final result  $u_a$ . Rules named UNMATCHED, PRIORITY-LEFT, PRIORITY-LEFT, and PRIORITY-UNAVAILABLE take care of scanning the

**Table 2.** Semantics for action  $A$  under  $\Pi$  including the semantics format.

PUT-APPLY: $\frac{\Pi = v_\pi : \text{put } U \text{ altered by result func } \lambda_a \quad \text{match}(u, U) \quad v_\pi \in V \quad u_a = \lambda_a(u) \quad T' = T; V : u_a}{T, \Pi \vdash \text{put } V : u \rightarrow T', \Pi \triangleright V : u_a}$	
AGG-PUT-APPLY: $\frac{\Pi = v_\pi : \text{aput } \lambda_D, U' \text{ altered by template func } \lambda_U \text{ tuple func } \lambda_u \text{ result func } \lambda_a \quad \text{match}(U, U') \quad T_1 = \{V : u \in T \mid \text{match}(u, \lambda_U(U)) \wedge v_\pi \in V\} \quad T' = T \setminus T_1 \quad T_2 = \{V : \lambda_u(u) \mid V : u \in T_1\} \quad u_a = \lambda_a(\lambda_D(\{u \mid V : u \in T_2\})) \quad T'' = T'; \{v_\pi\} : u_a}{T, \Pi \vdash \text{aput } \lambda_D, U \rightarrow T'', \Pi \triangleright \{v_\pi\} : u_a}$	
AGG-GET-APPLY: $\frac{\Pi = v_\pi : \text{aget } \lambda_D, U' \text{ altered by template func } \lambda_U \text{ tuple func } \lambda_u \text{ result func } \lambda_a \quad \text{match}(U, U') \quad T_1 = \{V : u \in T \mid \text{match}(u, \lambda_U(U)) \wedge v_\pi \in V\} \quad T' = T \setminus T_1 \quad T_2 = \{V : \lambda_u(u) \mid V : u \in T_1\} \quad u_a = \lambda_a(\lambda_D(\{u \mid V : u \in T_2\}))}{T, \Pi \vdash \text{aget } \lambda_D, U \rightarrow T', \Pi \triangleright \{v_\pi\} : u_a}$	
AGG-QUERY-APPLY: $\frac{\Pi = v_\pi : \text{aqry } \lambda_D, U' \text{ altered by template func } \lambda_U \text{ tuple func } \lambda_u \text{ result func } \lambda_a \quad \text{match}(U, U') \quad T_1 = \{V : u \in T \mid \text{match}(u, \lambda_U(U)) \wedge v_\pi \in V\} \quad T_2 = \{V : \lambda_u(u) \mid V : u \in T_1\} \quad u_a = \lambda_a(\lambda_D(\{u \mid V : u \in T_2\}))}{T, \Pi \vdash \text{aqry } \lambda_D, U \rightarrow T, \Pi \triangleright \{v_\pi\} : u_a}$	
UNMATCHED: $\frac{\Pi = v_\pi : \text{none} \vee (\Pi = v_\pi : A_2 \text{ altered by } D_U D_u D_a \wedge A_1 \neq A_2) \vee \Pi = \mathbf{0}}{T, \Pi \vdash A_1 \rightarrow T, \Pi}$	
PRIORITY-RIGHT: $\frac{T, \Pi_1 \vdash A \rightarrow T, \Pi_1 \quad T, \Pi_2 \vdash A \rightarrow T', \Pi'_2 \triangleright V : u}{T, \Pi_1; \Pi_2 \vdash A \rightarrow T', \Pi_1; \Pi'_2 \triangleright V : u}$	PRIORITY-LEFT: $\frac{T, \Pi_1 \vdash A \rightarrow T', \Pi'_1 \triangleright V : u}{T, \Pi_1; \Pi_2 \vdash A \rightarrow T', \Pi'_1; \Pi_2 \triangleright V : u}$
PRIORITY-UNAVAILABLE: $\frac{T, \Pi_1 \vdash A \rightarrow T, \Pi_1 \quad T, \Pi_2 \vdash A \rightarrow T, \Pi_2}{T, \Pi_1; \Pi_2 \vdash A \rightarrow T, \Pi_1; \Pi_2}$	

policy as list. It is up to the embedding in an actual host language to decide what to do with the results. For example, in our implementation, if the policy enforcement yields no result, the action is simply ignored.

### 3 Privacy Models

The design of our language has been driven by inspecting a variety of privacy models, first and foremost  $k$ -anonymity and  $(\varepsilon, \delta)$ -differential privacy. We show in this section how those models can be adopted in our approach. The original definitions have been adapted from databases to our tuple space setting.

**$k$ -anonymity.** The essential idea of  $k$ -anonymity [15, 20, 22] is to provide anonymity guarantees beyond hiding sensitive fields by ensuring that, when

information on a data set is released, every individual data item is indistinguishable from at least  $k - 1$  other data items. In our motivational examples, for instance, this could be helpful to protect the correlation between devices and their distances from an attacker that can observe the position and number of devices in a zone and can obtain the list of distances within a zone through a query.  $k$ -anonymity is often defined for tables in a database, here it shall be adapted to templates  $U$  instead. We start by defining  $k$ -anonymity as a property of  $\mathcal{T}$ : roughly,  $k$ -anonymity requires that every tuple  $u$  cannot be distinguished from at least  $k - 1$  other tuples. Distinguishability of tuples is captured by an equivalence relation  $=_t$ . Note that  $=_t$  is not necessarily as strict as tuple equality: two tuples  $u$  and  $u'$  may be different but equivalent, in the sense that they can be related exactly by the same, and possibly external, data. In our setting,  $k$ -anonymity is formalized as follows.

**Definition 1 ( $k$ -anonymity).** *Let  $k \in \mathbb{N}^+$ ,  $\mathcal{T}$  be a multiset of tuples, and let  $=_t$  be an equivalence relation on tuples.  $\mathcal{T}$  has  $k$ -anonymity for  $=_t$  if:*

$$\forall u \in \mathcal{T}. |\{u' \in \mathcal{T}_U \mid u' =_t u\}| \geq k$$

In other words, the size of the non-empty equivalence classes induced by  $=_t$  is at least  $k$ . We say that a multiset of tuples  $\mathcal{T}$  has  $k$ -anonymity if  $\mathcal{T}$  has  $k$ -anonymity for  $=_t$  being tuple equality (the finest equivalence relation on tuples).  $k$ -anonymity is not expected to be a property of the tuple space itself, but of the release of data provided by the operations `aqry`, `aget` and `aput`. In particular, we say that  $k$ -anonymity is provided by a policy  $\Pi$  on a tuple space  $\mathcal{T}$  when for every query based on the above operations the released result  $u_a$  (cf. Fig. 4) has  $k$ -anonymity. Note that this does only make sense if the result  $u_a$  is a multiset of tuples, which could be the case when the aggregation function is a multiset operation like multiset union. Policies can be used to enforce  $k$ -anonymity on specific queries. Consider for instance the previously mentioned example of the attacker trying to infer information about distances and positions of devices. Assume the device information is stored in tuples  $(x, y, i, j, d)$  where  $(x, y)$  are actual coordinates of the devices,  $(i, j)$  represents the zone in the grid and  $d$  is the computed distance to the closest PoI. Suppose further that we want to provide access to a projection of those tuples by hiding the actual positions and providing zone and distance information. Hiding the positions is not enough and we want to provide 2-anonymity on the result. We can do so with the following policy:

```

1 2-anonymity:
2   aqry mset_union, float, float, int, int, float altered by
3   tuple func (fun x y i j d -> if anonymity(2) (i j d) else nil))

```

**Listing 1.7.** Example of  $k$ -anonymity for  $k = 2$ .

where `anonymity(k)` checks  $k$ -anonymity on the provided view  $\mathcal{T}_2$  (cf. Fig. 4), according to Definition 1. Basically, the enforcement of the policy will ensure that we provide the expected result, if in each zone there are at least two devices with the same computed distance, otherwise the query produces the empty set.

**$(\varepsilon, \delta)$ -differential Privacy.** Differential privacy techniques [7] aim at protecting against attackers that can perform repeated queries with the intention of inferring information about the presence and/or contribution of single data item in a data set. The main idea is to add controlled noise to the results of queries so to reduce the amount of information that such attackers would be able to obtain. Data accuracy is hence sacrificed for the sake of privacy. For instance, in the motivational example of the distance gradient, differential privacy can be used to approximate the result of the aggregations performed by the gradient computation. This is done in order to minimize leakage about the actual positions and distance of each neighbouring device. Differential privacy is a property of a randomized algorithm, where the data set is used to give enough state information in order to increase indistinguishably. Randomization arises from privacy protection mechanisms based on e.g. sampling and adding randomly distributed noise. The property requires that performing a query for all possible neighbouring subsets of some data set, the addition (or removal) of a single data item produces almost indistinguishable results. Differential privacy is often presented in terms of histogram representations of databases not suitable for our purpose. We present in the following a reformulation of differential privacy for our setting. Let  $\mathbf{P}[\mathcal{A}(\mathcal{T}) \in S]$  denote the probability that the output  $\mathcal{A}(\mathcal{T})$  of a randomized algorithm  $\mathcal{A}$  is in  $S$  when applied to  $\mathcal{T}$ , where  $S \subseteq \mathbf{R}(\mathcal{A})$  and  $\mathbf{R}(\mathcal{A})$  is the codomain of  $\mathcal{A}$ . In our setting  $\mathcal{A}$  should be seen as the execution of an aggregated query, and that randomization arises from random noise addition.  $(\varepsilon, \delta)$ -differential privacy in our setting is then defined as the following property.

**Definition 2** ( $(\varepsilon, \delta)$ -differential privacy). *Let  $\mathcal{A}$  be a randomized algorithm,  $\mathcal{T}$  be a tuple space,  $e$  be Euler's number, and  $\varepsilon$  and  $\delta$  be real numbers.  $\mathcal{A}$  satisfies  $(\varepsilon, \delta)$ -differential privacy if and only if for any two tuple spaces  $\mathcal{T}_1 \subseteq \mathcal{T}$  and  $\mathcal{T}_2 \subseteq \mathcal{T}$  such that  $\|\mathcal{T}_1 \ominus \mathcal{T}_2\|_{\tau_u} \leq 1$ , and for any  $S \subseteq \mathbf{R}(\mathcal{A})$ , the following holds:*

$$\mathbf{P}[\mathcal{A}(\mathcal{T}_1) \in S] \leq e^\varepsilon \cdot \mathbf{P}[\mathcal{A}(\mathcal{T}_2) \in S] + \delta$$

Differential privacy can be enforced by policies that add a sufficient amount of random noise to the result of the queries. There are several noise addition algorithms that guarantee differential privacy. A common approach is based on the *global sensitivity* of data set for an operation and a *differentially private mechanism* which uses the global sensitivity to add the noise. Global sensitivity measures the largest possible distance between neighbouring subsets (i.e. differing in exactly one tuple) of a tuple space, given an operation. The differentially private mechanism uses this measure to distort the result when the operation is applied. To define a notion of sensitivity in our setting, assume that for every basic type  $\tau$  there is a *norm* function  $\|\cdot\| : \tau_u \rightarrow \mathbb{R}$  which maps every tuple into a real number. This is needed in order to define a notion of difference between tuples. We are now ready to define a notion of sensitivity for a given aggregate operator  $\lambda_D$ .

**Definition 3 (Sensitivity).** Let  $\mathcal{T}$  be a tuple space,  $\lambda_D : \tau_{u'} \rightarrow \tau_{u^*}$  be an aggregation function, and  $p \in \mathbb{N}^+$ . The  $p^{\text{th}}$ -global sensitivity  $\Delta_p$  of  $\lambda_D$  is defined as:

$$\Delta_p(\lambda_D) = \max_{\substack{\forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{T} \\ \mathcal{T}_1 \cap \mathcal{T}_2 = 1}} \sqrt[p]{\sum_{i \in \{0, \dots, |\tau_{u'}|\}} \|\lambda_D(\mathcal{T}_1)_i\| - \|\lambda_D(\mathcal{T}_2)_i\|}^p} \quad (1)$$

Roughly, Eq. (1) is expressing that the sensitivity scale of an aggregate operator is determined by the largest value differences between all fields of the aggregated tuples. The global sensitivity can then be used to introduce Laplace noise according to the well-known *Laplace mechanism*, which provides  $(\epsilon, 0)$ -differential privacy.

**Definition 4 (Laplace noise addition).** Let  $\mathcal{T}$  be a tuple space,  $\lambda_D : \tau_u^* \rightarrow \tau_{u'}$  be an aggregation function,  $\oplus : \tau_{u'} \times \tau_{u'} \rightarrow \tau_{u'}$  be an addition operator for type  $\tau_{u'}$ ,  $\epsilon \in ]0, 1]$ ,  $p \in \mathbb{N}^+$ , and  $\mathbf{Y} = (Y_1, \dots, Y_i, \dots, Y_n)$  be a tuple of random variables that are independently and identically distributed according to the Laplace distribution  $Y_i \sim \mathcal{L}(0, \Delta_p(\lambda_D)/\epsilon)$ . The Laplace noise addition function  $\text{laplace}_{\mathcal{T}, \lambda_D, \epsilon}$  is defined by:

$$\text{laplace}_{\mathcal{T}, \lambda_D, \epsilon}(u) = u \oplus \mathbf{Y} \quad (2)$$

Note that the function is parametric with respect to the noise addition operator  $\oplus$ . For numerical values  $\oplus$  is just ordinary addition. In general, for  $\oplus$  to be meaningful, one has to define it for any type. For complex types such as strings, structures or objects this is not trivial, and either one has to have a well-defined  $\oplus$  or other mechanisms should be considered for complex data types.

Consider again our motivational example of distance gradient computation, we can define a policy to provide differential privacy on the aggregated queries of each round of the computation as follows:

```

1 edp:
2   qry minD, float, float, int, int, float altered by
3   result func (fun x y i j d -> (laplace minD 0.9 (x, y, d)))

```

**Listing 1.8.** Example of  $(\epsilon, 0)$ -differential privacy policy.

The policy controls queries aiming at retrieving the information  $(x, y)$  coordinates,  $(i, j)$  zone and distance  $d$  of the device that is closest to a PoI, obtained by the aggregation function `minD`. The query returns only the coordinates of such device and its distance, after distorting them with Laplace noise by function `laplace`, implemented according to Definition 4 (with the tuple space being the provided view  $\mathcal{T}_2$ , cf. Fig. 4).

More in general, the enforcement of policies of the form

```

1   qry  $\lambda_D$ , U altered by
2   template func  $D_U$ 
3   tuple func  $D_u$ 
4   result func (fun u -> (laplace  $\lambda_D$   $\epsilon$  u))

```

**Listing 1.9.** Schema for  $(\epsilon, 0)$ -differential privacy policies.

provides  $(\varepsilon, 0)$ -differential privacy on the view of the tuple space (cf.  $\mathcal{T}_2$  in Fig. 4) against aggregated queries based on the aggregation function  $\lambda_D$ .

## 4 Aggregation Policies at Work

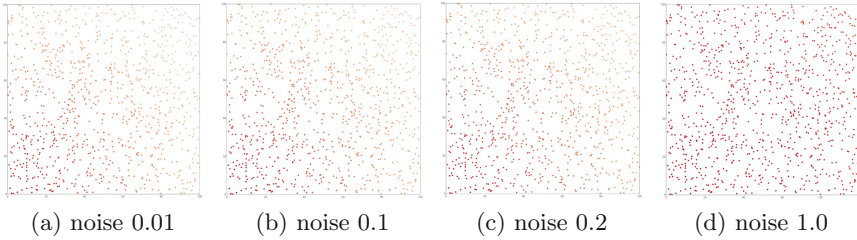
To showcase the applicability of our approach to aggregate computing applications, we describe in this section a proof-of-concept implementation of our policy language and its enforcement mechanism in a tuple space library (cf. Sect. 4.1), and the implementation of one of the archetypal self-organizing building blocks used in aggregate programming, namely the computation of a distance gradient field, that we use also to benchmark the library (cf. Sect. 4.2).

### 4.1 Implementation of a Proof-of-Concept Library

The open-source library we have implemented is available for download, installation and usage at <https://github.com/pSpaces/goSpace>. The main criteria for choosing Go was that it provides a reasonable balance between language features and minimalism needed for a working prototype. Features that were considered important included concurrent processes, a flexible reflection system and a concise standard library. The `goSpace` project was chosen because it provided a basic tuple space implementation, and had the fundamental features, such as addition, retrieval and querying of tuples based on templates, and it also provides derived features such as retrieval and querying of multiple tuples. Yet, `goSpace` itself was modified in order to provide additional features needed for realizing the policy mechanism. One of the key features of the implementation is a form of code mobility that allows to transfer functions across different tuple spaces. This was necessary to serve as a foundation for allowing user-defined aggregate functions across multiple tuple spaces. Further, the library was implemented to be slightly more generic than what is given in Sect. 2 and can in principle be applied to other data structures beyond tuple spaces and aggregation operators on tuple space. Currently, our `goSpace` implementation supports policies for the actions `aput`, `aget` and `aqry` but it can be easily extended to support additional operations.

### 4.2 Protecting Privacy in a Distance Gradient

We have implemented the case study of the distance field introduced in Sect. 1 as a motivational example. In our implementation, the area where devices and PoIs are placed, is discretized as a grid of zones; each device and PoI has a position and is hence located in a zone. The neighbouring relation is given by the zones: two devices are neighbours if their zones are incident. Devices can only detect PoIs in their own zone and devices cannot communicate directly with each other: they use a tuple space to share their information. Each device publishes in the tuple space their information (position, zone and computed distance) labelled with a privacy policy. The aggregation performed in each round uses the `aqry`

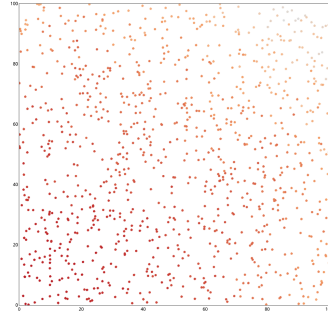


**Fig. 5.** Distance gradients with aggregation policies based on noise addition. (Color figure online)

operation with an aggregation function that selects the tuple with the smallest distance to a PoI.

Different policies can be considered. The identity policy would simply correspond to the typical computation of the field as seen in the literature. Basically, all devices and the tuple space are considered to be trustworthy and no privacy guarantees are provided. Another possibility would be to consider that devices, and other agents that want to exploit the field, cannot be fully trusted with respect to privacy issues. A way to address this situation would be to consider policies that hide or distort the result of the aggregated queries used in each round.

We have performed several experiments with our case study and we have observed, as expected, that such policies may affect accuracy (due to noise addition) and performance (due to the overhead of the policy enforcement mechanism). Some results are depicted in Figs. 2, 5 and 6. In particular the figures show experiments for a scenario with 1000 devices and a discretization of the map into a  $100 \times 100$  grid. Figure 2 shows the result where data is not protected but is provided as-it-is, while Fig. 5 shows results that differ in the amount of noise added to the distance obtained from the aggregated queries. This is regulated by a parameter  $x$  so that the noise added is drawn from a uniform distribution in  $[-x * d, x * d]$ ,



**Fig. 6.** Gradient with noise

where  $d$  is the diameter of each cell of the grid (actually  $\sqrt{10 * 10}$ ). In the figures, each dot represents a device and the color intensity is proportional to the distance to the PoI, which is placed at  $(0, 0)$ . Highest intensity corresponds to distance 0, while lowest intensity corresponds to the diameter of the area ( $\sqrt{200}$ ). The results with more noise (Fig. 5(d)) make it evident how noise can affect data accuracy: the actual distance seems to be the same for all nodes. However, Fig. 5, which shows the same data but where the color intensity goes from 0 to the

maximum value in the field, reveals that the price paid for providing more privacy does not affect much the field: the gradient towards the PoI is still recognizable.

## 5 Conclusion

We have designed and implemented a policy language which allows to succinctly express and enforce well-understood privacy models in the syntactic category such as  $k$ -anonymity, and in the semantic category such as  $(\epsilon, \delta)$ -differential privacy. Aggregate operations and templates defined for a tuple space were used to give a useful abstraction for aggregate programming. Even if not shown here, our language allows to express additional syntactic privacy models such as  $\ell$ -diversity [15],  $t$ -closeness [13, 21] and  $\delta$ -presence [8]. Our language does not only allow to adopt the above mentioned privacy models but it is flexible enough to specify and implement additional user-defined policies. The policy language and its enforcement mechanism have been implemented in a publicly available tuple space library. The language presented here has been designed with minimality in mind, and with a focus on the key aspects related to aggregation and privacy protection. Several aspects of the language can be extended, including richer operations to compose policies and label tuples, user-dependent and context-aware policies, tuple space localities and polyadic operations (e.g. to aggregate data from different sources as usual in aggregate computing paradigms). We believe that approaches like ours are fundamental to increase the security and trustworthiness of distributed and coordinated systems.

**Related Work.** We have been inspired by previous works that enriched tuple space languages with access control mechanism, in particular SCEL [6, 16] and KLAIM [3, 11]. We have also considered database implementations with access control mechanisms amenable for the adoption of privacy models. For example, [5] discusses the development strategies for FBAC (fine-grained access control) frameworks in NoSQL databases and showcases applications for MongoDB, the Qapla policy framework [18] provides a way to manipulate DBMS queries at different levels of data granularity and allows for transformations and query rewriting similar to ours, and PINQ [17] uses LINQ, an SQL-like querying syntax, to express queries that can apply differential privacy embedded in C#. With respect to databases our approach provides a different granularity to control operations, for instance our language allows to easily define template-dependent policies. Our focus on aggregate programming has been also highly motivated by the emergence of aggregate programming and its application to domains of increasing interest such as the IoT [1]. As far as we know, security aspects of aggregate programming are considered only in [4] where the authors propose to enrich aggregate programming approaches with trust and reputation systems to mitigate the effect of malicious data providers. Those considerations are related to data integrity and not to privacy. Another closely related work is [9] where the authors present an extension to a tuple space system with privacy properties based on cryptography. The main difference with respect to our work is in the different privacy models and guarantees considered.



**Future Work.** One of the main challenges of current and future privacy protection systems for distributed systems, such as the one we have presented here, is their computational expensiveness. We plan to carry out a thorough performance evaluation of our library. We plan in particular to experiment with respect to different policies and actions. It is well known that privacy protection mechanism may be expensive and finding a right trade-off is often application-dependent. Part of the overhead in our library is due to the preliminary status of our implementation where certain design aspects have been done in a naive manner to prioritize rapid prototyping over performance optimizations, e.g. use of strong cryptographic hashing, use of standard library concurrent maps and redundancies in some data structures. This makes room for improvement and we expect that the performance of our policy enforcement mechanism will be significantly improved. More in general, finding the optimal  $k$ -anonymity is an NP-hard problem. There is however room for improvements. For instance, [12] provides an approximation algorithm. This algorithm could be adapted if enforcement is needed. An online differentially private algorithm, namely *private multiplicative weights algorithm*, is given in [7]. Online algorithms are worth of investigation since interactions with  $\mathcal{T}$  are inherently online. Treatment of functions and functional data in differential privacy setting can be found [10]. We are currently investigating online efficient algorithms to improve the performance of our library.

## References

1. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. *Computer* **48**(9), 22–30 (2015)
2. Beal, J., Viroli, M.: Building blocks for aggregate programming of self-organising applications. In: Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2014, London, United Kingdom, 8–12 September 2014, pp. 8–13. IEEE Computer Society (2014)
3. Bruns, G., Huth, M.: Access-control policies via Belnap logic: effective and efficient composition and analysis. In: Proceedings of CSF 2008: 21st IEEE Computer Security Foundations Symposium, pp. 163–176 (2008)
4. Casadei, R., Aldini, A., Viroli, M.: Combining trust and aggregate computing. In: Cerone, A., Roveri, M. (eds.) SEFM 2017. LNCS, vol. 10729, pp. 507–522. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-74781-1\\_34](https://doi.org/10.1007/978-3-319-74781-1_34)
5. Colombo, P., Ferrari, E.: Fine-grained access control within NoSQL document-oriented datastores. *Data Sci. Eng.* **1**(3), 127–138 (2016)
6. De Nicola, R., et al.: The SCEL language: design, implementation, verification. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems. LNCS, vol. 8998, pp. 3–71. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-16310-9\\_1](https://doi.org/10.1007/978-3-319-16310-9_1)
7. Dwork, C., Roth, A.: The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.* **9**(3–4), 211–487 (2013)
8. Ercan Nergiz, M., Clifton, C.:  $\delta$ -presence without complete world knowledge. *IEEE Trans. Knowl. Data Eng.* **22** (2010). <https://ieeexplore.ieee.org/document/4912209/>
9. Floriano, E., Alchieri, E., Aranha, D.F., Solis, P.: Providing privacy on the tuple space model. *J. Internet Serv. Appl.* **8**(1), 19:1–19:16 (2017)

10. Hall, R., Rinaldo, A., Wasserman, L.: Differential privacy for functions and functional data. *J. Mach. Learn. Res.* **14**(1), 703–727 (2013)
11. Hankin, C., Nielson, F., Nielson, H.R.: Advice from Belnap policies. In: *Computer Security Foundations Symposium*, pp. 234–247. IEEE (2009)
12. Kenig, B., Tassa, T.: A practical approximation algorithm for optimal k-anonymity. *Data Min. Knowl. Disc.* **25**(1), 134–168 (2012)
13. Li, N., Li, T., Venkatasubramanian, S.: t-closeness: privacy beyond k-anonymity and l-diversity. In: *International Conference on Data Engineering (ICDE)*, pp. 106–115 (2007)
14. Lluch-Lafuente, A., Loreti, M., Montanari, U.: Asynchronous distributed execution of fixpoint-based computational fields. *Log. Methods Comput. Sci.* **13**(1) (2017)
15. Machanavajjhala, A., Kifer, D., Gehrke, J., Venkatasubramanian, M.: l-diversity: privacy beyond k-anonymity (2014)
16. Margheri, A., Pugliese, R., Tiezzi, F.: Linguistic abstractions for programming and policing autonomic computing systems. In: *2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC) Ubiquitous Intelligence and Computing*, pp. 404–409 (2013)
17. McSherry, F.: Privacy integrated queries: an extensible platform for privacy-preserving data analysis. *Commun. ACM* **53**(9), 89–97 (2010)
18. Mehta, A., Elnikety, E., Harvey, K., Garg, D., Druschel, P.: QAPLA: policy compliance for database-backed systems. In: *26th USENIX Security Symposium (USENIX Security 2017)*, Vancouver, BC, pp. 1463–1479. USENIX Association (2017)
19. Official Journal of the European Union. Regulation (EU) 2016/679 of the European parliament and of the council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/EC (general data protection regulation), L119, pp. 11–88, May 2016
20. Samarati, P., Sweeney, L.: Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report, Harvard Data Privacy Lab (1998)
21. Soria-Comas, J., Domingo-Ferrer, J., Sánchez, D., Martínez, S.: t-closeness through microaggregation: strict privacy with enhanced utility preservation. *CoRR*, abs/1512.02909 (2015)
22. Sweeney, L.: k-anonymity: a model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl. Based Syst.* **10**(5), 557–570 (2002)
23. Viroli, M., Damiani, F.: A calculus of self-stabilising computational fields. In: Kühn, E., Pugliese, R. (eds.) *COORDINATION 2014*. LNCS, vol. 8459, pp. 163–178. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-662-43376-8\\_11](https://doi.org/10.1007/978-3-662-43376-8_11)



# Distributed Coordination Runtime Assertions for the Peer Model

eva Kühn<sup>1(✉)</sup>, Sophie Therese Radschek<sup>1(✉)</sup>, and Nahla Elaraby<sup>1,2(✉)</sup>

<sup>1</sup> TU Wien, Argentinierstr. 8, Vienna, Austria

{eva.kuehn,sophie.radschek,nahla.el-araby}@tuwien.ac.at

<sup>2</sup> Canadian International College – CIC, Cairo, Egypt

<http://www.complang.tuwien.ac.at/eva>

**Abstract.** Major challenges in the software development of distributed systems are rooted in the complex nature of coordination. Assertions are a practical programming mechanism to improve the quality of software in general by monitoring it at runtime. Most approaches today limit assertions to statements about local states whereas coordination requires reasoning about distributed states. The Peer Model is an event-based coordination programming model that relies on known foundations like shared tuple spaces, Actor Model, and Petri Nets. We extend it with distributed runtime invariant assertions that are specified and implemented using its own coordination mechanisms. This lifts the concept of runtime assertions to the level of coordination modeling. The concept is demonstrated by means of an example from the railway domain.

**Keywords:** Coordination model · Runtime assertions  
Distributed systems · Tuple space

## 1 Introduction

The development of coordination is complex because of the asynchronous nature of distributed systems. Coordination models like the Actor Model [2], Petri Nets [20], Reo [3] and the Peer Model [15] are well suited to model the interactions in concurrent systems. They allow reasoning about coordination at a certain abstraction level. Such a model-driven approach has the advantage to specify the system unambiguously, and to support the verification of system properties. Assertions are a practical programming mechanism to improve the quality of software in general by monitoring it at runtime. Most approaches today limit assertions to statements about local states whereas coordination requires reasoning about distributed states.

The objective of this paper is to introduce event-based, asynchronous, and distributed runtime assertions to a coordination model. This approach shall lift the concept of runtime assertions to the level of coordination modeling. A main problem is to find a good tradeoff between a very costly mechanism and a good

quality of verification. A further requirement is to provide a notation that is well integrated into the notation of the coordination model so that developers need not learn a different language. This way, the assertions can be directly added to the model of the application using the same modeling concepts. Keeping the number of concepts that a developer has to comprehend small contributes to usability [24].

We use the Peer Model (see Sect. 3) as reference coordination model notation and introduce an assertion mechanism based on its own modeling language. Checking of distributed assertions can in turn be considered a coordination problem. We propose therefore model transformation algorithms that automatically translate the assertion specifications into coordination constructs of the model. Thus they become part of the application model and can be implemented with the same mechanisms. In this way, the extended Peer Model is a contribution towards verification of practical distributed systems.

The paper is structured as follows: Sect. 2 summarizes related work on distributed assertion mechanisms. Section 3 gives an overview of the Peer Model. In Sect. 4 we extend the Peer Model by distributed runtime invariant assertions that are specified and implemented using its own coordination mechanisms. In Sect. 5 the concept is demonstrated by means of an example from the railway domain. Section 6 summarizes the results and Sect. 7 gives an outlook to future work.

## 2 Related Work

Proper assertions for verification of coordination models should be declarative, event-driven and distributed. Assertions express correctness properties that must hold in all system states. Coordination model assertions should be model-based and use the same or similar notation of the model to keep the number of concepts small. For concurrent distributed systems, run time assertions [6] provide high confidence that the properties to be verified are fulfilled. However, assertions in this case should be of asynchronous nature so that they neither disturb the execution nor interfere with user code and data by any means. They should be bootstrapped for implementation with native model mechanisms.

[9] introduces a solution for coordinating activities of multiple intelligent agents using the tuple space computational model. The tuple space-based problem solving framework is implemented on an Intel Hypercube iPSC/2 allowing multiple rule-based systems concurrently performing their dedicated interrelated tasks. Their approach to build a rule-based system resembles our approach towards distributed assertions, but on a different level of reasoning.

A language for determining correct behavior in distributed systems is described in [25]. This approach succeeds to verify distributed systems but introduces a different language and notation. Also, waiting states for operators' actions add much overhead to the run time check which is not acceptable in most systems, especially those of safety-critical nature.

Reusable assertions were developed in [13], where the assertions can be used at different abstraction levels. The approach handles the timing but does not provide asynchronous assertions for distributed system modules.

In [14] a lightweight modeling language (Alloy), based on first-order relational logic is used to model and analyze Reo connectors. Reo [3] is a coordination language based on components and connectors. The model presented in this work preserves the original structure of the Reo network, avoiding a complex translation effort. The approach handles basic channel types, compositional construction of more complex connectors, constraints on the environment, and networks exposing context-sensitive behavior. However, in Alloy, no asynchronous assertion mechanism is provided and the properties are defined in terms of first-order predicates, and not in the same model notation.

[4] investigates runtime verification where the properties are expressed in linear time temporal logic (LTL) or timed linear time temporal logic (TLTL). Runtime verification is identified in comparison to model checking and testing. A three-valued semantics (with truth values true, false, inconclusive) is introduced as an expressive interpretation indicating whether a partial observation of a running system meets an LTL or TLTL property. For LTL, a minimal size deterministic monitor is generated identifying a continuously monitored trace as either satisfying or falsifying a property as early as possible, similarly for TLTL. The approach is successful and provides foundation for real-time monitoring of system properties. The approach does not provide asynchronous assertions and uses a different notation from the model.

Mining of the simulation data as an effective solution for generation of assertions is presented in [5]. The research provides an efficient way to develop a highly trusted set of assertions and solves the incompleteness problem of Assertion Based Verification (ABV). The assertions are not based on the model notation and need effort of the developer to integrate them into the runtime. Distributed asynchronous mechanisms for assertions are not provided.

The work presented in [19] investigates the realization of infrastructures to maintain and access data and use it in assertions. Assertions can be managed across distributed system to support sophisticated assertions for intercommunications in distributed systems. A tool-chain is provided for programming assertions on interaction history written in regular expressions that incorporate inter-process and inter-thread behavior amongst multiple components in a distributed system. The mechanism is promising but not model-based.

A novel approach described in [26] uses machine learning for automatic assertion generation. The approach releases the burden of manually specifying the assertions, which is a time-consuming and error-prone process. Similarly the authors of [21] also deal with the problem of insufficiently written assertions and propose an automatic approach for assertion generation based on active learning. The approach targets complex Java programs, which cannot be symbolically executed. Test cases are used as a base for assertion generation and active learning to iteratively improve the generated assertions is applied. Both approaches are runtime-based and automatic, but they are not applicable for distributed systems where coordination needs to be defined and asserted asynchronously to validate the system functionality.

[8] introduces runtime verification as a complementary validation technique for component-based systems written in the BIP (Behavior, Interaction and Priority) framework. The proposed solution dynamically builds a minimal abstraction of the current runtime state of the system so as to lower the overhead. Monitors are directly generated as BIP components where the C code generator of BIP generate actual monitored C programs and remain compatible with previously proposed runtime verification frameworks. However, an adaptation layer is needed to adapt monitors generated by other existing tools.

A general approach to monitor the specifications of decentralized systems is presented in [7]. The specification is considered as a set of automata associated with monitors attached to different components of the decentralized system. A general monitoring algorithm is developed to monitor the specifications. Execution History Encoding (EHE) data structure is introduced as a middle ground between rewriting and automata evaluation. Rewriting is restricted to Boolean expressions, parameters are determined and their respective effect on the size of expressions and the upper bounds are fixed. A similar decentralized algorithm for runtime verification of distributed programs is introduced in [18]. The technique works on asynchronous distributed systems, where a global clock is not assumed. The specification language used is full LTL so that temporal properties can be monitored. The algorithm can also determine the verification verdict once a total order of events in the system under inspection can be constructed. The presented algorithm addresses shortcomings in other algorithms as it does not assume a global clock, is able to verify temporal properties and is sound and complete. However, the main concern is the large number of monitoring messages introducing an increased communication and memory overhead.

[12] proposes a session-based verification framework for concurrent and distributed ABS models. Applications are categorized with respect to the sessions in which they participate. Their behaviors are partitioned based on sessions, which include the usage of future. The presented framework extends the protocol types through adding terms suitable for capturing the notion of futures, accordingly the communication between different ABS endpoints can be verified by the corresponding session-based composition verification framework. Timing is handled but not in an asynchronous way.

In summary, none of the listed approaches provides assertions that are specified at the model level, are runtime-based, asynchronous and distributed, and the implementation of which can be bootstrapped with native model mechanisms. The motivation was therefore to use the Peer Model (see Sect. 3) as reference coordination model notation and to provide a full set of assertions that can be translated to the same model notation, whereby all mentioned requirements for assertions are fulfilled. This way, the intent is to extend the Peer Model to become capable to trustfully monitor distributed systems.

### 3 The Peer Model in a Nutshell

The Peer Model [15] is a coordination model that relies on known foundations like shared tuple spaces [10, 11, 17], Actor Model [2], and Petri Nets [20]. It clearly

separates coordination logic from business logic and is well suited to model reusable coordination solutions in form of patterns. It provides a refinement-based modeling approach where you start with the highest abstraction layer of your application model, and stepwise improve the model<sup>1</sup>. The main concepts are briefly explained in the following.

*Peer.* A peer relates to an actor in the Actor Model [2]. It is an autonomous worker with in- and outgoing mailboxes, termed peer input container (PIC) and peer output (POC) container.

*Container.* A container is a sub-space in the tuple space that stores entries that are written, read, or taken (i.e., read and removed) in local transactions.

*Entry.* Entries are the units of information passed between peers. An entry has system- and application-defined properties, which are name/value pairs. Examples for system-defined properties are e.g.: `ttl` (time-to-life; if it expires, the entry becomes an exception entry that wraps the original entry), and `fid` (flow id). All entries with the same `fid` belong to the same flow, i.e., they logically belong together. An exception entry is a special system-defined entry. If it is caused by the expiration of an entry `ttl` it inherits the `fid` of the entry causing the exception, and “wraps” the original entry in a property termed `entry`. The type of the exception is stored in the `excType` property.

*Link.* A link transports entries between two containers `c1` and `c2`. It selects entries from `c1` using a query. Link operations are `move` (take entries from `c1` and write them to `c2`), `copy` (read entries from `c1` and write them to `c2`), `test` (read entries from `c1`), `delete` (read and remove entries from `c1`), `create` (create entries of given type and count and write them to `c2`), and no operation specified (no container access; this link serves solely to perform tests on variables). A link can set or get variables (see below) and set or get properties of entries (if an operation was used that selected entries from `c1`). In these expressions also system-defined functions like `head()` (get first element of a list), `tail()` (get rest of a list) can be used. Finally, a link may have system-defined properties, e.g., `dest` (peer to whose PIC all selected entries shall be sent), and `mandatory` (flag whether the link is obligatory).

*Wiring.* The coordination behavior of a peer is explicitly modeled with wirings, which have some similarity with Petri Net transitions [20]. All wiring specifications are executed concurrently in so-called wiring instances. Each instance has an internal container termed Wiring Internal Container (WIC), which serves as a temporary, local entry collection. Each wiring instance is a local transaction [16] that transports entries between containers with help of links. A link from a PIC or POC to the WIC is termed “guard” (G) and a link from the WIC to a PIC or POC is termed “action” (A). The operational behavior of a wiring instance is the sequential execution of guards, and actions. The arrival of entries in peer space containers triggers the execution of guards. A wiring is triggered if

---

<sup>1</sup> There exists an Event-B-based model checker for the Peer Model [22].

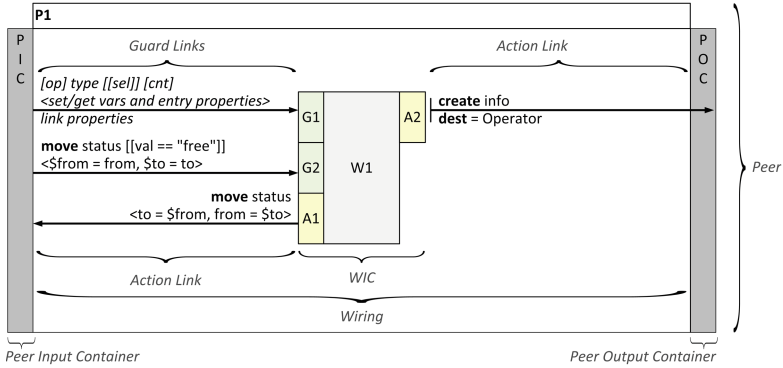


Fig. 1. Graphical notation for the main artifacts of the ground model.

all of its guard links are satisfied. Wirings also have system-defined properties. Each instance of a wiring obtains a new WIC. Wirings can treat exceptions.

*Query.* A query consists of: (1) entry type (mandatory), (2) an integer number (default is 1) specifying how many entries shall be selected, or ALL (all entries in the container that can be read or taken, possibly 0), and (3) a selector expression. It is fulfilled when enough entries with specified type match the selector. It selects entries with same type and of compatible flows [15]. The first link of a wiring determines the flow. All further links select only entries with same *fid*, or not *fid* set. A selector can use AND, OR and NOT operators.

*Variable.* There exist application- and system-defined variables. The former are denoted with “\$” and used in the scope of a wiring instance. The latter are denoted with “\$\$” and set by the runtime, e.g., \$\$PID (id of the current peer), \$\$WID (id of the current wiring), \$\$CNT (number of entries selected by a link query), and \$\$FID (current flow id of the wiring instance).

Figure 1 shows the graphical notation of the core concepts. It depicts one peer with id P1 and one wiring termed W1. The wiring possesses two guard links (G1, G2), and two action links (A1, A2) – represented by arrows. G1 (connecting PIC and WIC) depicts the general notation: Operation, query and the access to variables and entry properties are located above the respective arrow and properties are denoted below. Guard links are directed towards the WIC, and action links are pointing in the other direction. If entries are read (and removed) from a container, then the arrow base is connected to it; if entries are written into a container, then the arrow head is connected to it. Otherwise, a vertical bar is used on the respective side to indicate that no entries “go through”.

G2 shows a guard link that takes one entry of type status from the PIC, using a selector specifying that the val property of the entry must be set to free, and writes the entry into the WIC. It stores the from and to properties of the selected entry in local variables \$from and \$to. In action link A1 the wiring takes the above mentioned status entry from the WIC, swaps its to and from properties



with help of the local variables, and writes it back to the PIC. A2 creates one new entry of type `info`, and as the link property `dest` is set, the entry is sent to the PIC of the destination address, in this case the Operator peer. If `dest` is set, the arrow of the link goes through the POC. All system-defined terms are written in type `writer` style.

## 4 Introducing Peer Model Assertions

To improve the quality of specifications at the modeling level, the Peer Model is augmented with declarative invariant assertions. The goal of the assertion mechanism is to detect model inconsistencies at runtime. The trade-off between interference with normal execution, network traffic, and strict evaluation of assertions should be configurable. The mechanism shall be distributed, asynchronous and event-driven, assuming neither reliable nor ordered communication.

Assertions are statements about container states, because the shared coordination state of the distributed system is manifested in them. They follow the same principles as the Peer Model concepts (see Sect. 3), e.g., they have properties, are flow sensitive, and cause exception entries to be treated by the application. The consideration of the flow is fundamental, because the scope of each assertion check is restricted to one flow. This has the advantage that the correlation of distributed events can fully rely on the Peer Model's native flow mechanisms. Peers are assumed to report about assertion violations to one or more coordinators, which are peers in charge of controlling the assertion checking for a particular flow. Reporting considers actual and past states; liveness is out of scope of this mechanism.

We differentiate between assertions referring to containers of one single peer only (intra-peer assertions) and assertions involving multiple (local or distributed) peers (inter-peer assertions). For the former, violations are checked autonomously by the peer itself, for the latter, violation checking is understood as a distributed coordination problem. The proposed mechanism includes a translation of the declarative assertions to existing Peer Model constructs by a model translation algorithm (MTA) (see Sect. 4.1). For intra-peer assertion checks, respective wirings are statically added to the meta-model of the peers that monitor their local container states and create assertion exception entries on violation. The inter-peer assertion mechanism relies in principle on the same wiring type, but instead of exceptions, report entries are created and sent to coordinators, which in turn decide if exceptions must be created based on the status of received reports. Therefore, the overall assertion check can be reduced to an analogous wiring that checks a derived and normalized intra-peer assertion about these reports, generated by the MTA.

The required overhead consists only of carrying out the required tests when a new relevant event is received, storing the current reporting state and (in the inter-peer case) sending one report message to each coordinator on a new assertion violation. However, not all violations can be detected in case of communication failures. To cope with these issues and to support assertions about past

states, a history of the relevant events can be maintained in a dedicated internal peer container termed HIC. This history is created by extending the existing application wirings to capture all local changes and deletions of entries referenced by assertions in the HIC, which creates a certain overhead. However, the overhead can be reduced by deleting events that cannot be referenced any more by active assertions (e.g. a flow has expired), assigning time-to-live properties and introducing bounds on the depth of the history.

The expressiveness of the mechanism can be further enhanced by introducing variables shared between different sub-assertions (see below) of inter-peer assertions. This produces additional message overhead, as reports about property changes in relevant entries must be sent to all respective wirings that reference these properties by means of these variables. Variable reports are only sent, if the corresponding assertion is not violated, because – like in failing optional links – variables can only be assumed to have meaningful value, if the expression that sets them was successful. In addition, assertion wirings depend on variable reports to be present and are activated by each new variable report. The mentioned trade-off between strictness and expressiveness versus overhead is controlled by the assertion model. Namely, avoiding inter-peer variables, and not using assertions about the past and not demanding a history reduces the overhead to a minimum.

The coordinator(s) for inter-peer assertions are specified by means of assertion properties. Default is no coordinator, meaning that each sub-assertion creates local exceptions instead of sending reports to coordinators; the logic of how to handle them can be explicitly modeled. Here, the trade-off is between many coordinators with a larger message overhead and fewer coordinators leading to a potential bottle-neck and single point-of-failure.

The syntax of assertions is given in Table 1 and their operational semantics is defined with existing Peer Model constructs. An assertion is either a simple one (SAss) or a complex one (CAss), i.e., sub-assertions connected with logic operators AND, OR, NOT and  $\rightarrow$  (equivalent to  $\wedge$ ,  $\vee$ ,  $\neg$  and  $\implies$ ). Sub-assertions of inter-peer assertions are intra-peer assertions and sub-assertions of intra-peer assertions are container assertions. Container assertions are satisfied, if the container contains exactly *quantor* entries of *type* that satisfy the query selector *sel*. Note that *type sel* and *set/get* refer to the Peer Model’s link notation (see Fig. 1). Intra- and inter-peer assertions are satisfied if the propositional formula corresponding to the assertion – in which the propositions are the sub-assertions – is satisfied.

#### 4.1 Model Translation Algorithm (MTA)

Intra-peer assertions are translated into single wirings located in the peer(s) corresponding to the context. Each assertion corresponds to a wiring that checks its truth value and raises an exception on violation. Such assertion wirings define and use all variables used in the corresponding assertion and overhead variables to evaluate the truth value of the assertion. These overhead variables are: 1 variable for each container sub-assertion to store the evaluation value, 2 variables for each container sub-assertion to store the number of entries that fit the

**Table 1.** EBNF for assertion notation.

$Ass ::= CAss \text{ ‘[’ } properties \text{ ‘]’} \mid SAss \text{ ‘}\rightarrow\text{’ } Ass$	assertion
$CAss ::= SAss \mid CAss \text{ ‘AND’ } CAss \mid CAss \text{ ‘OR’ } CAss \mid \text{ ‘NOT’ } CAss$	complex assertion
$SAss ::= Ctx \text{ ‘\{’ } PeerAss \text{ ‘\}’}$	simple assertion
$Ctx ::= pid \mid QCtx \mid var$	context
$QCtx ::= \text{ ‘[’ } quantor \text{ ‘]’} \text{ [ } var \text{ ‘IN’ } ] \text{ ‘<’ } type \text{ ‘>’}$	quantified context
$PeerAss ::= PeerCAss \mid PeerSAss \text{ ‘}\rightarrow\text{’ } PeerAss$	peer assertion
$PeerCAss ::= PeerSAss \mid PeerCAss \text{ ‘AND’ } PeerCAss \mid PeerCAss \text{ ‘OR’ } PeerCAss \mid \text{ ‘NOT’ } PeerCAss$	peer complex assertion
$PeerSAss ::= Container \text{ ‘\{’ } ContainerAss \text{ ‘\}’} \mid \text{ ‘[[’ } sel \text{ ‘]’}$	peer simple assertion
$Container ::= \text{ ‘PIC’ } \mid \text{ ‘POC’ } \mid \text{ ‘HIC’}$	container
$ContainerAss ::= \text{ ‘[’ } quantor \text{ ‘]’ } type \text{ [ ‘[[’ } sel \text{ ‘]’ } ] \text{ [ ‘<’ } set/get \text{ ‘>’}$	container assertion
$properties ::= (* \text{ list of properties of form label = value } *)$	property list
$quantor ::= (* \text{ ALL, NONE, or a number (default = 1) } *)$	quantification
$var ::= (* \text{ local variable referring to a peer } *)$	variable
$pid ::= (* \text{ id of a concrete peer } *)$	peer id

type and query, 1 to store the overall assertion value and 1 to store whether a violation must be reported. The assertion and overhead variables are initialised on a dedicated mandatory guard. Each container sub-assertion is tested using 3 guards: (1) tests how many entries there are in the container that fulfil the query of the assertion and saves the number in a variable; this guard also sets the variables used by the assertion. (2) tests how many entries there are in the container that fulfil the type of the assertion. (3) compares the quantities for (1) and (2) according to the quantor in the container sub-assertion and sets the corresponding truth variable to true if it is fulfilled. (1) and (2) are mandatory, (3) is only executed if the sub-assertion is asserted. The truth value of the overall assertion is set to true on an optional guard according to the logic formula of the intra-peer assertion. If the overall intra-peer assertion is violated and no report has yet been issued, a report is created in the peer input container (PIC) and an exception is raised in the peer output container (POC).

Each intra-peer assertion creates an overhead of  $k$  wirings with  $3n + 2$  variables and  $3n + 5$  links each ( $n \dots \#$  of container sub-assertions and  $k \dots \#$  of peers in the context). For each violation, 2 additional entries are created.

Figure 2 shows the translated wiring of an intra-peer assertion. G1 defines a variable  $\$q_i$  for the result of each *ContainerAss* (see Table 1), a variable  $\$q$  for the result of the entire *PeerAss*, a variable  $\$report$  to reflect whether the assertion has already been reported, and initializes all application variables of the wiring, if any. G1 is created once per wiring. G<sub>j.k</sub> ( $j = 2, \dots, n + 1, k = 1, 2, 3$ ) are generated for each of the  $n$  *ContainerAss*: G<sub>j.1</sub> tests all entries fulfilling *type* and *sel* (see Fig. 1) of the assertion query and saves their count in the variable  $\$c_j$ . G<sub>j.2</sub> counts all entries that fulfill the type of the assertion query and saves the value in the variable  $\$ca_j$  which is needed for the quantifier check. G<sub>j.3</sub> checks whether the right quantity of entries satisfies the *sel* by comparing

$\$c_j$ ,  $\$ca_j$  and the quantor in the *ContainerAss* as follows: ( $quantor == \text{ALL}$  AND  $\$c_j == \$ca_j$ ) OR ( $quantor == \text{NONE}$  AND  $\$c_j == 0$ ) OR ( $\$c_j == quantor$ ).  $G_n + 2$  sets the variable  $\$q$  to the result of the overall assertion based on the values of  $\$q_1, \dots, \$q_n$ .  $G_n + 3$  checks whether an assertion violation has already been reported for this wiring and flow, which is the case if an `assReported` entry exists in the PIC. If the assertion is violated and has not been reported yet, A1 creates an `assReported` entry for the wiring and flow in the PIC. If the assertion is violated and has not been reported yet, A2 creates an exception with type `ASSERTION` for the wiring and flow in the POC.

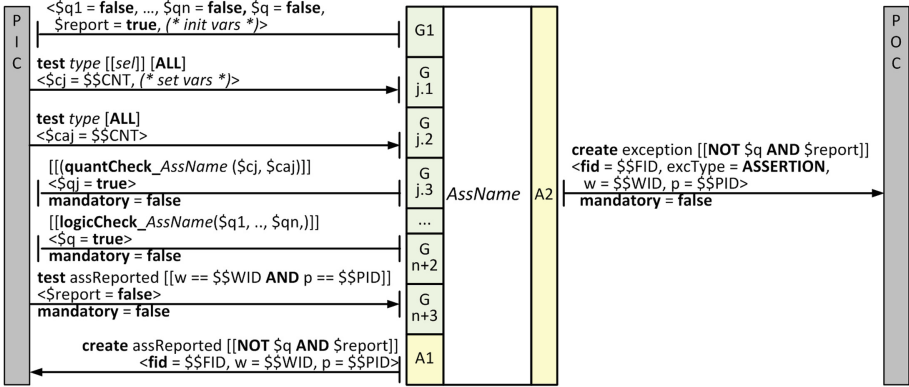


Fig. 2. Corresponding wiring for an intra-peer assertion.

Inter-peer assertions are translated analogously: Sub-assertions (intra-peer assertions) are treated individually and the overall inter-peer assertions evaluated by coordinators based on the evaluation value of the sub-assertions.

The sub-assertions are translated into intra-peer assertion wirings. But instead of raising exceptions, they send violation reports to coordinators when the evaluation value might cause the overall assertion to fail. The differences are reflected in A2, which creates an `assReport` entry and sends it to all coordinators, and in the selector of  $G_n + 2$ , which is inverted for negated assertions with quantor **ALL** and non-negated assertions with quantor **NONE** in the context.

The overall assertion is evaluated in dedicated wirings in coordinators that collect the corresponding sub-assertion values and combine them according to the propositional formula of the overall assertion. The resulting wiring is analogous to an intra-peer assertion wiring where the sub-assertions are checks on the number of assertion reports sent by other peers. Variable initialization, evaluation of the propositional skeleton, keeping track of violations and raising of exceptions is analogous to intra-peer wirings. For each sub-assertion, the number of reports sent by peers is counted ( $G_{2.1}$ ) and the numbers are compared according to the quantor and the variable set to true if the sub-assertion is fulfilled ( $G_{2.2}$ ).

Inter-peer assertions introduce  $c$  overall evaluation wirings ( $c \dots \#$  of coordinators) with  $2m + 5$  links and  $2m + 2$  variables each ( $m \dots \#$  of sub-assertions)

and  $\sum_1^m k_i$  intra-peer assertion wirings ( $k_i \dots \#$  of peers in the context for each sub-assertion) with  $3n_i + 5$  links and  $3n_i + 2$  variables ( $n_i \dots \#$  of container sub-assertions in the respective sub-assertions). For each violation at most  $\sum_1^m k_i$  messages are sent and  $2c + \sum_1^m 2k_i$  entries are created.

The use of HIC adds additional links to each wiring that modifies or deletes entries for which a HIC entry is assumed: The wiring gets additional links to retrieve, update and write back sequence numbers from the HIC for an overall sequence number (**os**) and a type sequence number (**ts**). (1) if the entry is taken from the container, modified and written back or written to another container, a copy of the entry as seen in the WIC is written to the history with **ts** and **os** set, and (2) if the entry is deleted, the delete is replaced by a **move** and it is subsequently moved into the HIC with **ts** and **os** set. All history entries of a wiring instance are assigned the same sequence numbers.  $l + 4$  additional links and 2 new variables are added to each wiring that modifies or deletes entries that must be kept in the history ( $l \dots \#$  of links that modify or delete entries relevant for the history).

Variables can be shared between sub-assertions. In that case, the wiring for the assertion that sets the variable, must get additional logic to report the values. The wirings for assertions that read the variables, must treat the variable reports and set their local wiring variables accordingly. For positive sub-assertions in the CNF, reports are sent if the corresponding assertion did not fail, for negated ones, they are sent if the assertion failed. Variables shared between contexts introduce an overhead of  $\sum_1^m (2o_i * k_i + k_{2j})$  additional links and at most  $\sum_1^m k_i \sum_1^{o_i} k_{2j}$  additional messages ( $m \dots \#$  sub-assertions,  $k_i \dots$  size of context setting a variable,  $k_{2j} \dots$  size of context reading a variable).

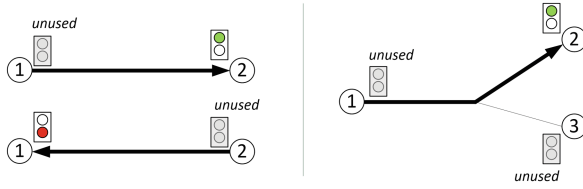
## 5 Proof-of-Concept: Railway Use Case

The selected use case refers to the reservation of rails (cf. [1, 23]). A *block* is either a track or a point (see Fig. 3). It has a unique identifier (**id**). A *track* has 2 connectors. A *point* has 3 connectors. A *connector* references a connector of another Block. A *signal* refers to a main signal; its value is red or green. It is valid at the exit of the Block and associated with a connector (dependent on the direction). A *route* is a sequence of connected Blocks. A *reserved route* is a route where all Blocks are exclusively reserved for this route.

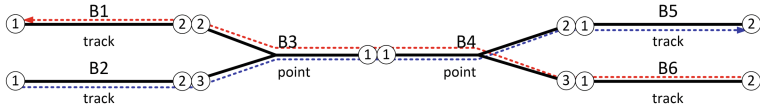
Three scenarios are selected: (1) Reserve Route: Mark all Blocks  $\text{Block}_1, \dots, \text{Block}_N$  for a route in the specified order. Check that the neighbor Blocks are physically connected via connectors, and set their direction appropriately. (2) Set Signals: Set signals on Blocks of a reserved route.

An example for a physical configuration of tracks is shown in Fig. 4. Each Block knows to which neighboring Blocks it is connected. The challenge is that there may exist many train Operators who try to reserve routes concurrently (see red and blue routes in Fig. 4). It must not happen that a safety constraint is violated (see informal specification of required assertions in Table 2). These assertions for the example use case were provided by a railway expert.

We assume that the Operator defines a **ttl** within which the route must be reserved. If not, the reservation fails, and the Operator issues the freeing of the



**Fig. 3.** Two tracks (left) and one point (right) with set direction and signals. (Color figure online)



**Fig. 4.** Sample configuration. Two routes: red and blue. (Color figure online)

already reserved Blocks of the route. The reservation of a route, the setting of signals of Blocks of the reserved route, and the freeing of the route belong to the same flow.

### 5.1 The Model

The Peer Model based representation is presented in the following using the graphical notation.

**Entries:** The entries used in the example are categorized into entries that represent the state of the distributed system, and entries that represent events which are sent between the peers as requests and answers. “bid” stands for Block id, and “cid” for connector id.

*State Entries:*

- config:
  - btyp: track or point
  - b1: bid of neighbor 1
  - c1: cid of neighbor 1
  - b2: bid of neighbor 2
  - c2: cid of neighbor 2
  - b3: bid of neighbor 3 (if point)
  - c3: cid of neighbor 3 (if point)
- status:
  - val: free or reserved
  - from: connector name
  - to: connector name
- signals: red or green
  - s1: signal at cid 1
  - s2: signal at cid 2
  - s3: signal at cid 3 (if point)

*Request/Answer Entries:*

- setDirection:
  - from: connector id
  - to: connector id
- setSignals:
  - s1: red or green
  - s2: red or green
  - s3: red or green
- markRoute:
  - route: sequence of Block ids
  - operator: peer id of operator
  - done: flag used by route marking
  - action: reserve or free
  - sender: peer id of sender (Operator or Block)
- ackRoute
- nackRoute

**Peers:** We define two kinds of peer roles: *Block* and *Operator*. Each track and point is modeled as a Block with unique block id. It accepts commands to change status, set signals and direction. The task of an Operator is to issue the setting of Block directions, the reservation and freeing of routes, and the setting of signals on reserved routes. All Operators act concurrently.

**Wirings:** Figures 5 and 6 show the wirings of Block peers.

*Init:* The initialization wiring (Fig. 5) is enabled exactly once when the Block peer is created. It creates entries to hold the local status (A1), signals (A2), and config (A3) and writes them to the peer’s PIC. Initially the status is free and does not belong to a flow, all signals are set to red, and the configuration of the Block is set based on the given peer’s configuration parameters.

*SetSignals:* This wiring (Fig. 5) sets the signals, provided that the Block is reserved for the current flow. It is triggered by receipt of a setSignals entry which it deletes from the PIC (G1). In (G2) it tests that its status says that it is reserved for this flow. In (G3) it takes the signals entry from the PIC and in (A1) writes the updated signals entry back to the PIC.

The marking of a route consists of the following wirings. Depending on the action of markRoute, the Blocks are either reserved or freed within a flow.

*ReserveRoute* (see Fig. 6) reacts on the receipt of a markRoute event in its PIC. We assume that a markRoute entry sent by an operator contains a valid route and has its ttl (time-to-live) property set. It checks that it has not been treated yet (i.e., the done flag is false), that the action is to reserve the route, remembers the sender in the variable \$s, the next Block of the route in the variable \$snext (G1), and initializes two variables \$f and \$t to empty. It takes its status entry (G2) from PIC, tests that in its configuration the val property equals free (G2). G3 tests the existence of one config entry in the PIC and remembers

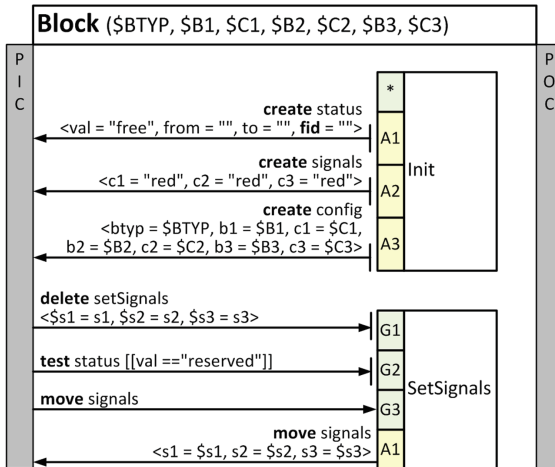


Fig. 5. Block Peer: Initialization, and setting of signals.

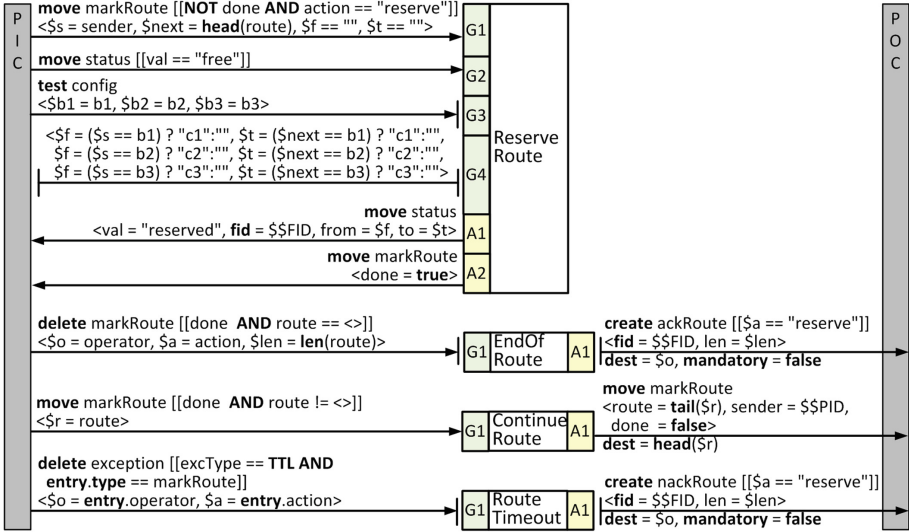


Fig. 6. Block Peer: Marking a route as reserved.

its properties in local variables termed  $\$b1$ ,  $\$b2$  and  $\$b3$ .  $G4$  initializes the local variables  $\$f$  and  $\$t$  to contain the connector id of the sender (if it was not the operator, i.e., sender is empty), and to the connector id of the next block. It updates the value of the status entry to be reserved, sets its flow id to the current flow, and the from and to properties to the respective connector ids (A1). Finally, it sets the done flag on the markRoute entry to **true** (A2) to denote that the status has been updated.

*FreeRoute* frees a reserved route. Its model is similar to *ReserveRoute*, with the only difference that it needs not check that the block was free before and updates the signals entry. It is therefore not shown.

*EndOfRoute* (see Fig. 6) fires after the Block's status has been set by *ReserveRoute* (i.e., done is **true**) and if the route has been completely treated, i.e., the route is empty (G1). It sends the reply entry *ackRoute* to the requesting Operator, provided that the action was to reserve the route. Otherwise no reply is sent (A1). Therefore the action link is not mandatory.

*ContinueRoute* (see Fig. 6) fires after the Block's status has been set by *ReserveRoute* and recursively processes the rest of a not yet empty route (G1). It sends the directive to mark the rest of the route to the next Block found in the route (A1), i.e., the head of the current route.

*RouteTimeout* (see Fig. 6) treats the **t1** exception of the markRoute entry (G1), and sends as reply *nackRoute* to the Operator (A1), if the action was to reserve the route, using the flow id of the markRoute entry, which is inherited by the exception entry.



**Table 2.** Railway use case assertions (informal description). N...assertion number, D...distributed assertion, P...refers to past

N	Assertion	D	P
1:	There must exist exactly one status entry at each Block	-	-
2:	There must exist exactly one config entry at each Block	-	-
3:	There must exist exactly one signals entry at each Block	-	-
4:	markRoute is received from “from” neighbor or from Operator	-	-
5:	The next Block of the route is the “to” Block neighbor	-	-
6:	A Block receiving setSignal is reserved within the same flow id	-	-
7:	If a Block is free then all its signals are red	-	-
8:	If a Block status is free then the flow id of its status must be null	-	-
9:	Reserved Blocks were free before	-	x
10:	Block freeing requires Operator to have received ack/nackRoute	-	x
11:	nackRoute is only sent after TTL of markRoute has expired	-	x
12:	If Operator got ackRoute, all Blocks of the route are reserved	x	-
13:	Signals are set after the Operator has received ackRoute for the flow	x	x

*CleanUp* (not shown) deletes outdated markRoute entries that want to free the Block.

## 5.2 Assertions

Table 2 describes the major assertions for the use case in an informal way. All of them were successfully modeled with the presented assertion mechanism, translated with MTA, and tested with the runnable specification of the Peer Model. The resulting Peer Model assertions are shown in Table 3. Of these, assertions 12 and 13 show the specification of the assertion property `excType`. For assertion 12 we specify that all Operators shall act as coordinators, and in assertion 13 all Blocks serve as coordinators.

### MTA Application

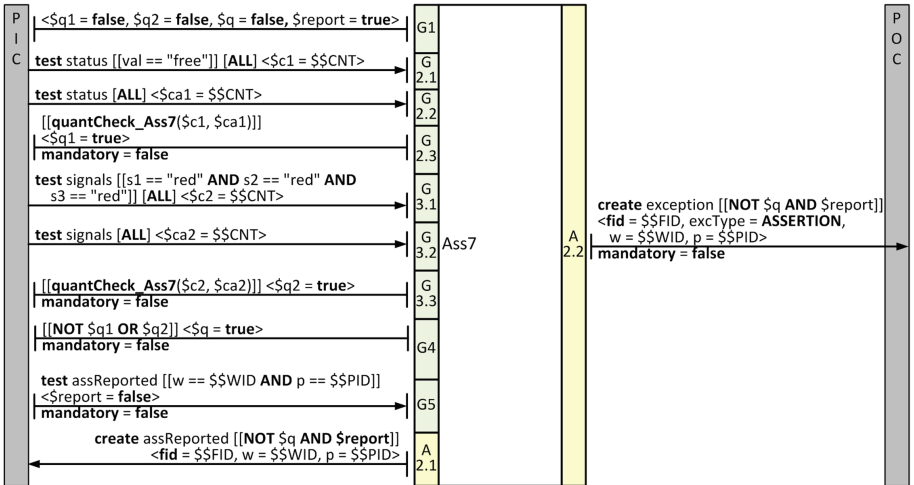
As an example for applying the developed model translation algorithms to the defined assertions, the resulting implementation of assertion 7 is shown in Fig. 7, which is an intra-peer one with two sub-assertions.

### Overhead Evaluation

We calculate the overhead by assuming one distributed railway application that has all shown example assertions. The introduced overhead is categorized by number of additional wirings and links. In average per assertion 1,08 more wirings and 11,15 links per Block, and 0,46 more wirings and 6,08 links per operator were needed as shown in Table 4.

**Table 3.** Railway use case assertions (formal specification).

1:	[ALL] ⟨Block⟩ { PIC { status } }
2:	[ALL] ⟨Block⟩ { PIC { config } }
3:	[ALL] ⟨Block⟩ { PIC { signals } }
4:	[ALL] ⟨Block⟩ { PIC { markRoute <\$s = sender> } → ( PIC { status <\$f = from> } AND PIC { config [label(\$f) == \$s] } ) }
5:	[ALL] ⟨Block⟩ { PIC { markRoute <\$next = head(route)> } → ( PIC { status <\$t = to> } AND PIC { config [label(\$t) == \$next] } ) }
6:	[ALL] ⟨Block⟩ { PIC { setSignal } → PIC { status [val == “reserved”] } }
7:	[ALL] ⟨Block⟩ { PIC { status [val == “free”] } → PIC { signals [s1 == “red” AND s2 == “red” AND s3 == “red”] } }
8:	[ALL] ⟨Block⟩ { PIC { status [val != “free” OR fid == “”] } }
9:	[ALL] ⟨Block⟩ { HIC { status [val == “reserved”] <\$ts = ts> } → HIC { status [ts == \$ts-1 AND val == “free”] } }
10:	[ALL] ⟨Block⟩ { PIC { markRoute [action == “free”] } } → [1] ⟨Operator⟩ { HIC { ackRoute } OR HIC { nackRoute } }
11:	[ALL] ⟨Operator⟩ { PIC { nackRoute } → HIC { exception [type == TTL AND entryType == markRoute] } }
12:	[ALL] \$O IN ⟨Operator⟩ { PIC { ackRoute } } → ( \$O { PIC { markRoute <\$r = route> } } AND [ALL] ⟨Block⟩ { [\$\$\$PID IN \$r] → PIC { status [val == “reserved”] } } ) [ coordinators = { [ALL] ⟨Operator⟩ } ]
13:	[ALL] ⟨Block⟩ { PIC { setSignal } } → [1] ⟨Operator⟩ { HIC { ackRoute } } [ coordinators = { [ALL] ⟨Block⟩ } ]


**Fig. 7.** MTA applied to assertion 7 (see Tables 2 and 3).

**Table 4.** Calculation of the overhead.

Assertion number	# wirings per Block	# wirings per Operator	# links per Block	# links per Operator
1	1	0	8	0
2	1	0	8	0
3	1	0	8	0
4	1	0	14	0
5	1	0	14	0
6	1	0	11	0
7	1	0	11	0
8	1	0	8	0
9	1	0	24	0
10	1	1	8	24
11	1	1	0	17
12	1	3	12	30
13	2	1	19	8
Total	14	6	145	79
Average	1,08	0,46	11,15	6,08
Median	1,00	0,00	11,00	0,00

## 6 Conclusion

In this paper we have presented an asynchronous runtime assertion mechanism for the Peer Model, which is an event-driven coordination model. Assertions are formulated in a declarative notation that relies on the query mechanism of the Peer Model. While their concept and syntax are new, they are closely related those of the link and do not pose an entirely new mechanism as the use of a separate tool or language would.

The proposed mechanism allows for lazy and strict assertion checking. In the former case, it might happen that certain errors are not detected due to race conditions. In the latter case, all errors can be detected during runtime, however, with the trade-off that a history of certain events must be kept.

Without history and shared variables, overhead is linear in the number of assertions to be checked and in the sizes of context they must be applied to. History imposes an overhead linear in the number of modifications to each entry type to be kept in history and shared variables increase the overhead quadratic in the size of the contexts and linear in the number of variables and sub-assertions.

In a first step, the application model is transformed so that the event history can be maintained, if required by assertions. In a second step, the assertions are implemented by mapping them to the concepts of the Peer Model, so that they become an integral part of the application model and can be understood as a coordination solution themselves.

In contrast to model checking, the proposed assertion mechanism is useful at both model definition and runtime. It makes no assumptions about the environment and can serve models of arbitrary size and distribution without encountering the state explosion problem.

The lazy mechanism is completely orthogonal to the application. The communication overhead depends on the distribution and number of the assertions. The strict mechanism causes additional computation in the wirings, however, only those events need to be captured in the history container that are referred to by (sub)assertions, that are newly obtained in a peer container, or updated or deleted by guards.

While the MTA is specific to the Peer Model, its concepts can be applied to assertion checking in coordination models or distributed systems in general. I.e., states are evaluated locally and evaluations compared instead of collecting global states, only evaluations and variable assignments that influence the overall evaluation of the assertion are shared and the mechanism is highly configurable.

As a proof-of-concept we have specified the major assertions for a use case from the railway domain and demonstrated that all could be captured.

## 7 Future Work

In future work, the MTA will be implemented and integrated into Peer Model implementations. Its correctness will be proven and its performance evaluated systematically. We will work on optimizations for overhead minimization produced by the model translation algorithm, investigate assertions that also refer to future events and dynamic changes of the meta model to support dynamic injection of assertions.

**Acknowledgements.** The authors would like to thank Anita Messinger for the discussions on train systems, and Stefan Craß for commenting this paper.

## References

1. Abrial, J.-R.: Train systems. In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) *Rigorous Development of Complex Fault-Tolerant Systems*. LNCS, vol. 4157, pp. 1–36. Springer, Heidelberg (2006). [https://doi.org/10.1007/11916246\\_1](https://doi.org/10.1007/11916246_1)
2. Agha, G.A.: *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge (1990)
3. Arbab, F.: Reo: a channel-based coordination model for component composition. *Math. Struct. Comput. Sci.* **14**(3), 329–366 (2004)
4. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* **20**(2), 14:1–14:64 (2011)
5. Chao, H., Li, H., Song, X., Wang, T., Li, X.: On evaluating and constraining assertions using conflicts in absent scenarios. In: *26th IEEE Asian Test Symposium, ATS 2017, Taipei City, Taiwan, 27–30 November 2017*, pp. 195–200 (2017)

6. Din, C.C., Owe, O., Bubel, R.: Runtime assertion checking and theorem proving for concurrent and distributed systems. In: *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, Lisbon, Portugal, 7–9 January 2014, pp. 480–487 (2014)
7. El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017*, pp. 125–135. ACM (2017)
8. Falcone, Y., Jaber, M., Nguyen, T.H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the bip framework with formally-proved sound and complete instrumentation. *Softw. Syst. Model.* **14**(1), 173–199 (2015)
9. Faruk Polat, R.A.: A multi-agent tuple-space based problem solving framework. *J. Syst. Softw.* **47**(5), 11–17 (1999)
10. Gelernter, D.: Generative communication in Linda. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **7**(1), 80–112 (1985)
11. Gelernter, D., Carriero, N.: Coordination languages and their significance. *Commun. ACM (CACM)* **35**(2), 96–107 (1992)
12. Kamburjan, E., Din, C.C., Chen, T.-C.: Session-based compositional analysis for actor-based languages using futures. In: Ogata, K., Lawford, M., Liu, S. (eds.) *ICFEM 2016*. LNCS, vol. 10009, pp. 296–312. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-47846-3\\_19](https://doi.org/10.1007/978-3-319-47846-3_19)
13. Kasuya, A., Tesfaye, T.: Verification methodologies in a TLM-to-RTL design flow. In: *DAC 2007*, pp. 199–204. ACM (2007)
14. Khosravi, R., Sirjani, M., Asoudeh, N., Sahebi, S., Iravanchi, H.: Modeling and analysis of Reo connectors using alloy. In: Lea, D., Zavattaro, G. (eds.) *COORDINATION 2008*. LNCS, vol. 5052, pp. 169–183. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-68265-3\\_11](https://doi.org/10.1007/978-3-540-68265-3_11)
15. Kühn, E.: Reusable coordination components: reliable development of cooperative information systems. *Int. J. Coop. Inf. Syst.* **25**(4), 1740001-1–1740001-32 (2016)
16. Kühn, E.: Flexible transactional coordination in the peer model. In: Dastani, M., Sirjani, M. (eds.) *FSEN 2017*. LNCS, vol. 10522, pp. 116–131. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-68972-2\\_8](https://doi.org/10.1007/978-3-319-68972-2_8)
17. 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 International Conference on Autonomous Agents and Multi-agent Systems (AAMAS)*, pp. 625–632. IFAAMAS (2009)
18. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: *2015 IEEE International Parallel and Distributed Processing Symposium*, pp. 494–503, May 2015
19. Newsham, Z., de Oliveira, A.B., Petkovich, J.C., Rehman, A.S.U., Tchamgoue, G.M., Fischmeister, S.: Intersert: assertions on distributed process interaction sessions. In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)* (2017)
20. Petri, C.A.: *Kommunikation mit Automaten*. Ph.D. thesis, Technische Hochschule Darmstadt (1962)
21. Pham, L.H., Thi, L.L.T., Sun, J.: Assertion generation through active learning. In: *39th IEEE International Conference on Software Engineering Companion*, pp. 155–157. IEEE/ACM (2017)
22. Radschek, S.T.: A usable formal methods tool chain for safety critical concurrent systems design. Master's thesis, TU Wien (2018, submitted)

23. Reichl, K., Fischer, T., Tummeltshammer, P.: Using formal methods for verification and validation in railway. In: Aichernig, B.K.K., Furia, C.A.A. (eds.) TAP 2016. LNCS, vol. 9762, pp. 3–13. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-41135-4\\_1](https://doi.org/10.1007/978-3-319-41135-4_1)
24. Scheller, T., Kühn, E.: Automated measurement of API usability: the API Concepts Framework. *Inf. Softw. Technol.* **61**, 145–162 (2015)
25. Tjang, A., Oliveira, F., Martin, R.P., Nguyen, T.D.: A: an assertion language for distributed systems. In: Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems, PLOS 2006. ACM (2006)
26. Wang, C., He, F., Song, X., Jiang, Y., Gu, M., Sun, J.: Assertion recommendation for formal program verification. In: 41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017, Turin, Italy, 4–8 July 2017, vol. 1, pp. 154–159 (2017)



# Active Objects for Coordinating BSP Computations (Short Paper)

Gaétan Hains<sup>1</sup>, Ludovic Henrio<sup>2</sup>, Pierre Leca<sup>1,2</sup>(✉), and Wijnand Suijlen<sup>1</sup>

<sup>1</sup> Huawei Technologies, Paris Research Center, Paris, France  
[pierre.leca@huawei.com](mailto:pierre.leca@huawei.com)

<sup>2</sup> Université Côte d'Azur, CNRS, I3S, Sophia-Antipolis, France

**Abstract.** Among the programming models for parallel and distributed computing, one can identify two important families. The programming models adapted to data-parallelism, where a set of coordinated processes perform a computation by splitting the input data; and coordination languages able to express complex coordination patterns and rich interactions between processing entities. This article takes two successful programming models belonging to the two categories and puts them together into an effective programming model. More precisely, we investigate the use of active objects to coordinate BSP processes. We choose two paradigms that both enforce the absence of data-races, one of the major sources of error in parallel programming. This article explains why we believe such a model is interesting and provides a formal semantics integrating the notions of the two programming paradigms in a coherent and effective manner.

**Keywords:** Parallelism · Programming models · Active objects · BSP

## 1 Introduction

This article presents our investigations on programming paradigms mixing efficient data parallelism, and rich coordination patterns. We propose a programming methodology that mixes a well-structured data-parallel programming model, BSP [17] (Bulk Synchronous Parallel), and actor-based high-level interactions between asynchronous entities. This way, we are able to express in a single programming model several tightly coupled data-parallel algorithms interacting asynchronously. More precisely we design an active-object language where each active object can run BSP code, the communication between active objects is remote method invocation, while it is delayed memory read/write inside the BSP code. These two programming models were chosen because of their properties: BSP features predictable performance and absence of deadlocks under simple hypotheses. Active objects have only a few sources of non-determinism and provide high-level asynchronous interactions. Both models ensure *absence of data races*, thus our global model features this valuable property. The benefits

we expect from our mixed model are the enrichment of efficient data-parallel BSP with both service-like interactions featured by active objects, and elasticity. Indeed, scaling a running BSP program is not often possible or safe, while adding new active objects participating to a computation is easy.

This short paper presents the motivation for this work and an analysis of related programming models in Sect. 2. Then a motivating example is shown in Sect. 3. The main contribution of this paper is the definition of a core language for our programming model, presented in Sect. 4. An implementation of the language as a C++ library is under development.

## 2 Context and Motivation

### 2.1 Active Objects and Actors

The actor model [1] is a parallel execution model focused on task parallelism. Actor-based applications are made of independent entities, each equipped with a different process or thread, interacting with each other through asynchronous messages passing. Active objects hide the concept of message from the language: they call each other through typed method invocations. A call is asynchronous and returns directly giving a Future [11] as a placeholder for its result. Since there is only one thread per active object, requests cannot run in parallel. The programmer is thus spared from handling mutual exclusion mechanisms to safely access data. This programming model is adapted to the development of independent components or services, but is not always efficient when it comes to data-parallelism and data transmission. An overview of active object languages is provided by [5], focusing on languages which have a stable implementation and a formal semantics. ASP [7] is an active object language that was implemented as the ProActive Java library [3]. Deterministic properties were proven in ASP when no two requests can arrive in a different order on the same active object.

Several extensions to the active object model enable controlled parallelism inside active objects [2, 8, 12]. Multi-active objects is an extension of ASP [12] where the programmer can declare that several requests can run in parallel on the same object. This solution relies on the correctness of program annotation to prevent data-races but provides efficient data-parallelism inside active objects. Parallel combinators of Encore [8] also enable some form of restricted parallelism inside an active object, mostly dedicated to the coordination of parallel tasks. A set of parallel operators is proposed to the programmer, but different messages still have to be handled by different active objects. This restricted parallelism does not provide local data parallelism.

### 2.2 Bulk Synchronous Parallel

BSP is another parallel execution model, it defines algorithms as a sequence of supersteps, each made of three phases: computation, communication, and synchronization. BSP is adapted to the programming of data-parallel applications,



but is limited in terms of application elasticity or loose coupling of computing components as it relies on the strong synchronization of all computing entities.

Interactions between BSP processes occur through communication primitives sending messages or performing one-sided Direct Remote Memory Access (DRMA) operations, reading or writing the memory of other processes without their explicit consent. BSP is generally used in an SPMD (Single Program Multiple Data) way, it is suitable for data-parallelism as processes may identically work on different parts of a data. BSPLib is a C programming library for writing BSP programs [13], it features message passing and DRMA. Variants of BSPLib exist such as the Paderborn University BSP (PUB) library [6], or BSPonMPI [14]. The PUB library offers subset synchronization, but this feature is argued against by [10] in the context of a single BSP data-parallel algorithm. Using subset synchronization to coordinate different BSP algorithms in the same system seems possible but even more error prone. Formal semantics were defined for BSPLib DRMA [15] and the PUB library [9].

### 2.3 Motivation and Objectives

The SPMD programming model in general and BSP are well adapted for the implementation of specific algorithms, but composing different such algorithms in a single application requires coordination capabilities that are not naturally provided by the SPMD approach. Such coordination is especially difficult to implement in BSPLib because a program starts with all processes in a single synchronization group. For any communication to occur, all processes of the application need to participate in the same synchronization barrier, making it difficult to split a program into parallel tasks with different synchronization patterns. The PUB library can split communication groups to synchronize only some of the processes, but still lacks high level libraries for coordinating the different groups. On the other hand, asynchronous message sending of active objects is appropriate for running independent tasks, but inefficient when there are many exchanged messages inside a given group of process or following a particular communication pattern.

In this paper, we use active objects for wrapping BSP tasks, allowing us to run different BSP algorithms in parallel without requiring them to participate in the same synchronization. Active objects provide coordination capabilities for loosely coupled entities and can be used to integrate BSP algorithms into a global application. To our knowledge, this is the first model using active objects to coordinate BSP tasks.

Among related works, programming languages based on stream processing, like the StreamIt [16] language, feature data parallelism. While splitting a program into independent tasks could be considered similar to our approach, stream processing languages do not feature the strong synchronization model of BSP. They are also less convenient for service-like interaction between entities, particularly when those sending queries are not determined statically. In summary, ABSP features an interesting mix between locally constrained parallelism using BSP (with fixed number of processes and predefined synchronization pattern),

```

1  class IPActor : public activebsp::ActorBase {
2  public:
3      double ip(vector<double> v1, vector<double> v2) {
4          // ...
5          for (int i = 0; i < bsp_nprocs(); ++i) { // Split data
6              int beg = (v1.size()/bsp_nprocs()) * i,
7                  end = (v1.size()/bsp_nprocs()) * (i+1);
8              bsp_put(i, &v1[beg], -x-part, 0, (end-beg)*sizeof(double));
9              bsp_put(i, &v2[beg], -y-part, 0, (end-beg)*sizeof(double));
10         }
11
12         bsp_run(&IPActor::bspinprod);
13
14         return -alpha;
15     }
16
17     void bspinprod() {
18         // call BSPedupack inner product, returns result on all p
19         -alpha = bspip(bsp_nprocs(), bsp_pid(), -n-part, -x-part, -y-part);
20     }
21 };
22
23 int main() {
24     // ...
25     vector<double> v;
26     // ...
27     Proxy<IPActor> actorA = createActor<IPActor> ({1,2});
28     Proxy<MultActor> actorB = createActor<MultActor>({3,4});
29
30     Future<double> f1 = actorA.ip(v,v);
31     double ip = f1.get();
32     Future<vector<double>> f2 = actorB.multiply_all(v, ip);
33     v = f2.get();
34     // ...
35 }

```

**Fig. 1.** A ABSP example

and flexible service oriented interactions featured by active objects (more flexible but still with some reasonable guarantees). This makes our approach quite different from StreamIt and other similar languages.

### 3 Example

In this section we show an example written using our C++ library under development (Fig. 1). This library uses MPI for actor communications and reuses the BSPonMPI library for BSP communications.

We chose C++ because we put higher priority on the efficiency of BSP data-parallel code, C++ allows us to re-use BSP implementations written in C while allowing objects and more transparent serialization. An implementation mixing incompatible languages would, at this point, yield unnecessary complexity in our opinion. We chose a motivating example based on this implementation instead of the formal language of the next section to show the re-use of existing code and because we think it is more convincing.

This example shows how an active object can encapsulate process data and how its function interface can act as a parameterized sequential entry point to a parallel computation. We also show the result of a call being used to call another active object to do another computation, which is not shown.

The IPActor class interfaces the inner product implementation included in the BSPedupack software package [4]. We only show parts of the code we deem interesting to present our model. Object variables begin by ‘\_’, their declarations are not shown. We assume `bsp_nprocs()` divides `v1.size()`.

In the *main* function, the MPI process of pid 0 creates two active objects with two processes each, see the parameter of the *createActor* primitive. Then the *ip* function of the first one is called with vector *v* as the two parameters. This asynchronous call returns with a future *f1*. The *ip* function of this active object is then called sequentially. Using BSP primitives, the input vectors are split among the processes of the active object. Then a *bsp\_run* primitive is used to run *bspinprod* in parallel. It calls the *bspip* function of BSPedupack. Immediately after the call on the first object, the main method requests the result with a *get* primitive on *f1*, blocking until the result is ready. This result is sent to another active object as request parameter.

## 4 A Core Language for Coordinating BSP Computations

### 4.1 Syntax

ABSP is our core language for expressing the semantics of BSP processes encapsulated inside active objects. Its syntax is shown in Fig. 2, *x* ranges over variable names, *m* over method names,  $\alpha, \beta$  over actor names, *f* over future names, and *i, j, k, N* over integers that are used as process identifiers or number of processes. A program *P* is made of a main method and a set of object classes with name *Act*, each having a set of fields and a set of methods. The main method identifies the starting point of the program. Each method *M* has a return type, a name, a set of parameters *x*, and a body. The body is made of a set of local variables and a statement. Types *T* and terms are standard for object languages, except that **new** creates an active object, **get** accesses a future, and *v.m*( $\bar{v}$ ) performs an asynchronous method invocation on an active object and creates a future. The operators for dealing with BSP computations are: **BSPrun**(*m*) that triggers the parallel execution of several instances of the method *m*; **sync** delimits BSP supersteps; and **bsp\_put** writes data on a parallel instance, to prevent data-races the effect of **bsp\_put** is delayed to the next **sync**. **bsp\_get** is the reverse of **bsp\_put**, reading remote data instead of writing it. Sequence is denoted; and is associative with a neutral element **skip**. Each statement can be written as *s*; *s'* with *s* neither **skip** nor a sequence.

**Design Choices.** We chose to specify a FIFO request service policy like in ASP because it exists in several implementations and makes programming easier. In ABSP, all objects are active, a richer model using passive objects or concurrent object groups [5] would be more complex. We choose a simple semantics for futures: futures are explicit and typed by a parametric type with a simple **get** operator. We chose to model DRMA-style communications although message passing also exists in BSPlib; modelling messages between processes hosted on the same active object would raise no additional difficulty.

$P ::= \overline{\text{Act}\{\overline{T\ x\ M}\}} \{\overline{T\ x\ s}\}$	program
$T ::= \text{Int} \mid \text{Bool} \mid \text{Act} \mid \text{Fut} < T >$	type
$M ::= T\ m(\overline{T\ x}) \{\overline{T\ x\ s}\}$	method
$s ::= \text{skip} \mid x = z \mid \text{if } v \{s\} \text{ else } \{s\} \mid s ; s$	statements
$\mid \text{return } v \mid \text{BSPrun}(m) \mid \text{sync} \mid \text{bsp\_put}(v, v, x)$	
$z ::= e \mid v.m(\overline{v}) \mid \text{new Act}(N, \overline{v}) \mid \text{get } v$	rhs of assign
$e ::= v \mid v \oplus v$	expressions
$v ::= x \mid \text{null} \mid \text{integer-values}$	atoms

Fig. 2. Static syntax of ABSP

$cn ::= \overline{\alpha(N, A, p, \overline{q}, \overline{Upd})} \overline{f(\perp)} \overline{f(w)}$	configuration
$p ::= \emptyset \mid q : ([\overline{i} \mapsto \overline{Task}] ; j \mapsto \overline{Task})$	current request service
$q ::= (f, m, \overline{w})$	request id
$\text{Task} ::= \{\ell   s\}$	task
$w ::= x \mid \alpha \mid f \mid \text{null} \mid \text{integer-values}$	runtime values
$\ell, a ::= [\overline{x} \mapsto \overline{w}]$	local store
$A ::= [\overline{i} \mapsto \overline{a}]$	object fields
$e ::= w \mid v \oplus v$	runtime expressions
$\text{Upd} ::= (\overline{i_{src}}, v, \overline{i_{dst}}, \overline{x})$	DRMA operations

Fig. 3. Runtime Syntax of ABSP (terms identical to the static syntax omitted).

## 4.2 Semantics

The semantics of ABSP is expressed as a small-step operational semantics (Fig. 4); it relies on the definition of *runtime configurations* which represent states reached during the intermediate steps of the execution. The syntax of configurations and runtime terms is defined in Fig. 3. Statements and expressions are the same as in the static syntax except that they can contain runtime values.

A runtime configuration is an unordered set of active objects and futures where futures can either be unresolved or have a future value associated. An active object has a name  $\alpha$ , a number  $N$  of processes involved in  $\alpha$ , these processes are numbered  $[0..N-1]$ ,  $A$  associates each pid  $i$  to a set of field-value pairs  $a$ . It has the form  $(0 \mapsto [x \mapsto \text{true}, y \mapsto 1], 1 \mapsto [x \mapsto \text{true}, y \mapsto 3])$  for example meaning that the object at pid 0 has two fields  $x$  and  $y$  with value **true** and 1, and the object at pid 1 has the same fields with different values. Note that the object has the same fields in every pid. The function  $A(i)$  allows us to select the element  $a$  at position  $i$ .  $\overline{q}$  is the request queue of the active object. The active object might be running at most one request at a time. If it is not running a request, then  $p = \emptyset$ . Otherwise  $p = q : ([\overline{i} \mapsto \overline{Task}] ; j \mapsto \overline{Task})$  where  $q$  is the identity of the request being served, and  $([\overline{i} \mapsto \overline{Task}] ; j \mapsto \overline{Task})$  is a two level mapping of processes to tasks that have to be performed to serve the request. The first level represents parallel execution, it maps process identifiers to tasks, the second represents sequential execution and contains a single process identifier and task.

$$\begin{array}{c}
\frac{w \text{ is not a variable}}{\llbracket w \rrbracket_\ell = w} \quad \frac{x \in \text{dom}(\ell)}{\llbracket x \rrbracket_\ell = \ell(x)} \quad \frac{\llbracket v \rrbracket_\ell = k \quad \llbracket v' \rrbracket_\ell = k'}{\llbracket v \oplus v' \rrbracket_\ell = k \oplus k'} \\
\\
\text{NEW} \\
\frac{\llbracket \bar{v} \rrbracket_{a+\ell} = \bar{w} \quad \bar{y} = \text{fields}(\text{Act})}{A = [j \mapsto (\bar{y} \mapsto \bar{w}, \text{pid} \mapsto j, \text{nprocs} \mapsto N) \mid j \in [0..N-1]] \quad \beta \text{ fresh}} \\
\mathcal{R}_k[a, \ell, x = \text{new Act}(N, \bar{v}) ; s] \rightarrow \mathcal{R}_k[a, \ell, x = \beta ; s] \beta(N, A, \emptyset, \emptyset, \emptyset) \\
\\
\text{IF-TRUE} \quad \frac{\llbracket v \rrbracket_{a+\ell} = \text{true}}{\mathcal{R}_k[a, \ell, \text{if } v \{s_1\} \text{ else } \{s_2\} ; s] \rightarrow \mathcal{R}_k[a, \ell, s_1 ; s]} \quad \text{IF-FALSE} \quad \frac{\llbracket v \rrbracket_{a+\ell} \neq \text{true}}{\mathcal{R}_k[a, \ell, \text{if } v \{s_1\} \text{ else } \{s_2\} ; s] \rightarrow \mathcal{R}_k[a, \ell, s_2 ; s]} \\
\\
\text{GET} \quad \frac{\llbracket v \rrbracket_{a+\ell} = f}{\mathcal{R}_k[a, \ell, y = \text{get } v ; s] f(w) \rightarrow \mathcal{R}_k[a, \ell, y = w ; s] f(w)} \quad \text{ASSIGN} \quad \frac{\llbracket e \rrbracket_{a+\ell} = w \quad (a+\ell)[x \mapsto w] = a' + \ell'}{\mathcal{R}_k[a, \ell, x = e ; s] \rightarrow \mathcal{R}_k[a', \ell', s]} \\
\\
\text{INVK} \\
\frac{\llbracket v \rrbracket_{a+\ell} = \beta \quad \llbracket \bar{v} \rrbracket_{a+\ell} = \bar{w} \quad f \text{ fresh}}{\mathcal{R}_k[a, \ell, x = v.\mathbf{m}(\bar{v}) ; s] \beta(N, A, p, \bar{q}, \text{Upd}) \rightarrow \mathcal{R}_k[a, \ell, x = f ; s] \beta(N, A, p, \bar{q} :: (f, m, \bar{w}), \text{Upd}) f(\perp)} \\
\\
\text{SERVE} \\
\frac{\text{bind}(\alpha, m, \bar{v}) = \{\ell | s\} \quad i = \text{head}(N)}{\alpha(N, A, \emptyset, (f, m, \bar{v}) :: \bar{q}', \text{Upd}) \text{ cn} \rightarrow \alpha(N, A, (f, m, \bar{v}) : (\emptyset ; i \mapsto \{\ell | s\}), \bar{q}', \text{Upd}) \text{ cn}} \\
\\
\text{BSPRUN} \\
\frac{\text{bind}(\alpha, m, \emptyset) = \{\ell' | s'\}}{\alpha(N, A, q : (\emptyset ; i \mapsto \{\ell | \text{BSPrun}(m) ; s\}), \bar{q}', \text{Upd}) \text{ cn} \rightarrow \alpha(N, A, q : (\{k \mapsto \{\ell' | s'\} | k \in [0..N-1]\} ; i \mapsto \{\ell | s\}), \bar{q}', \text{Upd}) \text{ cn}} \\
\\
\text{RETURN-VALUE} \\
\frac{\llbracket v \rrbracket_{A(i)+\ell} = w}{\alpha(N, A, (f, m, \bar{w}) : (\emptyset ; i \mapsto \{\ell | \text{return } v ; s\}), \bar{q}, \text{Upd}) f(\perp) \text{ cn} \rightarrow \alpha(N, A, \emptyset, \bar{q}, \text{Upd}) f(w) \text{ cn}} \\
\\
\text{RETURN-SUB-TASK} \\
\frac{\alpha(N, A, q : (\bar{i} \mapsto \overline{\text{Task}}) \uplus \{k \mapsto \{\ell | \text{return } v ; s\} ; j \mapsto \text{Task}'\}, \bar{q}, \text{Upd}) \text{ cn}}{\rightarrow \alpha(N, A, q : (\bar{i} \mapsto \text{Task} ; j \mapsto \text{Task}'), \bar{q}, \text{Upd}) \text{ cn}} \\
\\
\text{SYNC} \\
\frac{A' = [j \mapsto A(j) [(y \mapsto \llbracket v \rrbracket_{A(i)}) \mid (i, v, j, y) \in \text{Upd}] \mid j \in I]}{\alpha(N, A, q : (\bar{k} \mapsto \{\ell_k | \text{sync} ; s_k\} ; i \mapsto \text{Task}), \bar{q}', \text{Upd}) \text{ cn} \rightarrow \alpha(N, A', q : (\bar{k} \mapsto \{\ell_k | s_k\} ; i \mapsto \text{Task}), \bar{q}', \emptyset) \text{ cn}} \\
\\
\text{BSP-GET} \quad \frac{\llbracket v \rrbracket_{a+\ell} = i}{\mathcal{D}_k[a, \ell, \text{bsp.get}(v, x_{\text{src}}, x_{\text{dst}}) ; s, \text{Upd}] \rightarrow \mathcal{D}_k[a, \ell, s, \text{Upd} \cup (i, x_{\text{src}}, k, x_{\text{dst}})]} \quad \text{BSP-PUT} \quad \frac{\llbracket v \rrbracket_{a+\ell} = i \quad \llbracket v_{\text{src}} \rrbracket_{a+\ell} = v'}{\mathcal{D}_k[a, \ell, \text{bsp.put}(v, v_{\text{src}}, x_{\text{dst}}) ; s, \text{Upd}] \rightarrow \mathcal{D}_k[a, \ell, s, \text{Upd} \cup (k, v', i, x_{\text{dst}})]}
\end{array}$$

Fig. 4. Semantics of ABSP.

Tasks in each process consist of a local environment  $\ell$  and a current statement  $s$ . For example,  $p = q : ([k \mapsto \{\ell_k | s_k\} | k \in [0..N - 1]] ; i \mapsto \{\ell | s\})$  means that the current request  $q$  first requires the parallel execution on all processes of  $[0..N - 1]$  of their statements  $s_k$  in environments  $\ell_k$ ; then the process  $i$  will recover the execution and run the statement  $s$  in environment  $\ell$ . Concerning future elements, these have two possible forms:  $f(\perp)$  for a future being computed, or  $f(w)$  for a resolved future with the computed result  $w$ .

We adopt a notation inspired from reduction contexts to express concisely a point of reduction in an ABSP configuration. A global reduction context  $\mathcal{R}_k[a, \ell, s]$  is a configuration with four holes: a process number  $k$ , a set  $a$  of fields, a local store  $\ell$ , and a statement  $s$ . It represents a valid configuration where the statement  $s$  is at a reducible place, and the other elements can be used to evaluate the statement. This reduction context uses another reduction context focusing on a single request service and picking the reducible statement inside the current tasks. This second reduction context  $\mathcal{C}_k[\ell, s]$  will allow us to conveniently define rules evaluating the current statement in any of the two execution levels, it provides a single entry for two possible options: the sequential level and the parallel one. It also defines that the parallel level is picked first instead of the sequential one if it is not empty. The two reduction contexts are defined as follows:

$$\begin{aligned} \mathcal{R}_k[a, \ell, s] &::= \alpha(N, A \uplus [k \mapsto a], q : \mathcal{C}_k[\ell, s], \bar{q}, Upd) \text{ cn} \\ \mathcal{C}_k[\ell, s] &::= (\emptyset ; k \mapsto \{\ell | s\}) \quad | \quad (\bar{i} \mapsto \overline{Task}) \uplus [k \mapsto \{\ell | s\}]; j \mapsto Task \end{aligned}$$

Taking the assignment as example, it applies in two kinds of configurations:  $\alpha(N, A \uplus [k \mapsto a], q : (\bar{i} \mapsto \overline{Task}) \uplus [k \mapsto \{\ell | x = e ; s\}]; j \mapsto Task), \bar{q}, Upd) \text{ cn}$  and  $\alpha(N, A \uplus [k \mapsto a], q : (\emptyset ; k \mapsto \{\ell | x = e ; s\}), \bar{q}, Upd) \text{ cn}$ . Using contexts both greatly simplifies the notation and spares us from having to duplicate rules.

To help defining DRMA operations, we will also use  $\mathcal{D}_k[a, \ell, s, Upd]$ , which is an extension of  $\mathcal{R}_k[a, \ell, s]$  exposing the  $Upd$  field. It is defined as:

$$\mathcal{D}_k[a, \ell, s, Upd] ::= \alpha(N, A \uplus [k \mapsto a], q : \mathcal{C}_k[\ell, s], \bar{q}, Upd) \text{ cn}$$

We use the notation  $[\bar{i} \mapsto \overline{Task}] \uplus [k \mapsto \{\ell | s\}]$  to access and modify the local store and current statement of a process  $k$ . Just as a statement can be decomposed into a sequence  $s; s'$  with the associative property, the task mapping can be decomposed into  $[\bar{i} \mapsto \overline{Task}] \uplus [k \mapsto Task]$ , we use the disjoint union  $\uplus$  to work on a single process disjoint from the rest.

The first three rules of the semantics define an evaluation operator  $\llbracket e \rrbracket_\ell$  that evaluates an expression  $e$  using a variable environment  $l$ . We rely on  $\text{dom}(l)$  to retrieve the set of variables declared in  $l$ . While these rules involve a single variable environment, we often use the notation  $a + l$  to involve multiple variable environments, e.g.  $\llbracket e \rrbracket_{a+l}$ . It is important to note that  $\llbracket e \rrbracket_{a+l} = w$  implies that  $w$  is not a variable, it can only be an object or future name, `null`, or an integer value.

**New** creates a new active objects on  $N$  processes with parameters  $v$ , used to initialize object fields. We use  $\text{fields}(Act)$  to retrieve names and rely on the

declaration ordering to assign values to the right variables. We also add a unique process identifier and  $N$ , respectively as *pid* and *nprocs*. The new active object  $\beta$  is then initialized with  $N$  processes and the resulting object environment  $A$ .

**Assign** is used to change the value of a variable. The expression  $e$  is evaluated using the evaluation operator, producing value  $w$ . Since variable  $x$  can either be updated in the object or local variable environments, we use the notation  $(a + \ell)[x \mapsto w] = a' + \ell'$  to update either of these and retrieve both updated environments  $a'$  and  $\ell'$ . They replace old ones in the object configuration.

**If-True and If-False** reduces an **if** statement to  $s1$  or  $s2$  according to the evaluation of the boolean expression  $v$ .

**Get** retrieves the value associated with a future  $f$ . The future must be resolved. If the future has been resolved with value  $w$ , the **get** statement is replaced by  $w$ .

**Invk** invokes an existing active object and creates a future associated to the result. This rule requires  $v$  to be evaluated into an active object, then enqueues a new request in this object. A new unresolved future  $f$  is added to the configuration. Allowing self-invocation would require a simple adaptation of this rule. Parameters  $\bar{v}$  that are passed to the method are evaluated locally, into  $\bar{w}$ . The request queue of the active object  $\beta$  is then appended with a triplet containing a new future identifier  $f$  associated to the request, the method  $m$  to call and the parameters  $\bar{w}$ .

**Serve** processes a queued request. To prevent concurrent execution of different requests, the active object is required to be idle (with the current request field empty). A request  $(f, m, \bar{v})$  is dequeued to build and execute a new sequential environment  $i \mapsto \{l|s\}$ ; it relies on **bind** to build this environment. The process  $i$  responsible for the sequential environment is called the *head*, it is the master process responsible for serving requests.

**BSPRun** starts a new parallel environment from the current active object  $\alpha$  and the method  $m$ . Every process of the active object is going to be responsible for executing one instance of the same task  $\{l'|s'\}$ . All parallel processes start with the same local variable environment and the same statement to execute.

**Return-Value** resolves a future. The expression  $v$  is first evaluated into a value  $w$  that is associated with the future  $f$ . The current request field is emptied, allowing a new request to be processed.

**Return-Sub-Task** terminates one parallel task. The process that returned is removed from the set of tasks running in parallel. When the last process is removed, the sequential context can be evaluated.

**Sync** ends the current superstep, the **sync** statement must be reached on every pid  $k$  of the parallel execution context before this rule can be reduced. DRMA operations that were requested since the last superstep and stored in the *Upd* field as  $(i, v, j, y)$  quadruplets are taken into account. They are used to update the object variable environment  $A$  into  $A'$  such that variable  $y$  of pid  $j$  is going

to take the value  $v$  as evaluated in process  $i$ , for every such quadruplet. As  $Upd$  is an unordered set, these updates are performed in any order.

**BSP-Get** requests to update a local variable with the value of a remote one. We write a DRMA quadruplet such that the variable  $x_{src}$  of the remote process  $i$  is going to be read into the variable  $x_{dst}$  of the current pid  $k$  during the next synchronisation step.

**BSP-Put** requests to write a local value into a remote variable. The value to be written is evaluated into  $v$ , and a new update quadruplet is created in  $Upd$ . It will be taken into account upon the next `sync`.

While race conditions exist in ABSP, like in active object languages and in BSP with DRMA, the language has no data race. Indeed, the only race conditions are message sending between active objects, and parallel emission of update requests. The first one results in a non-deterministic ordering in a request queue, and the second in parallel accumulation of update orders in an unordered set. Updates are performed in any order upon synchronisation but additional ordering could be enforced, e.g. based on the time-stamp of the update.

## 5 Current Status and Objectives

We presented a new programming model for the coordination of BSP processes. It consists of an actor-like interaction pattern between SPMD processes. Each actor is able to run an SPMD algorithm expressed in BSP. The active-object programming model allowed us to integrate these notions together by using object and methods as entry points for asynchronous requests and for data-parallel algorithms. We have shown an example of this model that features two different BSP tasks coordinated through dedicated active objects. This example also shows the usage of an experimental C++ library implementing this model that relies on MPI for flexible actor communications and a BSPlib-like implementation for intra-actor data-parallel computations.

The semantics proposed in this paper will allow us to prove properties of the programming model. Already, by nature both active objects and BSP ensure the absence of data-races and thus our programming model inherits this crucial property. To further investigate race-conditions, we should formally identify the sources of non-determinism in ABSP and show that only concurrent request sending to the same AO and DRMA create non-determinism. Another direction of research could focus on the verification of `sync` statements, checking they can only be invoked in a parallel context.

## References

1. Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge (1986)
2. Azadbakht, K., de Boer, F.S., Serbanescu, V.: Multi-threaded actors. In: *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8–9 June 2016*, pp. 51–66 (2016)



3. Baduel, L., et al.: Programming, composing, deploying for the grid. In: Cunha, J.C., Rana, O.F. (eds.) *Grid Computing: Software Environments and Tools*, pp. 205–229. Springer, London (2006). [https://doi.org/10.1007/1-84628-339-6\\_9](https://doi.org/10.1007/1-84628-339-6_9)
4. Bisseling, R.: *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. OUP, Oxford (2004)
5. Boer, F.D., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. *ACM Comput. Surv.* **50**, 76:1–76:39 (2017)
6. Bonorden, O., Juurlink, B., von Otte, L., Rieping, I.: The paderborn university BSP (PUB) library. *Parallel Comput.* **29**(2), 187–207 (2003)
7. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004*, pp. 123–134. ACM, New York (2004)
8. Fernandez-Reyes, K., Clarke, D., McCain, D.S.: ParT: an asynchronous parallel abstraction for speculative pipeline computations. In: Lluch Lafuente, A., Proença, J. (eds.) *COORDINATION 2016*. LNCS, vol. 9686, pp. 101–120. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-39519-7\\_7](https://doi.org/10.1007/978-3-319-39519-7_7)
9. Gava, F., Fortin, J.: Formal semantics of a subset of the paderborn’s BSPLib. In: *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 269–276 (2008)
10. Hains, G.: Subset synchronization in BSP computing. In: *PDPTA*, vol. 98, pp. 242–246 (1998)
11. Halstead Jr., R.H.: Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* **7**(4), 501–538 (1985)
12. Henrio, L., Huet, F., István, Z.: Multi-threaded active objects. In: De Nicola, R., Julien, C. (eds.) *COORDINATION 2013*. LNCS, vol. 7890, pp. 90–104. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-38493-6\\_7](https://doi.org/10.1007/978-3-642-38493-6_7)
13. Hill, J.M.D., McColl, B., Stefanescu, D.C., Goudreau, M.W., Lang, K., Rao, S.B., Suel, T., Tsantilas, T., Bisseling, R.H.: BSPLib: the BSP programming library. *Parallel Comput.* **24**, 1947–1980 (1998)
14. Suijlen, W.J., BISSELING, R.: BSPonMPI (2013). <http://bsponmpi.sourceforge.net>
15. Tesson, J., Loulergue, F.: Formal semantics of DRMA-style programming in BSPLib. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) *PPAM 2007*. LNCS, vol. 4967, pp. 1122–1129. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-68111-3\\_119](https://doi.org/10.1007/978-3-540-68111-3_119)
16. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: a language for streaming applications. In: Horspool, R.N. (ed.) *CC 2002*. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-45937-5\\_14](https://doi.org/10.1007/3-540-45937-5_14)
17. Valiant, L.G.: A bridging model for parallel computation. *CACM* **33**(8), 103 (1990)



# Boosting Transactional Memory with Stricter Serializability

Pierre Sutra<sup>2</sup>(✉), Patrick Marlier<sup>1</sup>, Valerio Schiavoni<sup>1</sup>(✉) ,  
and François Trahay<sup>2</sup> 

<sup>1</sup> University of Neuchâtel, Neuchâtel, Switzerland  
{patrick.marlier, valerio.schiavoni}@unine.ch

<sup>2</sup> Télécom SudParis, Évry, France  
{pierre.sutra, francois.trahay}@telecom-sudparis.eu

**Abstract.** Transactional memory (TM) guarantees that a sequence of operations encapsulated into a transaction is atomic. This simple yet powerful paradigm is a promising direction for writing concurrent applications. Recent TM designs employ a time-based mechanism to leverage the performance advantage of invisible reads. With the advent of many-core architectures and non-uniform memory (NUMA) architectures, this technique is however hitting the synchronization wall of the cache coherency protocol. To address this limitation, we propose a novel and flexible approach based on a new consistency criteria named stricter serializability (SSER<sup>+</sup>). Workloads executed under SSER<sup>+</sup> are opaque when the object graph forms a tree and transactions traverse it top-down. We present a matching algorithm that supports invisible reads, lazy snapshots, and that can trade synchronization for more parallelism. Several empirical results against a well-established TM design demonstrate the benefits of our solution.

**Keywords:** Transactional memory · NUMA · Stricter serializability

## 1 Introduction

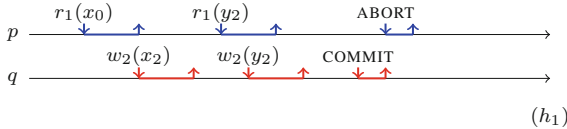
The advent of chip level multiprocessing in commodity hardware has pushed applications to be more and more parallel in order to leverage the increase of computational power. However, the art of concurrent programming is known to be a difficult task [27], and programmers always look for new paradigms to simplify it. Transactional Memory (TM) is widely considered as a promising step in this direction, in particular thanks to its simplicity and programmer's friendliness [11].

---

This research is partly supported by the Rainbow FS project of Agence Nationale de la Recherche, France, number ANR-16-CE25-0013-01a.

The engine that orchestrates concurrent transactions run by the application, i.e., the concurrency manager, is one of the core aspects of a TM implementation. A large number of concurrency manager implementations exists, ranging from pessimistic lock-based implementations [1, 21] to completely optimistic ones [22], with [29] or without multi-version support [2]. For application workloads that exhibit a high degree of parallelism, these designs tend to favor optimistic concurrency control. In particular, a widely accepted approach consists in executing tentatively invisible read operations and validating them on the course of the transaction execution to enforce consistency. For performance reasons, another important property is disjoint-access parallelism (DAP) [12]. This property ensures that concurrent transactions operating on disjoint part of the application do not contend in the concurrency manager. Thus, it is key to ensure that the system scales with the numbers of cores.

From a developer’s point of view, the interleaving of transactions must satisfy some form of correctness. Strict serializability (SSER) [24] is a consistency criteria commonly encountered in database literature. This criteria ensures that committed transactions behave as if they were executed sequentially, in an order compatible with real-time. However, SSER does not specify the conditions for aborted transactions. To illustrate this point, let us consider history  $h_1$  where transaction  $T_1 = r(x); r(y)$  and  $T_2 = w(x); w(y)$  are executed respectively by processes  $p$  and  $q$ . In this history,  $T_1$  aborts after reading inconsistent values for  $x$  and  $y$ . Yet,  $h_1$  is compliant with SSER.



Opacity (OPA) was introduced [17] to avoid the erratic behavior of so-called *doomed transactions*, i.e., transactions which eventually abort (such as  $T_1$  in history  $h_1$ ).<sup>1</sup> In addition to SSER, OPA requires that aborted transactions observe a prefix of the committed transactions. This is the usual consistency criteria for TM.

Achieving OPA is known to be expensive, even for weak progress properties on the transactions [30]. In particular, ensuring that a transaction always sees a consistent snapshot when reads are invisible generally asks to re-validate the read set after each read operation, or to rely on a global clock. The former approach increases the time complexity of execution. The latter is expensive in multi-core/multi-processors architecture, due to the synchronization wall.

In this paper, we address these shortcomings with a new consistency criteria, named *stricter serializability* (SSER<sup>+</sup>). This criteria extends strict serializability by avoiding specifically the inconsistency illustrated in history  $h_1$ . We describe in detail a corresponding TM algorithm that ensures invisible reads, and permits

<sup>1</sup> Allowing  $T_1$  to return both  $x_0$  and  $y_2$  may have serious consequences in a non-managed environment. As pointed out in [17], transaction  $T_1$  may compute a division by 0, leading the program to crash.

transactions to commit as long as they do not contend with conflicting transactions. We further validate our design by means of a full implementation of SSER<sup>+</sup> and several experiments. Our results show that when the workloads are embarrassingly parallel, SSER<sup>+</sup> offers close-to-optimum performance.

**Outline.** This paper is organized as follows. Section 2 introduces our system model assumptions and defines precisely SSER<sup>+</sup>. The corresponding transactional memory algorithm and a formal proof of correctness are presented in Sect. 3. We present the evaluation of our prototype against several benchmarks in Sect. 4. We survey related work in Sect. 5, before concluding in Sect. 6.

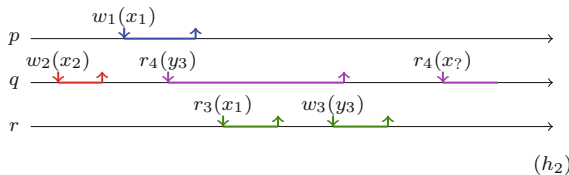
## 2 A New Consistency Criteria

This section is organized in two parts. The first part (Sect. 2.1) present the elements of our system model as well as the notions of contention and binding (Sect. 2.2). In the second part (Sects. 2.3 and 2.4), we formulate our notion of stricter serializability and study its applicability.

### 2.1 System Model

Transactional memory (TM) is a recent paradigm that allows multiple processes to access concurrently a shared memory region. Each process manipulates *objects* in the shared memory with the help of transactions. When a process starts a new transaction, it calls operation *begin*. Then, the process executes a sequence of *read* and *write* operations on the shared objects according to some internal logic. Operation *read*( $x$ ) takes as input an object  $x$  and returns either a value in the domain of  $x$  or a flag ABORT to indicate that the transition aborts. A write *write*( $x, v$ ) changes  $x$  to the value  $v$  in the domain of  $x$ . This operation does not return any value and it may also abort. At the end of the transaction execution, the process calls *tryCommit* to terminate the transaction. This calls returns either COMMIT, to indicate that the transaction commits, or ABORT if the transaction fails.

A *history* is a sequence of invocations and responses of TM operations by one or more processes. As illustrated with history  $h_2$  below, a history is commonly depicted as parallel timelines, where each timeline represents the transactions executed by a process. In history  $h_2$ , process  $p$ ,  $q$  and  $r$  execute respectively transactions  $T_1 = w(x)$ ,  $T_2 = w(x)$  then  $T_4 = r(y); r(x)$ , and  $T_3 = r(x); r(y)$ . All the transactions but  $T_4$  complete in this history. For simplicity, a complete transaction that is not explicitly aborted in a history commits immediately after its last operation. We note  $com(h)$  the set of transactions that commit during history  $h$ . In the case of history  $h_2$ , we have  $com(h_2) = \{T_1, T_2, T_3\}$ .



A history induces a real-time order between transactions (denoted  $\prec_h$ ). The order  $T_i \prec_h T_j$  holds when  $T_i$  terminates in  $h$  before  $T_j$  begins. For instance in history  $h_2$ , transaction  $T_1$  precedes transaction  $T_3$ . When two transactions are not related with real-time, they are *concurrent*.

A *version* is the state of a shared object as produced by the write of a transaction. This means that when a transaction  $T_i$  writes to some object  $x$ , an operation denoted  $w_i(x_i)$ , it creates the version  $x_i$  of  $x$ . Versions allow to uniquely identify the state of an object as observed by a read operation, e.g.,  $r_3(x_1)$  in  $h_2$ . When a transaction  $T_i$  reads version  $x_j$ , we say that  $T_i$  *read-from* transaction  $T_j$ .

Given some history  $h$  and some object  $x$ , a version order on  $x$  for  $h$  is a total order over the versions of  $x$  in  $h$ . By extension, a version order for  $h$  is the union of all the version orders for all the objects (denoted  $\ll_h$ ). For instance, in history  $h_2$  above, we may consider the version order  $(x_2 \ll_{h_2} x_1)$ .

Consider an history  $h$  and some version order  $\ll_h$ . A transaction  $T_i$  *depends on* some transaction  $T_j$ , written  $T_i \supseteq T_j$  when  $T_j$  precedes  $T_i$ ,  $T_i$  reads-from  $T_j$ , or such a relation holds transitively. Transaction  $T_i$  *anti-depends* from  $T_j$  on object  $x$ , when  $T_i$  reads some version  $x_k$ ,  $T_j$  writes version  $x_j$ , and  $x_k$  precedes  $x_j$  in the version order  $(x_k \ll_h x_j)$ . An anti-dependency between  $T_i$  and  $T_j$  on object  $x$  is a *reverse-commit anti-dependency* (for short, RC-anti-dependency) [20] when  $T_j$  commits before  $T_i$ , and  $T_i$  writes some object  $y \neq x$ .<sup>2</sup>

To illustrate the above definitions, consider again history  $h_2$ . In this history, transaction  $T_3$  depends on  $T_1$  and  $T_2$ . On the other hand, if  $x_2 \ll_h x_1$  holds and  $T_4$  reads  $x_2$ , then this transaction exhibits an anti-dependency with  $T_1$ . This anti-dependency becomes an RC-anti-dependency if  $T_4$  executes an additional step during which it writes some object  $z \neq x$ .

Over the course of its execution, a transaction reads and writes versions of the shared objects. The set of versions read by the transaction forms its *read set* (or snapshot). The versions written define the *write set*.

A transaction observes a *strictly consistent* snapshot [5] when it never misses the effects of some transaction it depends on. In detail, the snapshot of transaction  $T_i$  in history  $h$  is strictly consistent when, for every version  $x_j$  read by  $T_i$ , if  $T_k$  writes version  $x_k$ , and  $T_i$  depends on  $T_k$ , then  $x_k$  is followed by  $x_j$  in the version order.

## 2.2 Contention and Bindings

Internally, a transactional memory is built upon a set of *base objects*, such as locks or registers. When two transactions are concurrent, their steps on these base objects interleave. If the two transactions access disjoint objects and the TM is disjoint-access parallel, no contention occurs. However, in the case they access the same base object, they may slow down each other.

A transactional read is *invisible* when it does not change the state of the base objects implementing it. With invisible reads, read contention is basically

<sup>2</sup> In this paper, we consider a slight generalization of an RC-anti-dependency as defined in [20], where  $T_j$  does not read  $x$  prior to its update.

free. From a performance point of view, this property is consequently appealing, since workloads exhibit in most case a large ratio of read operations.

When two transactions are concurrently writing to some object, it is possible to detect the contention and abort preemptively one of them. On the other hand, when a read-write conflict occurs, a *race condition* occurs between the reader and the writer. If the read operation takes place after the write, the reader is bound to use the version produced by the writer.

**Definition 1 (Binding).** *During a history  $h$ , when a transaction  $T_i$  reads some version  $x_j$  and  $T_i$  is concurrent to  $T_j$ , we say that  $T_i$  is bound to  $T_j$  on  $x$ .*

When a transaction  $T_i$  is bound to another transaction  $T_j$ , to preserve the consistency of its snapshot,  $T_i$  must read the updates and causal dependencies of  $T_j$  that are intersecting with its read set. This is for instance the case of transaction  $T_4$  in history  $h_2$ , where this transaction is bound to  $T_3$  on  $y$ . As a consequence,  $T_4$  must return  $x_1$  as the result of its read on  $x$ , or its snapshot will be inconsistent.

Tracking this causality relation is difficult for the contention manager as it requires to inspect the read set, rely on a global clock, or use large amount of metadata. We observe that this tracking is easier if each version read prior the binding is either, accessed by the writer, or one of its dependencies. In which case, we will say that the binding is fair.

**Definition 2 (Fair binding).** *Consider that in some history  $h$  a transaction  $T_i$  is bound to a transaction  $T_j$  on some object  $x$ . This binding is fair when, for every version  $y_k$  read by  $T_i$  before  $x_j$  in  $h$ ,  $T_j \geq T_k$  holds.*

Going back to history  $h_3$ , the binding of  $T_4$  to  $T_3$  on  $y$  is fair. Indeed, this transaction did not read any data item before accessing the version of  $y$  written by  $T_3$ . When the binding is fair, the reader can leverage the metadata left by the writer to check prior versions it has read and ensure the consistency of later read operations. In the next section, we formalize this idea with the notion of stricter serializability.

### 2.3 Stricter Serializability

In what follows, we introduce and describe in detail  $SSER^+$ , the stricter serializability consistency criteria that we build upon in the remainder of this paper. As strict serializability,  $SSER^+$  requires that committed transactions form a sequential history which preserves the real-time order. In addition, it prohibits transactions to view inconsistencies unless one of their bindings is unfair.

**Definition 3 (Strict serialization graph).** *Consider some version order  $\ll_h$  for  $h$ . Below, we define a relation  $<$  to capture all the relations over  $com(h)$  induced by  $\ll_h$ .*

$$T_i < T_{j \neq i} \triangleq \bigvee T_i \prec_h T_j \quad (1)$$

$$\bigvee \exists x : \bigvee r_j(x_i) \in h \quad (2)$$

$$\bigvee \exists T_k : x_k \ll_h x_j \wedge \bigvee T_k = T_i \quad (3)$$

$$\bigvee r_i(x_k) \in h \quad (4)$$

Relation can be either a partial or a total order over the committed transactions in  $h$ . The serialization graph of history  $h$  induced by  $\ll_h$ , written  $\text{SSG}(h, \ll_h)$ , is defined as  $(\text{com}(h), <)$ .

In the above definition, (1) is a real-time order between  $T$  and  $T'$ , (2) a read-write dependency, (3) a version ordering, and (4) an anti-dependency.

**Definition 4 (Stricter serializability).** A history  $h$  is stricter serializable ( $h \in \text{SSER}^+$ ) when (i) for some version order  $\ll_h$ , the serialization graph  $(\text{com}(h), <)$  is acyclic, and (ii) for every transaction  $T_i$  that aborts in  $h$ , either  $T_i$  observes a strictly consistent snapshot in  $h$ , or one of its bindings is unfair.

Opacity (OPA) and strict serializability (SSER) coincide when aborted transactions observe strictly consistent snapshots. As a consequence of the above definition, a stricter serializable history during which all the aborted transactions exhibit fair bindings is opaque.

**Proposition 1.** For a history  $h \in \text{SSER}^+$ , if every transaction  $T$  in  $h$  exhibits fair bindings then  $h \in \text{OPA}$  holds.

Proposition 1 offers a convenient property on histories that, when it applies, allows to reach opacity. The next section characterizes a class of applications for which this property holds. In other words, we give a robustness criteria [6] against  $\text{SSER}^+$ .

## 2.4 Applicability

In what follows, we give some details about the model of application we are interested with. Then, we present our robustness criteria and prove that it applies to  $\text{SSER}^+$ .

**Model of Application.** The state of an object commonly includes *references* to one or more objects in the shared memory. These references between objects form the *object graph* of the application.

When performing a computation, a process traverses a *path* in the object graph. To this end, the process knows initially an immutable *root* object in the graph. Starting from this root, the process executes a traversal by using the references stored in each object.

For some transaction  $T$ , a *path* is the sequence of versions  $\pi$  that  $T$  accesses. It should satisfy that (i) the first object in  $\pi$  corresponds to the immutable root of the object graph, and (ii) for all  $x_i \in \pi$ , some  $y_j <_\pi x_i$  includes a reference to  $x_i$ .

**A Robustness Criteria.** To define our criteria, we focus specifically on  $\text{SSER}^+$  implementations that allow invisible reads. As pointed out earlier, this restriction is motivated by performance since most workloads are read-intensive. In this context, the result of Hans et al. [20] tells us that it is not possible to jointly achieve (i)  $\text{SSER}$ , (ii) read invisibility, (iii) minimal progressiveness, and (iv) accept RC-anti-dependencies. As a consequence, we remove histories that exhibit such a pattern from our analysis; hereafter, we shall note these histories RCAD.

Let us consider the property  $\mathcal{P}$  below on a TM application. In what follows, we prove that if  $\mathcal{P}$  holds and the TM does not accept RC-anti-dependencies, then it is robust against  $\text{SSER}^+$ .

- ( $\mathcal{P}$ ) The object graph forms initially a tree and every transaction maintains this invariant.

Let  $\mathcal{T}$  be some set of transactions for which property  $\mathcal{P}$  holds.  $H_{\mathcal{T}}$  refers to histories built upon transactions in  $\mathcal{T}$ . We wish to establish the following result:

**Proposition 2.**  $(\text{SSER}^+ \cap H_{\mathcal{T}} \setminus \text{RCAD}) \subset \text{OPA}$ .

To state this result, we note  $h$  some history in  $H_{\mathcal{T}} \cap \text{SSER}^+$ . Since  $h$  is serializable, there exists some linearization  $\lambda$  of  $\text{com}(h)$  equivalent to  $h$ . For a transaction  $T_i$  in  $\lambda$ , we let  $\pi_i$  and  $\pi'_i$  be the paths (if any) from the root to  $x$  before and after transaction  $T_i$ . By property  $\mathcal{P}$ , if such a path exists it is unique, because each transaction preserves that the object graph is a tree.

**Lemma 1.** *If transaction  $T_i$  reaches  $x$  in  $h$ , then for every  $y_j$  in  $\pi_i \cup \pi'_i$ , the dependency  $T_i \succeq T_j$  holds.*

*Proof.* There are two cases to consider:

- ( $y_j \in \pi_i$ ) Property  $\mathcal{P}$  implies that either  $y_j$  is the root, or  $T_i$  reads the version  $z_k$  right before  $y_j$  in  $\pi_i$ . Hence, by a short induction, transaction  $T_i$  reads all the versions in  $\pi_i$ .
- ( $y_j \in \pi'_i$ ) Assume that  $T_i$  accesses  $y_j$  and name  $z_k$  the version right before  $y_j$  in  $\pi'_i$ . Version  $z_k$  holds a reference to  $y_j$ . If this reference does not exist prior to the execution of  $T_i$ , object  $z$  was updated. Otherwise,  $T$  must reads  $z_k$  prior to accessing  $y_j$ .



**Lemma 2.** *If transaction  $T_i$  aborts in  $h$  then all its bindings are fair.*

*Proof* (By induction.). Define  $x$  and  $T_j$  such that  $T_i$  is bound to  $T_j$  on  $x$  and assume that all the prior bindings of  $T_i$  are fair.

First, consider that either  $(\pi_i = \pi_j)$  or  $(\pi_i = \pi'_j)$  is true. Choose some  $y_k$  read before  $x_j$  in  $\pi_i$ . By Lemma 1, since  $y_k \in (\pi_j \cup \pi'_j)$  is true, the dependency  $T_j \succeq T_k$  holds.

Otherwise, by our induction hypothesis, all the bindings of  $T_i$  prior  $x_j$  are fair. It follows that transaction  $T_i$  observes a strictly consistent snapshot in  $h$  up to  $r_i(x_j)$ . Hence, there exists a committed transaction  $T_k$  such that  $\pi_i$  is the path to  $x$  after transaction  $T_k$  in  $\lambda$  (i.e.,  $\pi'_k = \pi_i$ ).

Depending on the relative positions of  $T_j$  and  $T_k$  in  $\lambda$ , there are two cases to consider. In both cases, some transaction  $T_l$  between  $T_j$  and  $T_k$  modifies the path to  $x$  in the object graph.

- $(T_j <_\lambda T_k)$  Without lack of generality, assume that  $T_l$  is the first transaction to modify  $\pi'_j$ . Transaction  $T_l$  and  $T_j$  are concurrent in  $h$  and  $T_l$  commits before  $T_j$ . This comes from the fact that  $T_l$  must commit before  $T_i$  in  $h$ ,  $T_j$  is concurrent to  $T_i$  in  $h$  and  $T_j$  is before  $T_l$  in  $\lambda$ . Then, since  $T_l$  modifies  $\pi'_j$  and the two transactions are concurrent,  $T_l$  must update an object read by  $T_j$ . It follows that  $h$  exhibits an RC-anti-dependency between  $T_j$  and  $T_l$ . Contradiction.
- $(T_k <_\lambda T_j)$  Choose some  $y_{k'}$  read before  $x_j$  in  $\pi_i$ . If  $y$  is still in  $\pi_j$ , then  $T_j$  reads at least that version of object  $y$ . Otherwise, consider that  $T_l$  is the first transaction that removes  $y$  from the path to  $x$  in the object graph. To preserve property  $\mathcal{P}$ ,  $T_l$  updates some object  $y'$  read by  $T_{k'}$  that was referring to  $y$ . Because  $h \notin \text{RCAD}$ , transaction  $T_{k'}$  cannot commit after  $T_l$ . Hence,  $T_j \succeq T_{k'}$  holds.

### 3 Algorithm

In this section, we present a transactional memory that attains  $\text{SSER}^+$ . Contrary to several existing TM implementation, our design does not require a global clock. It is weakly-progressive, aborting a transaction only if it encounters a concurrent conflicting transaction. Moreover, reads operations do not modify the base objects of the implementation (read invisibility).

We first give an overview of the algorithm, present its internals and justify some design choices. A correctness proofs follows. We close this section with a discussion on the parameters of our algorithm. In particular, we explain how to tailor it to be disjoint-access parallel.

### 3.1 Overview

Algorithm 1 depicts the pseudo-code of our construction of the TM interface at some process  $p$ . Our design follows the general approach of the lazy snapshot algorithm (LSA) [14], replacing the central clock with a more flexible mechanism. Algorithm 1 employs a deferred update schema that consists in two steps. A transaction first executes optimistically, buffering its updates. Then, at commit time, the transaction is certified and, if it commits, its updates are applied to the shared memory.

During the execution of a transaction, a process checks that the objects accessed so far did not change. Similarly to LSA, this check is lazily executed. Algorithm 1 executes it only if the shared object was recently updated, or when the transaction terminates.

### 3.2 Tracking Time

Algorithm 1 tracks time to compute how concurrent transactions interleave during an execution. To this end, the algorithm makes use of logical clocks. We model the interface of a *logical clock* with two operations:  $read()$  returns a value in  $\mathbb{N}$ , and  $adv(v \in \mathbb{N})$  updates the clock with value  $v$ . The sequential specification of a logical clock guarantees a single property, that the time flows forward: (*Time Monotonicity*) A read operation always returns at least the greatest value to which the clock advanced so far. In every sequential history  $h$ ,  $(res(read(), v) \in h) \rightarrow (v \geq \max(\{u : adv(u) \prec_h read()\} \cup \{0\}))$ .

Algorithm 1 associates logical clocks with both processes and transactions. To retrieve the clock associated with some object  $x$ , the algorithm uses function  $clock(x)$ . Notice that in the pseudo-code, when it is clear from the context,  $clock(x)$  is a shorthand for  $clock(x).read()$ .

The clock associated with a transaction is always local (line 2). In the case of a process, it might be shared or not (line 3). The flexibility of our design comes from this locality choice for  $clock(p)$ . When the clock is shared, it is linearizable. To implement an (obstruction-free) linearizable clock we employ the following common approach:

**(Construction 1).** Let  $x$  be a shared register initialized to 0. When  $read()$  is called, we return the value stored in  $x$ . Upon executing  $adv(v)$ , we fetch the value stored in  $x$ , say  $u$ . If  $v > u$  holds, we execute a compare-and-swap to replace  $u$  with  $v$ ; otherwise the operation returns. If the compare-and-swap fails, the previous steps are retried.

**Algorithm 1.** A SSER<sup>+</sup> transactional memory – code at process  $p$ 


---

```

1: Variables:
2:    $clock(T), rs(T), ws(T)$  // local to  $p$ 
3:    $clock(p)$  // shared

4:  $begin(T)$ 
5:    $clock(T).adv(\min_q clock(q))$ 
6:    $rs(T) \leftarrow \emptyset$ 
7:    $ws(T) \leftarrow \emptyset$ 

8:  $read(T, x)$ 
9:   if  $(x, d) \in ws(T)$  then
10:    return  $d$ 
11:    $(d, t) \leftarrow loc(x)$ 
12:   if  $isLocked(x)$ 
13:      $\vee (\exists(x, t') \in rs(T) : t \neq t')$  then
14:     return  $abort(t)$ 
15:   if  $t > clock(T) \wedge \neg extend(T, t)$  then
16:     return  $abort(t)$ 
17:    $rs(T)[x] \leftarrow t$ 
18:   return  $d$ 

19:  $write(T, x, d)$ 
20:   if  $\neg lock(x)$  then
21:     return  $abort(T)$ 
22:    $(-, t) \leftarrow loc(x)$ 
23:   if  $t > clock(T) \wedge \neg extend(T, t)$  then
24:     return  $abort(t)$ 
25:    $ws(T)[x] \leftarrow d$ 

26:  $tryCommit(T)$ 
27:   if  $\neg extend(T, clock(T))$  then
28:     return  $abort(T)$ 
29:   return  $commit(T)$ 

// Helpers
30:  $extend(T, t)$ 
31:   for all  $(x, t') \in rs(T)$  do
32:      $(-, t'') \leftarrow loc(x)$ 
33:     if  $isLocked(x) \vee t'' \neq t'$  then
34:       return  $false$ 
35:    $clock(T).adv(t)$ 
36:   return  $true$ 

37:  $abort(T)$ 
38:   for all  $(x, -) \in ws(T)$  do
39:      $unlock(x)$ 
40:   return ABORT

41:  $commit(T)$ 
42:   if  $ws(T) \neq \emptyset$  then
43:      $clock(T).adv(clock(T) + 1)$ 
44:      $clock(p).adv(clock(T))$ 
45:     for all  $(x, d) \in ws(T)$  do
46:        $loc(x) \leftarrow (d, clock(T))$ 
47:        $unlock(x)$ 
48:   return COMMIT

```

---

### 3.3 Internals

In Algorithm 1, each object  $x$  has a *location* in the shared memory, denoted  $loc(x)$ . This location stores a pair  $(d, t)$ , where  $t \in \mathbb{N}$  is a *timestamp*, and  $d$  is the actual content of  $x$  as seen by transactions. For simplicity, we shall name

hereafter a pair  $(d, t)$  a *version* of object  $x$ . Since the location of object  $x$  is unique, a single version of object  $x$  may exist at a time in the memory. As usual, we assume some transaction  $T_{\text{INIT}}$  that initializes for every object  $x$  the location  $loc(x)$  to  $(\perp, 0)$ . Furthermore, we consider that each read or write operation to some location  $loc(x)$  is atomic.

Algorithm 1 associates a lock to each object. To manipulate the lock-related functions of object  $x$ , a process  $p$  employs appropriately the functions  $lock(x)$ ,  $isLocked(x)$  and  $unlock(x)$ .

For every transaction  $T$  submitted to the system, Algorithm 1 maintains three local data structures:  $clock(T)$  is the logical clock of transaction  $T$ ;  $rs(T)$  is a map that contains its read set; and  $ws(T)$  is another map that stores the write set of  $T$ . Algorithm 1 updates incrementally  $rs(T)$  and  $ws(T)$  over the course of the execution. The read set serves to check that the snapshot of the shared memory as seen by the transaction is strictly consistent. The write set buffers updates. With more details, the execution of a transaction  $T$  proceeds as follows.

- When  $T$  starts its execution, Algorithm 1 initializes  $clock(T)$  to the smallest value of  $clock(q)$  for any process  $q$  executing the TM. Then, both  $rs(T)$  and  $ws(T)$  are set to  $\emptyset$ .
- When  $T$  accesses a shared object  $x$ , if  $x$  was previously written, its value is returned (line 10). Otherwise, Algorithm 1 fetches atomically the version  $(d, t)$ , as seen in location  $loc(x)$ . Then, the algorithm checks that (i) no lock is held on  $x$ , and (ii) in case  $x$  was previously accessed, that  $T$  observes the same version. If one of these two conditions fails, Algorithm 1 aborts transaction  $T$  (line 14). The algorithm then checks that the timestamp  $t$  associated to the content  $d$  is smaller than the clock of  $T$ . In case this does not hold (line 15), Algorithm 1 tries extending the snapshot of  $T$  by calling function  $extend()$ . This function returns *true* when the versions previously read by  $T$  are still valid. In which case,  $clock(T)$  is updated to the value  $t$ . If Algorithm 1 succeeds in extending (if needed) the snapshot of  $T$ ,  $d$  is returned and the read set of  $T$  updated accordingly; otherwise transaction  $T$  is aborted (line 16).
- Upon executing a write request on behalf of  $T$  to some object  $x$ , Algorithm 1 takes the lock associated with  $x$  (line 20), and in case of success, it buffers the update value  $d$  in  $ws(T)$  (line 25). The timestamp  $t$  of  $x$  at the time Algorithm 1 takes the lock serves two purposes. First, Algorithm 1 checks that  $t$  is lower than the current clock of  $T$ , and if not  $T$  is extended (line 23). Second, it is saved in  $ws(T)$  to ensure that at commit time the timestamp of the version of  $x$  written by  $T$  is greater than  $t$ .
- When  $T$  requests to commit, Algorithm 1 certifies the read set by calling function  $extend()$  with the clock of  $T$  (line 27). If this test succeeds, transaction  $T$  commits (lines 43 to 48). In such a case,  $clock(T)$  ticks to reach its final value (line 43). By construction, this value is greater than the timestamps of all the versions read or written by  $T$  (lines 14 and 23). Algorithm 1 updates the clock of  $p$  with the final value of  $clock(T)$  (line 44), then it updates the items written by  $T$  with their novel versions (line 46).

### 3.4 Guarantees

In this section, we assess the core properties of Algorithm 1. First, we show that our TM design is weakly progressive, i.e., that the algorithm aborts a transaction only if it encounters a concurrent conflicting transaction. Then, we prove that Algorithm 1 is stricter serializable.

(*Weak-progress*). A transaction executes under *weak progressiveness* [19], or equivalently it is *weakly progressive*, when it aborts only if it encounters a conflicting transaction. By extension, a TM is weakly progressive when it only produces histories during which transactions are weakly-progressive. We prove that this property holds for Algorithm 1.

In Algorithm 1, a transaction  $T$  aborts either at line 14, 16, 21, 24, or 28. We observe that in such a case either  $T$  observes an item  $x$  locked, or that the timestamp associated with  $x$  has changed. It follows that if  $T$  aborts then it observes a conflict with a concurrent transaction. From which we deduce that it is executing under weak progressiveness.

(*Stricter serializability*). Consider some run  $\rho$  of Algorithm 1, and let  $h$  be the history produced in  $\rho$ . At the light of its pseudo-code, every function defined in Algorithm 1 is wait-free. As a consequence, we may consider without lack of generality that  $h$  is complete, i.e., every transaction executed in  $h$  terminates with either a commit or an abort event. In what follows, we let  $\ll_h$  be the order in which writes to the object locations are linearized in  $\rho$ . We first prove that  $<$  is acyclic for this definition of  $\ll_h$ . Then, we show that, if a transaction does not exhibit any unfair binding, then it observes a strictly consistent snapshot. For some transaction, we shall note  $clock(T_i)_f$  the final value of  $clock(T)$ .

**Proposition 3.** *Consider two transactions  $T_i$  and  $T_{j \neq i}$  in  $h$ . If either  $T_i \trianglerighteq T_j$  or  $x_j \ll_h x_i$  holds, then  $clock(T_i)_f \geq clock(T_j)_f$  is true. In addition, if transaction  $T_i$  is an update that commits then the ordering is strict, i.e.,  $clock(T_i)_f > clock(T_j)_f$ .*

*Proof.* In each of the two cases, we prove that  $clock(T_i)_f \geq clock(T_i)_f$  holds before transaction  $T_i$  commits.

( $T_i \trianglerighteq T_j$ ). Let  $x$  be an object such that  $r_i(x_j)$  occurs in  $h$ . Since transaction  $T_i$  reads version  $x_j$ , transaction  $T_j$  commits. We observe that  $T_j$  writes version  $x_j$  together with  $clock(T_j)_f$  at  $loc(x)$  when it commits (line 46). As a consequence, when transaction  $T_i$  returns version  $x_i$  at line 18, it assigns  $clock(T_j)_f$  to  $t$  before at line 11. The condition at line 15 implies that either  $clock(T_i) \geq t$  holds, or a call to  $extend(T_i, t)$  occurs. In the latter case, transaction  $T_i$  executes line 35, advancing its clock up to the value of  $t$ .

( $x_j \ll_h x_i$ ). By definition, relation  $\ll_h$  forms a total order over all versions of  $x$ . Thus, we may reason by induction, considering that  $x_i$  is immediately after  $x_j$  in the order  $\ll_h$ . When  $T_j$  returns from  $w_j(x_j)$  at line 25, it holds a lock on  $x$ . This lock is released at line 47 after writing to  $loc(x)$ . As  $\ll_h$  follows

the linearization order,  $T_i$  executes line 20 after  $T_j$  wrote  $(x_j, clock(T_j)_f)$  to  $loc(x)$ . Location  $loc(x)$  is not updated between  $x_j$  and  $x_i$ . Hence, after  $T_i$  executes line 23,  $clock(T_i) \geq clock(T_j)$  holds.

Since a clock is monotonic, the relation holds forever. Then, if transaction  $T_i$  is an update that commits, it must execute line 43, leading to  $clock(T_i)_f > clock(T_i)_f$ .

**Proposition 4.** *History  $h$  does not exhibit any RC-anti-dependencies ( $h \notin RCAD$ )*

*Proof.* Consider  $T_i$ ,  $T_j$  and  $T_k$  such that  $r_i(x_k), w_j(x_j) \in h$ ,  $x_k \ll_h x_j$  and  $T_j$  commits before  $T_i$ . When  $T_j$  invokes *commit*, it holds a lock on  $x$ . This lock is released at line 47 after version  $x_j$  is written at location  $loc(x)$ . Then, consider the time at which  $T_i$  invokes *tryCommit*. The call at line 27 leads to fetching  $loc(x)$  at line 32. Since  $T_i$  reads version  $x_k$  in  $h$ , a pair  $(x_k, clock(T_k)_f)$  is in  $rs(T_i)$ . From the definition of  $\ll_h$  the write of  $(x_k, clock(T_k)_f)$  takes place before the write of version  $(x_j, clock(T_j)_f)$  in  $\rho$ . Hence,  $loc(x)$  does not contain anymore  $(x_k, clock(T_k)_f)$ . Applying Proposition 3,  $T_i$  executes line 34 and aborts at line 29.

**Proposition 5.** *Consider two transactions  $T_i$  and  $T_{j \neq i}$  in  $com(h)$ . If  $T_i < T_j$  holds, transaction  $T_i$  invokes *commit* before transaction  $T_j$  in  $h$ .*

*Proof.* Assume that  $T_i$  and  $T_j$  conflict of some object  $x$ . We examine in order each of the four cases defining relation  $<$ .

- $(T_i \prec_h T_j)$   
This case is immediate.
- $(\exists x : r_j(x_i) \in h)$   
Before committing,  $T_j$  invokes *extend* at line 27. Since  $T_j$  commits in  $h$ , it should retrieve  $(x_i, -)$  from  $loc(x)$  when executing line 32. Hence, transaction  $T_i$  has already executed line 46 on object  $x$ . It follows that  $T_i$  invokes *commit* before transaction  $T_j$  in history  $h$ .
- $(\exists x : x_i \ll_h x_j)$   
By definition of  $\ll_h$ , the write of version  $x_i$  is linearized before the write of version  $x_j$  in  $\rho$ . After  $T_i$  returns from  $w_i(x_i)$ , it owns a lock on object  $x$  (line 46). The object is then unlocked by transaction  $T_i$  at line 47. As a consequence, transaction  $T_i$  takes a lock on object  $x$  after  $T_i$  invokes operation *commit*. From which it follows that the claim holds.
- $(\exists x, T_k : x_k \ll_x x_j \wedge r_i(x_k))$   
Follows from Proposition 4.

**Theorem 1.** *History  $h$  belongs to  $SSER^+$ .*

*Proof.* Proposition 5 tells us that if  $T_i < T_j$  holds then  $T_i$  commits before  $T_j$ . It follows that the  $SSG(h, \ll_h)$  is acyclic.

Let us now turn our attention to the second property of SSER<sup>+</sup>. Assume that a transaction  $T_i$  aborts in  $h$ . For the sake of contradiction, consider that  $T_i$  exhibits fair bindings and yet that it observes a non-strictly consistent snapshot.

Applying the definition given in Sect. 2.1, there exist transactions  $T_j$  and  $T_k$  such that  $T_i \supseteq T_j$ ,  $r_i(x_k)$  occurs in  $h$  and  $x_k \ll_h x_j$ . Applying Proposition 5, if  $T_j \prec_h T_i$  holds, transaction  $T_i$  cannot observe version  $x_k$ . Thus, transaction  $T_j$  is concurrent to  $T_i$ . Moreover, by definition of  $T_i \supseteq T_j$ , there exist a transaction  $T_l$  (possibly,  $T_j$ ) and some object  $y$  such that  $T_i$  performs  $r_i(y_l)$  and  $T_l \supseteq T_j$ . In what follows, we prove that  $T_i$  aborts before returning  $y_l$ .

For starter, relation  $<$  is acyclic, thus  $x_k \neq y_l$  holds. It then remains to investigate the following two cases:

- $(r_i(y_l) \prec_h r_i(x_k))$   
From Proposition 5 and  $T_l \supseteq T_j$ , transaction  $T_j$  is committed at the time  $T_i$  reads object  $x$ . Contradiction.
- $(r_i(x_k) \prec_h r_i(y_l))$   
We first argue that, at the time  $T_i$  executes line 11, the timestamp fetches from  $loc(y)$  is greater than  $clock(T_i)$ .

*Proof.* First of all, observe that  $T_j$  is not committed at the time  $T_i$  reads object  $x$  (since  $x_k \ll_h x_j$  holds). Hence, denoting  $q$  the process that executes  $T_j$ ,  $clock(q) < clock(T_j)_f$  is true when  $T_i$  begins its execution at line 5. From the pseudo-code at line 5,  $clock(T_i) < clock(T_j)_f$  holds at the start of  $T_i$ . Because  $T_j$  is concurrent to  $T_i$ ,  $T_l$  is also concurrent to  $T_i$  by Proposition 5. Thus, as  $r_i(y_l)$  occurs,  $T_i$  is bound to  $T_l$  on  $y$ . Now, consider some object  $z$  read by  $T_i$  before  $y$ , and name  $z_r$  the version read by  $T_i$ . Since the binding of  $T_i$  to  $T_l$  is fair,  $T_l \supseteq T_r$  is true. Hence, applying Proposition 3, we have  $clock(T_r)_f < clock(T_l)_f$ . It follows that the relation  $clock(T_i) < clock(T_l)_f$  is true.

From what precedes, transaction  $T_i$  invokes *extend* at line 15. We know that transaction  $T_j$  is committed at that time (since  $T_l$  is committed and  $T_l \supseteq T_j$  holds). Thus, the test at line 33 fails and  $T_i$  aborts before returning  $y_l$ .

### 3.5 Discussion

Algorithm 1 replaces the global clock usually employed in TM architectures with a more flexible mechanism. For some process  $p$ ,  $clock(p)$  can be local to  $p$ , shared across a subset of the processes, or even all of them.

If processes need to synchronize too often, maintaining consistency among the various clocks is expensive. In this situation, it might be of interest to find a compromise between the cost of cache coherency and the need for synchronization. For instance, in a NUMA architecture, Algorithm 1 may assign a clock per hardware socket. Upon a call to  $clock(p)$ , the algorithm returns the clock defined for the socket in which the processor executing process  $p$  resides.

On the other hand, when the processes use a global clock, Algorithm 1 boils down to the original TinySTM implementation [14]. In this case, a read-only transaction always sees a strictly consistent snapshot. As a consequence, it can

commit right after a call to *tryCommit*, i.e., without checking its snapshot at line 27.

A last observation is that our algorithm works even if one of the processes takes no step. This implies that the calls to process clocks (at lines 5 and 44) are strictly speaking not necessary and can be skipped without impacting the correctness of Algorithm 1. Clocks are solely used to avoid extending the snapshot at each step where a larger timestamp is encountered. If process clocks are not used, when two transactions access disjoint objects, they do not contend on any base object of the implementation. As a consequence, such a variation of Algorithm 1 is disjoint-access parallel (DAP).

## 4 Evaluation

This section presents a performance study of our SSER<sup>+</sup> transactional memory described in Sect. 3. To conduct this evaluation we implemented and integrated our algorithm inside TINYSTM [14], a state-of-the-art software transactional memory implementation. Our modifications account for approximately 500 SLOC. We run Algorithm 1 in disjoint-access parallel mode. As explained in Sect. 3.5, in this variation the clocks of the processes are not accessed. A detailed evaluation of the other variations of Algorithm 1 is left for future work.

The experiments are conducted on an AMD Opteron48, a 48-cores machine with 256 GB of RAM. This machine has 4 dodeca-core AMD Opteron 6172, and 8 NUMA nodes. To evaluate the performance of our implementation on this multi-core platform, we use the test suite included with TINYSTM. This test suite is composed of several TM applications with different transaction patterns. The remainder of this section briefly describes the benchmarks and discuss our results. As a matter of a comparison, we also present the results achieved with the default TINYSTM distribution, (v1.0.5).

### 4.1 A Bank Application

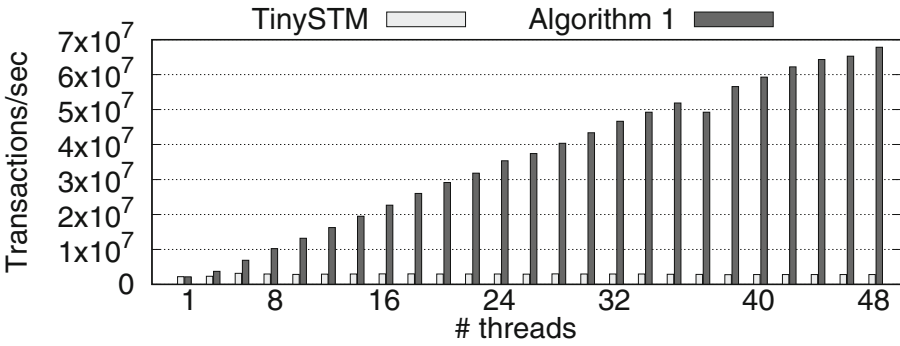
The bank benchmark consists in simulating transfers between bank accounts. A transaction updates two accounts, transferring some random amount of money from one account to another. A thread executing this benchmark performs transfers in closed-loop. Each thread is bound to some *branch* of the bank, and accounts are spread evenly across the branches. A *locality* parameter allows to tune the accounts accessed by a thread to do a transfer. This parameter serves to adjust the probability that a thread executes consecutive operations on the same data. More specifically, when locality is set to the value  $\rho$ , a thread executes a transfer in its branch with probability  $\rho$  and between two random accounts with probability  $(1 - \rho)$ . When  $\rho = 1$ , this workload is fully parallel.

Figure 1 presents the experimental results for the bank benchmark. In Fig. 1(a), we execute a base scenario with 10k bank accounts, and a locality of 0.8. We measure the number of transfers performed by varying the number of threads in the application. In this figure, we observe that the performance

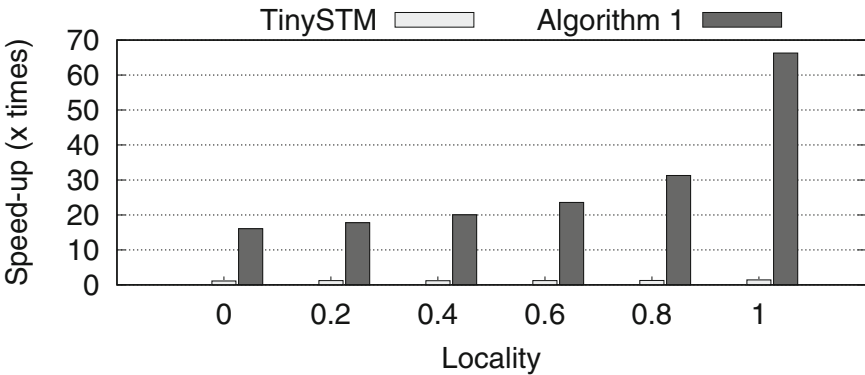


obtained with TINYSTM merely improves as the number of thread increases: 48 threads achieve 2.8 million transactions per second (MTPS), scaling-up from 2.2 MTPS with a single thread. Our implementation performs better: with 48 threads Algorithm 1 executes around 68 MTPS, executing  $\times 31$  more operations than with one thread.

To understand the impact of data locality on performance, we vary this parameter for a fixed number of threads. Figure 1(b) presents the speedup obtained when varying locality from 0, i.e., all the accounts are chosen at random, up to 1, where they are all chosen in the local branch. In this experiment, we fix the number of threads to 48, i.e. the maximum number of cores available on our test machine. As shown in Fig. 1(b), our TM implementation leverages the presence of data locality in the bank application. This is expected, since we use the disjoint-access parallel (DAP) variation of Algorithm 1. When locality increases, the contention in the application decreases. As a consequence of DAP, each thread works on independent data, thus improving performance.

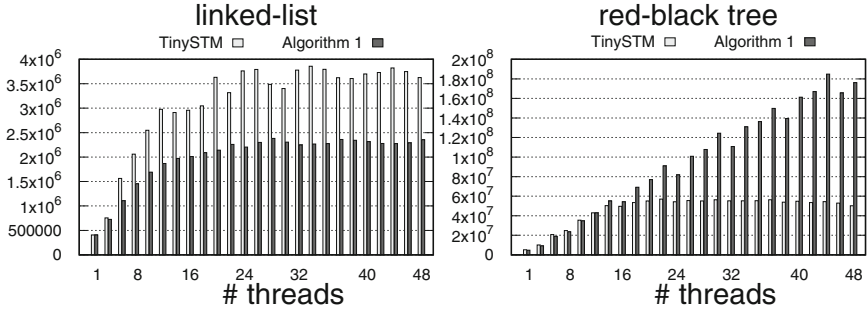


(a) Base scenario – locality set to 0.8



(b) Varying locality – using 48 threads

Fig. 1. Bank benchmark with fixed locality (a) and increasing locality values (b).



**Fig. 2.** Linked-list (left) and Red-Black tree (right) benchmarks. Y-axis: transactions/sec.

## 4.2 Linked-List

The linked-list benchmark consists in concurrently modifying a sorted linked-list of integers. Each thread randomly adds or removes an integer from the list. We run this benchmark for a range of 512 values, *i.e.* a thread randomly selects a value between  $-255$  and  $+256$  before doing an insertion/removal. The linked list is initialized to contain 256 integers. We report our results in Fig. 2 (left).

We observe that TINYSTM outperforms our implementation in the linked-list benchmark. This is due to the fact that, without proper clock synchronization, transactions tend to re-validate their reads frequently over their execution paths. In this scenario of high contention, it is (as expected) preferable to rely on a frequent synchronization mechanism such as the global clock used in TINYSTM. To alleviate this issue, one could adjust dynamically the clocks used in Algorithm 1 accordingly to contention. Such a strategy could rely on a global lock, similarly to the mechanism used to avoid that long transactions abort. We left the implementation of this optimization for future work

## 4.3 Red-Black Tree

The red-black tree benchmark is similar to the linked-list benchmark except that the values are stored in a self-balancing binary search tree. We run this benchmark with a range of  $10^7$  values, and a binary tree initialized with  $10^5$  values. Figure 2 (right) reports our results.

When using the original TINYSTM design, the performance of the application improves linearly up to 12 threads. It then stalls to approximately 50 MTPS due to contention on the global clock. In this benchmark, the likelihood of having two concurrent conflicting transactions is very low. Leveraging this workload property, our implementation of Algorithm 1, scales the application linearly with the number of threads. Algorithm 1 achieves 176 MTPS with 48 threads, improving performance by a  $\times 36$  factor over a single threaded execution.

## 5 Related Work

Transactional memory (TM) allows to design applications with the help of sequences of instructions that run in isolation one from another. This paradigm greatly simplifies the programming of modern highly-parallel computer architectures.

Ennals [13] suggests to build deadlock-free lock-based TMs rather than non-blocking ones. Empirical evidences [9] and theoretical results [18, 26] support this claim.

At first glance, it might be of interest that a TM design accepts all correct histories; a property named *permissiveness* [16]. Such TM algorithms need to track large dependencies [25] and/or acquire locks for read operations [2]. However, both techniques are known to have a significant impact on performance.

Early TM implementations (such as DSTM [23]) validate all the prior reads when accessing a new object. The complexity of this approach is quadratic in the number of objects read along the execution path. A time-based TM avoids this effort by relying on the use a global clock to timestamp object versions. Zhang et al. [33] compare several such approaches, namely TL2 [8], LSA [31] and GCC [32]. They provide guidelines to reduce unnecessary validations and shorten the commit sequence.

Multi-versioning [10, 15] brings a major benefit: allowing read-only transactions to complete despite contention. This clearly boosts certain workloads but managing multiple versions has a non-negligible performance cost on the TM internals. Similarly, invisible reads ensure that read operations do not contend in most cases. However, such a technique limits progress or the consistency criteria satisfied by the TM [3]. In the case of Algorithm 1, both read-only and updates transaction are certain to make progress only in the absence of contention.

New challenges arise when considering multicore architectures and cache coherency strategies for NUMA architectures. Clock contention [7] is one of them. To avoid this problem, workloads as well as TM designs should take into account parallelism [28]. Chan et al. [7] propose to group threads into zones, and that each zone shares a clock and a clock table. To timestamp a new version, the TL2C algorithm [4] tags it with a local counter together with the thread id. Each thread stores a vector of the latest timestamp it encountered. The algorithm preserves opacity by requiring that a transaction restarts if one of the vector entries is not up to date.

## 6 Conclusion

Transactional memory systems must handle a tradeoff between consistency and performance. It is impractical to take into account all possible combinations of read and write conflicts, as it would lead to largely inefficient solutions. For instance, accepting RCAD histories brings only a small performance benefits in the general case [20].

This paper introduces a new consistency criteria, named stricter serializability (SSER<sup>+</sup>). Workloads executed under SSER<sup>+</sup> are opaque when the object

graph forms a tree and transactions traverse it top-down. We present an algorithm to attain this criteria together with a proof of its correctness. Our evaluation based on a fully implemented prototype demonstrates that such an approach is very efficient in weakly-contended workloads.

## References

1. Afek, Y., Matveev, A., Shavit, N.: Pessimistic software lock-elision. In: Aguilera, M.K. (ed.) DISC 2012. LNCS, vol. 7611, pp. 297–311. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33651-5\\_21](https://doi.org/10.1007/978-3-642-33651-5_21)
2. Attiya, H., Hillel, E.: A single-version STM that is multi-versioned permissive. *Theory Comput. Syst.* **51**(4), 425–446 (2012). <https://doi.org/10.1007/s00224-012-9406-3>
3. Attiya, H., Hillel, E., Milani, A.: Inherent limitations on disjoint-access parallel implementations of transactional memory. In: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2009, pp. 69–78. ACM, New York (2009). <https://doi.org/10.1145/1583991.1584015>
4. Avni, H., Shavit, N.: Maintaining consistent transactional states without a global clock. In: Shvartsman, A.A., Felber, P. (eds.) SIROCCO 2008. LNCS, vol. 5058, pp. 131–140. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-69355-0\\_12](https://doi.org/10.1007/978-3-540-69355-0_12)
5. Bernstein, P.A., Goodman, N.: Timestamp-based algorithms for concurrency control in distributed database systems. In: Proceedings of the 6th International Conference on Very Large Data Bases, pp. 285–300, October 1980. <http://dl.acm.org/citation.cfm?id=1286887.1286918>
6. Cerone, A., Gotsman, A.: Analysing snapshot isolation. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, 25–28 July 2016, pp. 55–64 (2016). <https://doi.org/10.1145/2933057.2933096>
7. Chan, K., Wang, C.L.: TrC-MC: decentralized software transactional memory for multi-multicore computers. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), pp. 292–299 (2011). <https://doi.org/10.1109/ICPADS.2011.144>
8. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006). [https://doi.org/10.1007/11864219\\_14](https://doi.org/10.1007/11864219_14)
9. Dice, D., Shavit, N.: What really makes transactions faster. In: Proceedings of the 1st TRANSACT 2006 Workshop, vol. 8, p. 3 (2006)
10. Diegues, N., Romano, P.: Time-warp: lightweight abort minimization in transactional memory. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2014, pp. 167–178. ACM, New York (2014). <https://doi.org/10.1145/2555243.2555259>
11. Dragojević, A., Felber, P., Gramoli, V., Guerraoui, R.: Why STM can be more than a research toy. *Commun. ACM* **54**(4), 70–77 (2011). <https://doi.org/10.1145/1924421.1924440>
12. Ellen, F., Fatourou, P., Kosmas, E., Milani, A., Travers, C.: Universal constructions that ensure disjoint-access parallelism and wait-freedom. *Distrib. Comput.* **29**(4), 251–277 (2016). <https://doi.org/10.1007/s00446-015-0261-8>

13. Ennals, R.: Software transactional memory should not be obstruction-free. Technical report, Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report (2006)
14. Felber, P., Fetzer, C., Marlier, P., Riegel, T.: Time-based software transactional memory. *IEEE Trans. Parallel Distrib. Syst.* **21**(12), 1793–1807 (2010). <https://doi.org/10.1109/TPDS.2010.49>
15. Fernandes, S.M., Cachopo, J.a.: Lock-free and scalable multi-version software transactional memory. In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPOPP 2011*, pp. 179–188. ACM, New York (2011). <https://doi.org/10.1145/1941553.1941579>
16. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: Taubenfeld, G. (ed.) *DISC 2008. LNCS*, vol. 5218, pp. 305–319. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-87779-0\\_21](https://doi.org/10.1007/978-3-540-87779-0_21)
17. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008*, pp. 175–184. ACM, New York (2008). <https://doi.org/10.1145/1345206.1345233>
18. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008*, pp. 175–184. ACM, New York (2008)
19. Guerraoui, R., Kapalka, M.: The semantics of progress in lock-based transactional memory. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pp. 404–415. ACM, New York (2009). <https://doi.org/10.1145/1480881.1480931>
20. Hans, S., Hassan, A., Palmieri, R., Peluso, S., Ravindran, B.: Opacity vs TMS2: expectations and reality. In: Gavoille, C., Ilcinkas, D. (eds.) *DISC 2016. LNCS*, vol. 9888, pp. 269–283. Springer, Heidelberg (2016). [https://doi.org/10.1007/978-3-662-53426-7\\_20](https://doi.org/10.1007/978-3-662-53426-7_20)
21. Harris, T., Fraser, K.: Revocable locks for non-blocking programming. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005*, pp. 72–82. ACM, New York (2005). <https://doi.org/10.1145/1065944.1065954>
22. Hassan, A., Palmieri, R., Ravindran, B.: Optimistic transactional boosting. *SIGPLAN Not.* **49**(8), 387–388 (2014). <https://doi.org/10.1145/2692916.2555283>
23. Herlihy, M., Luchangco, V., Moir, M., Scherer, III, W.N.: Software transactional memory for dynamic-sized data structures. In: *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, PODC 2003*, pp. 92–101. ACM, New York (2003). <https://doi.org/10.1145/872035.872048>
24. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **12**(3), 463–492 (1990). <https://doi.org/10.1145/78969.78972>
25. Keidar, I., Perelman, D.: On avoiding spare aborts in transactional memory. In: *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2009*, pp. 59–68. ACM, New York (2009). <https://doi.org/10.1145/1583991.1584013>
26. Kuznetsov, P., Ravi, S.: Why transactional memory should not be obstruction-free. *CoRR abs/1502.02725* (2015)
27. Lee, E.A.: The problem with threads. *Computer* **39**(5), 33–42 (2006). <https://doi.org/10.1109/MC.2006.180>

28. Nguyen, D., Pingali, K.: What scalable programs need from transactional memory. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, pp. 105–118. ACM, New York (2017). <https://doi.org/10.1145/3037697.3037750>
29. Perelman, D., Byshevsky, A., Litmanovich, O., Keidar, I.: SMV: selective multi-versioning STM. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 125–140. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-24100-0\\_9](https://doi.org/10.1007/978-3-642-24100-0_9)
30. Ravi, S.: Lower bounds for transactional memory. Bull. EATCS **121** (2017)
31. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006). [https://doi.org/10.1007/11864219\\_20](https://doi.org/10.1007/11864219_20)
32. Spear, M.F., Marathe, V.J., Scherer, W.N., Scott, M.L.: Conflict detection and validation strategies for software transactional memory. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 179–193. Springer, Heidelberg (2006). [https://doi.org/10.1007/11864219\\_13](https://doi.org/10.1007/11864219_13)
33. Zhang, R., Budimlić, Z., Scherer, III, W.N.: Commit phase in timestamp-based STM. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008, pp. 326–335. ACM, New York (2008). <https://doi.org/10.1145/1378533.1378589>



# From Field-Based Coordination to Aggregate Computing

Mirko Viroli<sup>1</sup>(✉), Jacob Beal<sup>2</sup>, Ferruccio Damiani<sup>3</sup>, Giorgio Audrito<sup>3</sup>,  
Roberto Casadei<sup>1</sup>, and Danilo Pianini<sup>1</sup>

<sup>1</sup> Alma Mater Studiorum–Università di Bologna, Cesena, Italy  
{mirko.viroli, roby.casadei, danilo.pianini}@unibo.it

<sup>2</sup> Raytheon BBN Technologies, Cambridge, USA  
jakebeal@ieee.org

<sup>3</sup> Università di Torino, Turin, Italy  
{damiani, audrito}@di.unito.it

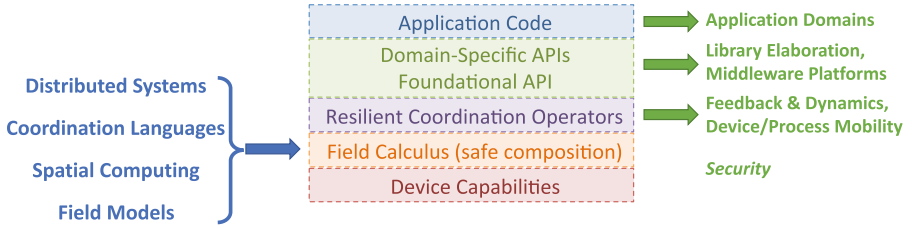
**Abstract.** Aggregate computing is an emerging approach to the engineering of complex coordination for distributed systems, based on viewing system interactions in terms of information propagating through collectives of devices, rather than in terms of individual devices and their interaction with their peers and environment. The foundation of this approach is the distillation of a number of prior approaches, both formal and pragmatic, proposed under the umbrella of field-based coordination, and culminating into the *field calculus*, a functional programming model for the specification and composition of collective behaviours with equivalent local and aggregate semantics. This foundation has been elaborated into a layered approach to engineering coordination of complex distributed systems, building up to pragmatic applications through intermediate layers encompassing reusable libraries of provably resilient program components. In this survey, we trace the development and antecedents of field calculus, review the current state of aggregate computing theory and practice, and discuss a roadmap of current research directions that we believe can significantly impact the agenda of coordination models and languages.

## 1 Introduction

As computing devices continue to become cheaper and more pervasive, the complexity of the distributed systems that run our world continues to increase. Over

---

This work has been partially supported by: EU Horizon 2020 project HyVar ([www.hyvar-project.eu](http://www.hyvar-project.eu)), GA No. 644298; ICT COST Action IC1402 ARVI ([www.cost-arvi.eu](http://www.cost-arvi.eu)); Ateneo/CSP D16D15000360005 project RunVar ([runvar-project.di.unito.it](http://runvar-project.di.unito.it)). This document does not contain technology or technical data controlled under either U.S. International Traffic in Arms Regulation or U.S. Export Administration Regulations.



**Fig. 1.** This survey reviews the development of field calculus from its antecedents (left), the current state of aggregate computing theory and practice as layered abstractions based on field calculus (middle), and current research directions on top of field calculus with respect to challenges in coordination models and languages (right).

the past several decades, we have moved from many people sharing a single computer to a computer for each person to many, mostly embedded and minimal-interface computing devices for each person. The only way to effectively engineer and coordinate the operation of such systems is to program and operate in terms of *aggregates* of devices rather than attempting to micro-manage each individual device. Moreover, as devices become more numerous, smaller, and more embedded, decentralisation brings new opportunities as well as new challenges—not only in terms of pervasive sensing/actuation/computation abilities, but also of increasing advantages in resilience, efficiency, and privacy.

Aggregate computing is an emerging approach, developed significantly within the coordination models and languages research community, that embraces this environment, building from a foundation on the *field calculus*, a functional programming model for the specification and composition of collective behaviours with formally equivalent local and aggregate semantics. Atop this foundation, a layered approach to engineering coordination of complex distributed systems has been constructed, first considering challenges of resilience, then pragmatism in the form of reusable libraries capturing common coordination patterns, and finally applications across a number of different domains. As the research on aggregate computing is becoming rather multi-faceted, we also envision a variety of research directions of high importance for distributed systems and specifically for coordination models and languages, both in theory, engineering methods and tools, and applications.

In this survey, we present a discussion of the past, present, and future of aggregate computing (Fig. 1). Section 2 begins by tracing the development of aggregate computing through its antecedents both in coordination research and in other areas, culminating in development of the field calculus. Section 3 then discusses the current state of aggregate computing theory and practice across its various abstraction layers. Finally, Sect. 4 presents a roadmap of current research directions on top of field calculus and with respect to challenges in coordination models and languages, and Sect. 5 summarises and concludes.



## 2 Coordination, Self-organisation, and Fields

In this section we review and discuss the conceptual, but also technical and technological, path that brought traditional coordination models for parallel computing, step-by-step to address the complexity of self-organising, large-scale deployed systems (Sect. 2.1). Then, we describe the emergence of field-based coordination (Sect. 2.2), and how, through the interaction with research works falling under the umbrella of space-based computation models (Sect. 2.3), this path ended up in the field calculus as discussed in next section.

### 2.1 Coordination Towards Self-organisation

**Generative Communication.** Coordination models are rooted into the idea that interaction among multiple, independent, and autonomous software systems (processes, components, somewhat generically called *agents* henceforth) could be conceived and designed as a space orthogonal to pure computation. Historically, many coordination models reify this idea into a concept of shared data space, working as a whiteboard, where processes of a parallel computing system can write and read information [36], enabling so-called *generative communication*. Linda [52] is universally recognised as the ancestor of a number of approaches to generative communication falling under the umbrella of tuple-based coordination models. The foundational idea of Linda was to have processes (on a centralised system) share information by writing and retrieving, with a suspensive semantics (the requestor is blocked until the query is satisfiable), data in form of ordered collection of possibly-heterogeneous knowledge chunks, i.e., tuples, from a shared (tuple-)space. Such data could be retrieved associatively, by querying through partial representations of the structure and content matching the desired piece of data (tuple template). The consequence is twofold: (i) decoupling in communication is strongly promoted, since no information about the sender, the space itself, and the tuple insertion time is required in order for communication to happen; and (ii) coordination is still possible in environments where information is vague, incomplete, inaccurate, or not entirely specified, due to the possibility to synchronise over a partial representation of knowledge.

**Programmable Coordination Rules.** The vision of tuple-based coordination as a shared knowledge repository used for agent coordination is further promoted by logic tuple-space models, where software agents coordinate through first-order tuples, and tuple spaces can be programmed as first-order logic theories. A prominent example of such approach is Shared Prolog [23], a framework for writing multi-processor Prolog systems. More generally, this view promotes the idea of equipping the shared space with some form of “intelligence”, i.e., in the form of an application logic that can manipulate data in the shared space and the way it can be accessed. Several Linda-inspired approaches tackle this issue by enabling programmability at the tuple-space level in order to express rules of coordination, and hence, pushing forward a notion of expressiveness of the

coordination media [24]. Among them, we find Law-Governed Interaction [65], MARS [26] and ReSpecT [71].

**Distribution.** All these approaches, however, do not explicitly focus on distributed systems, but on the coordination of centralised local components. As software components get spread across the system topology, so multiple tuple spaces can be distributed across the system environment, enabling distributed coordination abstractions, featuring mechanisms for event-based interactions, timing, and advanced data representation. This is the case with industrial systems like JavaSpaces [51] and TSpaces [97]. Lime [67], Klaim [44], and LogOp [63], take the approach a step further, by allowing to express the dynamic environment topology in a distributed setting, thus paving the way towards application of coordination models to pervasive computing system scenarios.

**Self-organising Coordination.** As coordination abstractions of various sorts (tuple spaces, channels, coordination artifacts [72,93]) are available in the distributed settings, one is directly faced with the problem of dealing with openness (hence, unexpectedness of environment changes, faults, and interactions), large-scale (possibly a huge number of agents and coordination abstractions to be managed), and intrinsic adaptiveness (as the ability of intercepting relevant events, and react to them so as to guarantee certain levels of overall system resilience). This calls for an approach of *self-organising coordination* [89], where coordination abstractions handle “local” interactions only (and typically use stochastic mechanisms to keep the coordination process always “up and running”), such that global and robust patterns of correct coordination behaviour can emerge—achieved by trading off by-design adaptiveness with inherent, automatic one.

Coordination models following this approach typically take their inspiration from complex natural systems (from physics through chemistry all the way to ethology) and reuse their foundational mechanisms. A primary source of inspiration for these systems is to be found in biology (social animals, and insects in particular), whose foraging techniques inspire the mechanisms that regulate coordination [27,78,80]. For instance, SwarmLinda [80] is a tuple-based middleware that brings the collective intelligence displayed by swarms of ants to computational mechanisms to guarantee efficient retrieval. Tuples are handled as sort of pheromones or items that ants (agents) continuously and opportunistically relocate. Chemical-inspiration is used in [87,88] to regulate the “activity level” of tuples, which drives the likelihood of their retrieval as well as their propagation rate. Ecology-inspiration is instead used in [81] to inject competition, composition, and disposal behaviour in the context of coordination of pervasive computing services.

## 2.2 Field-Based Coordination

Another important natural source of inspiration comes from physics: physics-inspired self-organising coordination systems rely on the notion of “field”

(gravitational field, electromagnetic field), which essentially provides a framework to handle (create, manipulate, combine) global-level, distributed data structures.

A notion of *coordination field* (or co-field) was initially proposed in [62] as a means to support self-organisation patterns of agent movement in complex environments: it was used as an abstraction over the actual environment, spread by both agents and the environment itself, and used by agents (which can locally perceive the value of fields) to properly navigate the environment. Based on this idea, the TOTA (Tuples On The Air) tuple-based middleware [61] was proposed to support field-based coordination for pervasive-computing applications. In TOTA each tuple, when inserted into a node of the network, is equipped with a content (the tuple data), a diffusion rule (the policy by which the tuple has to be cloned and diffused around) and a maintenance rule (the policy whereby the tuple should evolve due to events or time elapsing).

The *evolving tuples* model, presented in [79], is an extension to traditional Linda tuple spaces with the goal of supporting resource discovery in a pervasive system, relying on ideas inspired by TOTA. Evolution is firstly embedded in tuples by adding, to each field of the tuple, a name and a formula that specifies the field behaviour over time. Formulas support the if-then-else construct and arithmetic and boolean operators. Secondly, a new operation `evolve()` is introduced in the tuple space, which is responsible for applying formulas to tuples using contextual information.

One of the first works connecting field-based coordination with formalisation tools typical of coordination models and languages (i.e., process algebras and transition systems) is the  $\sigma\tau$ -Linda model [94], where agents can inject into the space “processes” that spread, collect and decay tuples, ultimately sustaining fields of tuples.

### 2.3 Spatial Computing Approaches: Towards the Field Calculus

More or less independently from the problem of finding suitable coordination models for distributed and situated systems, a number of works addressed similar problems in the more general attempt of building distributed intelligent systems by promoting higher abstractions of spatial collective adaptive systems. Works such as [14, 46, 64, 73] survey from various different viewpoints the many approaches that fall under this umbrella (including also some of the above mentioned coordination models), and which mainly organise in the following categories: methods that simplify programming of a collective by abstracting individual networked devices (e.g., SCEL [45], Hood [96], Butera’s “paintable computing” [25], and Meld [1]), spatial patterns and languages (e.g., Growing Point Language [35], geometric patterns in Origami Shape Language [68], self-healing geometries [34], or universal patterns [98]), tools to summarise and stream information over regions of space and time (e.g., TinyDB [60], Cougar [99], TinyLime [37], and Regiment [69]), and finally space-time computing models aiming at the manipulation of data structures diffused in space and evolving with time, e.g. targeting parallel computing (e.g., StarLisp [57], systolic computing

[48]) and topological computing (e.g., MGS [53,54]). Among them, space-time computing models based on the notion of computational fields were initially proposed in [12] and implemented in the Proto language. Combining techniques coming from the above approaches and generalising over Proto (which can be considered the archetypal spatial computing language due to its expressiveness and versatility), the field calculus has been proposed as a foundational model for the coordination of computational devices spread in physical environments.

### 3 From Field Calculus to Aggregate Computing

In this section, we discuss the current state of the art in aggregate computing, with the goal of presenting the full spectrum of results achieved without going into deep technical details—the reader can access code examples and tutorials, as well as formalisation of semantics, from the references provided. We begin with a review of its mathematical core in field calculus (Sect. 3.1), then discuss the construction of implementations of field calculus as the domain specific language Protelis (Sect. 3.2) and Scala support SCAFI (Sect. 3.3). Finally, we discuss the layered abstractions of aggregate programming built upon these foundations, from resilient operators to pragmatic libraries (Sect. 3.4).

#### 3.1 Field Calculus

**Basic Calculus.** The field calculus (FC) has been proposed in [92] as a minimal core calculus meant to capture the key ingredients of languages that make use of computational fields:<sup>1</sup> functional composition of fields, functions over fields, evolution of fields over time, construction of fields of values from neighbours, and restriction of a field computation to a sub-region of the network.

The field calculus is based on the idea of specifying aggregate system behaviour of a network of devices (where a dynamic neighbouring relation represents physical or logical proximity) by a functional composition of operators that manipulate (evolve, combine, restrict) computational fields. A key feature of the approach is that a specification can be interpreted locally or globally. Locally, it can be seen as describing a computation on an individual device, iteratively executed in asynchronous “computation rounds” comprising: reception of messages from neighbours, perception of contextual information through sensors, storing local state of computation, computing the local value of fields and spreading messages to neighbours. Globally, a field calculus expression  $e$  specifies a mapping (i.e., the computational field) associating each computation round of each device to the value that  $e$  assumes on that space-time event. This duality intrinsically supports the reconciliation between the local behaviour of each device and the emerging global behaviour of the whole network of devices [41,92].

The distinguished interaction model of this approach has been first formalised in [92] by means of a small-step operational semantics modelling single device

---

<sup>1</sup> Much as  $\lambda$ -calculus [32] captures the essence of functional computation and FJ [55] the essence of class-based object-oriented programming.

```

// distance from source region with nbrRange metric
def distanceTo(source) {
  rep (Infinity) { (dist) =>
    mux ( source, 0, minHood(nbr{dist} + nbrRange()) )
  }
}
// distance from source region, avoiding obstacle region
def distanceToWithObs(source, obstacle) {
  if (obstacle) { Infinity }{ distanceTo(source) }
}
// main expression
distanceWithObs(deviceId == 0, senseObs())

```

**Fig. 2.** Example field calculus code

computation (which is ultimately responsible for the whole network execution). The main technical novelty in this formalisation is that device state and message content are represented in an unified way as an annotated evaluation tree. Field construction, propagation, and restriction are then supported by local evaluation “against” the evaluation trees received from neighbours. Accessing these values is allowed by two specialised constructs:

- $\text{rep}(e_0)\{(x)\Rightarrow e\}$  which retrieves the value  $v$  computed for the whole  $\text{rep}$  expression in the last evaluation round (the value produced by evaluating the expression  $e_0$  is used at the first evaluation round) and updates it by the value produced by evaluating the expression obtained from  $e$  by replacing the occurrences of  $x$  by  $v$ ;
- $\text{nbr}\{e\}$ , which gathers the values computed by neighbours for expression  $e$  (from the respective evaluation trees) in their last round of computation into a *neighbouring field value*, which is a map from neighbour device identifiers to their correspondent values.

These constructs are backed by a data gathering mechanism accomplished through a process called *alignment*, which ensures appropriate message matching, i.e., that no two different  $\text{nbr}$  expressions can inadvertently “swap” their respective messages. This has the notable consequence that the two branches of an  $\text{if}$  statement in field calculus are executed in isolation: devices computing the “then” branch cannot communicate with a device computing the “else” branch, and viceversa.

Consider as an example Fig. 2. Function `distanceTo` takes as argument a field of booleans `source`, associating true to *source nodes*, and produces as result a field of reals, mapping each device to its minimum distance to a source node, computing relaxation of triangle inequality; namely: repetitively, and starting from infinity (construct  $\text{rep}$ ) everywhere, the distance on any node gets updated to 0 on source nodes (function  $\text{mux}(c, t, e)$  is a purely functional multiplexer which chooses  $t$  if  $c$  is true, or  $e$  otherwise), and elsewhere to the minimum (built-in `minHood`) of neighbours’ distance (construct  $\text{nbr}$ ) added with `nbrRange`,

a sensor for estimated distances. Function `distanceToWithObs` takes an additional argument, a field of booleans `obstacle`, associating true to *obstacle nodes*; it partitions the space of devices: on obstacle nodes it gives the field of infinity values, elsewhere it reuses computation of `distanceTo`. Because of alignment, the set of considered neighbours for `distanceTo` automatically discards nodes that evaluate the other branch of `if`, effectively making computation of distances circumvent obstacles. Finally, the main expression calls `distanceToWithObs` to compute distances from the node with `id` equal to 0, circumventing the devices where `senseObs` gives true.

The work in [41] (which is an extended and revised version of [92]) presents a type system, used to intercept ill-formed field-calculus programs. The type system, which builds on the HindleyMilner type system [39] for ML-like functional languages, is specified by a set of syntax-directed type inference rules. Being syntax-directed, the rules straightforwardly specify a variant of the Hindley-Milner type inference algorithm [39]. Namely, an algorithm that given a field calculus expression and type assumptions for its free variables: either fails (if the expression cannot be typed under the given type assumptions) or returns its principal type, i.e., a type such that all the types that can be assigned to an expression by the type inference rules can be obtained from the principal type by substituting type variables with types.

Types are partitioned in two sets: *types for expressions* and *types for functions* (built-in operators and user-defined functions)—this reflects the fact that the field calculus does not support higher order functions (i.e., functions are not values). Expression types are further partitioned in two sets: types for *local values* (e.g., the values produced by numerical literals) and types for *neighbouring field values* (e.g., the values produced by `nbr`-expressions).

The type system is proved to guarantee the following two properties:

- *Domain alignment*: On each device, the domain of every neighbouring field value arising during the reduction of a well-typed expression consists of the identifiers of the aligned neighbours and of the identifier of the device itself. In other words, information sharing is scoped to precisely implement the aggregate abstraction.
- *Type soundness*: The reduction of a well-typed expression terminates.

**Higher-Order Field Calculus.** The higher-order field calculus (HFC) [42] (see also [84]) is an extension of the field calculus with first-class functions. Its primary goal is to allow programmers to handle functions just like any other value, so that code can be dynamically injected, moved, and executed in network (sub)domains. Namely, in HFC:

- Functions can take functions as arguments and return a function as result (higher-order functions). This is key to define highly reusable building block functions, which can then be fully parameterised with various functional strategies.

- Functions can be created “on the fly” (anonymous functions). Among other applications, such functions can be passed into a system from the external environment, as a field of functions considered as input coming from a sensor modelling addition of new code into a device while the system is operating.
- Functions can be moved between devices (via the `nbr` construct) and the function to be executed can change over time (via `rep` construct), which allows one to express complex patterns of code deployment across space and time.
- A field of functions (possibly created on the fly and then shared by movement to all devices) can be used as an aggregate function operating over a whole spatial domain.

In considering fields of function values, HFC takes the approach in which making a function call acts as a branch, with each function in the range of the field applied only on the subspace of devices that hold that function. When the field of functions is constant, this implicit branch reduces to be precisely equivalent to a standard function call. This means that we can view ordinary evaluation of a function name (or anonymous function) as equivalent to creating a function-valued field with a constant value, then making a function call applying that field to its argument fields. This elegant transformation is one of the key insight of HFC, enabling first-class functions to be implemented with relatively minimal complexity.

In [42] the operational semantics of HFC is formalised, for computation within a single device, by a big-step operational semantics where each expression evaluates to an ordered tree of values tracking the results of all evaluated subexpressions. Moreover, [42] also presents a formalisation of network evolution, by a transition system on network configurations—transitions can either be firings of a device or network configuration changes, while network configurations model environmental conditions (i.e., network topology and inputs of sensors on each device) and the overall status of devices in the network at a given time.

**Behavioural Properties.** Since HFC is designed as a general-purpose language for spatially distributed computations, its semantics and type system guarantees do not prevent the formulation of ill-behaving programs. Thus, regularity properties have been isolated and studied for subsets of the core language. Among them, the established notion of *self-stabilisation* to correct states for distributed systems [47, 58, 59] plays a central role. This notion, defined in terms of properties of the transition system of network evolution, ensures that both (i) the evaluation of a program on an eventually constant input converges to a limit value in each device in finite time; (ii) this limit only depends on the input values, and not on the transitory input values that may have happened before that. When applied in a dynamically evolving system, a self-stabilising algorithm guarantees that whenever the input changes, the output reacts accordingly without spurious influences from past values.

In [40] (which is an extended version of [91]), a first self-stabilising fragment is isolated through a *spreading* operator, which minimises neighbour

values as they are monotonically updated by a *diffusion* function. This pattern can be composed arbitrarily with local operations, but no explicit `rep` and `nbr` expressions are allowed: nonetheless, several building blocks can be expressed inside this fragment, such as classic distance estimation. However, more self-stabilising programs and existing “building block” implementations are covered by the larger self-stabilising fragment introduced in [83] (which is an extended version of [86]). This fragment restricts the usage of `rep` statements to three specific patterns (converging, acyclic and minimising `rep`), roughly corresponding to the three main building blocks (time evolution, aggregation, distance estimation). Furthermore, a notion of *equivalence* and *substitutability* for self-stabilising programs is examined: on the one hand, this notion allows for practical optimisation of distributed programs by substitution of routines with equivalent but better-performing alternatives; on the other hand, this equivalence relation naturally induces a *limit* viewpoint for self-stabilising programs, complementing and integrating the two general (local and global) viewpoints by abstracting away the transitory characteristics and isolating the input-output mapping corresponding to the distributed algorithm. These viewpoints effectively constitute different semantic interpretations of a same program: operational semantics (local viewpoint), denotational semantics (global viewpoint), and eventual behaviour (limit viewpoint).

A fourth “continuous” viewpoint is considered in [20]: as the density of computing devices in a given area increases, assuming that each device takes inputs from a single continuous function on a space-time manifold, the output values may converge towards a limit continuous output. Programs with this property are called *consistent*, and have a “continuous” semantic interpretation as a transformation of continuous functions on space-time manifolds. Taking inspiration from self-stabilisation, this notion is relaxed for *eventually consistent* programs, which are only required to continuously converge to a limit except for a transitory initial part, *provided* that the inputs are constant (except for a transitory initial part). Eventual consistency can then be proved for all programs expressible in the GPI (gradient-following path integral) calculus, that is a restriction of the field calculus where the only coordination mechanism allowed is the GPI operator, a generalised variant of the distance estimation building block.

Up to this point, hence, validation of behavioural properties is mostly addressed “by construction”, namely, proving properties on simple building blocks or restricting the calculus to fragments. It is a future work to consider the applicability of techniques such as the formal basis in [59], or model-based analysis such as [7].

### 3.2 Protelis: A DSL for Field Calculus

The concrete usage of HFC in application development is conditioned by the availability of practical languages, embedding an interpreter or compiler, as well as handling runtime aspects such as communication, interfacing with the operating system, and integration with existing software. Protelis [77] provides one such implementation, including: (i) a concrete HFC syntax; (ii) an interpreter



and a virtual machine; (iii) a device interface abstraction and API; and (iv) a communication interface abstraction and API.

In Protelis, the parser translates a Protelis source code file into a valid representation of the HFC semantics. Then, the program, along with an execution context, is fed to the virtual machine that executes the Protelis interpreter at regular intervals. The execution context API defines the interface towards the operating system, including (with ancillary APIs) an abstraction of the device capabilities and the communication system. This architecture has been proven to make the language easy to port across diverse contexts, both simulated (Alchemist [76] and NASA World Wind [21]) and real-world [33].

The entire Protelis infrastructure is developed in Java and hosted on the Java Virtual Machine (JVM). The motivation behind one such choice is twofold: first, the JVM is highly portable, being available on a variety of architectures and operating systems; second, the Java world is rich in libraries that can be directly used within Protelis, with little or no need of writing new libraries for common tasks.

The model-to-model translation between the Protelis syntax and the HFC interpreter is operated by leveraging the Xtext framework [22]. Along the parser machinery, this framework is able to generate most of the code required for implementing Eclipse plug-ins: one such plug-in is available for Protelis, assisting the developer through code highlighting, completion suggestions, and early error detection.

The language syntax is designed with the idea of lowering the learning curve for the majority of developers, and as such it is inspired by languages of the C-family (C, C++, Java, C#...), with some details borrowed from Python. Code can be organised in modules (or namespaces) whose name must reflect the directory structure and the file name. Modules can contain functions and a main script. The code snippet in Fig. 3 offers a panorama on the ordinary and field-calculus specific features of Protelis, including the ability of importing libraries and static methods, using functions as higher-order values in `let` constructs and by `apply`, tuple and string literals, lambdas, built-ins (e.g., `minHood`, and `mux`), and field calculus constructs `rep` and `nbr`.

Function definitions are prefixed by the `def` keyword, and they are visible by default only in the local module. In order for other modules to access them, the keyword `public` must be explicitly specified. Other modules can be imported, as well as Java static methods. Types are not specified explicitly: in fact, Protelis is duck-typed—namely, type-checked at run-time through reflection mechanisms. The language offers literals for commonly used numeric values, tuples, and strings. Instance methods can be invoked on any expression with the same “dot” syntax used in Java. Higher order support includes a compact syntax for lambda expressions, closures, function references, functions as parameters and function application. Lastly, context properties, including device capabilities, are accessible through the `self` keyword. Environment variables can be accessed via the short syntax `env`.

```

import protelis:coord:spreading // Import other modules
import java.lang.Math.sqrt // Import static Java methods
def privateFun(my, params) {
  my + params // Infix operators, duck typing
}
public def availableOutside() { // externally visible
  privateFun(1, 2); // Function call
  let aFun = privateFun; // Variable definition, function ref
  aFun.apply("a", "str"); // String literals, application
  let tup = [NaN, pi, e]; // Tuple literals, built-in numbers
  // lambda expressions, closures, method invocation:
  let inc3 = v -> {privateFun(v, tup.size())}
}
// MAIN SCRIPT
let myid = self.getDeviceUID(); // Access to device info
if (myid < 1000) { // Domain separation
  rep (x <- self.nextRandomDouble()) { // Stateful computation
    // Java static method call
    mux (sqrt(x) < 0.5) { // mux executes both branches
      // Library call, field gathering and reduction
      minHood(nbr(env.has("source")))
    } else { Infinity }
  } < 10
} else { // Mandatory else: every expression returns a value
  false // booleans
}

```

**Fig. 3.** Example Protelis code showcasing detailed syntactic aspects

A relevant asset of Protelis is its recently developed library “protelis-lang” [50], streamlining the implementation of several algorithms found in literature devoted to development of distributed systems. Among others, it includes several implementations of self-stabilising building functions [18,83], such as `distanceTo` to estimate distances, `broadcast` to send alerts, `summarize` to perform distributed sensing, and so on. Notably, the library also includes machinery for “aligning” aggregate computing programs along arbitrary keys, separating and mixing domains in a finer way than the `if` construct allows. These constructs, based on the `alignedMap` primitive of Protelis, enable highly dynamic meta-algorithms to be written, that open to new possibilities such as `multiInstance` [50], or allow for increased resilience and adaptation as in the case of `timeReplicated` [75].

### 3.3 SCAFI: An API for the Scala Programming Ecosystem

From a pragmatism viewpoint, it is highly desirable to bridge the gap between field calculus-based DSLs and mainstream programming platforms and languages that embody, among others, the functional, object-oriented, and

actor-based paradigms (i.e., the nowadays reference styles for in-the-small, in-the-large, and concurrent/distributed programming, respectively). Indeed, this can be critical to foster adoption, reducing accidental complexity through coherent syntax, semantics, and toolset, and paving the way to a more integrated programming experience.

External DSLs such as Protelis, despite the aid provided by DSL frameworks like Xtext [22], can require a lot of development and maintenance effort, since they must cover aspects ranging from language design to typing, and proper tooling must be provided to enable full interoperability with the target platform in static, runtime, and debugging contexts. By contrast, internal DSLs are an interesting alternative, for they are expressed in the host language and are de facto equivalent to an API: they more seamlessly interoperate, and reuse the syntax, semantics, typing, and tools of their host language, at the expense of reduced flexibility due to the constraints exerted by the host environment.

Such considerations of pragmatism, reuse, and interoperability motivate SCAFI (Scala Fields) [30], an aggregate computing framework including a field-calculus DSL internal to the Scala programming language [70], also integrated into the Alchemist meta-simulator [29], as well as an actor-based platform for distributed aggregate systems [31,90]. The choice of Scala as the host language was inspired by its *(i)* interoperability across the JVM platform, *(ii)* seamless integration of the object-oriented and functional paradigms, with support for lightweight component-based programming (cf., traits and self-types), *(iii)* advanced features for type-safe library development (cf., implicits, generic type constraints), *(iv)* syntax flexibility and sugar (cf., by-name arguments), allowing to create fluent DSL-like APIs; and *(v)* prominent role in the scene of distributed computing frameworks (cf., Akka, Kafka, Spark). Concerning the platform perspective, instead, the use of actor-based abstractions is instrumental to the integration of aggregate-level functionality into existing distributed systems (e.g., developed with more traditional techniques), by exposing collective coordination events and data through message or event-like interfaces [31].

Working with a general-purpose, multi-paradigm programming language like Scala gives to the hands of developers quite a lot of flexibility and power for what concerns design and implementation of field libraries and programs. Consider the example in Fig. 4 for a taste of the programming style, including definition of a reusable block `G` (extending distance calculation [15,83]), type-class-style assumptions on arguments via context bound “[`V: Bounded`]”, tuples by syntax `(.,.)`, and pattern matching (`case .. =>`).

An `AggregateProgram` instance acts simply as a function from an abstract `Context` to an `Export`. Hence, for a platform to support local execution of field computations it is just a matter of instantiating an aggregate program (possibly mixing in components to provide access to platform-level functionality), preparing contextual information (i.e., previous state, sensor data, and messages from neighbours), and running a computation round according to the device lifecycle.

```

trait BlockG { // Component
  self: FieldCalculus with StandardSensors => // Dependencies

  // Generic function with type-class constraint on V
  def G[V: Bounded](source: Boolean,
                    field: V,
                    acc: V => V,      // Function type
                    metric: => Double // By-name parameter
                    ): V =           // Return type
    rep((Double.MaxValue, field)) {
      case (dist, value) => // Function by pattern matching
        mux(source) {
          (0.0, field) // Tuple syntax sugar for Tuple2(_,_)
        }{
          minHoodPlus { // Requires (Double,V) to be Bounded
            (nbr { dist } + metric, acc(nbr { value })))
          }
        }
    }._2 // Selects 2nd element of tuple
}

class Program extends AggregateProgram
  with StandardSensors with BlockG { // Mixins
  def main: Double = // Program entry point
    distanceTo(isSource)

  def isSource = sense[Boolean]("source")

  def distanceTo(source: Boolean): Double =
    G(source, 0.0, _ + nbrRange, nbrRange)
}

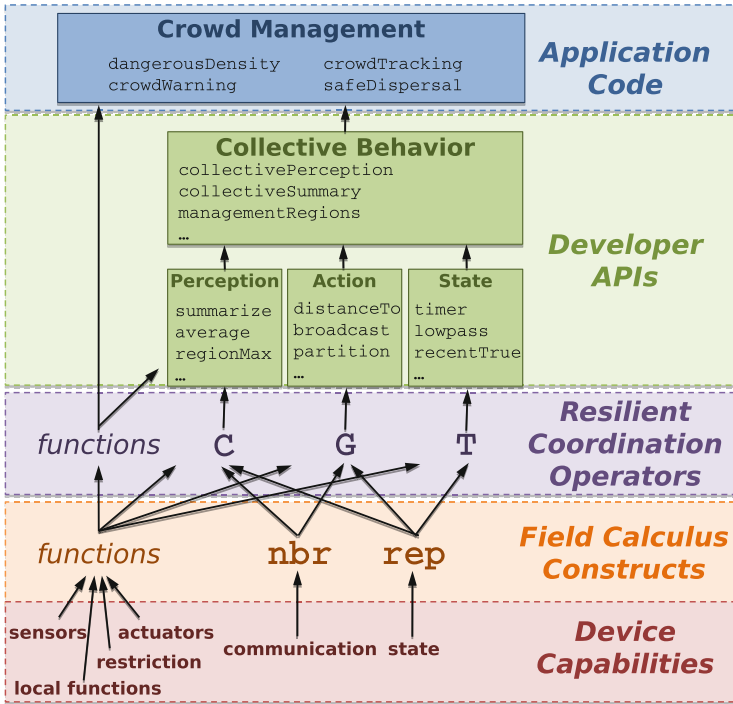
```

Fig. 4. Example SCAFI code

### 3.4 Aggregate Programming

Building upon these theoretical and pragmatic foundations, aggregate programming [15] elaborates a layered architecture that aims to dramatically simplify the design, creation, and maintenance of complex distributed systems. This approach is motivated by three key observations about engineering complex coordination patterns:

- composition of modules and subsystems must be simple and transparent;
- different subsystems need different coordination mechanisms for different regions and times;
- mechanisms for robust coordination should be hidden “under the hood”, where programmers are not required to interact with them.



**Fig. 5.** Aggregate programming abstraction layers. The software and hardware capabilities of particular devices are used to implement aggregate-level field calculus constructs. These constructs are used to implement a limited set of building-block coordination operations with provable resilience properties, which are then wrapped and combined together to produce a user-friendly API for developing situated IoT (Internet-of-Things) systems—picture adapted from [15].

Field calculus (and its language incarnations) provides mechanisms for the first two, but is too general to guarantee resilience and too mathematical and succinct in its syntax for direct programming to be simple.

Aggregate programming thus proposes two additional abstraction layers, as illustrated in Fig. 5, for hiding the complexity of distributed coordination in complex networked environments. First, the “resilient coordination operators” layer plays a crucial role both in hiding the complexity and in supporting efficient engineering of distributed coordination systems. First proposed in [18], it is inspired by the approach of combinatory logic [38], the catalogue of self-organisation primitives in [49], and work on self-stabilising fragments of the field calculus [40, 83, 91]. Notably, three key operators within this self-stabilising fragment cover a broad range of distributed coordination patterns: operator *G* is a highly general information spreading and “outward computation” operation, *C* is its inverse, a general information collection operation, and *T* implements bounded state evolution and short-term memory.

Above the resilience layer, aggregate programming libraries [50,86] capture common patterns of usage and more specialised and efficient variants of resilient operators to provide a more user-friendly interface for programming. This definition of well-organised layers of abstractions with predictable compositional semantics thus aims to foster (i) *reusability*, through generic components; (ii) *productivity*, through application-specific components; (iii) *declarativity*, through high-level functionality and patterns; (iv) *flexibility*, through low-level and fine-grained functions; and (v) *efficiency*, through multiple components with coherent substitution semantics [83,86].

Within these two layers, development has progressed from an initial model built only around the spreading of information to a growing system of composable operators and variants. The first of these operator/variant families to be developed centred around the problems of spreading information, since interaction in aggregate computing is often structured in terms of information flowing through collectives of devices. A major problem thus lies in regulating such spreading, in order to take into account context variation, and in rapidly adapting the spreading structure in reaction to changes in the environment and in the system topology. Here, the gradient (i.e., the field of minimum distances from source nodes) in its generalised form in the  $\mathbf{G}$  operator is what captures, in a distributed way, a notion of “contextual distance” instrumental to calculate information diffusion, and forms the basis for key interaction patterns, such as outward/inward bounded broadcasts and dynamic group formation, as well as higher-level components built upon these.

The widespread adoption of gradient structures in algorithms stresses the importance of fast self-healing gradients [13], which are able to quickly recover good distance estimates after disruptive perturbations, and more “dependable” gradient algorithms in which stability is favoured by enacting a smoother self-healing behaviour [8]. Several alternative gradient algorithms have been developed, addressing two main issues. Firstly, the recovery speed after an input discontinuity, which has first been bounded to  $O(\text{diameter})$  time by CRF (constraint and restoring force) gradient [13], further improved to optimal for algorithms with a single-path communication pattern by BIS (bounded information speed) gradient [5], and refined to optimality for algorithms with a multi-path communication pattern by SVD (stale values detection) gradient [3]. Secondly, the smoothness and resilience to noise in inputs, first addressed by FLEX (flexible) gradient [8] and then refined and combined with improved recovery speed by ULT (ultimate) gradient [3].

To empower the aggregate programming tool-chain, other building blocks have been proposed and refined besides from gradients: consensus algorithms [11], centrality measures [4], leader election and partitioning [18], and most notably, *collection*. The collection building block  $\mathbf{C}$  progressively aggregates and summarises values spread throughout a network into a single value, e.g., the sum or other meaningful statistics. Based itself on distance estimation through gradients, a general *single-path* collection algorithm has been proposed in [18] granting self-stabilisation to a correct value, then *multi-path* collection has been

developed for improved resiliency in sum estimations [83], and finally refined to *weighted multi-path* collection [2] which is able to maintain acceptable whole network sums even in highly volatile environments. A different approach to collection has also proved to be effective for *minimum/maximum* estimates: overlapping replicas of non-self-stabilising *gossip* algorithms [75] (with an appropriately tuned interval of replication), thus combining the resiliency of these algorithms with self-stabilisation requirements.

## 4 Perspectives and Roadmap

Over the past decade, aggregate computing has moved from a fragmented collection of ideas and tools to a stable core calculus and a coherent layered framework for the engineering of distributed systems. Thus, even as the underlying theory continues to be developed, as shown in [85], a significant portion of research and development can shift to more pragmatic issues linked to applications and higher levels of the aggregate computing stack. In this section, we review a number of such research directions, which include elaboration of libraries (Sect. 4.1), techniques to control dynamics (Sect. 4.2), management of mobile devices and processes (Sect. 4.3), development of software platforms (Sect. 4.4), security (Sect. 4.5), and applications (Sect. 4.6).

### 4.1 Elaboration of Libraries

The most immediate and incremental line of future development for aggregate computing is the elaboration of the existing collection of libraries, to form a more broadly applicable and easier to use interface at the top of the aggregate computing stack. Some of these additions and refinements will be based on development of alternative implementations of core resilient building blocks (e.g., [2, 5, 75]), while others are expected to capturing common design patterns and necessary functionalities specific to particular application domains. No particular high-priority targets are suggested at present for this development, however. Instead, this process is expected to be a natural incremental progress of ongoing maturation and professionalisation driven by issues discovered as the other lines of future development outlined below exercise the existing libraries to expose their current shortcomings and needs for enhancement.

### 4.2 Understanding and Controlling Dynamics and Feedback

Much of the work to date on aggregate computing has focused on the converged properties of a system, such as self-stabilisation [47, 82] and eventual consistency [20]. These theoretical approaches, however, assume that the network of devices is often in a persistent quasi-stable state in which the set of devices, their connections to one another, and their environment all do not change for a significant length of time. In large scale systems, however, such quasi-stable states are typically rare and short-lived: there is almost always something changing with

respect to some device, thus constantly injecting perturbations into the system. Prior compositional safety analysis regarding self-stabilisation and eventual consistency also does not apply in the case of systems involving feedback, and many applications do require feedback either directly between building blocks or indirectly via interactions with the environment.

The control theory literature has many well-developed tools for analysing the response of complex systems under perturbation and in the presence of feedback. The mathematical frameworks for such tools are not straightforward to adapt for application to aggregate computing building blocks, but with careful work may often still be applied, e.g., through identification of appropriate Lyapunov functions to bound the convergence behaviour of a building block. Early work in this area shows promise, enabling analysis and prediction of aggregate computing systems with feedback between building blocks [56] and providing stability analysis and tight convergence bounds for particular applications of the **G** operator [43] and **C** operator [66]. An important area for future development is thus to expand these results to cover a large sublanguage of aggregate computing systems and to apply them in order to refine and improve the dynamical performance of building blocks.

### 4.3 Mobility of Devices and Processes

Another key area for expansion of aggregate computing, both in theory and practice, is better handling of mobility, both of devices and of processes dispersed through networks of devices. From a theoretical perspective, this is closely interwoven with the need for a deeper understanding of convergence dynamics, as systems with mobile devices or processes typically do not ever achieve the quasi-stable states required for self-stabilisation to hold. Instead, work to date has depended on the informal observation that “slow enough” mobility does not disrupt commonly used self-stabilising building blocks. Theoretical work is needed to predict and bound regions of stability and effects of perturbation, as well as to develop improved building block alternatives for conditions where the identified dynamics are unsatisfactory.

There is also a need to expand the existing building block libraries to support applications involving mobility. For controlling the physical motion of devices, a number of building blocks have been demonstrated or proposed throughout the swarm robotics and multi-agent systems literature, including a number already formulated as building blocks for aggregate computing (e.g., [6,9,10]). We may also consider systems in which the device is not the focus of mobility, but instead code and processes dynamically deploy, migrate, upgrade, and terminate during system operation, as considered for example in [15,95]. To effectively support mobility in aggregate computing, the large volume of prior work on algorithms and strategies for such systems needs to be systematised and organised, analysed for compositional safety and bounds on convergence, and adapted for use in aggregate computing based on the results of analysis.



#### 4.4 Software Platforms

Aggregate computing targets a number of application scenarios, generally characterised by inherent distribution, heterogeneity, mobility and lack of stable infrastructure (including computation, storage, and networking media). Hence, proper middleware or software platform is paramount to ease the development and deployment of applications as well as support their management at runtime [90]. Moreover, such a layer is the ideal place where to encapsulate cross-cutting concerns such as security, privacy, monitoring, fault tolerance and so on.

Though the problem of a middleware is common to almost any distributed computing effort, there are some issues (e.g., those discussed in this section, like mobility and control) and opportunities specifically related to aggregate computing and coordination that deserve attention. In particular, consider the aggregate programming model: it achieves a certain degree of declarativity by abstracting over a number of details such as, for instance, the specifics of neighbourhood-based communication and the order and frequency of micro-level activities sustaining application execution—details that can be delegated to corresponding platform services for topology management, scheduling and round execution. This abstraction provides a lot of flexibility on the platform side, which is free to apply optimisations of various sorts, from simpler (e.g., avoid broadcasting redundant messages) to more complex ones. In fact, the most relevant insight here is the ability of running aggregate computing systems according to different execution strategies [90], from fully peer-to-peer, where end-devices directly communicate between one another and run by themselves their piece of aggregate logic, to completely centralised solutions where, instead, end-devices act only as managers for sensors and actuators, sending perceptions upstream to one or more servers which run computations on their behalf and ultimately propagate actuation data downstream.

Crucially, this flexibility paves the way towards an opportunistic and QoS-driven exploitation of available infrastructural resources, as well as to intrinsic adaptation of application execution to forthcoming multi-layer architectures involving edge, fog, and cloud interfaces [90]—as required to deal with emerging IoT scenarios. For instance, an aggregate system specification can be mapped to a system of actors [31] where each actor is responsible for a specific aspect of the overall computation and communication and can be migrated to different machines while preserving coordination by automatically adapting the bindings [90]. A lot of interesting future work is expected to be carried out in order to put such theory of adaptive execution coordination into practice.

#### 4.5 Security

Security is a critical concern in computer science in general and especially in open environments, such as those envisioned in pervasive computing and IoT scenarios involving vast numbers of devices administered by individuals and

organisations with no particular knowledge of security. This problem is multifaceted and requires carefully thought, full-stack solutions that also consider orthogonal issues such as, for instance, the cost of security-related computational tasks in resource-constrained devices.

A number of security issues, not strictly related to coordination, of prominent importance in real-world, trustworthy systems, can be addressed in the middleware layer and through proper deployment solutions. For example, support is needed to enable safe code mobility and execution, as proposed in [42], which may be required in scenarios characterised by significant dynamicity requirements or demands for automatic deployment of new functionality. Another key theme is confidentiality: privacy properties on the propagated and collected data need to be understood and guaranteed, otherwise participation may be hindered. Additionally, despite the decentralised and inherently scalable nature of aggregate systems, availability issues need to be considered, according to the specifics of applications, especially with respect to nodes playing a crucial role in algorithms (e.g., sources, hubs, collectors, region leaders).

Regarding application-level interaction, since coordination activity in aggregate computing is substantially based on a premise of cooperation between the participating entities, it is often sensitive to attacks that may trigger epidemic deviation. That is, what is the extent to which agents and their data can be trusted? In order to assess and mitigate the impact of voluntary or involuntary misbehaviour, adoption of computational trust has proven useful [28] and applicable even in decentralised settings, in which no central authority is available to certify recipients and endpoints, and in scenarios where seamless opportunistic interaction is the norm. Much work remains, however, to develop these initial proposals into a fully articulated theory and practice for the security of aggregate computing systems that takes into account confidentiality, integrity, availability, and authenticity issues.

#### 4.6 Applications and Pragmatics

Finally, the core goal all along for the aggregate computing research thrust has been to enable simpler, faster development of more resilient distributed applications. Having developed both its theoretical foundations and the layered system of algorithms and libraries exploiting those foundations, one of the major directions of current and future work is indeed to apply these developments to real-world problems across a variety of domains.

One key application area, previously discussed in [15] and other works, is pervasive or IoT scenarios in dense urban environments. As the density of communicating devices increases, their interactions put pressure on the available fixed infrastructure and the opportunities for local interaction increase. This is particularly acute during transient events when demand and the available infrastructure become mismatched, such as during festivals or sporting events when the number of people packed into an area spikes, or during natural disasters and other emergencies when the available infrastructure may be degraded. One of the

critical challenges in this area is simply to access the potential peer-to-peer capabilities of devices, which are often closed platforms and are currently typically configured primarily for asymmetrical communication with fixed infrastructure or individually connected personal networks. These constraints are both loosening over time as app infrastructures continue to spread and develop on many platforms. Finally, the benefits of distribution must be effectively balanced with tight energy budgets on many devices and the continuous value of non-local interactions enabled by cloud connections.

Another important emerging application area is control of drones and other unmanned vehicles, driven by the rapidly increasing availability of high-quality platforms at various levels of cost and capability. With the emergence of highly capable autopilots, the need for detailed human control is decreased and it becomes desirable to shift from the current typical practice of multiple people commanding a single platform toward a single person controlling many platforms. Aggregate computing is a natural fit for approaching multi-platform control, using paradigms such as those discussed in [6,9]. In implementation, however, the challenges of mobility become acute as one considers rapid physical movements. Likewise, a better understanding of convergence dynamics and feedback will be needed. Work in this space will also demand significant elaborations in aggregate computing libraries, adapting manoeuvres from the applicable literature and doctrine into additional composable building block components. Finally, there are also major pragmatic issues to be addressed in platform interfaces, including a plethora of standards, safety issues, and appropriate incorporation of resource and manoeuvring constraints.

Agent-based planning uses similar principles, computing plans for future actions over an aggregate of agents. This generalisation, however, typically also connects representations of future plans, tasks, goals, and environment into the aggregate [95], as some combination of additional virtual devices in the aggregate and virtual fields that devices can interact with. Examples include the poly-agent approach to modelling and planning [74] and agent-based sharing of airborne sensors [16,17]. When agent-based planning is centralised, managing projections and tasks is straightforward; when distributed across physical agents, however, there are important questions to be addressed regarding where projections and tasks should be hosted, to what degree they should be duplicated, and how to synchronise information between duplicates.

Aggregate computing can also be applied to more conventional networked systems. In this case, the links between neighbours are defined by (not particularly spatial) physical network connections, virtual network relationships such as in an overlay network, or else logical relationships such as interaction patterns between services. As long as the number of such neighbours is relatively constrained, such that sending regular updates to neighbours is not problematic, many of the same sorts of coordination approaches that work in other application areas can work in areas such as these as well. Examples of applications in this space include coordinating recovery operations for networks of enterprise services [33], coordinating a checkpoint-based “rewind and replay”

across interacting services to undo the effects of a cyber-attack [19], and integrating applications across intermittently connected distributed cloud nodes [19]. In this domain, in most cases it is not cost-effective to try to write or refactor entire services and applications into an aggregate computing paradigm. Instead, aggregate computing appears better used as meta-level coordination and control service, helping to determine things like when and where to migrate services across machines, how many instances of a service should be used, how to rendezvous between services that need to communicate, and so on. Future work in this space is thus likely to focus on extending libraries to better support various coordination paradigms, particularly with distributed graph algorithms for supporting coordination regarding dependencies and information flows, and on the pragmatics of interfacing with complex legacy applications.

In addition to the four presented here, aggregate computing offers potential value in many other application domains as well: it is likely to offer value in any domain with an increasing number and potential volatility in collections of devices capable of communicating locally. The ongoing continuation of miniaturisation and embedding of computational devices means this is likely to apply in most areas of human endeavour, to one degree or another. Across all such domains, just as in the four domains described in detail, it is likely to be the case that aggregate computing will not be the focus of the system but rather, much like any other specialised library, used as a modular component: and most specifically, as a component providing a *coordination service*. A critical challenge for the future, then, will be to continue shaping and improving libraries and interface patterns in response to the needs of these application domains, in order to allow aggregate computing to become as invisible as possible in the actual process of systems engineering.

## 5 Conclusions

Aggregate computing is a potentially powerful approach to the engineered distributed systems, emerging from the distillation of a wide variety of approaches to coordination into the field calculus. This mathematical core then serves as the basis for a layered approach to pragmatic development of composable and resilient distributed systems. The future of aggregate programming involves both continued development of its core theoretical tools as well as work to realise its potential across a wide range of important application domains.

## References

1. Ashley-Rollman, M.P., Goldstein, S.C., Lee, P., Mowry, T.C., Pillai, P.: Meld: a declarative approach to programming ensembles. In: International Conference on Intelligent Robots and Systems (IROS), pp. 2794–2800. IEEE (2007)
2. Audrito, G., Bergamini, S.: Resilient blocks for summarising distributed data. In: ALP4IoT Workshop, to Appear on EPTCS Online (2017)

3. Audrito, G., Casadei, R., Damiani, F., Viroli, M.: Compositional blocks for optimal self-healing gradients. In: 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pp. 91–100. IEEE (2017)
4. Audrito, G., Damiani, F., Viroli, M.: Aggregate graph statistics. In: ALP4IoT Workshop, to Appear on EPTCS Online (2017)
5. Audrito, G., Damiani, F., Viroli, M.: Optimally-self-healing distributed gradient structures through bounded information speed. In: Jacquet, J.-M., Massink, M. (eds.) COORDINATION 2017. LNCS, vol. 10319, pp. 59–77. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-59746-1\\_4](https://doi.org/10.1007/978-3-319-59746-1_4)
6. Bachrach, J., Beal, J., McLurkin, J.: Composable continuous-space programs for robotic swarms. *Neural Comput. Appl.* **19**(6), 825–847 (2010)
7. Bakhshi, R., Cloth, L., Fokkink, W., Haverkort, B.R.: Mean-field framework for performance evaluation of pushpull gossip protocols. *Perform. Eval.* **68**(2), 157–179 (2011). *Advances in Quantitative Evaluation of Systems*
8. Beal, J.: Flexible self-healing gradients. In: Symposium on Applied Computing, pp. 1197–1201. ACM (2009)
9. Beal, J.: A tactical command approach to human control of vehicle swarms. In: AAAI Fall Symposium: Human Control of Bioinspired Swarms (2012)
10. Beal, J.: Superdiffusive dispersion and mixing of swarms. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **10**(2), article 10 (2015)
11. Beal, J.: Trading accuracy for speed in approximate consensus. *Knowl. Eng. Rev.* **31**(4), 325–342 (2016)
12. Beal, J., Bachrach, J.: Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intell. Syst.* **21**, 10–19 (2006)
13. Beal, J., Bachrach, J., Vickery, D., Tobenkin, M.: Fast self-healing gradients. In: Symposium on Applied Computing, pp. 1969–1975. ACM (2008)
14. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, Chap. 16, pp. 436–501. IGI Global (2013). A longer version <http://arxiv.org/abs/1202.5509>
15. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. *IEEE Comput.* **48**(9), 22–30 (2015)
16. Beal, J., Usbeck, K., Loyall, J., Metzler, J.: Opportunistic sharing of airborne sensors. In: International Conference on Distributed Computing in Sensor Systems (DCOSS), pp. 25–32. IEEE (2016)
17. Beal, J., Usbeck, K., Loyall, J., Rowe, M., Metzler, J.: Adaptive task reallocation for airborne sensor sharing. In: International Workshops on Foundations and Applications of Self\* Systems (FAS\*W), pp. 168–173. IEEE (2016)
18. Beal, J., Viroli, M.: Building blocks for aggregate programming of self-organising applications. In: 8th International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW), pp. 8–13 (2014)
19. Beal, J., Viroli, M.: Aggregate programming: from foundations to applications. In: Bernardo, M., De Nicola, R., Hillston, J. (eds.) SFM 2016. LNCS, vol. 9700, pp. 233–260. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-34096-8\\_8](https://doi.org/10.1007/978-3-319-34096-8_8)
20. Beal, J., Viroli, M., Pianini, D., Damiani, F.: Self-adaptation to device distribution in the Internet of Things. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **12**(3), 12 (2017)
21. Bell, D.G., Kuehnel, F., Maxwell, C., Kim, R., Kasraie, K., Gaskins, T., Hogan, P., Coughlan, J.: NASA world wind: opensource GIS for mission operations. In: Aerospace Conference. IEEE (2007)

22. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2E. Packt Publishing, Birmingham (2016)
23. Brogi, A., Ciancarini, P.: The concurrent language, Shared Prolog. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **13**(1), 99–123 (1991)
24. Busi, N., Ciancarini, P., Gorrieri, R., Zavattaro, G.: Coordination models: a guided tour. In: Omicini, A., Zambonelli, F., Klusch, M., Tolksdorf, R. (eds.) *Coordination of Internet Agents: Models, Technologies, and Applications*, Chap. 1, pp. 6–24. Springer, Heidelberg (2001). [https://doi.org/10.1007/978-3-662-04401-8\\_1](https://doi.org/10.1007/978-3-662-04401-8_1)
25. Butera, W.: Programming a paintable computer. Ph.D. thesis, MIT, Cambridge, USA (2002)
26. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: a programmable coordination architecture for mobile agents. *IEEE Internet Comput.* **4**(4), 26–35 (2000)
27. Casadei, M., Viroli, M., Gardelli, L.: On the collective sort problem for distributed tuple spaces. *Sci. Comput. Program.* **74**(9), 702–722 (2009)
28. Casadei, R., Aldini, A., Viroli, M.: Combining trust and aggregate computing. In: Cerone, A., Roveri, M. (eds.) *SEFM 2017. LNCS*, vol. 10729, pp. 507–522. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-74781-1\\_34](https://doi.org/10.1007/978-3-319-74781-1_34)
29. Casadei, R., Pianini, D., Viroli, M.: Simulating large-scale aggregate MASs with Alchemist and Scala. In: *Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 1495–1504. IEEE (2016)
30. Casadei, R., Viroli, M.: Towards aggregate programming in Scala. In: *1st Workshop on Programming Models and Languages for Distributed Computing*, p. 5. ACM (2016)
31. Casadei, R., Viroli, M.: Programming actor-based collective adaptive systems. In: *Programming with Actors - State-of-the-Art and Research Perspectives. Lecture Notes in Computer Science*, vol. 10789. Springer (2018, to appear)
32. Church, A.: A set of postulates for the foundation of logic. *Ann. Math.* **33**(2), 346–366 (1932)
33. Clark, S.S., Beal, J., Pal, P.: Distributed recovery for enterprise services. In: *9th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pp. 111–120. IEEE (2015)
34. Clement, L., Nagpal, R.: Self-assembly and self-repairing topologies. In: *Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open* (2003)
35. Coore, D.: Botanical computing: a developmental approach to generating inter connect topologies on an amorphous computer. Ph.D. thesis, MIT, Cambridge, MA, USA (1999)
36. Corkill, D.: Blackboard systems. *J. AI Expert* **9**(6), 40–47 (1991)
37. Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L., Picco, G.P.: Mobile data collection in sensor networks: the TinyLime middleware. *Elsevier Pervasive Mob. Comput. J.* **4**, 446–469 (2005)
38. Curry, H., Feys, R.: *Combinatory Logi*. North-Holland, Amsterdam (1958)
39. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: *Symposium on Principles of Programming Languages (POPL)*, pp. 207–212. ACM (1982)
40. Damiani, F., Viroli, M.: Type-based self-stabilisation for computational fields. *Log. Methods Comput. Sci.* **11**(4) (2015)
41. Damiani, F., Viroli, M., Beal, J.: A type-sound calculus of computational fields. *Sci. Comput. Program.* **117**, 17–44 (2016)
42. Damiani, F., Viroli, M., Pianini, D., Beal, J.: Code mobility meets self-organisation: a higher-order calculus of computational fields. In: Graf, S., Viswanathan, M. (eds.) *FORTE 2015. LNCS*, vol. 9039, pp. 113–128. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-19195-9\\_8](https://doi.org/10.1007/978-3-319-19195-9_8)

43. Dasgupta, S., Beal, J.: A Lyapunov analysis for the robust stability of an adaptive Bellman-Ford algorithm. In: 55th IEEE Conference on Decision and Control (CDC), pp. 7282–7287. IEEE (2016)
44. De Nicola, R., Ferrari, G., Pugliese, R.: KLAIM: a kernel language for agent interaction and mobility. *IEEE Trans. Softw. Eng. (TOSE)* **24**(5), 315–330 (1998)
45. De Nicola, R., Loretì, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: the SCEL language. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **9**(2), 7:1–7:29 (2014)
46. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **1**(2), 223–259 (2006)
47. Dolev, S.: *Self-Stabilization*. MIT Press, Cambridge (2000)
48. Engstrom, B.R., Cappello, P.R.: The SDEF programming system. *J. Parallel Distrib. Comput.* **7**(2), 201–231 (1989)
49. Fernandez-Marquez, J.L., Di Marzo Serugendo, G., Montagna, S., Viroli, M., Arcos, J.L.: Description and composition of bio-inspired design patterns: a complete overview. *Nat. Comput.* **12**(1), 43–67 (2013)
50. Francia, M., Pianini, D., Beal, J., Viroli, M.: Towards a foundational API for resilient distributed systems design. In: *International Workshops on Foundations and Applications of Self\* Systems (FAS\*W)*. IEEE (2017)
51. Freeman, E., Hupfer, S., Arnold, K.: *JavaSpaces Principles, Patterns, and Practice: Principles, Patterns and Practices*. The Jini Technology Series. Addison-Wesley Longman, Boston (1999)
52. Gelernter, D.: Generative communication in Linda. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **7**(1), 80–112 (1985)
53. Giavitto, J.-L., Godin, C., Michel, O., Prusinkiewicz, P.: Computational models for integrative and developmental biology. Technical report 72-2002, Univerite d’Evry, LaMI (2002)
54. Giavitto, J.-L., Michel, O., Cohen, J., Spicher, A.: Computations in space and space in computations. In: Banâtre, J.-P., Fradet, P., Giavitto, J.-L., Michel, O. (eds.) *UPP 2004*. LNCS, vol. 3566, pp. 137–152. Springer, Heidelberg (2005). <https://doi.org/10.1007/11527800.11>
55. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **23**(3), 396–450 (2001)
56. Kumar, A., Beal, J., Dasgupta, S., Mudumbai, R.: Toward predicting distributed systems dynamics. In: *International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, pp. 68–73. IEEE (2015)
57. Lasser, C., Massar, J., Miney, J., Dayton, L.: *Starlisp Reference Manual*. Thinking Machines Corporation, Cambridge (1988)
58. Lafuente, A.L., Loretì, M., Montanari, U.: A fixpoint-based calculus for graph-shaped computational fields. In: Holvoet, T., Viroli, M. (eds.) *COORDINATION 2015*. LNCS, vol. 9037, pp. 101–116. Springer, Cham (2015). <https://doi.org/10.1007/978-3-319-19282-6.7>
59. Lluch-Lafuente, A., Loretì, M., Montanari, U.: Asynchronous distributed execution of fixpoint-based computational fields. *Log. Methods Comput. Sci.* **13**(1) (2017)
60. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.* **36**(SI), 131–146 (2002)

61. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: the TOTA approach. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **18**(4), 1–56 (2009)
62. Mamei, M., Zambonelli, F., Leonardi, L.: Co-fields: towards a unifying approach to the engineering of swarm intelligent systems. In: Petta, P., Tolksdorf, R., Zambonelli, F. (eds.) *ESAW 2002. LNCS (LNAI)*, vol. 2577, pp. 68–81. Springer, Heidelberg (2003). [https://doi.org/10.1007/3-540-39173-8\\_6](https://doi.org/10.1007/3-540-39173-8_6)
63. Menezes, R., Snyder, J.: Coordination of distributed components using LogOp. In: *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, vol. 1, pp. 109–114. CSREA Press (2003)
64. Menezes, R., Tolksdorf, R.: Adaptiveness in Linda-Based coordination models. In: Di Marzo Serugendo, G., Karageorgos, A., Rana, O.F., Zambonelli, F. (eds.) *ESOA 2003. LNCS (LNAI)*, vol. 2977, pp. 212–232. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-24701-2\\_15](https://doi.org/10.1007/978-3-540-24701-2_15)
65. Minsky, N.H., Ungureanu, V.: Law-Governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **9**(3), 273–305 (2000)
66. Mo, Y., Beal, J., Dasgupta, S.: Error in self-stabilizing spanning-tree estimation of collective state. In: *International Workshops on Foundations and Applications of Self\* Systems (FAS\*W)*, pp. 1–6. IEEE (2017)
67. Murphy, A.L., Picco, G.P., Roman, G.C.: LIME: a coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **15**(3), 279–328 (2006)
68. Nagpal, R.: Programmable self-assembly: constructing global shape using biologically-inspired local interactions and Origami mathematics. Ph.D. thesis, MIT, Cambridge, MA, USA (2001)
69. Newton, R., Welsh, M.: Region streams: functional macroprogramming for sensor networks. In: *Workshop on Data Management for Sensor Networks*, pp. 78–87 (2004)
70. Odersky, M., Rompf, T.: Unifying functional and object-oriented programming with Scala. *Comm. ACM* **57**(4), 76–86 (2014)
71. Omicini, A., Denti, E.: From tuple spaces to tuple centres. *Sci. Comput. Program.* **41**(3), 277–294 (2001)
72. Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., Tummolini, L.: Coordination artifacts: environment-based coordination for intelligent agents. In: *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 286–293. IEEE Computer Society (2004)
73. Omicini, A., Viroli, M.: Coordination models and languages: from parallel computing to self-organisation. *Knowl. Eng. Rev.* **26**(1), 53–59 (2011)
74. Parunak, H.V.D., Brueckner, S.: Concurrent modeling of alternative worlds with polyagents. In: Antunes, L., Takadama, K. (eds.) *MABS 2006. LNCS (LNAI)*, vol. 4442, pp. 128–141. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-76539-4\\_10](https://doi.org/10.1007/978-3-540-76539-4_10)
75. Pianini, D., Beal, J., Viroli, M.: Improving gossip dynamics through overlapping replicates. In: Lluch Lafuente, A., Proença, J. (eds.) *COORDINATION 2016. LNCS*, vol. 9686, pp. 192–207. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-39519-7\\_12](https://doi.org/10.1007/978-3-319-39519-7_12)
76. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. *J. Simul.* **7**(3), 202–215 (2013)
77. Pianini, D., Viroli, M., Beal, J.: Protelis: practical aggregate programming. In: *Symposium on Applied Computing*, pp. 1846–1853. ACM (2015)



78. Pianini, D., Virruso, S., Menezes, R., Omicini, A., Viroli, M.: Self organization in coordination systems using a WordNet-based ontology. In: 4th International Conference on Self-Adaptive and Self-Organizing Systems (SASO). IEEE (2010)
79. Stovall, D., Julien, C.: Resource discovery with evolving tuples. In: International Workshop on Engineering of Software Services for Pervasive Environments: In Conjunction with the 6th ESEC/FSE Joint Meeting, ESSPE, pp. 1–10. ACM, New York (2007)
80. Tolksdorf, R., Menezes, R.: Using swarm intelligence in Linda systems. In: Omicini, A., Petta, P., Pitt, J. (eds.) ESAW 2003. LNCS (LNAI), vol. 3071, pp. 49–65. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-25946-6\\_3](https://doi.org/10.1007/978-3-540-25946-6_3)
81. Viroli, M.: On competitive self-composition in pervasive services. *Sci. Comput. Program.* **78**(5), 556–568 (2013)
82. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. arXiv preprint [arXiv:1711.08297](https://arxiv.org/abs/1711.08297) (2017)
83. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. *ACM Trans. Model. Comput. Simul. (TOMACS)* (2018, to appear)
84. Viroli, M., Audrito, G., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. arXiv preprint [arXiv:1610.08116](https://arxiv.org/abs/1610.08116) (2016)
85. Viroli, M., Beal, J.: Resiliency with aggregate computing: state of the art and roadmap. In: Workshop on FORMAL methods for the quantitative Evaluation of Collective Adaptive SysTems (FORECAST) (2016)
86. Viroli, M., Beal, J., Damiani, F., Pianini, D.: Efficient engineering of complex self-organising systems by self-stabilising fields. In: 9th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pp. 81–90, September 2015
87. Viroli, M., Casadei, M.: Biochemical tuple spaces for self-organising coordination. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 143–162. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-02053-7\\_8](https://doi.org/10.1007/978-3-642-02053-7_8)
88. Viroli, M., Casadei, M., Montagna, S., Zambonelli, F.: Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **6**(2), 14:1–14:24 (2011)
89. Viroli, M., Casadei, M., Omicini, A.: A framework for modelling and implementing self-organising coordination. In: ACM Symposium on Applied Computing (SAC), pp. 1353–1360 (2009)
90. Viroli, M., Casadei, R., Pianini, D.: On execution platforms for large-scale aggregate computing. In: International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct, pp. 1321–1326. ACM (2016)
91. Viroli, M., Damiani, F.: A calculus of self-stabilising computational fields. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 163–178. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-662-43376-8\\_11](https://doi.org/10.1007/978-3-662-43376-8_11)
92. Viroli, M., Damiani, F., Beal, J.: A calculus of computational fields. In: Canal, C., Villari, M. (eds.) ESOC 2013. CCIS, vol. 393, pp. 114–128. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-45364-9\\_11](https://doi.org/10.1007/978-3-642-45364-9_11)
93. Viroli, M., Omicini, A., Ricci, A.: Engineering MAS environment with artifacts. In: Weyns, D., Parunak, H.V.D., Michel, F. (eds.) 2nd International Workshop “Environments for Multi-Agent Systems” (E4MAS 2005), AAMAS 2005, Utrecht, The Netherlands, 26 July 2005

94. Viroli, M., Pianini, D., Beal, J.: Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 212–229. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-30829-1\\_15](https://doi.org/10.1007/978-3-642-30829-1_15)
95. Viroli, M., Pianini, D., Ricci, A., Croatti, A.: Aggregate plans for multiagent systems. *Int. J. Agent-Oriented Softw. Eng.* **4**(5), 336–365 (2017)
96. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: 2nd International Conference on Mobile Systems, Applications, and Services. ACM (2004)
97. Wyckoff, P., McLaughry, S.W., Lehman, T.J., Ford, D.A.: T spaces. *IBM J. Res. Dev.* **37**(3 – Java Techonology), 454–474 (1998)
98. Yamins, D.: A theory of local-to-global algorithms for one-dimensional spatial multi-agent systems. Ph.D. thesis, Harvard, Cambridge, MA, USA (2007)
99. Yao, Y., Gehrke, J.: The Cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record* **31**(3), 9–18 (2002)

## Author Index

- Arbab, Farhad 142  
Audrito, Giorgio 1, 252
- Beal, Jacob 1, 252  
Bravetti, Mario 21
- Casadei, Roberto 252  
Castegren, Elias 162  
Ciatto, Giovanni 51  
Clarke, Dave 162
- Damiani, Ferruccio 1, 252  
Darquennes, Denis 81  
De Nicola, Rocco 110  
Dokter, Kasper 142
- Elaraby, Nahla 200
- Fernandez-Reyes, Kiko 162  
Ferrari, Gianluigi 110
- Hains, Gaétan 220  
Henrio, Ludovic 220
- Jacquet, Jean-Marie 81
- Kaminskas, Linas 181  
Kühn, eva 200
- Leca, Pierre 220  
Linden, Isabelle 81  
Lluch Lafuente, Alberto 181  
Louvel, Maxime 51
- Mariani, Stefano 51  
Marlier, Patrick 231
- Omicini, Andrea 51
- Pianini, Danilo 252  
Pugliese, Rosario 110
- Radschek, Sophie Therese 200
- Schiavoni, Valerio 231  
Suijlen, Wijnand 220  
Sutra, Pierre 231
- Tiezzi, Francesco 110  
Trahay, François 231
- Viroli, Mirko 1, 252  
Vo, Huu-Phuc 162
- Zambonelli, Franco 51  
Zavattaro, Gianluigi 21