EPTCS 414

Proceedings of the

17th

Interaction and Concurrency Experience

Groningen, The Netherlands, 21st June 2024

Edited by: Clément Aubert, Cinzia Di Giusto, Simon Fowler and Violet Ka I Pun

Published: 11th December 2024
DOI: 10.4204/EPTCS.414
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

Clément Aubert
Augusta University, USA

Cinzia Di Giusto
Université Côte d'Azur, CNRS, I3S Sophia Antipolis, FR

Simon Fowler
University of Glasgow School of Computing Science, UK

Violet Ka I Pun
Western Norway University of Applied Sciences, NO

This volume contains the proceedings of ICE'24, the 17th Interaction and Concurrency Experience, which was held in Groningen, the Netherlands, as a satellite event of DisCoTec'24. The previous editions of ICE were affiliated with ICALP'08 (Reykjavik, Iceland), CONCUR'09 (Bologna, Italy), DisCoTec'10 (Amsterdam, The Netherlands), DisCoTec'11 (Reykjavik, Iceland), DisCoTec'12 (Stockholm, Sweden), DisCoTec'13 (Florence, Italy), DisCoTec'14 (Berlin, Germany), DisCoTec'15 (Grenoble, France), DisCoTec'16 (Heraklion, Greece), DisCoTec'17 (Neuchâtel, Switzerland), DisCoTec'18 (Madrid, Spain), DisCoTec'19 (Lyngby, Denmark), DisCoTec'20 (virtual, hosted by the University of Malta), DisCoTec'21 (virtual, hosted by the University of Malta), DisCoTec'22 (hybrid event, hosted in Lucca, Italy), and DisCoTec'23 (Lisbon, Portugal). The details of previous editions, proceedings and special issues as well as the composition of the steering committee can be found on the new series website, at `https://ice-workshop.github.io/`.

The ICE workshop series features a distinguishing review and selection procedure: PC members are encouraged to interact, anonymously, with authors. This year these interactions took place on the HotCRP platform which combines paper selection features with forum-like interactions. As in the past editions, the forum discussion during the review and selection phase of ICE'24 considerably improved the accuracy of the feedback from the reviewers and the quality of accepted papers, and offered the basis for lively discussion during the workshop. The time and effort spent on the interaction between reviewers and authors is rewarding for all parties. The discussions on HotCRP made it possible to resolve misunderstandings at an early stage of the review process, to discover and correct mistakes in key definitions, and to improve examples and presentation.

The 2024 edition of ICE included double anonymous reviewing of original research papers, in order to increase fairness and avoid bias in reviewing. Research papers were submitted with authors' names and identifying details removed, and authors were given anonymous access to a dedicated section of HotCRP. Each paper was reviewed by three PC members, and altogether 3 papers were accepted for publication – plus 1 oral presentation which is not part of this volume. We were proud to host one invited talk by Jorge A. Pérez. The full paper of the invited talk is included in this volume, together with the final versions of the research papers, which takes into account the discussion at the workshop.

We would like to thank the authors of all the submitted papers for their interest in the workshop. We thank Jorge A. Pérez for accepting our invitation to present his recent work. We are grateful for the efforts of the PC members:

- Matthew Alan Le Brun (University of Glasgow, UK)
- Kirstin Peters (Augsburg University, DE)
- Carla Ferreira (NOVA University Lisbon, PT)
- Eva Graversen (University of Southern Denmark, DK)
- **Bas van den Heuvel (Karlsruhe University of Applied Sciences / University of Freiburg, DE)** ⋆

C. Aubert, C. Di Giusto, S. Fowler & V. K. I Pun (Eds.):
17th Interaction and Concurrency Experience (ICE 2024)
EPTCS 414, 2024, pp. ii–iii, doi:10.4204/EPTCS.414.0

- Gerard Tabone (University of Malta, MT)
- Ping Hou (University of Oxford, UK)
- Sergueï Lenglet (Université de Lorraine – Université Sorbonne Paris-Nord, FR)
- Emanuele D'Osualdo (Max Planck Institute for Software Systems, DE)
- Seyed Hossein Haeri (IOG & University of Bergen, NO)
- Ivan Prokić (Faculty of Technical Sciences, University of Novi Sad, RS)
- Luc Edixhoven (Open University of the Netherlands, NL)
- Felix Stutz (University of Luxembourg, LU)
- Lorenzo Gheri (University of Liverpool, UK)
- Andreia Mordido (LASIGE, University of Lisbon, PT)

⋆ **The *ICE 2024 Outstanding PC Member Award* was awarded this year to Bas van den Heuvel!**

Previous ICE Outstanding PC members include:

- ICE 2023: *Sergueï Lenglet*
- ICE 2022: *Duncan Paul Attard*
- ICE 2021: *Ivan Prokić*

We thank the ICE steering committee and the *DisCoTec'24 organizers*, in particular the general chair, Jorge A. Pérez, for providing an excellent environment for the preparation and staging of the event. Finally, we thank the editors of *EPTCS* for the publication of these post-proceedings.

A Gentle Overview of Asynchronous Session-based Concurrency: Deadlock Freedom by Typing

Bas van den Heuvel
Karlsruhe University of Applied Sciences, Karlsruhe, and University of Freiburg, Freiburg, Germany

Jorge A. Pérez
University of Groningen, The Netherlands

While formal models of concurrency tend to focus on synchronous communication, asynchronous communication is relevant in practice. In this paper, we will discuss asynchronous communication in the context of session-based concurrency, the model of computation in which *session types* specify the structure of the two-party protocols implemented by the channels of a communicating process. We overview recent work on addressing the challenge of ensuring the *deadlock-freedom* property for message-passing processes that communicate asynchronously in cyclic process networks governed by session types. We offer a gradual presentation of three typed process frameworks and outline how they may be used to guarantee deadlock freedom for a concurrent functional language with sessions.

1 Introduction

The purpose of this paper is to overview recent work on new verification techniques that enforce deadlock freedom for *message-passing processes*. We target concurrent systems that form cyclic process networks and that communicate *asynchronously*, governed by protocols expressed as *session types* [43, 25].

We rely on *process calculi* as formal models of concurrency, as they provide a firm foundation for specifying and analyzing message-passing programs and for principled designs of programming abstractions involving concurrent, interactive behavior. We are also interested in asynchronous communication, which, from the standpoint of distributed systems, can be informally described as the kind of process communication in which no global clock is assumed; as such, an observer has no way of knowing if the message they have sent has been received. These intuitions can be precisely formulated in a language-independent way (cf. [42]). Unsurprisingly, asynchronous communication is of clear practical relevance: it is the standard in most distributed systems and web-based applications nowadays.

Despite this pragmatic interest, process calculi such as the π-calculus tend to focus on synchronous communication, rather than on asynchronous communication. In fact, as we discuss later on, the study of formalisms such as the asynchronous π-calculus originated from an interest in the essential ingredients of the synchronous π-calculus. There is a vast literature on the asynchronous π-calculus as a 'fundamental core' of the π-calculus, its behavioral theory, and its relationship with the original synchronous π-calculus, in particular from the point of view of relative (or comparative) expressiveness (see, e.g., [23, 4, 36, 1, 33, 32, 37, 5]). In the π-calculus, asynchronous communication admits an elegant and economical formulation: asynchronous processes can be defined simply by decreeing that in output-prefixed processes $x[v];P$ the continuation P can only correspond to the inactive process 0. This ensures that there are no processes that are blocked by an output action, but also that a process $x[v];0$ can be regarded as an 'output particle' that has been emitted but not yet received by some intended receiver. These particles could then be arranged into some suitable structure, such as a queue or a stack [3].

C. Aubert, C. Di Giusto, S. Fowler & V. K. I Pun (Eds.):
17th Interaction and Concurrency Experience (ICE 2024)
EPTCS 414, 2024, pp. 1–20, doi:10.4204/EPTCS.414.1

Asynchronous communication is also relevant for *session-based concurrency*, which can be described as the model of interaction in which processes exchange messages following some predetermined protocols specified as session types. Session types specify sequences of input and output actions, possibly recursive, which define *communication structures* between two or more interacting parties. Session-type systems leverage those structures to ensure that interacting processes always respect their intended specifications and never exhibit issues such as message mismatches (e.g., ill-formatted data), message duplication and loss, out-of-order messages, race conditions, and *deadlocks*—the insidious situation in which processes are permanently blocked, awaiting indefinitely a message that will never arrive. The use of session types for excluding deadlocks in asynchronous processes is the central theme in our work.

Session-based concurrency has been widely studied using (variants of) the π-calculus, which provide a simple yet rigorous framework for developing verification techniques for message-passing programs. In this context, asynchronous communication usually has a less economical definition than in the untyped setting: processes/programs are typically defined together with some runtime entities, such as buffers, which explicitly account for the in-transit messages by following the structure of their corresponding sessions [15, 16]. Different designs for these buffers, accounting for different levels of granularity, are possible [28]. This treatment has direct consequences on the notions of causality that govern reasoning over well-typed processes: actions from different sessions should be independent from each other, but actions (including outputs) within a session should follow the ordering described by their session type. As a result, the machinery required by asynchronous sessions entails some notational burden, for instance when formulating the meta-theoretical results for well-typed processes. Moving to an asynchronous setting has also important consequences for central notions, such as *subtyping*, which is decidable under a synchronous semantics but becomes undecidable in the asynchronous case [29].

In this paper, we are interested in verification techniques for ensuring that asynchronous session processes are *deadlock free*. Just as session-based concurrency integrates elements and concepts originating from different areas (concurrency theory, process calculi, type systems, programming languages), we consider an amalgamation of two separate developments, which leads to a clean formulation of asynchronous communication in which deadlock-freedom guarantees hold for a wide class of processes.

On the one hand, we consider logical correspondences, in the style of Curry-Howard, that connect session types and linear logic [8, 46]. This line of work provides in particular clean foundations for both the analysis of deadlock freedom and for asynchronous communication. Indeed, as shown by DeYoung *et al.* [14], in a logically-motivated setting, asynchronous communication enjoys a remarkably economical formulation, in which output particles represent the intended communication structure using continuation passing. On the other hand, we consider type systems for the π-calculus that exclude deadlocks by considering *priority-based* approaches [26, 34], which avoid vicious cycles in advanced communication patterns. These two strands of work are distinct in nature but complementary nevertheless; in particular, it is known that priority-based approaches to deadlock freedom are strictly more powerful than logic-based approaches [12, 13]. In this paper we show how the proposed amalgamation enables a fresh understanding of the key insights involved in enforcing deadlock freedom in an asynchronous setting, gradually going from no enforcement, to enforcement restricted to tree-like topologies, and culminating in deadlock freedom for the cyclic topologies of processes that abound in practice (such as those present in parallel algorithms).

Structure of the document. Section 2 gives a high-level discussion on the interplay of asynchronous communication and (session) protocols. Section 3 presents AP: a π-calculus with asynchronous communication whose session-type system enforces conformance to session protocols but does not exclude

deadlocks. Sections 4 and 5 build upon AP to gradually illustrate the essentials of deadlock freedom by typing. We first introduce ACP, an asynchronous variant of Wadler's CP [46] that enforces deadlock freedom for processes that form tree-like networks. Then, we present the key ideas underlying APCP, an enhancement of AP with a priority-based approach to typing, which enforces deadlock freedom also for processes that form cyclic networks. Section 6 briefly discusses LAST^{n}, a functional language with asynchronous sessions (based on the language by Gay and Vasconcelos [16]), for which deadlock freedom can be guaranteed via a correct translation into APCP. Section 7 collects some final remarks.

Origin of the results. This paper is intended as a gentle introduction to our journal paper [21], which offers a full treatment of APCP, its meta-theoretical results, formal connections with LAST^{n}, and comparisons with related works. In particular, Sections 5 and 6 collect selected results first reported in [21]. For the sake of presentation, here we consider typed calculi without recursive processes and recursive session types, which are included in [21]. The discussion in Section 2 and the asynchronous variant of CP presented in Section 4 are new to this presentation.

2 Encodings as (Session) Protocols and Asynchronous Communication

In the theory of the π-calculus, asynchronous communication was not the first choice. The π-calculus was introduced as a calculus of synchronous, channel-based communication, from which asynchrony arose as a (syntactic) limitation—a sort of afterthought. Much of what we know about asynchronous communication in this setting actually comes from studies investigating the (non-)existence of (correct) *encodings* of synchronous into asynchronous communication. Perhaps unsurprisingly, the theory of session types followed a similar path: it was first formulated using programming models with synchronous communication; the interest in asynchrony came later, and continues to be relevant, especially as session types have found their way into (mainstream) programming languages [2].

Encodings between process calculi can intuitively be seen as *protocols*: given a step in a source calculus, an encoding gives a precise sequence of steps in the target calculus that represents it. A bit of history may be instructive here. Shortly after the (synchronous) π-calculus was introduced, researchers sought to determine the *essential ingredients* of interaction and concurrency. An initial subject of study was *polyadicity*—the ability to send and receive finite lists of names in a single communication step:

$$x[v_1,\ldots,v_k];P \mid x(y_1,\ldots,y_k);Q \longrightarrow P \mid Q\{v_1/y_1\}\ldots\{v_k/y_k\}$$

where $x[\tilde{v}]$ and $x(\tilde{y})$ denote output and input prefixes, respectively, and ';' and '|' denote sequential and parallel composition, respectively. The question is then whether the polyadic π-calculus can be encoded into the *monadic* variant, in which at most one value can be exchanged. Milner [31] gave the following protocol for the exchange of a list of values $v_1,\ldots,v_k$ over name x: create a fresh name s, send s over x, and then use s to individually transmit each v_i (with $i \in 1\ldots k$) using a monadic communication. Hence, this protocol represents a single k-adic communication with $k+1$ monadic communications, using s as a private session to avoid interferences. Interestingly, as simple and plausible as this protocol looks, its correctness is not obvious, in particular if one considers *full abstraction*: as shown by Quaglia and Walker [40], using types for monadic processes is essential for establishing a sound and complete correspondence between source (polyadic) processes and their corresponding target (monadic) processes.

This brings us back to the issue of asynchronous communication. Studies on asynchronous variants of the π-calculus originated from the question: can a synchronous communication discipline be represented in the simpler and more pragmatic asynchronous setting, in which output is not a blocking

operation? For the π-calculus without choice constructs, two encodings/protocols were independently proposed by Boudol [4] and by Honda and Tokoro [23] (who studied it for a core language for objects).

We briefly review these two encodings, following the presentation by Van Glabbeek [17], writing $[\![\cdot]\!]_{\mathtt{B}}$ and $[\![\cdot]\!]_{\mathtt{HT}}$, respectively (in both cases, the encoding is a homomorphism for other process constructs):

$$[\![x[y];P]\!]_{\mathtt{B}} = (\nu u)(x[u] \mid u(w);(w[y] \mid [\![P]\!]_{\mathtt{B}})) \qquad [\![x[y];P]\!]_{\mathtt{HT}} = x(u);(u[y] \mid [\![P]\!]_{\mathtt{HT}})$$
$$[\![x(z);P]\!]_{\mathtt{B}} = x(u);(\nu w)(u[w] \mid w(z);[\![P]\!]_{\mathtt{B}}) \qquad [\![x(z);P]\!]_{\mathtt{HT}} = (\nu w)(x[w] \mid w(z);[\![P]\!]_{\mathtt{HT}})$$

The encodings adopt different approaches to represent synchronous communication. In Boudol's encoding/protocol, a single communication step is represented using three steps in the asynchronous calculus: first on x and then on the two fresh names u and w (on which the source value y is finally communicated). Observe that the direction of actions is preserved: an output is encoded into an output, and same for input. Honda and Tokoro's encoding/protocol is simpler. It only involves one fresh name but it does not preserve directionality: indeed, the encoding of input takes the initiative by sending a freshly created name on which the communication of y will occur. Despite these differences, both encodings are correct with respect to Gorla's correctness criteria [18], as shown in [17].

In our opinion, the view of encodings as protocols is insightful in itself, but also because we are interested in session types as a way of specifying protocols for (deadlock-free) concurrent processes. In this respect, there are two salient points worth making.

- First, there is a tension between asynchronous communication and session types: while the former aims at *unconstraining* behaviors (by ensuring that only inputs are blocking points in communication), session types aim at *constraining* behaviors, for a good reason: to ensure that processes conform to some intended communication structure. This tension does not entail a conflict, but it does have a consequence: the incorporation of session types in asynchronous process calculi results in a model that stands "in between" synchrony and asynchrony but differs from both [28]: (output) actions from different sessions should be independent from each other, but (output) actions within a session should follow the ordering described by their session type.
- Second, it is known that the choice between synchronous and asynchronous communication directly influences deadlock-freedom analyses [10]: the asynchronous setting appears as the most convenient scenario in which to develop techniques for ensuring deadlock freedom for (session) processes, as it involves the least amount of blocking operations. Also, when considering sessions, asynchronous communication makes differences between different sessions even more prominent. To see this, consider the following process in a synchronous setting:

  $$P = (\nu xy)(\nu uw)(u[z];y[v];P_1 \mid x(v);w(z);P_2)$$

  where we use the restriction (νxy) to declare x and y as dual endpoints of a session. Hence, in P we have that x and y form one session, different from the session formed by u and w. Also, P consists of two sub-processes, both with blocking operations (outputs in the left sub-process, inputs in the right sub-process); the cyclic dependencies induced by these operations make P deadlocked.

  Having an output on a session that blocks an output from another session is difficult to justify. While a variant of P in which these two outputs are swapped (as in $y[v];u[z];P_1$) would immediately solve the issue, a more fundamental observation is that P is not directly expressible in an asynchronous setting. In fact, in an asynchronous setting we could have a process like

  $$(\nu xy)(\nu uw)(u[z] \mid y[v] \mid P_1 \mid x(v);w(z);P_2)$$

  that expresses the dependency between the two sessions in a deadlock-free manner.

Process syntax:

$$
\begin{array}{rlll}
P,Q ::= & x[a,b] & \text{send} & \mid\ x(y,z);P \quad \text{receive} \\
\mid & x[b] \lhd j & \text{selection} & \mid\ x(z) \rhd \{i : P\}_{i \in I} \quad \text{branch} \\
\mid & P \mid Q & \text{parallel} & \mid\ (\nu xy)P \quad \text{restriction} \\
\mid & 0 & \text{inaction} & \mid\ [x \leftrightarrow y] \quad \text{forwarder}
\end{array}
$$

Structural congruence:

$$\frac{P \equiv_\alpha Q}{P \equiv Q}\ [\text{SC-ALPHA}] \qquad \frac{}{P \mid 0 \equiv P}\ [\text{SC-PAR-UNIT}] \qquad \frac{}{P \mid Q \equiv Q \mid P}\ [\text{SC-PAR-COMM}] \qquad \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R}\ [\text{SC-PAR-ASSOC}]$$

$$\frac{x,y \notin \text{fn}(P)}{P \mid (\nu xy)Q \equiv (\nu xy)(P \mid Q)}\ [\text{SC-SCOPE}] \qquad \frac{}{(\nu xy)(\nu zw)P \equiv (\nu zw)(\nu xy)P}\ [\text{SC-RES-COMM}] \qquad \frac{}{(\nu xy)P \equiv (\nu yx)P}\ [\text{SC-RES-SYMM}]$$

$$\frac{}{(\nu xy)0 \equiv 0}\ [\text{SC-RES-INACT}] \qquad \frac{}{[x \leftrightarrow y] \equiv [y \leftrightarrow x]}\ [\text{SC-FWD-SYMM}] \qquad \frac{}{(\nu xy)[x \leftrightarrow y] \equiv 0}\ [\text{SC-RES-FWD}]$$

Reduction:

$$\frac{}{(\nu xy)(x[a,b] \mid y(a',b');Q) \longrightarrow Q\{a/a',b/b'\}}\ [\text{RED-SEND-RECV}]$$

$$\frac{j \in I}{(\nu xy)(x[b] \lhd j \mid y(b') \rhd \{i : Q_i\}_{i \in I}) \longrightarrow Q_j\{b/b'\}}\ [\text{RED-SEL-BRA}] \qquad \frac{y \neq z}{(\nu xy)([x \leftrightarrow z] \mid P) \longrightarrow P\{z/y\}}\ [\text{RED-FWD}]$$

$$\frac{P \equiv P' \qquad P' \longrightarrow P' \qquad Q' \equiv Q}{P \longrightarrow Q}\ [\text{RED-SC}] \qquad \frac{P \longrightarrow Q}{(\nu xy)P \longrightarrow (\nu xy)Q}\ [\text{RED-RES}] \qquad \frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R}\ [\text{RED-PAR}]$$

Figure 1: The AP process language: syntax and reduction semantics.

Having presented a high-level discussion on the interplay between encodings/protocols, asynchronous communication, and deadlock freedom, we now move on to formally presenting a basic process model of asynchronous communication and its corresponding type system.

3 Asynchronous Processes

We start by presenting a calculus of *asynchronous processes*, dubbed AP. We define its syntax, reduction semantics, and session-type system. Well-typed processes perform their ascribed session protocols but may run into deadlocks. As such, AP provides a basic framework for developing more sophisticated

typing disciplines that enforce deadlock freedom, namely ACP and APCP, to be introduced later on.

Syntax. We write $a, b, c, \ldots, x, y, z, \ldots$ to denote *names* (or *endpoints*); by convention we use the early letters of the alphabet for the objects of output-like constructs. Also, we write $\tilde{x}, \tilde{y}, \tilde{z}, \ldots$ to denote finite sequences of names. With a slight abuse of notation, we sometimes write $x_i \in \tilde{x}$ to refer to a specific element in the sequence $\tilde{x}$. Also, we write $i, j, k, \ldots$ to denote *labels* for choices, and $I, J, K, \ldots$ to denote finite sets of labels. In AP, communication is *asynchronous* (cf. [23, 24, 4]) and *dyadic*: each communication involves the transmission of a pair of names, usually interpreted as a message name and a continuation name. We use $P, Q, \ldots$ to denote processes.

Figure 1 (top) gives the syntax of processes. The send $x[a,b]$ emits along x a message name a and a continuation name b. The receive $x(y,z);P$ blocks until along x a message and continuation name are received (referred to in P as the placeholders y and z, respectively), binding y and z in P. The selection $x[b] \triangleleft i$ sends along x a label i and a continuation name b. The branch $x(z) \triangleright \{i : P_i\}_{i \in I}$ blocks until it receives along x a label $i \in I$ and a continuation name (referred to in P_i as the placeholder z), binding z in each P_i. In the rest of this paper, we refer to sends, receives, selections, and branches as *prefixes* (even though sends and selections do not prefix a continuation process). We refer to sends and selections collectively as *outputs*, and to receives and branches as *inputs*.

The process $P \mid Q$ denotes the parallel composition of P and Q. Restriction $(\nu xy)P$ binds x and y in P, thus declaring them as the two names of a channel and enabling communication (cf. [45]). The process 0 denotes inaction. The forwarder $[x \leftrightarrow y]$ is a primitive copycat process that links together names x and y.

Names are free unless otherwise stated (i.e., unless they are bound somehow). We write $\mathrm{fn}(P)$ for the set of free names of P, and $\mathrm{bn}(P)$ for the set of bound names of P. Also, we write $P\{x/y\}$ to denote the capture-avoiding substitution of the free occurrences of y in P for x. We write sequences of substitutions $P\{x_1/y_1\} \ldots \{x_n/y_n\}$ as $P\{x_1/y_1, \ldots, x_n/y_n\}$.

Reduction semantics. The reduction relation for processes ($P \longrightarrow Q$) formalizes how complementary outputs/inputs on connected names may synchronize. As usual for π-calculi, reduction relies on *structural congruence* ($P \equiv Q$), which relates processes with minor syntactic differences. Structural congruence is the smallest congruence on the syntax of processes (Figure 1 (top)) satisfying the axioms in Figure 1 (middle).

Structural congruence defines the following properties for processes. Processes are equivalent up to α-equivalence (Rule [SC-ALPHA]). Parallel composition is associative (Rule [SC-PAR-ASSOC]) and commutative (Rule [SC-PAR-COMM]), with unit 0 (Rule [SC-PAR-UNIT]). A parallel process may be moved into or out of a restriction as long as the bound channels do not occur free in the moved process (Rule [SC-SCOPE]): this is *scope inclusion* and *scope extrusion*, respectively. Restrictions on inactive processes may be dropped (Rule [SC-RES-INACT]), and the order of names in restrictions and of consecutive restrictions does not matter (Rules [SC-RES-SYMM] and [SC-RES-COMM], respectively). Forwarders are symmetric (Rule [SC-FWD-SYMM]), and equivalent to inaction if both names are bound together through restriction (Rule [SC-RES-FWD]).

We define the reduction relation $P \longrightarrow Q$ by the axioms and closure rules in Figure 1 (bottom). We write $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$. Rule [RED-SEND-RECV] synchronizes a send and a receive on connected names and substitutes the message and continuation names. Rule [RED-SEL-BRA] synchronizes a selection and a branch: the received label determines the continuation process, substituting the continuation name appropriately. Rule [RED-FWD] implements the forwarder as a substitution. Rules [RED-SC], [RED-RES], and [RED-PAR] close reduction under structural congruence, restriction, and

parallel composition, respectively.

Type system. AP types processes by assigning binary session types to names. Following Curry-Howard interpretations, we present session types as linear logic propositions (cf., e.g., Caires *et al.* [9], Wadler [46], Caires and Pérez [7], and Dardha and Gay [11]).

Definition 1 (Session Types for AP)**.** *The following grammar defines the syntax of* session types A, B.

$$A, B ::= A \otimes B \mid A \parr B \mid \oplus\{i : A\}_{i\in I} \mid \&\{i : A\}_{i\in I} \mid \bullet$$

A name of type $A \otimes B$ (resp. $A \parr B$) first sends (resp. receives) a message name of type A and a continuation name of type B. A name of type $\oplus\{i : A_i\}_{i\in I}$ selects a label $i \in I$ and sends a continuation name of type A_i. A name of type $\&\{i : A_i\}_{i\in I}$ offers a choice: after receiving a label $i \in I$, the continuation name should behave as A_i. We write $\bullet$ to denote the type of a session protocol that is finished, i.e., a session that is *closed*. Closed sessions are usually typed using the linear logic units 1 and $\bot$ (cf., e.g., [8, 11]), but AP does not associate any process behavior with closed sessions, so we follow Caires [6] in conflating 1 and $\bot$ to the single, self-dual type $\bullet$.

Duality, the cornerstone notion of session types and linear logic, ensures that the two names of a channel have complementary behaviors.

Definition 2 (Duality)**.** *The* dual *of session type A, denoted $\overline{A}$, is defined inductively as follows:*

$$\overline{A \otimes B} \triangleq \overline{A} \parr \overline{B} \qquad \overline{\oplus\{i : A_i\}_{i\in I}} \triangleq \&\{i : \overline{A_i}\}_{i\in I} \qquad \overline{\bullet} \triangleq \bullet$$
$$\overline{A \parr B} \triangleq \overline{A} \otimes \overline{B} \qquad \overline{\&\{i : A_i\}_{i\in I}} \triangleq \oplus\{i : \overline{A_i}\}_{i\in I}$$

Judgments are of the form

$$P \vdash \Gamma$$

where P is a process and Γ is a context that records assignments of types to channels of the form $x : A$. A judgment $P \vdash \Gamma$ then means that P can be typed in accordance with the type assignments for names recorded in Γ. The context Γ obeys *exchange* (assignments may be silently reordered), but disallows *weakening* (all assignments must be used, except names typed with $\bullet$) and *contraction* (assignments may not be duplicated). The empty context is written $\varnothing$. In writing $\Gamma, x : A$ we assume that $x \notin \mathrm{dom}(\Gamma)$.

Figure 2 gives the typing rules. We describe the typing rules from a *bottom-up* perspective. Rule [TYP-SEND] types a send; this rule does not have premises to provide a continuation process, leaving the free message and continuation names to be bound to a continuation process using Rules [TYP-PAR] and [TYP-RES] (both of which will be discussed next). Similarly, Rule [TYP-SEL] types a selection, where the continuation name is free. Rules [TYP-RECV] and [TYP-BRA] type receives and branches, respectively.

In the tradition of simply-typed π-calculi [41], we have two separate rules for parallel composition and restriction. Rule [TYP-PAR] types the parallel composition of two processes that do not share assignments on the same names. Rule [TYP-RES] types a restriction, where the two restricted names must be of dual type. Rule [TYP-END] implements a constrained form of weakening, which silently removes a closed name from the typing context. Rule [TYP-INACT] types an inactive process with no names. Rule [TYP-FWD] types forwarding between names of dual type.

AP satisfies an important form of type soundness that guarantees consistency of typing across structural congruence and reduction. This property is key to proving safety properties such as session fidelity (correct implementation of assigned session types) and communication safety (no message mismatches).

Theorem 1 (Type Preservation for AP)**.** *Given $P \vdash \Gamma$ and Q such that $P \equiv Q$ or $P \longrightarrow Q$, we have $Q \vdash \Gamma$.*

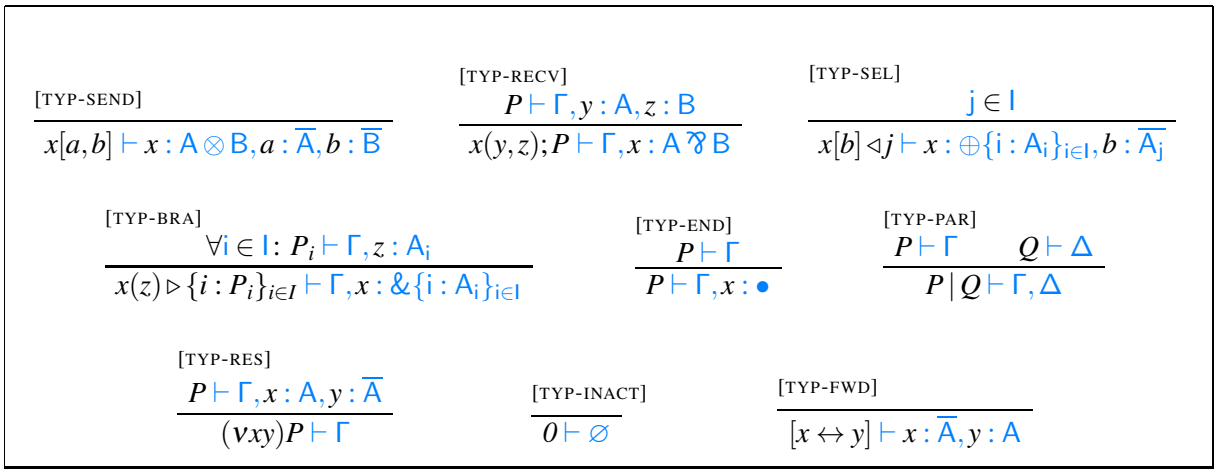


Figure 2: The typing rules of AP.

Deadlock freedom is a fundamental property for message-passing processes. However, typing in AP is too permissive to guarantee deadlock freedom, as illustrated by the following example.

Example 1. *Consider the process*

$$(\nu xy)(\nu uw)(x(v,x');u[a,b] \mid w(z,w');y[c,d])$$

which can be considered as the "hello world" of deadlocked message-passing processes. The process is deadlocked because the left receive (on x) is waiting for the right send (on y), which is blocked by a receive (on w) waiting for the left send (on u), which is blocked by the left receive. We refer to such a state as a cyclic dependency. *The corresponding typing derivation, given below, is valid in* AP*; for brevity, the right subtree (analogous to the left one) is omitted and superscripts on rule labels indicate a number of repeated applications of the same rule.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{u[a,b] \vdash u : \bullet \otimes \bullet, a : \bullet, b : \bullet}\,[\text{TYP-END}]^2}{u[a,b] \vdash v : \bullet, x' : \bullet, u : \bullet \otimes \bullet, a : \bullet, b : \bullet}\,[\text{TYP-RECV}]}{x(v,x');u[a,b] \vdash x : \bullet \parr \bullet, u : \bullet \otimes \bullet, a : \bullet, b : \bullet} \qquad \vdots \quad w(z,w');y[c,d] \vdash w : \bullet \parr \bullet, y : \bullet \otimes \bullet, c : \bullet, d : \bullet}{x(v,x');u[a,b] \mid w(z,w');y[c,d] \vdash x : \bullet \parr \bullet, u : \bullet \otimes \bullet, a : \bullet, b : \bullet, w : \bullet \parr \bullet, y : \bullet \otimes \bullet, c : \bullet, d : \bullet}\,[\text{TYP-PAR}]$$
$$\dfrac{}{(\nu xy)(\nu uw)(x(v,x');u[a,b] \mid w(z,w');y[c,d]) \vdash a : \bullet, b : \bullet, c : \bullet, d : \bullet}\,[\text{TYP-RES}]^2$$

Note that deadlock freedom is usually guaranteed for closed *processes, i.e., without free names. The process above is not closed, but it can be trivially closed because all its free names are typed* $\bullet$.

In fact, the cyclic dependencies introduced in the example above are the only source of deadlock in our process calculus. As we will see in the next two sections, there are multiple ways for typing to guarantee deadlock freedom by ruling out cyclic dependencies.

4 Asynchronous CP

ACP is an asynchronous variant of Wadler's Classical Processes (CP) [46]. It can be obtained from AP by a minor yet crucial modification to Figure 2: Rules [TYP-PAR] and [TYP-RES] are replaced by the

following Rule [TYP-CUT], which combines parallel composition and restriction:

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, y : \overline{A}}{(\nu xy)(P \mid Q) \vdash \Gamma, \Delta}\ [\text{TYP-CUT}]$$

As we will see, replacing Rules [TYP-PAR] and [TYP-RES] with Rule [TYP-CUT] guarantees deadlock freedom simply by ruling out all possible cyclic dependencies, because it ensures that every pair of processes shares at most a single pair of names with dual behaviors. We shall write $\overset{C}{\vdash}$ instead of $\vdash$ to denote the difference between the two type systems.

The different typing for parallel composition has an effect on the semantics of (typable) processes. Indeed, note that most structural congruence rules in Figure 1 (middle) do not preserve typing under $\overset{C}{\vdash}$ (e.g., the left process in Rule [SC-SCOPE] is not typable at all). Therefore, we define an alternative structural congruence for ACP, denoted $\overset{C}{\equiv}$, with Rules [SC-ALPHA] and [SC-FWD-SYMM] as in Figure 1 (middle) and the following rules:

$$\frac{}{(\nu xy)(P \mid Q) \overset{C}{\equiv} (\nu yx)(Q \mid P)}\ [\text{SC-CUT-SYMM}] \qquad \frac{y \notin \text{fn}(Q) \qquad w \notin \text{fn}(P)}{(\nu xy)(P \mid (\nu zw)(Q \mid R)) \overset{C}{\equiv} (\nu zw)(Q \mid (\nu xy)(P \mid R))}\ [\text{SC-CUT-ASSOC-L}]$$

$$\frac{y \notin \text{fn}(R) \qquad z \notin \text{fn}(P)}{(\nu xy)(P \mid (\nu zw)(Q \mid R)) \overset{C}{\equiv} (\nu zw)((\nu xy)(P \mid Q) \mid R)}\ [\text{SC-CUT-ASSOC-R}]$$

Accordingly, we define reduction for ACP, denoted $\overset{C}{\longrightarrow}$, as in Figure 1 (bottom), except that Rule [RED-SC] uses $\overset{C}{\equiv}$. We write $P \overset{C}{\not\longrightarrow}$ whenever there exists no Q such that $P \overset{C}{\longrightarrow} Q$. We have the following:

Theorem 2 (Type Preservation for ACP)**.** *Given $P \overset{C}{\vdash} \Gamma$ and Q such that $P \overset{C}{\equiv} Q$ or $P \overset{C}{\longrightarrow} Q$, we have $Q \overset{C}{\vdash} \Gamma$.*

Example 2. *Recall the deadlocked process from Example 1. Under $\overset{C}{\vdash}$, this process is not typable, because its two parallel sub-processes are connected on two pairs of names. We can create a well-typed variant by splitting one of the sub-processes into a parallel composition instead of sequence, such that the resulting two sub-processes can be connected to the original left sub-process on a single pair of names each:*

$$(\nu xy)\big((\nu uw)(x(v,x');u[a,b] \mid w(z,w');0) \mid y[c,d]\big)$$

The corresponding derivation is shown below; for readability, we omit the left subtree, which is identical to that in Example 1 (writing $\overset{C}{\vdash}$ instead of $\vdash$).

$$\dfrac{\dfrac{\vdots}{x(v,x');u[a,b] \overset{C}{\vdash} x : \bullet \parr \bullet, u : \bullet \otimes \bullet, a : \bullet, b : \bullet} \qquad \dfrac{\dfrac{\dfrac{}{0 \overset{C}{\vdash} \varnothing}\ [\text{TYP-INACT}]}{0 \overset{C}{\vdash} z : \bullet, w' : \bullet}\ [\text{TYP-END}]^2}{w(z,w');0 \overset{C}{\vdash} w : \bullet \parr \bullet}\ [\text{TYP-RECV}]}{(\nu uw)(x(v,x');u[a,b] \mid w(z,w');0) \overset{C}{\vdash} x : \bullet \parr \bullet, a : \bullet, b : \bullet}\ [\text{TYP-CUT}] \qquad \dfrac{}{y[c,d] \overset{C}{\vdash} y : \bullet \otimes \bullet, c : \bullet, d : \bullet}\ [\text{TYP-SEND}]}{(\nu xy)\big((\nu uw)(x(v,x');u[a,b] \mid w(z,w');0) \mid y[c,d]\big) \overset{C}{\vdash} a : \bullet, b : \bullet, c : \bullet, d : \bullet}\ [\text{TYP-CUT}]$$

As a result, the cyclic dependency from Example 1 is broken, and the process is deadlock free. Note that we use 0 to accommodate the standalone receive.

The following result captures common definitions of deadlock freedom (cf., e.g., [26]), where, e.g., when a process contains a non-blocked output it will reduce until an input on an opposite endpoint is non-blocked and communication can take place.

Theorem 3 (Deadlock Freedom for ACP)**.** *Given* $P \overset{\mathtt{C}}{\vdash} \varnothing$*, if* $P \overset{\mathtt{C}}{\not\longrightarrow}$*, then* $P \overset{\mathtt{C}}{\equiv} 0$*.*

Commuting conversions. The sequent calculus of linear logic induces a form of proof equivalence known as *commuting conversions*, where rule applications may be commuted past each other while preserving assumptions and conclusion. Caires and Pfenning [8] noticed that some commuting conversions correspond to structural congruences in type systems such as ACP. For example, Rule [SC-CUT-ASSOC-L] commutes two applications of Rule [TYP-CUT]. On the other hand, other commuting conversions induce more significant process transformations, e.g., they change the order of blocking prefixes. Writing $\rightsquigarrow$ to denote such transformations and annotating processes with relevant free names, we have, e.g.,

$$(\nu xy)((\nu aa')(\nu bb')z[a,b];(P_{a'} \mid Q_{b',x}) \mid R_y) \rightsquigarrow (\nu aa')(\nu bb')z[a,b];(P_{a'} \mid (\nu xy)(Q_{b',x} \mid R_y))$$

by commuting the application of the synchronous variant of Rule [TYP-SEND] past the application of Rule [TYP-CUT]. These conversions do not correspond to structural congruences but to typed behavioral equivalences (cf. [38, 39]). Importantly, this behavioral characterization of commuting conversions holds under synchronous communication, where outputs are blocking. Interestingly, DeYoung *et al.* [14] discovered that, under asynchronous communication, some of these latter commuting conversions that involve outputs correspond to simple structural congruences. This is the case for our previous example:

$$(\nu xy)((\nu bb')((\nu aa')(z[a,b] \mid P_{a'}) \mid Q_{b',x}) \mid R_y) \overset{\mathtt{C}}{\equiv} (\nu bb')((\nu aa')(z[a,b] \mid P_{a'}) \mid (\nu xy)(Q_{b',x} \mid R_y))$$

As discussed in Section 2, to study deadlock freedom at its core, it is desirable to have a setting with the least possible "amount of blockage". Hence, the above discoveries confirm that deadlock freedom is best studied under asynchronous communication, for synchronous communication entails unnecessary blockages by outputs leading to artificial sources of deadlock, whereas asynchronous communication adequately considers inputs as the only source of blockage.

5 Priorities for AP

We now consider APCP [21], which uses the same process language as AP: syntax, structural congruence, and reduction are defined exactly as in Figure 1. Following [26, 11], we extend the session types of AP (Definition 1) with *priority* annotations on binary connectives. Priorities are natural numbers. Intuitively, prefixes whose type has lower priority should not be blocked by those with higher priority.

We write $\pi, \rho, \ldots$ to denote priorities, and ω to denote the ultimate priority that is greater than all other priorities and cannot be increased further. That is, for every $\pi \in \mathbb{N}$, $\omega > \pi$ and $\omega + \pi = \omega$. Also, by abuse of notation, we reuse $\mathsf{A}, \mathsf{B}, \ldots$ to denote APCP session types.

Definition 3 (Session Types for APCP)**.** *The following grammar defines the syntax of* session types A, B*.*

$$\mathsf{A}, \mathsf{B} ::= \mathsf{A} \otimes^{\pi} \mathsf{B} \mid \mathsf{A} \parr^{\pi} \mathsf{B} \mid \oplus^{\pi}\{\mathsf{i} : \mathsf{A}\}_{\mathsf{i} \in \mathsf{I}} \mid \&^{\pi}\{\mathsf{i} : \mathsf{A}\}_{\mathsf{i} \in \mathsf{I}} \mid \bullet$$

Session types retain the same meaning as in Section 4. A name of type $\bullet$ does not require a priority, as closed names do not exhibit behavior and thus are non-blocking.

The priority of a type is determined by the priority of its outermost connective:

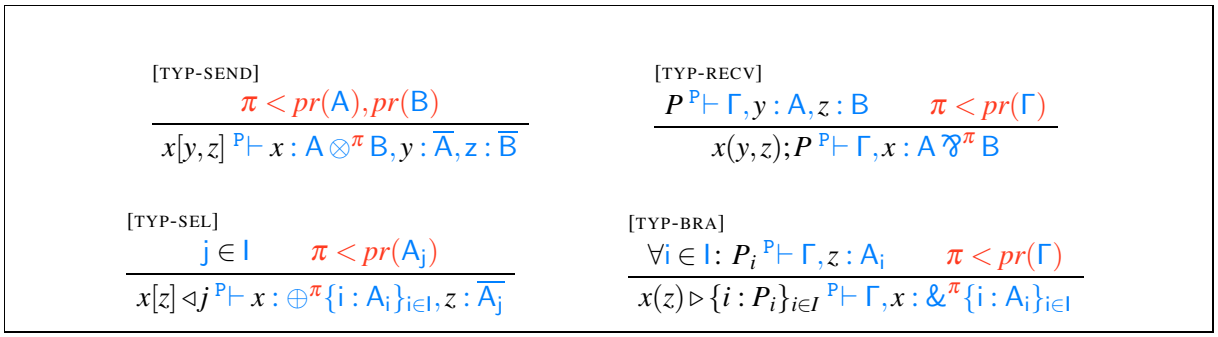


Figure 3: Typing rules of APCP that add priority conditions to the AP typing rules in Figure 2.

Definition 4 (Priorities). *For session type* A, $pr(\mathsf{A})$ *denotes its* priority*:*

$$pr(\mathsf{A} \otimes^{\pi} \mathsf{B}) \triangleq pr(\mathsf{A} \parr^{\pi} \mathsf{B}) \triangleq pr(\oplus^{\pi}\{\mathsf{i} : \mathsf{A}_\mathsf{i}\}_{\mathsf{i}\in\mathsf{I}}) \triangleq pr(\&^{\pi}\{\mathsf{i} : \mathsf{A}_\mathsf{i}\}_{\mathsf{i}\in\mathsf{I}}) \triangleq \pi \qquad pr(\bullet) \triangleq \omega$$

The priority of $\bullet$ is the constant ω: the type denotes the "final", non-blocking part of protocols. Although the connectives $\otimes$ and $\oplus$ also denote non-blocking prefixes, they do block their continuation until they are received. Hence, their priority is not constant.

APCP typing judgments are denoted $P \vdash^{\mathsf{P}} \Gamma$. We write $pr(\Gamma)$ to denote the least of the priorities of all types in Γ:

$$pr(\varnothing) \triangleq \omega \qquad pr(\Gamma, x : \mathsf{A}) \triangleq min(pr(\Gamma), pr(\mathsf{A})).$$

The typing rules of APCP ensure that prefixes with lower priority are not blocked by those with higher priority. To this end, they enforce the following laws:

1. Outputs with priority π must have messages and continuations with priority strictly larger than π;
2. A prefix typed with priority π must be prefixed only by inputs with priority strictly smaller than π;
3. Dual prefixes leading to a synchronization must have equal priorities.

The typing rules for APCP are the same as those for AP in Figure 2, and have the same meaning. To enforce the laws above, conditions on priorities are needed: Figure 3 shows the modified typing rules for prefixes. Rules [TYP-SEND] and [TYP-SEL] require that the priority of the subject is lower than the priorities of both objects (continuation and payload)—this enforces Law 1. In Rules [TYP-RECV] and [TYP-BRA], the used name's priority must be lower than the priorities of the other types in the continuation's typing context—this enforces Law 2. Law 3 is enforced by extending type duality (Definition 2) to require that $\mathsf{A} = \overline{\mathsf{B}}$ not only if the sequences of actions in A and B are complementary, but also that their priority annotations match up perfectly. This is then implicitly enforced by Rules [TYP-RES] and [TYP-FWD] (omitted from Figure 3).

We have the following results:

Theorem 4 (Type Preservation for APCP). *Given* $P \vdash^{\mathsf{P}} \Gamma$ *and* Q *such that* $P \equiv Q$ *or* $P \longrightarrow Q$*, we have* $Q \vdash^{\mathsf{P}} \Gamma$.

Theorem 5 (Deadlock Freedom for APCP). *Given* $P \vdash^{\mathsf{P}} \varnothing$*, if* $P \not\longrightarrow$*, then* $P \equiv 0$.

Example 3. *Recall the deadlocked process from Example 1:*

$$(\nu xy)(\nu uw)(x(v,x'); u[a,b] \mid w(z,w'); y[c,d])$$

An attempt to type this process under $\overset{\mathsf{P}}{\vdash}$ *reveals the cyclic dependency: the dual receive on* x *and send on* y *have priority* π*, the dual send on* u *and receive on* w *have priority* ρ*, and the two applications of Rule* [TYP-RECV] *contradictorily require* $\pi < \rho$ *and* $\rho < \pi$*.*

On the other hand, recall the non-deadlocked variant from Example 2:

$$(\nu xy)\big((\nu uw)(x(v,x');u[a,b] \mid w(z,w');0) \mid y[c,d]\big)$$

In this case, $\overset{\mathsf{P}}{\vdash}$ *only requires* $\pi < \rho$ *but not* $\rho < \pi$*, so no cyclic dependency is detected and the process is considered well typed.*

By abuse of notation, we write $\vdash$, $\overset{\mathsf{C}}{\vdash}$, and $\overset{\mathsf{P}}{\vdash}$ to denote the *sets of processes* that are well typed by the typing rules of AP, ACP, and APCP, respectively. These classes are related by strict inclusions:

Theorem 6 (Comparative Expressiveness)**.** *We have* $\overset{\mathsf{C}}{\vdash} \subset \overset{\mathsf{P}}{\vdash} \subset \vdash$.

The example above is typable under $\overset{\mathsf{C}}{\vdash}$, so we illustrate how much more expressive APCP is than ACP with a variant of Milner's cyclic scheduler [30] (inspired by [11, Example 1]), which is not typable under $\overset{\mathsf{C}}{\vdash}$. This variant is finite and has a fixed number of three participants; the interested reader can consult [21] for a recursive cyclic scheduler with $n \geq 2$ participants.

Example 4. *We construct a ring of three partial schedulers* A_i*, each of which simultaneously invokes a worker* B_i *and waits for the others to finish. Each scheduler* A_i *(*$1 \leq i \leq 3$*) communicates with its worker on name* w_i *(connected to the worker's name* z_i*), with its left neighbor on name* x_{i-1} *(*$x_0 \triangleq x_3$*), and with its right neighbor on name* y_i*. The schedulers and workers are defined as follows, writing '*$\_$*' for unused names:*

$$A_1 \triangleq (\nu a_1 y_1')(\nu b_1 w_1')(\nu c_1 \_) \big(y_1[a_1] \triangleleft start \mid w_1[b_1] \triangleleft start \mid w_1'(\_) \triangleright \{done : y_1'[c_1] \triangleleft done \mid x_3(x_3') \triangleright \{start : x_3'(\_) \triangleright \{done : 0\}\}\}\big)$$

$$A_i \triangleq (\nu a_i y_i')(\nu b_i w_i')(\nu c_i \_) \big(x_{i-1}(x_{i-1}') \triangleright \{start : w_i[b_i] \triangleleft start \mid y_i[a_i] \triangleleft start \mid w_i'(\_) \triangleright \{done : x_{i-1}'(\_) \triangleright \{done : y_i'[c_i] \triangleleft done\}\}\}\big) \qquad 2 \leq i < 3$$

$$B_i \triangleq (\nu d_i \_)(z_i(z_i') \triangleright \{start : doSomething(); z_i'[d_i] \triangleleft done\}) \qquad 1 \leq i \leq 3$$

Thus, A_1 *starts the routine by signaling its right neighbor and worker. It then waits for its worker to finish, and signals its right neighbor it is done. Only in the end does it wait for a start signal from its left neighbor, and then wait for it to finish. The other two schedulers are defined identically. They wait for a start signal from their left neighbor, and then start their worker. After its worker is done, it signals to its right neighbor, and waits for its left neighbor to finish. Workers wait for a start signal, do some task (*$doSomething()$ *stands for some arbitrary computation), and signal that they are finished afterwards.*

A complete, cyclic scheduler Sched is then formed by connecting the partial schedulers on channels between x_i *and* y_i *for* $1 \leq i \leq 3$*.*

$$Sched \triangleq (\nu x_1 y_1)(\nu x_2 y_2)(\nu x_3 y_3)\big((\nu w_1 z_1)(A_1 \mid B_1) \mid (\nu w_2 z_2)(A_2 \mid B_2) \mid (\nu w_3 z_3)(A_3 \mid B_3)\big)$$

To type Sched, we assign the following types to names for $1 \leq i \leq 3$*:*

- $w_i : \oplus^{\pi_i}\{start : \&^{\pi_i'}\{done : \bullet\}\}$,
- $x_i : \&^{\rho_i}\{start : \&^{\rho_i'}\{done : \bullet\}\}$,
- $y_i : \oplus^{\sigma_i}\{start : \oplus^{\sigma_i'}\{done : \bullet\}\}$

- $z_i : \&^{\phi_i}\{\mathsf{start} : \oplus^{\phi_i'}\{\mathsf{done} : \bullet\}\}$.

Applications of Rule [TYP-RES] *require that* $\pi_i = \phi_i$, $\pi_i' = \phi_i'$, $\rho_i = \sigma_i$, *and* $\rho_i' = \sigma_i'$, *for* $1 \leq i \leq 3$. *Applications of Rules* [TYP-SEL] *and* [TYP-BRA] *then require:*

- $\rho_1 < \rho_1'$, *and* $\pi_1 < \pi_1' < \rho_1', \rho_3$;
- $\rho_1 < \pi_2, \rho_2, \pi_2', \rho_2'$, $\pi_2 < \pi_2'$, $\rho_2 < \rho_2'$, *and* $\pi_2' < \rho_1', \rho_2'$;
- $\rho_2 < \pi_3, \rho_3, \pi_3', \rho_3'$, $\pi_3 < \pi_3'$, $\rho_3 < \rho_3'$, *and* $\pi_3' < \rho_2', \rho_3'$.

We verify that these requirements are consistent, so $\mathsf{Sched} \overset{\mathtt{P}}{\vdash} \varnothing$. *Hence,* $\mathsf{Sched} \in \overset{\mathtt{P}}{\vdash} \setminus \overset{\mathtt{C}}{\vdash}$, *yet the process is deadlock free (following Theorem 5).*

6 Asynchronous Functional Sessions

With APCP we have established a solid foundation for asynchronous processes that are deadlock free by typing (Theorem 5). To bring these foundations closer to programming calculi, we consider LAST: a concurrent λ-calculus in which asynchronous message-passing is governed using session types [16]. In LAST, which stands for Linear Asynchronous Session Types, channels can form cyclic connections and deadlock freedom is not guaranteed by typing.

The variant LAST$^{\mathrm{n}}$ that we consider here is, in spirit, the functional variant of AP (Section 3): it is obtained via translation into AP, resulting in a call-by-name semantics (rather that LAST's call-by-value semantics) and an explicit treatment of variable substitution. Prior works ensure deadlock freedom for synchronous variants of LAST by extending the type system with priorities [35, 27]. Rather than accommodating these contributions to the asynchronous setting, as in Section 5, for LAST$^{\mathrm{n}}$ we develop an alternative approach to deadlock freedom: LAST$^{\mathrm{n}}$ translates into AP, and operational correctness of the translation guarantees that well typedness under $\overset{\mathtt{P}}{\vdash}$ implies deadlock freedom for the source program.

We illustrate the call-by-name semantics and explicit substitutions in LAST$^{\mathrm{n}}$ using a simple example.

Example 5. *Using standard* λ*-calculus notation, the following sequence of term reductions (*$\overset{\mathtt{M}}{\longrightarrow}$*) and structural congruences (*$\overset{\mathtt{M}}{\equiv}$*) illustrate the call-by-name semantics and explicit substitution (*$\{\!\{\ldots\}\!\}$*) of* LAST$^{\mathrm{n}}$*:*

$$\begin{aligned}(\lambda x.x\ (\lambda y.y))\ ((\lambda w.w)\ (\lambda z.z)) &\overset{\mathtt{M}}{\longrightarrow} (x\ (\lambda y.y))\{\!\{((\lambda w.w)\ (\lambda z.z))/x\}\!\}\\ &\overset{\mathtt{M}}{\equiv} (x\{\!\{((\lambda w.w)\ (\lambda z.z))/x\}\!\})\ (\lambda y.y)\\ &\overset{\mathtt{M}}{\longrightarrow} ((\lambda w.w)\ (\lambda z.z))\ (\lambda y.y)\\ &\overset{\mathtt{M}}{\longrightarrow} (w\{\!\{(\lambda z.z)/w\}\!\})\ (\lambda y.y) \overset{\mathtt{M}}{\longrightarrow} (\lambda z.z)\ (\lambda y.y)\\ &\overset{\mathtt{M}}{\longrightarrow} z\{\!\{(\lambda y.y)/z\}\!\} \overset{\mathtt{M}}{\longrightarrow} \lambda y.y\end{aligned}$$

Notice how every function application immediately evolves into an explicit substitution, instead of first reducing the argument to a value as a call-by-value semantics would.

LAST$^{\mathrm{n}}$ programs are configurations $C, D, \ldots$ of parallel threads executing functional terms $M, N, \ldots$ connected by buffered channels for asynchronous message-passing. The semantics of configurations is defined in terms of a reduction relation, denoted $C \overset{\mathtt{C}}{\longrightarrow} D$. We write $\overset{\mathtt{C}}{\longrightarrow}^*$ for the reflexive, transitive closure of $\overset{\mathtt{C}}{\longrightarrow}$, and $C \overset{\mathtt{C}}{\not\longrightarrow}$ if there is no D such that $C \overset{\mathtt{C}}{\longrightarrow} D$. Next, we briefly discuss the types of LAST$^{\mathrm{n}}$, and thereafter focus on the deadlock-freedom guarantee through translation. We refer to [21] for details about the language and type system of LAST$^{\mathrm{n}}$.

Types can be divided into functional types T, U and session types S:

$$\begin{array}{rcl|rcl|rcl|rcl|l} T,U & ::= & T \times U & \text{pair} & T \multimap U & \text{function} & 1 & \text{unit} & S & \text{session} & \\ S & ::= & !T.S & \text{send} & ?T.S & \text{receive} & \oplus\{i : T\}_{i\in I} & \text{select} & \&\{i : T\}_{i\in I} & \text{branch} & \mathsf{end} \end{array}$$

Configurations may contain at most one main thread with a return type, denoted $\blacklozenge M$, and arbitrarily many unit-typed child threads, denoted $\lozenge N$. Configurations are then typed $\Gamma \vdash^{c}_{\phi} C : T$. The annotation ϕ can be $\blacklozenge$ or $\lozenge$, denoting whether C contains a main thread or not, respectively. The typing context Γ is a list of variable-type assignments $x : U$, and T is the configuration's return type ($T = 1$ if $\phi = \lozenge$).

We translate configurations into AP processes. Our translation crucially relies on the typing of configurations, in particular to correctly translate buffers. Note that AP processes have no functional behavior, so as usual our translation is parametric on a dedicated name on which the source configuration's return type can be observed. In the following we describe the translation focussing on types, operational correctness, and deadlock freedom; again, we refer to [21] for details on other aspects.

Definition 5 (Translation)**.** *The translation of typed* LAST^{n} *configurations into* AP *processes is denoted* $[\![\Gamma \vdash^{c}_{\phi} C : T]\!]_z$*, where* z *is the name on which* T *can be observed.*

Although this translation does not return typed processes, it does preserve typing. Notice that LAST^{n} and AP have different types, and so the connection between LAST^{n} configurations and AP processes (Definition 5) can be captured by a translation on types: it reflects how the translation on configurations adds additional synchronizations that ensure that the behavior of configurations is soundly captured by the "more concurrent" nature of AP (i.e., the translation does not add behavior not present in the source configuration). This way, the following definition illustrates the work done by the translation to ensure operational soundness (discussed hereafter). Given $\Gamma \vdash^{c}_{\phi} C : T$, we refer to types in Γ as *context types*.

Definition 6 (Translation: Types)**.** *The translation of* LAST^{n} *types is denoted* $[\![T]\!]$*, and* $(\!|T|\!)$ *for context types. They are defined mutually, by induction on the structure of* T.

$$\begin{array}{lll} (\!|T|\!) \triangleq \bullet \otimes \overline{[\![T]\!]} & & \\ [\![T \times U]\!] \triangleq \overline{(\!|T|\!)} \otimes \overline{(\!|U|\!)} & [\![T \multimap U]\!] \triangleq (\!|T|\!) \parr [\![U]\!] & [\![1]\!] \triangleq \bullet \\ [\![!T.S]\!] \triangleq \bullet \otimes (\!|T|\!) \parr \overline{(\!|S|\!)} & [\![\oplus\{i : S_i\}_{i\in I}]\!] \triangleq \bullet \otimes \&\{i : \overline{(\!|S_i|\!)}\}_{i\in I} & [\![\mathsf{end}]\!] \triangleq \bullet \otimes \bullet \\ [\![?T.S]\!] \triangleq \overline{(\!|T|\!)} \otimes \overline{(\!|S|\!)} & [\![\&\{i : S_i\}_{i\in I}]\!] \triangleq \oplus\{i : \overline{(\!|S_i|\!)}\}_{i\in I} & \end{array}$$

Context type translation extends naturally to typing contexts $(\!|\Gamma|\!)$.

Example 6. *We illustrate the translation of types by means of an example:*

$$\begin{aligned} [\![(T \times S) \multimap !T.S]\!] &= (\!|T \times S|\!) \parr [\![!T.S]\!] \\ &= (\bullet \otimes \overline{[\![T \times S]\!]}) \parr \bullet \otimes (\!|T|\!) \parr \overline{(\!|S|\!)} \\ &= (\bullet \otimes \overline{\overline{(\!|T|\!)} \otimes \overline{(\!|S|\!)}}) \parr \bullet \otimes (\bullet \otimes \overline{[\![T]\!]}) \parr \bullet \otimes \overline{\overline{[\![S]\!]}} \\ &= (\bullet \otimes (\!|T|\!) \parr (\!|S|\!)) \parr \bullet \otimes (\bullet \otimes \overline{[\![T]\!]}) \parr \bullet \parr [\![S]\!] \\ &= (\bullet \otimes (\bullet \otimes \overline{[\![T]\!]}) \parr \bullet \otimes \overline{[\![S]\!]}) \parr \bullet \otimes (\bullet \otimes \overline{[\![T]\!]}) \parr \bullet \parr [\![S]\!] \end{aligned}$$

Figure 4 breaks down the resulting AP *type and explains it in terms of the associated behavior of a process translated from a term implementing the type.*

The first property of the translation is type preservation, under $\vdash$:

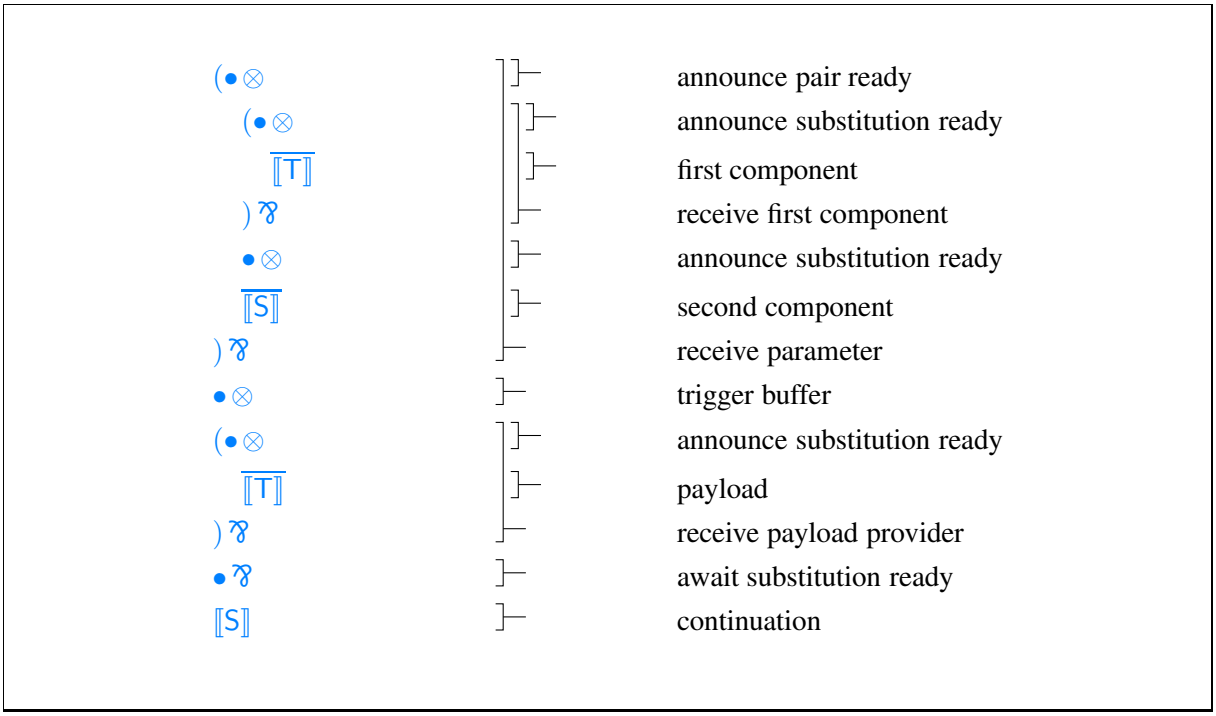


Figure 4: The translation of $(\mathsf{T} \times \mathsf{S}) \multimap !\mathsf{T}.\mathsf{S}$, given in Example 6, in detail.

Theorem 7 (Translation: Type Preservation)**.** *Suppose given* $P = [\![\Gamma \overset{C}{\vdash}{}^{\phi} \mathrm{C} : \mathsf{T}]\!]_z$. *Then* $P \vdash (\!|\Gamma|\!), z : [\![\mathsf{T}]\!]$.

Let us now make precise what we mean by our translation being operationally sound. The property is based on Gorla's criteria for correct translations [18]. We focus on operational correctness, which encodes a correspondence between the behavior of a source program and its translation. Operational soundness then means that any behavior of the translation is reflected by behavior of the source program. In context of our translation, this means that at any state of execution of a translated configuration, another state can be reached that is the translation of a configuration reachable from the source configuration. Note that the other direction, where any behavior of the source program is preserved by the translation (operational completeness), holds as well but is omitted for brevity.

Theorem 8 (Translation: Operational Soundness)**.** *Suppose given* $P = [\![\Gamma \overset{C}{\vdash}{}^{\phi} \mathrm{C} : \mathsf{T}]\!]_z$, *and suppose* $P \longrightarrow^* Q$. *Then there exists* D *such that* $\mathrm{C} \overset{C}{\longrightarrow}{}^* \mathrm{D}$ *and* $Q \longrightarrow^* Q'$ *for* $Q' = [\![\Gamma \overset{C}{\vdash}{}^{\phi} \mathrm{D} : \mathsf{T}]\!]_z$.

As discussed before, the type system for LAST$^{\mathrm{n}}$ cannot guarantee deadlock freedom by itself. However, we can rely on APCP for an indirect result. Operational soundness is key here, as it guarantees that if a translated configuration is able to reduce, then so is the source configuration. We are then able to prove deadlock freedom for closed configurations (with empty typing context and unit return type) if their translation is well typed under $\overset{P}{\vdash}$. The idea is that a closed configuration translates into a closed AP process. Well typedness under $\overset{P}{\vdash}$ (more stringent than under $\vdash$) then guarantees deadlock freedom (Theorem 5), which transfers back to the source configuration through operational soundness. Hence, we state a deadlock-freedom result for LAST$^{\mathrm{n}}$ in the spirit of Theorems 3 and 5 (recall: given $P \overset{P}{\vdash} \varnothing$, if $P \not\longrightarrow$, then $P \equiv 0$), where the inactive process 0 corresponds to a main thread of unit value $\blacklozenge\,()$.

Theorem 9 (Deadlock Freedom for LAST$^{\mathrm{n}}$)**.** *If* $[\![\varnothing \overset{C}{\vdash}{}^{\blacklozenge} \mathrm{C} : 1]\!]_z$ *is well typed under* $\overset{P}{\vdash}$, *then* $\mathrm{C} \overset{C}{\not\longrightarrow}$ *implies* $\mathrm{C} \overset{C}{\equiv} \blacklozenge\,()$.

Hence, Theorem 9 defines a *proof technique* for enforcing deadlock freedom for LAST^{n} programs, which, given a program, first applies the translation into asynchronous processes and then checks for typability in APCP (i.e., under $^{P}\vdash$). Our deadlock-freedom result is conditional in that it is contingent on correct typability in APCP. A practical procedure for type reconstruction/inference would be needed to correctly associate priorities to the session types obtained using the translation in Definition 6.

7 Conclusion

This paper has presented an overview of recent work on static verification techniques for message-passing processes with asynchronous communication. Based on our results in [21], we have provided a unified presentation for different forms of deadlock enforcement based on type systems. Our presentation involves four typed languages: AP, ACP, APCP, and LAST^{n}.

We started by presenting AP, a simple session π-calculus with asynchronous communication, whose typing discipline enforces conformance to session protocols but does not exclude deadlocks. As such, AP is representative of a class of typed process frameworks that make the conscious decision of imposing minimal conditions over the processes/programs that can be typed (cf. [16, 45, 44])—in these frameworks, the programmer has wide agency to write typable programs, but also the responsibility of ensuring that their programs enjoy correctness guarantees that go beyond protocol conformance.

Subsequently, we presented ACP, an asynchronous variant of Wadler's CP, which results from AP via a simple but crucial modification: the two separate typing rules for process composition and restriction in AP are replaced in ACP by a single rule that coalesces both constructs following the cut rule in linear logic (Rule [TYP-CUT]). ACP is based on the logical correspondence that connects session types and *classical* linear logic; its formulation follows the presentation by DeYoung *et al.* [14], which was given in the setting of *intuitionistic* linear logic. To our knowledge, an asynchronous version of CP had not been presented before, so this can be considered an original (yet modest) contribution of this paper. Owing to its logical foundations, ACP inherits the known expressiveness limitations of its predecessors: it can only enforce deadlock freedom for processes that form tree-like topologies.

APCP then arises as another variant of AP, aimed at overcoming these limitations by adopting the key features from priority-based approaches. This ensures that APCP can enforce deadlock freedom for processes forming cyclic topologies. We briefly discussed how AP and APCP provide a suitable foundation for a correct interpretation of a concurrent functional calculus with asynchronous sessions, dubbed LAST^{n}. The fact that AP, ACP, and APCP determine a wide spectrum of techniques for enforcing deadlock freedom is important when considering LAST^{n}. In fact, AP suffices for a basic concurrent interpretation of functional sessions; if the interest is in deadlock-free behaviors, then APCP (and, to some extent, ACP) can provide an indirect approach based on a correct typed translation.

Here we have focused on typing disciplines for binary (two-party) protocols. Further applications of APCP include the analysis of *multiparty* (n-ary) protocols, which involve more than two parties and for which the analysis of deadlock freedom is both important and challenging, especially when processes can interleave actions from different sessions and exchange references to sessions in communications (aka *delegation*), as supported in AP, ACP, and APCP. Interestingly, APCP provides a basis for the decentralized (static) analysis of process implementations of multiparty protocols, as developed in [20]; this decentralized analysis, in turn, can be adapted also to a setting with dynamic (run-time) verification, to consider (untyped) asynchronous processes whose behavior is governed by a monitoring infrastructure based on session types [22]. The PhD thesis of the first author provides a unified account of APCP—its theory and applications—, but also detailed comparisons with related works [19].

Acknlowedgments We are grateful to the organizers and participants of ICE'24 for their comments and to Juan C. Jaramillo for discussions and comments on previous versions of this document.

The research described here has been supported by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 ('Unifying Correctness for Communicating Software').

References

[1] Roberto M. Amadio, Ilaria Castellani & Davide Sangiorgi (1998): *On Bisimulations for the Asynchronous Pi-Calculus*. *Theoretical Computer Science* 195(2), pp. 291–324, doi:10.1016/S0304-3975(97)00223-5.

[2] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos & Nobuko Yoshida (2016): *Behavioral Types in Programming Languages*. *Found. Trends Program. Lang.* 3(2-3), pp. 95–230, doi:10.1561/2500000031.

[3] Romain Beauxis, Catuscia Palamidessi & Frank D. Valencia (2008): *On the Asynchronous Nature of the Asynchronous Pi-Calculus*. In Pierpaolo Degano, Rocco De Nicola & José Meseguer, editors: *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, *Lecture Notes in Computer Science* 5065, Springer, pp. 473–492, doi:10.1007/978-3-540-68679-8_29.

[4] Gérard Boudol (1992): *Asynchrony and the Pi-calculus*. Research Report RR-1702, INRIA.

[5] Diletta Cacciagrano, Flavio Corradini & Catuscia Palamidessi (2007): *Separation of Synchronous and Asynchronous Communication via Testing*. *Theoretical Computer Science* 386(3), pp. 218–235, doi:10.1016/J.TCS.2007.07.009.

[6] Luís Caires (2014): *Types and Logic, Concurrency and Non-Determinism*. Technical Report MSR-TR-2014-104, In Essays for the Luca Cardelli Fest, Microsoft Research.

[7] Luís Caires & Jorge A. Pérez (2017): *Linearity, Control Effects, and Behavioral Types*. In Hongseok Yang, editor: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, *Lecture Notes in Computer Science* 10201, Springer, pp. 229–259, doi:10.1007/978-3-662-54434-1_9.

[8] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, *Lecture Notes in Computer Science* 6269, Springer, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[9] Luís Caires, Frank Pfenning & Bernardo Toninho (2016): *Linear Logic Propositions as Session Types*. *Mathematical Structures in Computer Science* 26(3), pp. 367–423, doi:10.1017/S0960129514000218.

[10] Mario Coppo, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2007): *Asynchronous Session Types and Progress for Object Oriented Languages*. In Marcello M. Bonsangue & Einar Broch Johnsen, editors: *Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, *Lecture Notes in Computer Science* 4468, Springer, pp. 1–31, doi:10.1007/978-3-540-72952-5_1.

[11] Ornela Dardha & Simon J. Gay (2018): *A New Linear Logic for Deadlock-Free Session-Typed Processes*. In Christel Baier & Ugo Dal Lago, editors: *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, *Lecture Notes in Computer Science* 10803, Springer, pp. 91–109, doi:10.1007/978-3-319-89366-2_5.

[12] Ornela Dardha & Jorge A. Pérez (2015): *Comparing Deadlock-Free Session Typed Processes*. In Silvia Crafa & Daniel Gebler, editors: *Proceedings of the Combined 22th International Workshop on Expressiveness in*

Concurrency and 12th Workshop on Structural Operational Semantics, EXPRESS/SOS 2015, Madrid, Spain, 31st August 2015, *EPTCS* 190, pp. 1–15, doi:10.4204/EPTCS.190.1.

[13] Ornela Dardha & Jorge A. Pérez (2022): *Comparing Type Systems for Deadlock Freedom*. *Journal of Logical and Algebraic Methods in Programming* 124, p. 100717, doi:10.1016/J.JLAMP.2021.100717.

[14] Henry DeYoung, Luís Caires, Frank Pfenning & Bernardo Toninho (2012): *Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication*. In Patrick Cégielski & Arnaud Durand, editors: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, *LIPIcs* 16, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 228–242, doi:10.4230/LIPICS.CSL.2012.228.

[15] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous & Nobuko Yoshida (2009): *Objects and Session Types*. *Information and Computation* 207(5), pp. 595–641, doi:10.1016/J.IC.2008.03.028.

[16] Simon J. Gay & Vasco T. Vasconcelos (2010): *Linear Type Theory for Asynchronous Session Types*. *Journal of Functional Programming* 20(1), pp. 19–50, doi:10.1017/S0956796809990268.

[17] Rob J. van Glabbeek (2018): *On the Validity of Encodings of the Synchronous in the Asynchronous π-Calculus*. *Information Processing Letters* 137, pp. 17–25, doi:10.1016/J.IPL.2018.04.015.

[18] Daniele Gorla (2010): *Towards a Unified Approach to Encodability and Separation Results for Process Calculi*. *Information and Computation* 208(9), pp. 1031–1053, doi:10.1016/J.IC.2010.05.002.

[19] Bas van den Heuvel (2024): *Correctly Communicating Software: Distributed, Asynchronous, and Beyond*. Ph.D. thesis, University of Groningen / University of Groningen, doi:10.33612/diss.929078700.

[20] Bas van den Heuvel & Jorge A. Pérez (2022): *A Decentralized Analysis of Multiparty Protocols*. *Science of Computer Programming* 222, p. 102840, doi:10.1016/J.SCICO.2022.102840.

[21] Bas van den Heuvel & Jorge A. Pérez (2024): *Asynchronous Session-Based Concurrency: Deadlock-freedom in Cyclic Process Networks*. *Logical Methods in Computer Science* 20(4), doi:10.46298/LMCS-20(4:6)2024.

[22] Bas van den Heuvel, Jorge A. Pérez & Rares A. Dobre (2023): *Monitoring Blackbox Implementations of Multiparty Session Protocols*. In Panagiotis Katsaros & Laura Nenzi, editors: *Runtime Verification - 23rd International Conference, RV 2023, Thessaloniki, Greece, October 3-6, 2023, Proceedings*, *Lecture Notes in Computer Science* 14245, Springer, pp. 66–85, doi:10.1007/978-3-031-44267-4_4.

[23] Kohei Honda & Mario Tokoro (1991): *An Object Calculus for Asynchronous Communication*. In Pierre America, editor: *ECOOP'91 European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991, Proceedings*, *Lecture Notes in Computer Science* 512, Springer, pp. 133–147, doi:10.1007/BFB0057019.

[24] Kohei Honda & Mario Tokoro (1992): *On Asynchronous Communication Semantics*. In Mario Tokoro, Oscar Nierstrasz & Peter Wegner, editors: *Object-Based Concurrent Computing, ECOOP'91 Workshop, Geneva, Switzerland, July 15-16, 1991, Proceedings*, *Lecture Notes in Computer Science* 612, Springer, pp. 21–51, doi:10.1007/3-540-55613-3_2.

[25] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, *Lecture Notes in Computer Science* 1381, Springer, pp. 122–138, doi:10.1007/BFB0053567.

[26] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In Christel Baier & Holger Hermanns, editors: *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, *Lecture Notes in Computer Science* 4137, Springer, pp. 233–247, doi:10.1007/11817949_16.

[27] Wen Kokke & Ornela Dardha (2023): *Prioritise the Best Variation*. *Logical Methods in Computer Science* 19(4), doi:10.46298/LMCS-19(4:28)2023.

[28] Dimitrios Kouzapas, Nobuko Yoshida & Kohei Honda (2011): *On Asynchronous Session Semantics*. In Roberto Bruni & Jürgen Dingel, editors: *Formal Techniques for Distributed Systems - Joint 13th IFIP WG*

6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings, *Lecture Notes in Computer Science* 6722, Springer, pp. 228–243, doi:10.1007/978-3-642-21461-5_15.

[29] Julien Lange & Nobuko Yoshida (2017): *On the Undecidability of Asynchronous Session Subtyping*. In Javier Esparza & Andrzej S. Murawski, editors: *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, *Lecture Notes in Computer Science* 10203, pp. 441–457, doi:10.1007/978-3-662-54458-7_26.

[30] Robin Milner (1989): *Communication and Concurrency*. PHI Series in Computer Science, Prentice Hall.

[31] Robin Milner (1993): *The Polyadic π-Calculus: A Tutorial*. In Friedrich L. Bauer, Wilfried Brauer & Helmut Schwichtenberg, editors: *Logic and Algebra of Specification*, Springer, Berlin, Heidelberg, pp. 203–246, doi:10.1007/978-3-642-58041-3_6.

[32] Uwe Nestmann (2000): *What Is a "Good" Encoding of Guarded Choice?* *Information and Computation* 156(1-2), pp. 287–319, doi:10.1006/INCO.1999.2822.

[33] Uwe Nestmann & Benjamin C. Pierce (2000): *Decoding Choice Encodings*. *Information and Computation* 163(1), pp. 1–59, doi:10.1006/INCO.2000.2868.

[34] Luca Padovani (2014): *Deadlock and Lock Freedom in the Linear π-Calculus*. In Thomas A. Henzinger & Dale Miller, editors: *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, ACM, pp. 72:1–72:10, doi:10.1145/2603088.2603116.

[35] Luca Padovani & Luca Novara (2015): *Types for Deadlock-Free Higher-Order Programs*. In Susanne Graf & Mahesh Viswanathan, editors: *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, *Lecture Notes in Computer Science* 9039, Springer, pp. 3–18, doi:10.1007/978-3-319-19195-9_1.

[36] Catuscia Palamidessi (2003): *Comparing the Expressive Power of the Synchronous and Asynchronous Pi-Calculi*. *Mathematical Structures in Computer Science* 13(5), pp. 685–719, doi:10.1017/S0960129503004043.

[37] Catuscia Palamidessi, Vijay A. Saraswat, Frank D. Valencia & Björn Victor (2006): *On the Expressiveness of Linearity vs Persistence in the Asychronous Pi-Calculus*. In: *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, IEEE Computer Society, pp. 59–68, doi:10.1109/LICS.2006.39.

[38] Jorge A. Pérez, Luís Caires, Frank Pfenning & Bernardo Toninho (2012): *Linear Logical Relations for Session-Based Concurrency*. In Helmut Seidl, editor: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, *Lecture Notes in Computer Science* 7211, Springer, pp. 539–558, doi:10.1007/978-3-642-28869-2_27.

[39] Jorge A. Pérez, Luís Caires, Frank Pfenning & Bernardo Toninho (2014): *Linear Logical Relations and Observational Equivalences for Session-Based Concurrency*. *Information and Computation* 239, pp. 254–302, doi:10.1016/J.IC.2014.08.001.

[40] Paola Quaglia & David Walker (2005): *Types and Full Abstraction for Polyadic* pi-*Calculus*. *Information and Computation* 200(2), pp. 215–246, doi:10.1016/J.IC.2005.03.004.

[41] Davide Sangiorgi & David Walker (2001): *The Pi-Calculus - a Theory of Mobile Processes*. Cambridge University Press.

[42] Peter Selinger (1997): *First-Order Axioms for Asynchrony*. In Antoni W. Mazurkiewicz & Józef Winkowski, editors: *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, *Lecture Notes in Computer Science* 1243, Springer, pp. 376–390, doi:10.1007/3-540-63141-0_26.

[43] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-Based Language and Its Typing System*. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou & Sergios Theodoridis, editors: *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, *Lecture Notes in Computer Science* 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118.

[44] Peter Thiemann & Vasco T. Vasconcelos (2016): *Context-Free Session Types*. In Jacques Garrigue, Gabriele Keller & Eijiro Sumii, editors: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, ACM, pp. 462–475, doi:10.1145/2951913.2951926.

[45] Vasco T. Vasconcelos (2012): *Fundamentals of Session Types*. *Information and Computation* 217, pp. 52–70, doi:10.1016/J.IC.2012.05.002.

[46] Philip Wadler (2012): *Propositions as Sessions*. In Peter Thiemann & Robby Bruce Findler, editors: *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, ACM, pp. 273–286, doi:10.1145/2364527.2364568.

An Overview of the Decentralized Reconfiguration Language Concerto-D through its Maude Formalization

Farid Arfi Hélène Coullon
IMT Atlantique, Inria, LS2N, UMR 6004, F-44000 Nantes, France
{farid.arfi, helene.coullon}@imt-atlantique.fr

Frédéric Loulergue
Univ. Orléans, INSA CVL, LIFO EA 4022, France
frederic.loulergue@univ-orleans.fr

Jolan Philippe
IMT Atlantique, Inria, LS2N, UMR 6004, F-44000 Nantes, France
jolan.philippe@imt-atlantique.fr

Simon Robillard
LIRMM, CNRS, Université de Montpellier, France
simon.robillard@umontpellier.fr

We propose an overview of the decentralized reconfiguration language CONCERTO-D through its Maude formalization. CONCERTO-D extends the already published CONCERTO language. CONCERTO-D improves on two different parameters compared with related work: the decentralized coordination of numerous local reconfiguration plans which avoid a single point of failure when considering unstable networks such as edge computing, or cyber-physical systems (CPS) for instance; and a mechanized formal semantics of the language with Maude which offers guarantees on the executability of the semantics. Throughout the paper, the CONCERTO-D language and its semantics are exemplified with a reconfiguration extracted from a real case study on a CPS. We rely on the Maude formal specification language, which is based on rewriting logic, and consequently perfectly suited for describing a concurrent model.

1 Introduction

Running and maintaining large-scale distributed software is now a commonplace activity, but managing the inherent complexity of this task requires dedicated tools, models, and languages. The complexity is particularly apparent when distributed software needs to be reconfigured during execution, either to satisfy changing requirements or to carry out maintenance operations.

The DevOps community (and associated tools) as well as component-base software engineering (CBSE) are the main domains focussing on the deployment and reconfiguration of distributed software systems. A reconfiguration consists of a set of actions to execute on the different pieces of software, distributed across the network, to lead the system in the new desired configuration (i.e., state). In these domains, actions are almost always orchestrated by a central coordinator [25, 9], i.e., an entity that keeps track of the actions to apply and their dependencies, as well as the global state of the distributed system.

However, a centralized model is necessarily limited in terms of resilience, as it creates a single point of failure. For instance, in the context of constrained (e.g., energy, communications) cyber-physical systems [21, 22], or edge computing where network disconnections are commonplace, as well as in the

C. Aubert, C. Di Giusto, S. Fowler & V. K. I Pun (Eds.):
17th Interaction and Concurrency Experience (ICE 2024)
EPTCS 414, 2024, pp. 21–38, doi:10.4204/EPTCS.414.2

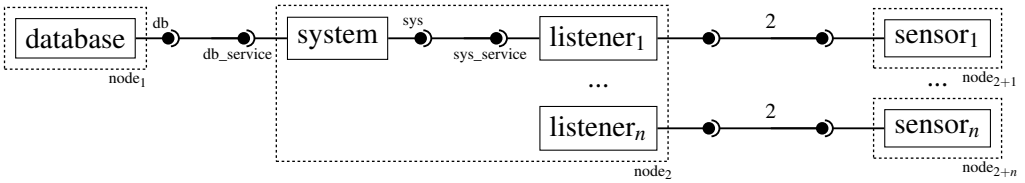


Figure 1: Full overview of the CPS use case with $2+n$ nodes hosting *database*, *system*, *n listeners*, and *n sensors*.

context of large-scale projects where cross-DevOps teams [16, 23, 25, 26] have to collaborate, decentralized reconfiguration models are preferred.

With this work, we extend the semantics of the reconfiguration language CONCERTO [6] and turn it into a decentralized model called CONCERTO-D, by extending the semantics to describe the specifics of communication and synchronization between components. In CONCERTO-D multiple coordinators collaborate to achieve their respective local reconfigurations (one for each node). Both CONCERTO [4, 5] and CONCERTO-D [23, 21, 22] have been the subject of experimental studies to validate the approach and compare it to related work [10, 25]. In particular, and while this is not the main subject of this paper, CONCERTO and CONCERTO-D allow better parallel execution of reconfiguration actions compared to the related work, thus leading to faster reconfigurations.

To support the development of this decentralized semantics, it appeared necessary to go beyond a pen-and-paper approach and to provide a mechanized formalization of the semantics. To this end, we used Maude [8, 20], a language based on rewriting logic, and consequently perfectly suited to describing a concurrent model. Developing and formalizing the semantics of CONCERTO-D in Maude helped us manage the complexity of the model, and clarify it where needed, but it also allowed us to generate tools such as an interpreter and a model-checker for these semantics. A Maude program describes a logical theory, while a Maude computation consists of logical deductions using the axioms specified in the theory.

Throughout the paper, we will use an example taken from a real case study from the literature: a wildlife monitoring system that utilizes sensors to capture animal sounds [19] illustrated in Figure 1. These sensors are linked to a gateway and calibrated to specific sound frequencies. Reconfiguration is regularly required in this system to adjust the listening frequency of each sensor. However, during the reconfiguration process, the sensor must temporarily cease listening. Similarly, if the gateway fails to receive data from a sensor, the sensor is required to halt its observation activities. The collected data is stored in a remote database. In such a use case, disconnections are common between listeners and sensors. A central coordination of the reconfiguration program is typically a single point of failure, which makes the system fully inactive during unavailability. Furthermore, when facing disconnections, it is more difficult to maintain the global state of the system, which slows down the process: synchronization with the central entity is required even for purely local actions. A decentralized language such as CONCERTO-D is preferable in such a context.

In this paper, we give an overview of the semantics of CONCERTO-D through its Maude formalization. The complete Maude specification with all the rules is available at `https://doi.org/10.5281/zenodo.12786127`.

First, Section 2 presents how the life cycles of components, and the creation and modification of a components assembly are formalized in CONCERTO-D. Second, Section 3 gives an overview of the main semantics rules of CONCERTO-D and how are formalized communications in CONCERTO-D. Finally,

Section 4 and Section 5 respectively give the related work and a conclusion on this contribution.

2 Control components and reconfiguration language

This section presents the structural concepts of CONCERTO-D as well as its reconfiguration language. Some elements of the formalization in Maude are included, as well as examples based on our case study.

2.1 Control components

In CONCERTO-D, a distributed software system is modeled as an assembly of control component instances, linked together via *ports* (i.e., interfaces), to represent dependencies in their life cycles (e.g., data exchanges, or other interactions between components). There are two types of ports in such connections: *provide* ports and *use* ports. A provide port represents information or a resource offered by a component when the port is active. Conversely, a use port indicates that a component requires some information or resources to perform.

Each piece of software (i.e., component, service), identified by a unique identifier, is an instance of a control component type, that models a component's life cycle as a set of *places* and *transitions*. Each port is bound to a subset of the life cycle (i.e., places and transitions) that is called a group. Places represent milestones of the life cycle. A specific initial place serves as the starting point for the component's life cycle. Transitions represent concrete actions to perform between places (e.g., admin commands).

The dynamic nature of components is captured by their *behaviors*. A behavior is a subset of the component's transitions. At any given point, a control component instance executes one behavior, i.e., only the transitions in this behavior can be fired. Requests to change the active behavior for a component (see reconfiguration actions in Section 2.2) are queued and executed in the order in which they are received: a behavior remains active until no more transitions in it can be fired, at which point the behavior is changed to the next requested behavior, if any. This mechanism makes up the behavioral interface of components.

Informal example. Figure 1 depicts an assembly modeling our use case, where distributed sensors interact with a system through listeners. Physically, the components are distributed on several nodes. The first node serves as a host for the *database* component responsible for storing recorded data obtained from sensors. The second node comprises the components `system` and `listener`. The *system* component establishes a connection with the database through a *use* port (represented using the UML notation), enabling the usage of the database's service. Furthermore, the *system* is interconnected with *n listeners*, each corresponding to a program responsible for monitoring and reconfiguring remote sensors. These listeners are connected to `system` via their use port. Finally, each $\texttt{sensor}_i$ is hosted on a node and is connected to its associated $\texttt{listener}_i$.

Figure 2 shows the internals of some components of this assembly, omitting others (i.e., *database*, *system*) for clarity. Each listener has four places: `off`, `paused`, `configured`, and `running`; and three behaviors `deploy`, `update`, and `destroy`. The sensors have five places: `off`, `provisioned`, `installed`, `configured`, and `running`; and also three behaviors: `start`, `pause`, `stop`. Each listener has two connections with its respective sensor. Through the first connection, the listener exposes the configuration to apply on the sensor (e.g., frequency of listening). Through the second connection, the listener offers a service to which data can be sent by the sensor when observing.

Maude specification. Let us present the definitions of sorts, subsorts, and operators used to encode the above syntactic aspects of the CONCERTO-D model, starting from elementary concepts to the construction of a net of CONCERTO-D components. In Maude, a type hierarchy can be defined using the keywords `sort` and `subsort`. An operator definition starts with the keyword `op` followed by the operator name and type signature; several operators with the same signature can be defined using `ops`.

A component type is defined by a set of places, an initial place, and a set of transitions. The definition also includes the behaviors (sets of transitions) of the component type, and its use and provide ports, each bound to a group of places. In the definition below, some details are omitted for simplicity.

We use predefined data structures and the module system of Maude to import the parameterized module `SET{Place}` and define the sort `Places` to be a supersort of `SET{Place}`. Generally, given a sort `T`, we let `Ts` stand for `SET{T}` and `QT` stand for `LIST{T}`.

```
sorts Place InitialPlace Transition Behavior UsePort ProvidePort GroupUse GroupProvide
      ComponentType .
sorts Places Transitions Behaviors GroupUses GroupProvides .
subsort InitialPlace < Place .

--- [...]
op b(_) : Transitions -> Behavior [ctor] .
op g(_?_) : UsePort Places -> GroupUse [ctor] .
op g(_!_) : ProvidePort Places -> GroupProvide [ctor] .

op { places: _,
     initial: _,
     --- [...]
     transitions: _,
     behaviors: _,
     groupUses: _,
     groupProvides: _
   } :
   Places InitialPlace
   --- [...]
   Transitions Behaviors GroupUses GroupProvides ->
   ComponentType
  [ctor] .
```

Example in Maude. We can now describe the component type `sensor` (an instance of which is displayed on the right of Figure 2). We first define a few constants corresponding to the element of the component type, then the type itself:

```
ops Running Configured Installed Provisioned Off : -> Place .
op Off : -> InitPlace .
--- [...]
ops RcvService ConfigService : -> UsePort .
ops Start11 Start12 Start13 Start2 Start3 Start4 Pause1 Stop1 : -> Transition .
ops Start Pause Stop : -> Behavior .

eq Deploy = b(Start11,Start12,Start13,Start2,Start3,Start4) .
eq Pause = b(Pause1) .
eq Stop = b(Stop1) .
op sensor : -> ComponentType .
```

```
eq sensor =
  { places: Running, Configured, Installed, Provisioned, Off,
    initial: Off,
    --- [...]
    transitions: (Start11,Start12,Start13,Start2,Start3,Start4,Pause1,Stop1),
    behaviors: (Start, Pause, Stop),
    groupUses: g(RcvService \lstinline[language=maude,basicstyle=\ttfamily]! (
        Configured, Running)),
               g(ConfigService \lstinline[language=maude,basicstyle=\ttfamily]! (
                   Configured, Installed, Running)),
    groupProvides: empty } .
```

2.2 Reconfiguration language and assembly of components

To allow the modification of assemblies, CONCERTO-D proposes a simple imperative language for writing reconfiguration programs, that offers four commonly used topological actions to create or modify the existing assemblies: $add(id_c, c)$, $del(id_c)$ (creation and deletion of control component instances), $con(id_{c1}, u, id_{c2}, p)$, $dcon(id_{c1}, u, id_{c2}, p)$ (connection and disconnection between two control component instances), where id_c is an instance identifier, c a component type, u a use port, p a provide port. Besides these actions, two additional actions manage the execution of behaviors: $pushB(id_c, b, id_b)$ to request the execution of a behavior b on the component instance id_c; and $wait(id_c, id_b)$ to synchronize onto the end of a given behavior. id_c is a component instance identifier, and id_b the identifier of a given *pushB*.

As a decentralized coordination model, CONCERTO-D considers a system of n nodes, and a partition of component instances over these nodes. Each node operates its own local CONCERTO-D controller and runs its own reconfiguration program. Concrete communications allow the synchronization of actions between paired components. This is an evolution of previous work on the CONCERTO model [6], in which a single central entity executed the reconfiguration and synchronization of components, and communications were implicit. Two examples of CONCERTO-D reconfiguration programs in our case study are given in Listings 1 and 2.

Listing 1: Listeners reconfiguration on $node_2$

```
for i in range(nb_listener):
  pushB(listener_i, update, 2+i*2)
  pushB(listener_i, deploy, 3+i*2)
```

Listing 2: One sensor reconfiguration on $node_{2+i}$

```
pushB(sensor_i, pause, 0)
wait(listener_i, 2+i*2)
pushB(sensor_i, start,1)
```

In CONCERTO-D reconfiguration, programs apply changes to the current assembly of components, i.e., component instances and their connection, and to the queue of requested behaviors for each component instance. A specific instance of a component (and its state at any given point of the execution) is specified by its component type, a queue of identified behaviors to be executed by the instance, and a marking that indicates the places reached and transitions fired.

```
sorts Instance Id PushedBehavior TransitionEnding Marking .

op (_;_) : Id Behavior  -> PushedBehavior [ctor] .
op m(_,_,_) : Places Transitions TransitionEndings  -> Marking [ctor] .
op  {   type: _,
```

```
        queue: _,
        marking: _
    } :
    ComponentType List{PushedBehavior} Marking -> Instance [ctor] .
```

To illustrate this, the definition below describes an instance `sensor1` of type `sensor`, in a state where only the place `running` is marked, and a single behavior `pause` is pending in the queue of behaviors. This corresponds to the state (0) in Figure 2.

```
eq instanceS1 = { type: sensor,
                  queue: (0 ; Pause),
                  marking: m(Running, empty, empty) } .
```

In order to describe an assembly, it is also necessary to specify the connections between the ports of its component instances, through their identifiers.

```
sort Connection .
sorts Connections .
op (_,_)--(_,_) : Id UsePort Id ProvidePort -> Connection [ctor] .
```

Here is an example of such connections between `node2` and `node3`:

```
eq connectionSL1 = (sensor1, RcvService)--(listener1, Rcv) .
eq connectionSL2 = (sensor1, ConfigService)--(listener1, Config) .
```

We now define actions that can be executed to perform a reconfiguration on a CONCERTO-D assembly.

```
sorts Action Program .
subsort List{Action} < Program .

op add(_,_) : Id ComponentType -> Action [ctor] .
op del(_) : Id -> Action [ctor] .
op pushB(_,_,_) : Id Behavior Id -> Action [ctor] .
op con(_) : Connection -> Action [ctor] .
op dcon(_) : Connection -> Action [ctor] .
op wait(_,_) : Id Id -> Action [ctor] .
```

The reconfiguration program given in Listing 2 (as instantiated specifically for `node3`) can thus be specified in Maude as follows:

```
eq programNode3 =
pushB(sensor1, start, 0) wait(listener1, 4) pushB(sensor1, start, 1) .
```

3 Elements of operational semantics

CONCERTO-D is equipped with a small-step semantics. We first illustrate the execution of a reconfiguration on our use case to give an intuition of this semantics.

We consider deployed and running components, i.e., the places `running` are marked in all listener and sensor components. From there, we aim to trigger an update of each sensor's listening frequency. Hence, each listener has to pause to change its configuration, forcing the sensors to pause as well.

Listings 1 and 2 give the reconfiguration programs respectively for all listeners (all hosted on the same node $node_2$) and one sensor i (corresponding to the update of the frequency of each sensor). These

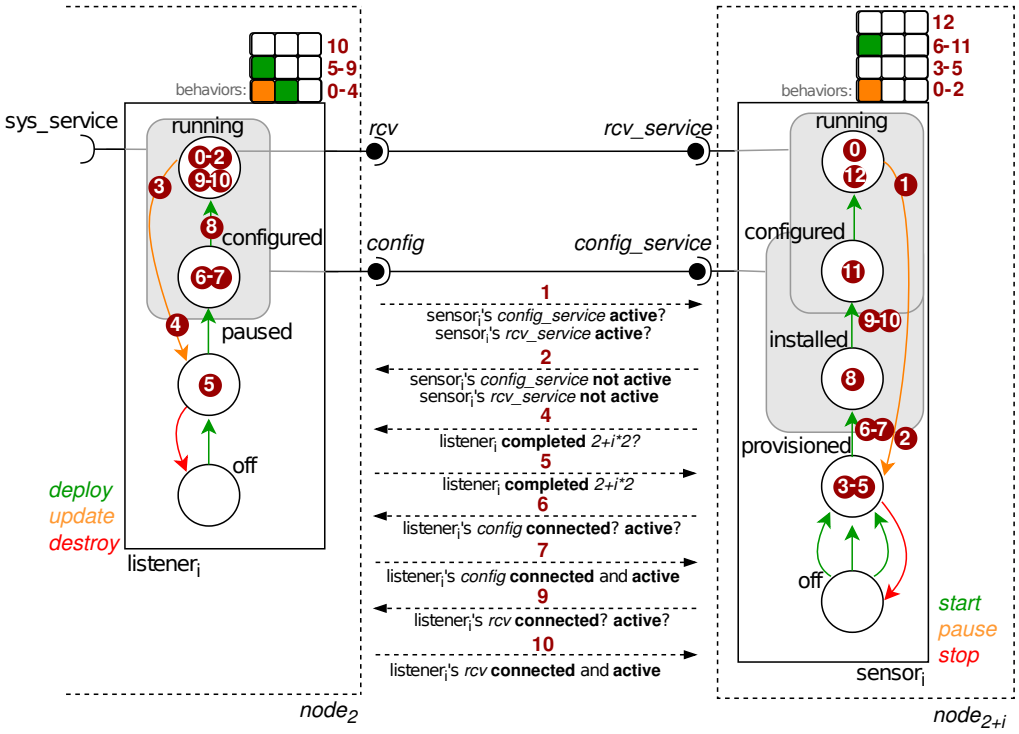


Figure 2: Control components of `listener`$_i$ on *node*$_2$ and `sensor`$_i$ from *node*$_{2+i}$. State example when applying the reconfiguration plans of Listings 1 and 2.

programs are executed concurrently on each node. The execution of these scripts is illustrated in Figure 2 to give the reader an intuition of the semantics of CONCERTO-D. Some possible steps of the execution are represented by a number representing the current configuration of the system (i.e., a snapshot). Using these numbers, three pieces of information are given: (i) the marking, represented by the red dots on marked places and transitions; (ii) the status of the behavior queue, and (iii) the coordination information required between nodes introduced by decentralized execution of CONCERTO-D. In the following, we describe the state of each configuration according to its number. We decompose each state, and we highlight communication steps using the notation Δ.

0. Both *listener*$_i$ and *sensor*$_i$ services are running. Thus, the `running` places are marked. The behaviors to trigger the update are pushed in the queue, i.e., `update` and `deploy` for *listener*$_i$, and `pause` for *sensor*$_i$. The behavior `start` for *sensor*$_i$ is not pushed due to wait action of a behavior of *listener*$_i$.

1. By triggering `pause`, *sensor*$_i$ leaves the place `running`.

Δ *listener*$_i$ needs to know if any component is using its ports before executing `update`. Then, *listener*$_i$ requests information on *sensor*$_i$'s use ports *rcv_service* and *config_service*, to know if they are active or not.

2. *sensor*$_i$ ends its previously fired transition.

Δ *sensor*$_i$ answers *listener*$_i$ about its use ports *rcv_service* and *config_service*, indicating that both are inactive.

3. *listener*$_i$ can begin the `update` behavior, therefore deactivating its provide ports. *sensor*$_i$ removes the `pause` behavior from its queue, as no more transitions in this behavior can be fired.

Δ The information sent in 2 is received by *listener*$_i$, allowing it to start its `update`.

4. *listener*$_i$ finishes its transition related to its `update` behavior.

Δ *sensor*$_i$ is now waiting the behavior identified by 2+i*2, and runs on *listener*$_i$, to be ended. Then it sends a request to *listener*$_i$ to know if the behavior 2+i*2 (`update`) is completed.

5. The behavior `update` of *listener*$_i$ is retrieved from its queue.

Δ *sensor*$_i$ is informed that this behavior is completed, which allows it to continue its execution, i.e., to push and execute the transitions of the `start` behavior.

6. After pushing the behavior `start` and completing its first transition.

Δ *sensor*$_i$ sends a request to determine whether the port *config* of *listener*$_i$ is connected and active, before entering in *config_service* group.

7. Δ *sensor*$_i$ receives information on *listener*$_i$'s port *config*. It is now active and connected.

8. *sensor*$_i$ enters the place `installed`. *listener*$_i$ begins the activation of its service.

9. *listener*$_i$ activates its service and provides the port *rcv*.

Δ *sensor*$_i$ sends a request asking whether the port *rcv* is active and connected before entering the place `configured`.

10. *listener*$_i$ removes the `deploy` behavior from its queue.

Δ *sensor*$_i$ get the information that *listener*$_i$ provides the port *rcv*.

11. *sensor*$_i$ enters the place `configured`.

12. *sensor*$_i$ reactivates its service and removes the `start` behavior from its queue.

The rest of this section gives some formal elements of the small-step semantics of CONCERTO-D. Because of space reasons, we cannot fully detail all rules of the semantics. Instead, the section first illustrates one execution rule at a reconfiguration program level, one execution rule at a component level, and then details communications, i.e., the main formalization challenge in a decentralized model as CONCERTO-D.

3.1 Execution and synchronization of reconfiguration actions

Let us first describe the semantics rules that govern the execution of actions in reconfiguration programs. Recall that each node executes its own reconfiguration program, affecting components hosted on that node. Actions to add and delete component instances are straightforward. Actions for connections and disconnections of components are also fairly standard. They only modify the view that one node has of the component's topology. When components to (dis)connect are located on different nodes, both nodes should eventually execute their own similar (dis)connection actions, however, the node-local views of the topology do not have to be kept continuously synchronized. Finally, we turn to the `pushB` action, which is specific to the CONCERTO-D model. The rule in Listing 3 gives the semantic of that action, describing the modification of a node (as defined in Section 3.2). The action adds a behavior request to the queue of behaviors of one component. The target component is indicated by its identifier, and the behavior type is indicated, as well as an identifier. The latter is used for synchronization points, in order to distinguish between multiple executions of the same behavior type.

The behavior requests determine the active behavior of a component instance at a given point, which plays a role in the evolution of this component. We now describe the rules of this evolution. As described

in Section 2, the life cycles of components are modeled by places (milestones in the reconfiguration) and transitions between places (reconfiguration actions). A marking on places and transitions indicates the current state of the reconfiguration. When a place is marked, its outgoing transitions can be fired. The place is then unmarked, and the fired transitions are marked. Conversely, when all the incoming transitions toward a place are finished, those transitions are unmarked and the place is entered. This model is well suited to represent concurrent execution, in particular, multiple transitions can be marked simultaneously, corresponding to parallel execution of reconfiguration actions.

Listing 3: The rule to execute the `pushB` action. The rule applies to a node (as defined in Section 3.2) but rewrites a single component instance in that node.

```
crl [PushingBehavior] :
 < --- [...]
   instances: (Id1 ↦ { type: Ct, queue: Qb, marking: M }) , Isx,
   --- [...]
   program: pushB(Id1, B, IdB) RPx,
   --- [...]  >
  =>
 < --- [...]
   instances: ( Id1 ↦ { type: Ct, queue: append(Qb,(IdB ; B)), marking: M } ), Isx,
   --- [...]
   program: RPx,
   --- [...]
 >
  if (B in (Ct).behaviors) .
```

This evolution of markings is desribed by four rules: *FiringTransitions*, *EndingTransition*, *EnteringPlace*, and *FinishingBehavior*. For space reasons, we only detail one of these rules: *Firing Transitions* given in Listing 4. It modifies the marking in one component on one node, namely unmarking a place `P` and marking the outgoing transitions of `Ts` (only those transitions that belong to the active behavior of the component). Note that this is a conditional rule, as indicated by the keyword `crl` and the boolean conditions after the rule. The first condition merely ensures that there are transitions to fire. The second condition relies on the predicate `isSafeToFire` to check dependencies towards other components, modeled by ports, are not violated by the firing of the transitions. In particular, this predicate is true only if the firing of the transitions does not lead to deactivating a provide port that is being used, i.e., connected to an active use port. The rule to end a transition (not given) is somewhat similar but instead requires that use ports attached to the reached place are connected to an active provide port. Thus, ports impose inter-component synchronization conditions on the execution.

Listing 4: The rule to fire a transition. As before, this rule modifies a component instance in a node.

```
crl [FiringTransitions] :
    < nodeInventory: Idsx,
      instances: (Id1 ↦  {   type: Ct,
                             queue: (IdB ; b(TsB)) Qb,
                             marking: m((P, Ps),Ts,Tes)
                         }) , Isx,
      connections: Csx,
      --- [...] >
     =>
    < nodeInventory: Idsx,
```

```
      instances: (Id1 ↦ {   type: Ct,
                            queue: (IdB ; b(TsB)) Qb,
                            marking: m(Ps,union(Ts, getTransitionsofPlace(TsB,P) ),
                                Tes)
                        }), Isx ,
      connections: Csx,
      --- [...] >
    if (   getTransitionsofPlace(TsB,P) =/= empty and
           isSafeToFire( Idsx,
                         m(Ps,union(Ts, getTransitionsofPlace(TsB,P) ),Tes) ,
                         (Id1 ↦    {   type: Ct,
                                       queue: (IdB ; b(TsB)) Qb,
                                       marking: m(P, Ps,Ts,Tes)
                                   } ,Isx),
                         eStatex,
                         getPConnectionsofCInst(Id1, Csx))) .
       ) .
```

3.2 Communications

Firing or ending a transition on one component may require checking the activity status of a port on another component. In previous work [6], the coordination was assumed to be carried out by a central entity that kept track of the status of every port. Instead, CONCERTO-D is meant to represent a decentralized process, it is therefore necessary to explicitly model communications between the nodes that host components.

The following boolean information may be needed for synchronization, and can be called on remote nodes: (1) the completion of a *dcon* action; (2) the completion of a behavior (i.e., *wait* action); (3) the activity of a use or provide port.

When evaluating the status of one of the above queries (such as the status of a port in the predicate `safeToFire`), we distinguish the case where the element of interest (*dcon* action, behavior, port etc.) belongs to the local node, from the case where messages must be exchanged between nodes:

```
eq question(R1,Ids,RcvAns, Is,Cs,RP ) = if (isLocal(R1,Ids)) then
                                            localQuestion(R1,Is,Cs,RP)
                                        else externQuestion(R1,RcvAns) fi .
```

Essentially, each CONCERTO-D node maintains a localized perspective of its components and communicates with neighboring nodes (i.e., other CONCERTO-D controllers hosting connected control components). In CONCERTO-D, an asynchronous message-based communication model, increasingly favored in distributed systems, manages the communication [21]. To achieve asynchronous communications each node is equipped with an incoming queue of messages. Messages must transit through this queue before being effectively received (this assumes that the order of messages is preserved).

A message can either be a `Question` or an `Answer`. The former is aimed at a specified component instance and contains one of several possible queries while an `Answer` gives the boolean value corresponding to a question. It can take the values true, false.

Listing 5: Constructors for Question, Answer and Message

```
sorts Query .
op isActive(_) : Port  -> Query [ctor] .
```

```
op isRefusing(_) : Port -> Query [ctor] .
op isConnected(_) : Connection -> Query [ctor] .
op isCompleted(_) : Id -> Query [ctor] .
op onDisconnect(_) : Connection -> Query [ctor] .
op [ dst: _ , query: _ ] : Id Query -> Question [ctor] .
op [ question: _, value: _ ] : Question Bool -> Answer [ctor] .
op mkMsg : Question -> Message [ctor] .
op mkMsg : Answer -> Message [ctor] .
```

For example, the question `[ dst: listener1, query: isActive(Rcv)]` can be sent by node3 (which hosts `sensor1`) to node2 (which hosts `listener1`) to check whether the port `Rcv` of `listener1` is active. This corresponds to the state 9 in Figure 2. The answer `[ req: [ dst: listener1, query : isActive(Rcv)], value: true ]` is the answer which will be returned at state (10) to indicate that the port is indeed active. For more details about the remaining queries, the reader can refer to [6].

A node is specified by the set of identifiers of all the component instances involved in the reconfiguration programs, a mapping (called an inventory) that associates these identifiers to their associated nodes, the connections between local component instances and external instances, and the local reconfiguration program to be executed on the node. A node is additionally specified by five parameters used for communication: (1) a local vision of the state of external components `externState`; (2) a queue of pending questions waiting for an answer `pendingQuestions`; (3) a queue of outgoing questions asked to other nodes `outgoingQuestions`; (4) a queue of incoming messages `incomingMsgs` including incoming questions asked by other nodes and answers received from other nodes to previously sent questions; and finally (5) a history of previously sent questions (to avoid redundant messages for already pending requests). Received answers are stored in `externState` as a mapping that associates requests to the value of the answer. This represents the local node's vision of the status of other components. This information is updated each time external information is needed in a semantics rule.

Listing 6: Local configuration

```
op < nodeInventory: _,
     instances: _,
     connections: _,
     program: _,
     externState: _,
     pendingQuestions: _,
     outgoingQuestions: _,
     history: _,
     incomingMsgs: _
   > :
   Ids Instances Connections Program Map{Question, Bool} List{Question} List{
       Question} Set{Question} List{Message}
    -> LocalConfiguration
```

For example, the description of the local configuration of `node3` in state 0 is as follows:

```
eq ConfNode3 =
   < nodeInventory: sensor1,
     instances: { sensor1 ↦ instanceS1 },
     connections: { connectionSL1, connectionSL2 },
     program: programNode3,
     externState: eStateNode3,
```

```
      pendingQuestions: nil,
      outgoingQuestions: nil,
      history: empty,
      incomingMsgs: nil >
```

where instanceS1, connectionSL1, connectionSL2 and programNode3 are the elements previously described for our use case in Sections 2 and 3.2. sensor1 is the identifier of instanceS1 and eStateNode3 is the external state (i.e., mapping of questions and received answers sent by node3). Following the deployment steps that preceded the reconfiguration state 0, eStateNode3 of node3 is described as follows:

```
eq eStateNode3 =
    [ dst: listener1, query: isRefusing(Rcv) ] ↦ false ,
    [ dst: listener1, query: isConnected(connectionSL1) ] ↦ true ,
    [ dst: listener1, query: isRefusing(Config) ] ↦ false ,
    [ dst: listener1, query: isConnected(connectionSL2) ] ↦ true .
```

The rest of this section gives the execution semantics of communication in CONCERTO-D using Maude rewrite rules, which operate on a system of local configurations of nodes. The associated rules for communication are presented in listings 7, 8, 9 and 10. The rules consider two nodes x and y.

First, Listing 7 describes the rule [SendQuestion] sending of a request [dst: Dst, query: Q] from node x to node y. The destination node y is determined since the identifier of the instance of the request Dst appears in the inventory of node y. Sending the request implies adding to the incoming messages of y the message mkMsg([dst: Dst, query: Q]). The sent request will also be inserted in the history Hx of node x to avoid the request being sent multiple times while waiting for an answer.

Listing 7: The rule to send a question, that rewrites two nodes (unaffected variables are omitted).

```
rl [SendQuestion] :
  < --- [...]
    outgoingQuestions: [ dst: Dst, query: QY ] OutQx,
    history: Hx,
    --- [...]
  > ,
  < nodeInventory: (Dst, Idsy),
    ---[...]
    incomingMsgs: iMsgsy >
   =>
  < --- [...]
    outgoingQuestions: OutQx,
    history: ([ dst: Dst, query: Q ], Hx),
    --- [...]
  > ,
  < nodeInventory: (Dst, Ids),
    --- [...]
    incomingMsgs: append(iMsgsy, mkMsg([ dst: Dst, query: QY ])) >
```

Second, Listing 8 describes the sending of an answer from node x to node y concerning a question Q previously sent by y. The destination node y is chosen when the concerned request Q is in its history of sent requests, and has not yet received in an answer to this request (!not occurs(mkMsg(Q), iMsgsy)!). The value of the answer is computed on node x by the function localQuestion and sent to node y by placing the answer in the incoming messages of node y.

It is important to maintain the consistency of shared information, so if a node x sends information that could allow an external component y to change the status of one of its ports p, the previously recorded information about p on x is deleted. For example, when a component on node y asks x if one of its provide port p is active, it means that the associated use port u on y may be activated soon after the answer is sent by x. Consequently, the node x that sends the answer on p will simultaneously reset the information in its external state regarding u on y (of course, it should not yet assume that the use port is active). In Listing 8, this update is carried out by the function `resetState`.

Listing 8: The (conditional) rule to send an answer

```
crl [SendAnswer] :
  < --- [...]
    instances: Isx,
    connections: Csx,
    program: RPx,
    externState: eStatex,
    pendingQuestions: Q pendingQx,
    --- [...]
  > ,
  < --- [...]
    history: (Q, Hy),
    incomingMsgs: iMsgsy >
   =>
  < --- [...]
    instances: Isx,
    connections: Csx,
    program: RPx,
    externState:resetState(eStatex,getIdsForReset(Csx,Q,localQuestion(Q,Isx,Csx,RPx))),
    pendingQuestions: pendingQx,
    --- [...]
  > ,
  < --- [...]
    history: Hy,
    incomingMsgs: append(iMsgsy, mkMsg([ question: Q,
                                         value: localQuestion(Q,Isx,Csx,RPx) ]))
  >
 if (not occurs(mkMsg(Q), iMsgsy)) .
```

Third, Listing 9 describes the rule for receiving a question on a component local to `x`. This is started when the identifier `Dst` of the request `[dst: Dst, query: Q]` is in the set of identifiers of the components of node `x`. The request is evaluated and the answer is placed in the queue of pending questions.

Listing 9: The rule for receiving a question

```
rl [ReceiveQuestion] :
  < nodeInventory: (Dst, Idsx),
    --- [...]
    pendingQuestions: pendingQx,
    --- [...]
    incomingMsgs: mkMsg([dst: Dst, query: QY]) iMsgsx >
   =>
```

```
  < nodeInventory: (Dst, Ids),
    --- [...]
    pendingQuestions: append(pendingQx,[ dst: Dst, query: QY ]),
    incomingMsgs: iMsgsx >
```

Finally, Listing 10 describes the rule for receiving an answer to a question `Q` previously sent by node `x` (as indicated by its presence in the history). The answered value `val` is recorded (!insert(R, val, RcvA)!) and it replaces previous information about the result of request `Q`, if any. The request `Q` is removed from the history of node `x`, so that a similar request may be sent again in the future.

Listing 10: The rule for receiving an answer

```
rl [ReceiveAnswer] :
 < --- [...]
   externState: eStatex,
   --- [...]
   history: (Q,Hx),
   incomingMsgs: mkMsg([ question: Q, value: Val ]) iMsgsx >
     =>
 < --- [...]
   externState: insert(Q,Val,eStatex),
   --- [...]
   history: Hx,
   incomingMsgs: iMsgsx >
```

Example. To illustrate the communication protocol of the above semantics rules, a subpart of the example of Figure 2 is used with one listener and one sensor. In step 4, the `sensor` sends a question to the `listener` regarding the completion of its behavior `idb4`. The question is created following the `wait(listener, idb4)` action in the local reconfiguration program of the sensor, and is placed in `outgoingQuestion`. The question is `[dst: listener1, query: isCompleted(idb4)]`. The following communication rules are applied:

1. `[SendQuestion]` is applied: the question in `outgoingQuestion` on `sensor` is popped and placed into `history` of `sensor`. The `incomingMsgs` of `listener` appends the question received from `sensor`.
2. `[ReceiveQuestion]` is applied: The node `listener` appends the incoming question to its list `pendingQuestions`. The question is removed from `incomingMsgs`.
3. `[SendAnswer]` is applied: The question in `pendingQuestions` on `listener` is popped. The answer is created and pushed to the `incomingMsgs` of `sensor`. The answer message to the question is `[question: [dst: listener1, query: isCompleted(idb4)], value: true]`. The original question is removes from `history` on `sensor`.
4. `[ReceiveAnswer]` is applied: The answer received on `sensor` in `incomingMsgs` is removed and inserted (added or updated) to the `externState` of `sensor` as a mapping `[dst: listener, query: isCompleted(idb4)]` $\mapsto$ `true`.

Once completed a pushed behavior cannot be executed again as being uniquely identified. For this reason, no consistency issue can happen in this small example. The `resetState` of the rule `SendAnswer` has no consequences on `externState` in this case.

4 Related work

CONCERTO and CONCERTO-D can be considered as component models, such as defined in Component-Based Software Engineering (CBSE). As explained in [6, 9] CONCERTO, and by extension CONCERTO-D, differ from usual component models by modeling the life cycle of components instead of modeling the functional code of components. For this reason, both CONCERTO and CONCERTO-D are more comparable to DevOps approaches. Only one other component model can be compared to CONCERTO: Aeolus [10]. However, Aeolus is more limited than CONCERTO in terms of parallelism and concurrency. As for CONCERTO, and unlike CONCERTO-D, Aeolus is a centralized model, and it has been formalized manually.

Regarding DevOps approaches for deployments, a few contributions have studied a decentralized approach. In [16, 26] each component of the application expresses its dependencies with the other components in a central plan, distributed to the corresponding nodes, deploying their part of the application. Deployment executions are then coordinated between the nodes according to their dependencies. Here, the execution is decentralized as in CONCERTO-D. In [25], an extension of the DevOps tool Pulumi is presented to handle both the computation and execution of deployment and update programs in a decentralized manner. However, the three approaches above, and almost all DevOps tools, lack formal specifications: their language is defined by a unique implementation and informal documentation.

To our knowledge, there are only two DevOps contributions that offer formal semantics of their system configuration languages: SMARTFROG [2] a tool no longer maintained, and μPUPPET [14] a subset of PUPPET. The semantics of SMARTFROG is a denotational semantics of a *compiler* for a core SMARTFROG fragment. From the high-level language SF, which handles features such as inheritance, composition, references, etc., the compilation process produces a store, basically a tree of attribute-value pairs. These trees are also the abstractions for system states in SMARTFROG's semantics. The authors wrote three implementations (in Scala, Haskell, and OCaml) of the compiler guided by the semantics but not proved correct w.r.t. to the formal semantics (which is not mechanized). These implementations were randomly tested and a few implementation errors were found. They were also tested against the production compiler: it allowed them to find both a misunderstanding of the semantics of SMARTFROG and bugs in the production compiler. The semantics of μPUPPET is a small-step operational semantics. An implementation of a μPUPPET compiler exists, guided by the semantics but not proved correct w.r.t. the formal semantics, which is also not mechanized. The output of the compiler is a catalog, i.e., a structure close to the stores' output by SMARTFROG's compiler. A catalog is also an abstraction for a system state. To help debug μPUPPET manifests, Fu [13] also proposed an analysis of provenance [7] based on the μPUPPET operational semantics. In both cases, the formal semantics is not an executable artifact as is our proposal.

Khebbeb et al. also use Maude to formalize adaptation in the Cloud [18] and at the edge [17]. The motivation and approach are however very different from our work. There is no reconfiguration language: the goal is to automatically adapt resources depending on the load of the Cloud system. On the one hand, this work considers the provisioning and de-provisioning of virtual machines, an aspect we do not consider. On the other hand, their concept of service is very simplistic: a service is something that processes requests (and only the number of requests is formalized) and there are no connections between services. Their rewrite rules implement pre-defined elasticity strategies. While they perform model-checking on a small example (3 services, 1 VM, and 2 Fog nodes) [17], they neither provide any information about the number of states, nor the time required for the verification. In its complexity, our proposal is closer to work that models APIs for e.g., [27]. As Yu et al., to analyze interesting enough case studies, we will need to optimize the model-checking by implementing partial order reduction [12].

5 Conclusion and Future Work

In this paper, we proposed a formalization in Maude of the decentralized reconfiguration language CONCERTO-D, an extension of the centralized reconfiguration language CONCERTO [6]. In CONCERTO-D, the assembly of components is distributed among a set of nodes. Each node is responsible for a local reconfiguration program that may require some coordination with the nodes that host connected component instances. CONCERTO-D automatically manages the necessary communications between nodes via asynchronous communications, which involve nodes communicating by exchanging messages through buffers. The advantage of having a formal semantics with Maude is that it is executable and hence we can check by running examples that the semantics we designed behaves as we expect it to do.

This is however a first step. We ambition to verify properties of specific assemblies and their reconfiguration programs as well as to verify properties of the semantics itself.

To verify specific assemblies and their programs, Maude offers on-the-fly model-checking of LTL formulas. Preliminary experiments show that in its current form the semantics does not allows to scale up to the model-checking of realistic applications. We will first explore the application of partial order reduction techniques [12] to limit the state-space explosion problem. Another way to avoid such an explosion is to use symbolic model-checking techniques in particular verification by over-approximation of the set of descendants [15, 3]. In these approaches, the goal is to check that a set of bad states does not intersect the over-approximations of the sets of descendants. There are however constraints on how the (approximations of) sets of terms are described (for e.g. regular tree automata) and constraints on the rewriting rules describing the dynamic of the system.

Verifying properties of the CONCERTO-D model itself may require the use of interactive theorem proving (ITP). There are ongoing efforts to develop a dedicated interactive theorem prover for Maude [11] as well as to translate Maude specifications [24] into existing ITP specification languages. The main properties we plan to prove interactively are that the semantics preserves well-formed instances and well-formed systems, as well as that the consistency of the communicated information between nodes is ensured.

Finally, thanks to CONCERTO-D, CONCERTO could be transformed into a choreographic language (as defined in [1]). In a choreographic approach CONCERTO would be the choreography language. A compilation process would then automatically generate the CONCERTO-D programs (i.e., local projections) on the nodes, and guarantee that required communications between nodes will be performed when required. The formalization may be used to prove that the set of CONCERTO-D programs yielded by that process has a behavior that corresponds to the behavior original centralized CONCERTO program.

References

[1] Davide Ancona et al. (2016): *Behavioral Types in Programming Languages*. *Foundations and Trends in Programming Languages* 3, doi:10.1561/2500000031.

[2] Paul Anderson & Herry Herry (2016): *A Formal Semantics for the SmartFrog Configuration Language*. *J. Netw. Syst. Manag.*, doi:10.1007/s10922-015-9351-y.

[3] Yohan Boichut, Jacques Chabin & Pierre Réty (2019): *Towards more precise rewriting approximations*. *J. Comput. Syst. Sci.* 104, pp. 131–148, doi:10.1016/J.JCSS.2017.01.006.

[4] Maverick Chardet, Hélène Coullon & Christian Pérez (2020): *Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto*. In: *20th IEEE/ACM International Symposium*

on Cluster, Cloud and Internet Computing (CCGrid), IEEE, doi:10.1109/CCGrid49817.2020.00-59.

[5] Maverick Chardet, Hélène Coullon, Christian Pérez, Dimitri Pertin, Charlène Servantie & Simon Robillard (2020): *Enhancing Separation of Concerns, Parallelism, and Formalism in Distributed Software Deployment with Madeus*. hal: hal-02737859.

[6] Maverick Chardet, Hélène Coullon & Simon Robillard (2021): *Toward safe and efficient reconfiguration with Concerto*. *Sci. Comput. Program.*, doi:10.1016/j.scico.2020.102582. hal: hal-03103714.

[7] James Cheney, Laura Chiticariu & Wang Chiew Tan (2009): *Provenance in Databases: Why, How, and Where*. *Found. Trends Databases*, doi:10.1561/1900000006.

[8] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, Rubén Rubio & Carolyn Talcott (2024): *Maude manual (version 3.4)*. Available at `https://maude.lcc.uma.es/maude-manual/`.

[9] Hélène Coullon, Ludovic Henrio, Frédéric Loulergue & Simon Robillard (2023): *Component-Based Distributed Software Reconfiguration: A Verification-Oriented Survey*. *ACM Comput. Surv.*, doi:10.1145/3595376.

[10] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli & Gianluigi Zavattaro (2014): *Aeolus: a Component Model for the Cloud*. *Information and Computation*, doi:10.1016/j.ic.2014.11.002.

[11] F Durán, S Escobar, J Meseguer & J Sapina (2024): *An Inductive Theorem Prover for Maude Equational Theories*. Available at `https://nuitp.webs.upv.es/download/NuITP.pdf`.

[12] Azadeh Farzan & José Meseguer (2006): *Partial Order Reduction for Rewriting Semantics of Programming Languages*. In: *Workshop on Rewriting Logic and its Applications (WRLA)*, *ENTCS* 176, Elsevier, doi:10.1016/J.ENTCS.2007.06.008.

[13] Weili Fu (2019): *Semantics and provenance of configuration programming language μPuppet*. Ph.D. thesis, University of Edinburgh, UK. Available at `https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.798916`.

[14] Weili Fu, Roly Perera, Paul Anderson & James Cheney (2017): *muPuppet: A Declarative Subset of the Puppet Configuration Language*. In: *31st European Conference on Object-Oriented Programming (ECOOP)*, LIPIcs, doi:10.4230/LIPIcs.ECOOP.2017.12.

[15] Thomas Genet (1998): *Decidable Approximations of Sets of Descendants and Sets of Normal Forms*. In: *Rewriting Techniques and Applications (RTA)*, *LNCS* 1379, Springer, pp. 151–165, doi:10.1007/BFB0052368.

[16] Herry Herry, Paul Anderson & Michael Rovatsos (2013): *Choreographing configuration changes*. In: *9th International Conference on Network and Service Management (CNSM 2013)*, doi:10.1109/CNSM.2013.6727828.

[17] Khaled Khebbeb, Nabil Hameurlain & Faiza Belala (2020): *A Maude-Based rewriting approach to model and verify Cloud/Fog self-adaptation and orchestration*. *J. Syst. Archit.*, doi:10.1016/J.SYSARC.2020.101821.

[18] Khaled Khebbeb, Nabil Hameurlain, Faiza Belala & Hamza Sahli (2019): *Formal modelling and verifying elasticity strategies in cloud systems*. *IET Softw.* 13(1), doi:10.1049/IET-SEN.2018.5030.

[19] Vincent Lostanlen, Antoine Bernabeu, Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou & Mathieu Lagrange (2021): *Energy Efficiency is Not Enough:Towards a Batteryless Internet of Sounds*. In: *16th International Audio Mostly Conference*, doi:10.1145/3478384.3478408.

[20] Peter Csaba Ölveczky (2017): *Designing Reliable Distributed Systems*. Springer, doi:10.1007/978-1-4471-6687-0.

[21] Antoine Omond, Hélène Coullon, Issam Raïs & Otto Anshus (2023): *Leveraging Relay Nodes to Deploy and Update Services in a CPS with Sleeping Nodes*. In: *16th IEEE International Conference on Cyber, Physical and Social Computing (CPSCom)*, IEEE, doi:10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics60724.2023.00102. hal: hal-04372320.

[22] Antoine Omond, Issam Raïs & Hélène Coullon (2023): *Evaluating the energy consumption of adaptation tasks for a CPS in the Arctic Tundra*. In: *19th IEEE International Conference on Green Computing and Communications (GreenCom)*, IEEE, doi:10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics60724.2023.00122. hal: hal-04372340.

[23] Jolan Philippe, Antoine Omond, Hélène Coullon, Charles Prud'Homme & Issam Raïs (2024): *Fast Choreography of Cross-DevOps Reconfiguration with Ballet: A Multi-Site OpenStack Case Study*. In: *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE. hal: hal-04457484.

[24] Rubén Rubio & Adrián Riesco (2022): *Theorem proving for Maude specifications using Lean*. In: *International Conference on Formal Engineering Methods*, Springer, pp. 263–280, doi:10.1007/978-3-031-17244-1_16.

[25] Daniel Sokolowski, Pascal Weisenburger & Guido Salvaneschi (2021): *Automating Serverless Deployments for DevOps Organizations*. In: *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, doi:10.1145/3468264.3468575.

[26] Karoline Wild, Uwe Breitenbücher, Kálmán Képes, Frank Leymann & Benjamin Weder (2020): *Decentralized Cross-organizational Application Deployment Automation: An Approach for Generating Deployment Choreographies Based on Declarative Deployment Models*. In: *Advanced Information Systems Engineering (CAiSE)*, Springer, doi:10.1007/978-3-030-49435-3_2.

[27] Geunyeol Yu, Seunghyun Chae, Kyungmin Bae & Sungkun Moon (2024): *Formal Specification of Trusted Execution Environment APIs*. In: *Fundamental Approaches to Software Engineering*, Springer Nature Switzerland, doi:10.1007/978-3-031-57259-3_5.

Safe Composition of Systems of Communicating Finite State Machines

Franco Barbanera*
Dipartimento di Matematica e Informatica
University of Catania
franco.barbanera@unict.it

Rolf Hennicker
Institute for Informatics
LMU Munich
hennicke@pst.ifi.lmu.de

The *Participants-as-Interfaces* (PaI) approach to system composition suggests that participants of a system may be viewed as interfaces. Given a set of systems, one participant per system is chosen to play the role of an interface. When systems are composed, the interface participants are replaced by *gateways* which communicate to each other by forwarding messages. The PaI-approach for systems of asynchronous communicating finite state machines (CFSMs) has been exploited in the literature for binary composition only, with a (necessarily) unique forwarding policy. In this paper we consider the case of multiple system composition when forwarding gateways are not uniquely determined and their interactions depend on specific *connection policies* complying with a *connection model*. We represent connection policies as CFSM systems and prove that a bunch of relevant communication properties (deadlock-freeness, reception-error-freeness, etc.) are preserved by *PaI multicomposition*, with the proviso that also the used connection policy does enjoy the communication property taken into account.

1 Introduction

Concurrent/Distributed systems are hardly – especially nowadays – stand-alone entities. They are part of "jigsaws" never completely finished. Either in their design phase or after their deployment, they should be considered as *open* and ready for interaction with their environment, and hence with other systems. The possibility of extending and improving their functional and communication capabilities by composing them with other systems is also a crucial means against their obsolescence. Compositional mechanisms and techniques are consequently an important subject for investigation. As mentioned in [3], system composition investigations should focus on three relevant features of these mechanisms/techniques:

- *Conservativity:* They should alter as little as possible the single systems we compose.
- *Flexibility:* They should not be embedded into the systems we compose, i.e. they should be "system independent". In particular, they should allow to consider **any** system as potentially **open**.
- *Safety:* Relevant properties of the single systems should not be "broken" by composition.

A fairly general and abstract approach to binary composition of systems was proposed in [1] and dubbed afterwards *Participants-as-Interfaces* (PaI). Roughly, the composition is achieved by transforming two selected participants – one per system, say h and k, – into coupled forwarders (gateways), provided the participants exhibit "compatible" behaviours. The graphics in Fig. 1 illustrates the PaI idea for the binary case. If interface participant h of the first system S_1 can receive a message a from some participant of S_1 and interface participant k of the second system S_2 can send a to some participant of

*Partially supported by Project "National Center for HPC, Big Data e Quantum Computing", Programma M4C2, Investimento 1.3.

C. Aubert, C. Di Giusto, S. Fowler & V. K. I Pun (Eds.):
17th Interaction and Concurrency Experience (ICE 2024)
EPTCS 414, 2024, pp. 39–57, doi:10.4204/EPTCS.414.3

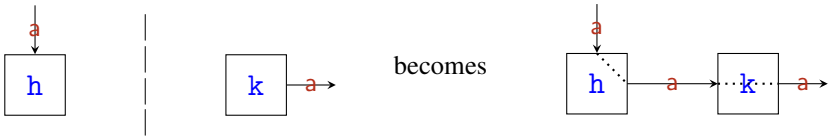


Figure 1: The PaI idea for binary composition

S_2, then the gateway replacing the first interface (also called h) will forward the received message to the gateway for k. How PaI works for multicomposition of systems will be illustrated in Section 2. It is worth remarking that the PaI approach to system composition does not expect any particular condition to be satisfied by a single participant in order to be used as an interface.

Conservativity as well as *flexibility* are definitely features of the PaI composition idea. Conservativity holds since all participants not acting as interfaces remain untouched and flexibility holds since, in principle, any participant can play the role of an interface. This fact is independent of the concrete formalism used for protocol descriptions and system designs/implementations. *Safety*, instead, can be checked only once we take into account a specific formalism. Such checks were carried out in a number of papers where two relevant formalisms for the description and verification of concurrent communicating systems were considered: MultiParty Session Types (MPST) [21, 22] and Communicating Finite State Machines (CFSM) [12]. Safety of the binary PaI approach was investigated for MPST in [4], where a synchronous communication model was considered. The PaI approach to multicomposition for MPST has been exploited in [3, 2], again for synchronous communications. In particular, in [2], a restricted notion of multiple connections in a client-server setting has been considered. For the synchronous MPST formalism used in those papers, PaI proved to be safe. The binary PaI approach for safety in systems of (standard) asynchronous CFSMs was taken into account in [1], whereas safety of PaI for a synchronous version of the CFSM formalism was investigated in [6, 7, 8], again for binary composition.

Contributions. In the present paper we investigate safety of PaI multicomposition for the asynchronous formalism of CFSMs. For this purpose we reuse the PaI multicomposition idea of [3] but realise it – instead of the synchronous MPST framework – in the asynchronous CFSM setting which needs completely different design and proof techniques. At the same time we go beyond the binary composition of asynchronous CFSMs of [1] and study multicomposition of CFSM systems. Clearly this goes also beyond the aforementioned papers [6, 7, 8] dealing with binary composition of synchronous CFSMs. In particular, in the asynchronous case different communication properties, like unspecified-reception freeness, are relevant.

A crucial role in our approach to multicomposition is played by *connection policies* which can be individually chosen by the system designer on the basis of a given concrete *connection model*. A connection model describes architectural aspects of compositions. It specifies which forwarding links between interface roles of different systems are meaningful from a static perspective. The concrete behavioural instantiation of such links, in terms of which message of an interface role, say h, is forwarded in which state of h to which interface role of another system, is determined by a connection policy which therefore also determines the construction of gateway CFSMs. The *multicomposition* of n systems of CFSMs is then simply defined by taking all CFSMs of the single systems but replacing each CFSM of an interface participant by its gateway CFSM. The use of connection models is methodologically important since it is more likely that a connection policy complying with a connection model will satisfy desired communication properties. For the proofs of our safety results, however, only the specifics of the chosen connection policy is relevant.

We show that a number of relevant communication properties (deadlock-freeness, orphan-message

freeness, unspecified-reception freeness, and progress) are preserved by PaI multicomposition of CFSM systems whenever the particular property is satisfied also by the connection policy used, which is formalised as a CFSM system itself. Apart from orphan-message-freeness preservation we need, however, an additional assumption which requires that interface participants do not have a state with at least one outgoing output action and one outgoing input action, a condition referred to in the literature as *no-mixed-state* [15]. We shall provide counterexamples illustrating the role played by the no-mixed-state condition in guaranteeing safety of composition. In contrast with deadlock-freeness, the stronger property of lock-freeness will be shown (by means of a counterexample) not to be preserved in general, even in absence of of mixed-states.

Outline. The main ideas underlying PaI multicomposition are intuitively described in Section 2. In Section 3 we recall the definitions of communicating finite state machine, communicating system and their related notions. There we also provide the definitions of a number of relevant communication properties. In Section 4, PaI multicomposition is formally defined on the basis of the definitions of connection policy and gateway. Our main results are presented in Section 5 including counterexamples spotting the role of the no-mixed-state condition and a counterexample for lock-freeness preservation. Section 6 concludes with a brief summary, by pointing out a few more approaches to system composition, and with hints for future work.

2 The PaI Approach to Multicomposition

In order to illustrate the idea underlying *PaI multicomposition*[1], we consider an example of [3] with four systems S_1, S_2, S_3 and S_4. As shown in Fig. 2, we have selected for each system one participant as an interface, named h, k, v and w. As in Fig. 1, we consider here only static aspects abstracting from dynamic issues, like the logical order of the exchanged messages, whose representation depends on the chosen formalism.

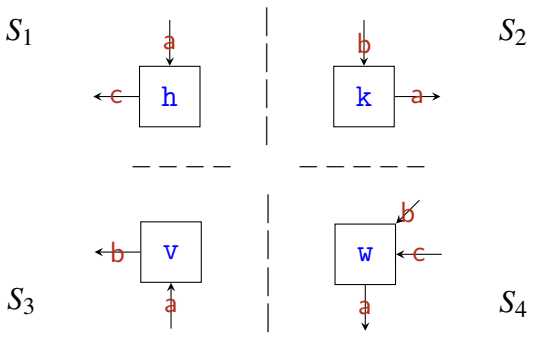


Figure 2: Four interface participants

Following the PaI approach, the composition of the four systems above consists in replacing the participants h, k, v and w, chosen as interfaces, by gateways. Note that a message, like a in S_1 sent to h, could be forwarded (unlike the binary case) to different other gateways. This means that a *connection policy* has to be set up in order to appropriately define the gateways. Such a policy primarily depends on which partner is chosen for the current message to be exchanged.

For what concerns the present example, one could decide that message a received by h has to be forwarded to w; the a received by v to k; the b received by k and w to v; the c received by w to h. Another possible choice could be similar to the previous one but for the forwarding of the messages a: the one received by h could be forwarded now to k whereas the one received by v could be forwarded to w. Such different "choices of partners", that we formalise by introducing the notion of *connection model*, can be graphically represented, respectively, by Choice A and Choice B in Fig. 3.

[1] It is of course possible to compose, two by two, several systems using binary composition, but in that way – by looking at systems as vertices and gateway connections as undirected edges – we can get only tree-like structures of systems.

Figure 3: Two possible choices of partners.

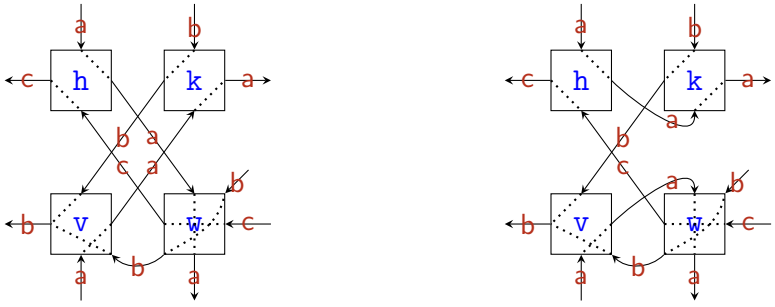


Figure 4: Two possible PaI multicompositions via gateways

The architecture of the resulting composed systems, according to the particular choices of partners (i.e. connection models), are represented by the diagrams in Fig. 4.

In both drawings of Fig. 4, the names h,k, etc. do now represent gateways. It is important to see that even if the original CFSMs for the participants in the single systems, like the CFSM for v in S_3, are given, the connection models and the drawings in Fig. 4 do not always determine how a gateway CFSM, modelling the dynamic forwarding strategy, should look like. This can be illustrated by looking at message b and participant v. No matter whether we consider Choice A or Choice B it is not determined when the gateway for v will accept b from k and when from w. For instance, a message b from w could be accepted by v only after two b's are received from k. Therefore, a given choice of partners needs, in general, to be "refined" – according to the formalism taken into account – into a specific *connection policy* taking care of the dynamic choice of partners.

This PaI approach to multicomposition has been exploited in [3] for a MPST formalism with synchronous communications. We are now going to realise PaI multicomposition in the context of CFSM systems with asynchronous communications.

3 Systems of Communicating Finite State Machines

Communicating Finite State Machines (CFSMs) is a widely investigated formalism for the description and analysis of distributed systems, originally proposed in [12]. CFSMs are a variant of finite state I/O-automata that represent processes which communicate by asynchronous exchanges of messages via FIFO channels. We now recall (partly following [15, 17, 24, 1]) the definitions of CFSM and system of

CFSMs.

We assume given a countably infinite set $\mathbf{P}_{\mathfrak{U}}$ of participant names (ranged over by $\mathtt{p},\mathtt{q},\mathtt{r},\mathtt{h},\mathtt{k},\ldots$) and a countably infinite alphabet $\mathbb{A}_{\mathfrak{U}}$ of messages (ranged over by a, b, c, l, m, ...).

Definition 3.1 (CFSM). *Let* $\mathbf{P}$ *and* $\mathbb{A}$ *be finite subsets of* $\mathbf{P}_{\mathfrak{U}}$ *and* $\mathbb{A}_{\mathfrak{U}}$ *respectively.*

i) The set $C_{\mathbf{P}}$ *of* channels *over* $\mathbf{P}$ *is defined by* $C_{\mathbf{P}} = \{\mathtt{pq} \mid \mathtt{p},\mathtt{q} \in \mathbf{P}, \mathtt{p} \neq \mathtt{q}\}$

ii) The set $Act_{\mathbf{P},\mathbb{A}}$ *of* actions *over* $\mathbf{P}$ *and* $\mathbb{A}$ *is defined by* $Act_{\mathbf{P},\mathbb{A}} = C_{\mathbf{P}} \times \{!,?\} \times \mathbb{A}$

The subject *of an output action* $\mathtt{pq}!\mathtt{m}$ *and of an input action* $\mathtt{qp}?\mathtt{m}$ *is* $\mathtt{p}$.

iii) A communicating finite-state machine over $\mathbf{P}$ and $\mathbb{A}$ *is a finite transition system given by a tuple*

$$M = (Q, q_0, \mathbb{A}, \delta)$$

where Q *is a finite set of states,* $q_0 \in Q$ *is the initial state, and* $\delta \subseteq Q \times Act_{\mathbf{P},\mathbb{A}} \times Q$ *is a set of transitions such that all the actions have the same subject, to which we refer as the* name *of* M.

We shall write $M_{\mathtt{p}}$ to denote a CFSM with name p. Where no ambiguity arises we shall refer to a CFSM by its name.

Notice that the above definition of CFSM is generic with respect to the underlying sets $\mathbf{P}$ and $\mathbb{A}$. This is necessary, since we shall not deal with a single system of CFSMs but with an arbitrary number of systems of CFSMs that can be *composed*. We shall write C and Act instead of $C_{\mathbf{P}}$ and $Act_{\mathbf{P},\mathbb{A}}$ when no ambiguity can arise. We assume $l, l', \ldots$ to range over Act; $\varphi, \varphi', \ldots$ to range over Act^* (the set of finite words over Act), and $w, w', \ldots$ to range over $\mathbb{A}^*$ (the set of finite words over $\mathbb{A}$). The symbol $\varepsilon\,(\notin \mathbb{A} \cup Act)$ denotes the empty word and $|\, v \,|$ the lenght of a word $v \in Act^* \cup \mathbb{A}^*$.

The transitions of a CFSM are labelled by actions; a label $\mathtt{sr}!\mathtt{a}$ represents the asynchronous sending of message a from machine s to r through channel sr and, dually, $\mathtt{sr}?\mathtt{a}$ represents the reception (consumption) of a by r from channel sr.

Given a CFSM $M = (Q, q_0, \mathbb{A}, \delta)$, we also define

$$\mathsf{in}(M) = \{\mathtt{a} \mid (\_, \_\_?\mathtt{a}, \_) \in \delta\} \quad \text{and} \quad \mathsf{out}(M) = \{\mathtt{a} \mid (\_, \_\_!\mathtt{a}, \_) \in \delta\}.$$

If M is a CFSM with name p, we also write $\mathsf{in}(\mathtt{p})$ for $\mathsf{in}(M)$ and $\mathsf{out}(\mathtt{p})$ for $\mathsf{out}(M)$. Note that, in concrete examples, the name of a CFSM together with its input and output messages can be graphically depicted as in Fig. 2.

A state $q \in Q$ with no outgoing transition is *final*; q is a *sending* (resp. *receiving*) state if it is not final and all outgoing transitions are labelled with sending (resp. receiving) actions; q is a *mixed* state if there are at least two outgoing transitions such that one is labelled with a sending action and the other one is labelled with a receiving action.

A *communicating system*, called "protocol" in [12], is a finite set of CFSMs. In [15, 17, 24] the names of the CFSMs in a system are called *roles*. In the present paper we call them *participants*.

The dynamics of a system is formalised as a transition relation on configurations, where a configuration is a pair of tuples: a tuple of states of the machines in the system and a tuple of buffers representing the content of the channels.

Definition 3.2 (Communicating system and configuration). *Let* $\mathbf{P}$ *and* $\mathbb{A}$ *be as in Def. 3.1.*

i) A communicating system (CS) over $\mathbf{P}$ and $\mathbb{A}$ *is a set* $S = (M_{\mathtt{p}})_{\mathtt{p} \in \mathbf{P}}$ *where for each* $\mathtt{p} \in \mathbf{P}$, $M_{\mathtt{p}} = (Q_{\mathtt{p}}, q_{0\mathtt{p}}, \mathbb{A}, \delta_{\mathtt{p}})$ *is a CFSM over* $\mathbf{P}$ *and* $\mathbb{A}$.

ii) A configuration *of a system* S *is a pair* $s = (\vec{q}, \vec{w})$ *where*

$$\vec{q} = (q_{\mathtt{p}})_{\mathtt{p} \in \mathbf{P}} \text{ with } q_{\mathtt{p}} \in Q_{\mathtt{p}}, \quad \text{and} \quad \vec{w} = (w_{\mathtt{pq}})_{\mathtt{pq} \in C} \text{ with } w_{\mathtt{pq}} \in \mathbb{A}^*.$$

The component $\vec{q}$ is the control state *of the system and $q_{\mathtt{p}} \in Q_{\mathtt{p}}$ is the* local state *of machine $M_{\mathtt{p}}$. The component $\vec{w}$ represents the state of the channels of the system and $w_{\mathtt{pq}} \in \mathbb{A}^*$ is the state of the channel $\mathtt{pq}$, i.e. the messages sent from $\mathtt{p}$ to $\mathtt{q}$. The initial configuration of S is $s_0 = (\vec{q}_0, \vec{\varepsilon})$ with $\vec{q}_0 = (q_{0_{\mathtt{p}}})_{\mathtt{p} \in \mathbf{P}}$.*

In the following we shall often denote a communicating system $(M_{\mathtt{p}})_{\mathtt{p} \in \{\mathtt{r}_i\}_{i \in I}}$ by $(M_{\mathtt{r}_i})_{i \in I}$.

Definition 3.3 (Reachable configuration)**.** *Let S be a communicating system over $\mathbf{P}$ and $\mathbb{A}$, and let $s = (\vec{q}, \vec{w})$ and $s' = (\vec{q'}, \vec{w'})$ be two configurations of S. Configuration s'* is reachable from *s by firing a transition with action l, written $s \xrightarrow{l} s'$, if there is $\mathtt{a} \in \mathbb{A}$ such that one of the following conditions holds:*

1. *$l = \mathtt{sr}!\mathtt{a}$ and $(q_{\mathtt{s}}, l, q'_{\mathtt{s}}) \in \delta_{\mathtt{s}}$ and*
   a) *for all $\mathtt{p} \neq \mathtt{s}$: $q'_{\mathtt{p}} = q_{\mathtt{p}}$ and*
   b) *$w'_{\mathtt{sr}} = w_{\mathtt{sr}} \cdot \mathtt{a}$ and for all $\mathtt{pq} \neq \mathtt{sr}$: $w'_{\mathtt{pq}} = w_{\mathtt{pq}}$;*
2. *$l = \mathtt{sr}?\mathtt{a}$ and $(q_{\mathtt{r}}, l, q'_{\mathtt{r}}) \in \delta_{\mathtt{r}}$ and*
   a) *for all $\mathtt{p} \neq \mathtt{r}$: $q'_{\mathtt{p}} = q_{\mathtt{p}}$ and*
   b) *$w_{\mathtt{sr}} = \mathtt{a} \cdot w'_{\mathtt{sr}}$ and for all $\mathtt{pq} \neq \mathtt{sr}$: $w'_{\mathtt{pq}} = w_{\mathtt{pq}}$.*

We write $s \to s'$ if there exists l such that $s \xrightarrow{l} s'$ and we write $s \not\to$ if no s' and no l exist with $s \xrightarrow{l} s'$. As usual, we denote the reflexive and transitive closure of $\to$ by $\to^*$. The set of reachable configurations *of S is $\mathsf{RC}(S) = \{s \mid s_0 \to^* s\}$.*

According to the above definition, communication happens via buffered channels following the FIFO principle.

The overall behaviour of a system can be described (at least) by the traces of configurations that are reachable from a distinguished initial one. Configurations may exhibit some pathological properties, like various forms of *deadlock* or *progress violation*, channels containing messages that will never be consumed (*orphan messages*) or just sent to a participant who is expecting another message to come (*unspecified receptions*). The goal of the analysis of communicating systems is to check whether such kinds of configurations are reachable or not. Although the desirable system properties are undecidable in general [12], sufficient conditions are known that are effectively checkable relying, for instance, on half-duplex communication [15], on the form of network topologies [16], or on synchronous compatibility checking [19].

We formalise now a number of relevant communication properties for systems of CFSMs that we shall deal with in the present paper.

Definition 3.4 (Communication properties)**.** *Let S be a communicating system, and let $s = (\vec{q}, \vec{w})$ be a configuration of S.*

i) s is a deadlock configuration *of S if $\vec{w} = \vec{\varepsilon}$ and $\forall \mathtt{p} \in \mathbf{P}$. $q_{\mathtt{p}}$ is a receiving state.*
I.e. all buffers are empty, but all machines are waiting for a message.
We say that S is deadlock-free *whenever, for any $s \in \mathsf{RC}(S)$, s is not a deadlock configuration.*

ii) s is an orphan-message configuration *of S if $\forall \mathtt{p} \in \mathbf{P}$. $q_{\mathtt{p}}$ is final and $\vec{w} \neq \vec{\varepsilon}$.*
I.e. each machine is in a final state, but there is still at least one non-empty buffer. We say that S is orphan-message free *whenever, for any $s \in \mathsf{RC}(S)$, s is not an orphan-message configuration.*

iii) s is an unspecified reception configuration *of S if $\exists \mathtt{r} \in \mathbf{P}$ such that*
   a) *$q_{\mathtt{r}}$ is a receiving state; and*

b) $\forall \mathtt{s} \in \mathbf{P}.[\,(q_{\mathtt{r}}, \mathtt{sr}?\mathtt{a}, q'_{\mathtt{r}}) \in \delta_{\mathtt{r}} \implies (|w_{\mathtt{sr}}| > 0 \ \wedge \ w_{\mathtt{sr}} \notin \mathtt{a} \cdot \mathbb{A}^*)\,]$.

I.e. there is a receiving state $q_{\mathtt{r}}$ which is prevented from receiving any message from any of its buffers. (In other words, in each channel $\mathtt{sr}$ from which role $\mathtt{r}$ could consume there is a message which cannot be received by $\mathtt{r}$ in state $q_{\mathtt{r}}$.) We say that S is reception-error free *whenever, for any $s \in \mathsf{RC}(S)$, s is not an unspecified reception configuration.*

iv) S satisfies the progress property *if for all $s = (\vec{q}, \vec{w}) \in \mathsf{RC}(S)$, either there exists s' such that $s \to s'$ or $(\forall \mathtt{p} \in \mathbf{P}.\ q_{\mathtt{p}}$ is final$)$.*

v) s is a $\mathtt{p}$-lock configuration *of S if $\mathtt{p} \in \mathbf{P}$, $q_{\mathtt{p}}$ is a receiving state and*

$\mathtt{p}$ *does not appear as subject in any label of any transition sequence from s*

i.e. $\mathtt{p}$ remains stuck in all possible transition sequences from s. We say that S is lock-free *whenever, for each $\mathtt{p} \in \mathbf{P}$ and each $s \in \mathsf{RC}(S)$, s is not a $\mathtt{p}$-lock configuration.*

Note that progress property (iv) implies deadlock-freeness. Moreover, an unspecified reception configuration is trivially a $\mathtt{p}$-lock for some $\mathtt{p}$. This immediately implies that lock-freeness implies reception-error-freeness. It is also straightforward to check that lock-freeness does imply both deadlock-freeness and progress. The other properties are mutually independent.

The above definitions of communication properties (i)–(iv) are the same as the properties considered in [17], though the above formulation of progress is slightly simpler but equivalent to the one in [17]. The notions of orphan message and unspecified reception are also the same as in [24]. The same notions of deadlock and unspecified reception are given in [15] and inspired by [12]. The deadlock notions in [12] and [24] coincide with [15] and [17] if the local CFSMs have no final states. Otherwise deadlock in [24] is weaker than deadlock above. A still weaker notion of deadlock configuration, and hence a stronger notion of deadlock-freeness, has been suggested in [28]. This deadlock notion has been formally related to the above communication properties in [1].

4 PaI Multicomposition of Communicating Systems

As described in Section 2, the PaI approach to multicomposition of systems consists in replacing, in each to-be-composed system, one participant identified as an interface by a forwarder (that we dub "gateway"). Any participant in a system, say $\mathtt{h}$, can be considered as an interface. This means that we can look at the CFSM $\mathtt{h}$ as an abstract description of what the system expects from a number of "outer" systems (the environment) through their respective interfaces. Hence, any message received by $\mathtt{h}$ from another participant $\mathtt{p}$ of the system (to which $\mathtt{h}$ belongs) is interpreted as a message to be forwarded to some other interface $\mathtt{h}'$ among the available ones. Conversely, any message sent from $\mathtt{h}$ to another participant $\mathtt{p}$ of the system (to which $\mathtt{h}$ belongs) is interpreted as a message to be received from some other interface $\mathtt{h}'$ and to be forwarded to $\mathtt{p}$.

In order to clarify the notions introduced in this section, we present below an example from [3], "implemented" here in the CFSM formalism.

Example 4.1 (Working example)**.** Let us consider the following four systems[2]:

System-1 with participants $\mathtt{h}_1$ and $\mathtt{p}$.

Participant $\mathtt{h}_1$ controls the entrance of customers in a mall (via some sensor). As soon as a customer enters, $\mathtt{h}_1$ sends a message start to the participant $\mathtt{p}$ which controls a display for advertisements.

[2]For the sake of simplicity, the example considers only systems with two or three participants. Our definitions and results are of course independent of the number of participants in the single systems.

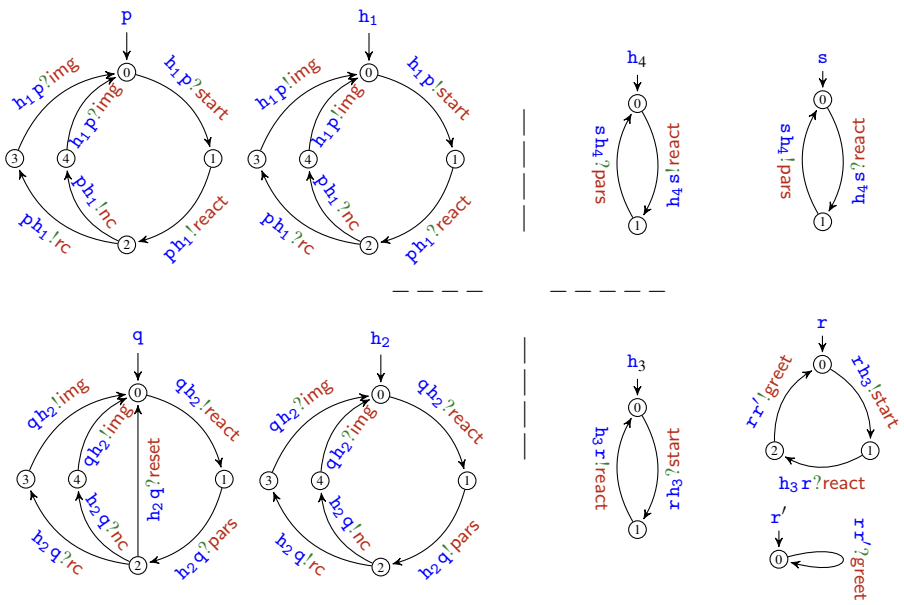


Figure 5: The four communicating systems formalising the systems of Example 4.1

On receiving the start message, p displays a general advertising image. Participant p does also control a sensor detecting emotional reactions as well as a card reader distinguishing regular from new customers. Such information, through the messages react, rc and nc is sent to h_1. Using that information h_1 sends to p a customised image, depending on the kind of the customer, through message img.

System-2 with participants h_2 and q.

Participant h_2 controls an image display. Images are provided by participant q according to some parameters sent by h_2 itself and depending on the reaction acquired by a sensor driven by q. Images are chosen also in terms of the kind of customers, on the basis of their cards. Participant q is able to receive a reset message too, even if h_2 cannot ever send it.

System-3 with participants h_3, r and r′.

Participant r controls a sensor detecting the entrance of people from a door. Once someone enters, a message start is sent by r to participant h_3 which turns on a light. The reaction of who enters, detected by a sensor driven by h_3, is sent back to r which, according to the reaction, communicates to r′ the greeting to be broadcasted from the loudspeakers.

System-4 with participants h_4 and s.

Some sensors driven by Participant h_4 acquire the first reactions of people getting into a hall adorned by several Christmas lights. Such reactions, sent to participant s through a message react, enable s to send to h_4 a set of parameters (pars) allowing the latter to adjust the lights of the hall.

The behaviours of the participants of the above systems – assuming an asynchronous model of communication – can be formalised as CFSMs. So the systems above can be formalised as the following communicating systems

$$S_1 = (M_{\mathtt{x}})_{\mathtt{x}\in\{\mathtt{h}_1,\mathtt{p}\}} \quad S_2 = (M_{\mathtt{x}})_{\mathtt{x}\in\{\mathtt{h}_2,\mathtt{q}\}} \quad S_3 = (M_{\mathtt{x}})_{\mathtt{x}\in\{\mathtt{h}_3,\mathtt{r},\mathtt{r}'\}} \quad S_4 = (M_{\mathtt{x}})_{\mathtt{x}\in\{\mathtt{h}_4,\mathtt{s}\}}$$

as described, anticlockwise, in Figure 5. ◇

Notation: We use the following notation to denote the above set of communicating systems: $\{S_i\}_{i\in\{1,2,3,4\}}$ where $S_i = (M_{\mathtt{x}})_{\mathtt{x}\in\mathbf{P}_i}$ with $\mathbf{P}_1 = \{\mathtt{h}_1,\mathtt{p}\}$, $\mathbf{P}_2 = \{\mathtt{h}_2,\mathtt{q}\}$, $\mathbf{P}_3 = \{\mathtt{h}_3,\mathtt{r},\mathtt{r}'\}$ and $\mathbf{P}_4 = \{\mathtt{h}_4,\mathtt{s}\}$.

The composition of a set of systems relies on a selection of participants, one for each system, considered as interfaces.

Definition 4.2 (Interfaces). *Let $\{S_i\}_{i\in I}$ be a set of communicating systems such that, for each $i \in I$, $S_i = (M_{\mathtt{x}})_{\mathtt{x}\in\mathbf{P}_i}$, where the $\mathbf{P}_i$'s are pairwise disjoint. A set of participants $H = \{\mathtt{h}_i\}_{i\in I} \subseteq \bigcup_{i\in I}\mathbf{P}_i$ is a* set of interfaces *for $\{S_i\}_{i\in I}$ whenever, for each $i \in I$, $\mathtt{h}_i \in \mathbf{P}_i$. An interface $\mathtt{h}_i$ has* no mixed states *if the CFSM $M_{\mathtt{h}_i}$ in S_i has no mixed states.*

Example 4.3. We choose $\{\mathtt{h}_i\}_{i\in\{1,2,3,4\}}$ as set of interfaces for the communicating systems of Figure 5. ◇

We introduce now the notion of *connection model*[3], formalising what we have informally called "choice of partners" in Section 2. A connection model is intended to specify the structural (architectural) aspects of possible "reasonable" connections between interfaces of systems. Connection models should be provided before systems are composed since they help the system designer to avoid blatantly unreasonable compositions. Formally, a connection model is a set of *connections*, where a connection is a triple $(\mathtt{h},\mathtt{a},\mathtt{h}')$ in which $\mathtt{h}$ and $\mathtt{h}'$ are, respectively, interfaces of two systems, say S and S', and $\mathtt{a}$ is an input message for $\mathtt{h}$ and an output message for $\mathtt{h}'$. Being $\mathtt{a}$ an input for $\mathtt{h}$, this participant is supposed to receive $\mathtt{a}$ from the "inside" of S, i.e. from another participant of S. As previously mentioned, PaI multicomposition relies on the idea that $\mathtt{a}$ can be forwarded to the interface of some other system. The connection $(\mathtt{h},\mathtt{a},\mathtt{h}')$ hence specifies that $\mathtt{h}'$ is one of the possible interfaces $\mathtt{a}$ can be forwarded to. This is sound since $\mathtt{a}$ is an output of $\mathtt{h}'$, i.e. it is sent by $\mathtt{h}'$ to some participant of S'. The actual composition will then rely on gateways (forwarders) which comply with the connection model taken into account.

Definition 4.4 (Connection model). *Let $\{S_i\}_{i\in I}$ be a set of communicating systems and let H be a set of interfaces for it.*

i) A connection model for H *is a ternary relation* $\mathrm{CM} \subseteq H \times \mathbb{A}_{!} \times H$ *such that, for each $\mathtt{h} \in H$ and $\mathtt{a} \in \mathbb{A}_{!}$,*

- $\mathtt{a} \in \mathsf{in}(\mathtt{h})$ *implies* $\exists\, \mathtt{h}' \in H$ *s.t.* $\mathtt{a} \in \mathsf{out}(\mathtt{h}')$ *and* $(\mathtt{h},\mathtt{a},\mathtt{h}') \in \mathrm{CM}$
- $\mathtt{a} \in \mathsf{out}(\mathtt{h})$ *implies* $\exists\, \mathtt{h}' \in H$ *s.t.* $\mathtt{a} \in \mathsf{in}(\mathtt{h}')$ *and* $(\mathtt{h}',\mathtt{a},\mathtt{h}) \in \mathrm{CM}$

where $\mathtt{h} \neq \mathtt{h}'$.
Elements of CM *are called* connections. *In particular, $(\mathtt{h},\mathtt{a},\mathtt{h}') \in \mathrm{CM}$ is called* connection for $\mathtt{a}$ (from $\mathtt{h}$ to $\mathtt{h}'$). *We also define* $\mathsf{Msg}(\mathrm{CM}) = \{\mathtt{a} \mid (\_,\mathtt{a},\_) \in \mathrm{CM}\}$ *and assume that any message $\mathtt{a} \in \mathsf{Msg}(\mathrm{CM})$ occurs in one of the interfaces in H either as an input or as an output.*

ii) A connection model CM *for H is* strong *if, for each $\mathtt{h} \in H$ and $\mathtt{a} \in \mathbb{A}_{!}$,*

- $\mathtt{a} \in \mathsf{in}(\mathtt{h})$ *implies* $\exists!\, \mathtt{h}' \in H$ *s.t.* $(\mathtt{h},\mathtt{a},\mathtt{h}') \in \mathrm{CM}$
- $\mathtt{a} \in \mathsf{out}(\mathtt{h})$ *implies* $\exists!\, \mathtt{h}' \in H$ *s.t.* $(\mathtt{h}',\mathtt{a},\mathtt{h}) \in \mathrm{CM}$.

where $\mathtt{h} \neq \mathtt{h}'$ and the unique existential quantifier '$\exists!$' stands for "there exists exactly one".

Connection models can be graphically represented by diagrams, like those used in Fig. 3.

[3] Such a notion was informally introduced in [3] in the setting of MultiParty Session Types.

Example 4.5 (Some connection models)**.** Let $H = \{\mathtt{h},\mathtt{k},\mathtt{v},\mathtt{w}\}$ be the set of interfaces for the systems $\{S_i\}_{i\in\{1,2,3,4\}}$ in Section 2. Fig. 3 represents the following connection models for H:

$$\begin{array}{rcl} \mathrm{CM_A} & = & \{(\mathtt{h},\mathtt{a},\mathtt{w}),(\mathtt{v},\mathtt{a},\mathtt{k}),(\mathtt{w},\mathtt{c},\mathtt{h}),(\mathtt{k},\mathtt{b},\mathtt{v}),(\mathtt{w},\mathtt{b},\mathtt{v})\} \\ \mathrm{CM_B} & = & \{(\mathtt{h},\mathtt{a},\mathtt{k}),(\mathtt{v},\mathtt{a},\mathtt{w}),(\mathtt{w},\mathtt{c},\mathtt{h}),(\mathtt{k},\mathtt{b},\mathtt{v}),(\mathtt{w},\mathtt{b},\mathtt{v})\} \end{array}$$

Obviously, both connection models are not strong, because of the presence of the connections $(\mathtt{k},\mathtt{b},\mathtt{v})$ and $(\mathtt{w},\mathtt{b},\mathtt{v})$.

Let us now provide a connection model for the systems in Fig. 5 with set of interfaces $H = \{\mathtt{h}_i\}_{i\in\{1,2,3,4\}}$. First we determine $\mathsf{in}(\mathtt{h}_1) = \{\mathtt{react},\mathtt{nc},\mathtt{rc}\}$, $\mathsf{out}(\mathtt{h}_1) = \{\mathtt{img},\mathtt{start}\}$, $\mathsf{in}(\mathtt{h}_2) = \{\mathtt{react},\mathtt{img}\}$, $\mathsf{out}(\mathtt{h}_2) = \{\mathtt{nc},\mathtt{rc},\mathtt{pars}\}$, $\mathsf{in}(\mathtt{h}_3) = \{\mathtt{start}\}$, $\mathsf{out}(\mathtt{h}_3) = \{\mathtt{react}\}$, and $\mathsf{in}(\mathtt{h}_4) = \{\mathtt{pars}\}$, $\mathsf{out}(\mathtt{h}_4) = \{\mathtt{react}\}$. A connection model for H is

$$\begin{array}{rcl} \mathrm{CM} & = & \{(\mathtt{h}_1,\mathtt{react},\mathtt{h}_4),(\mathtt{h}_3,\mathtt{start},\mathtt{h}_1),(\mathtt{h}_2,\mathtt{img},\mathtt{h}_1),(\mathtt{h}_1,\mathtt{nc},\mathtt{h}_2), \\ & & (\mathtt{h}_1,\mathtt{rc},\mathtt{h}_2),(\mathtt{h}_4,\mathtt{pars},\mathtt{h}_2),(\mathtt{h}_2,\mathtt{react},\mathtt{h}_3)\} \end{array}$$

The representation of CM is as in Fig. 6. Obviously, this connection model is strong. ◇

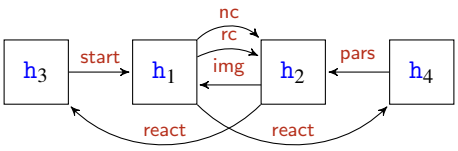


Figure 6: A connection model for the interfaces of Fig. 5.

When we have more than two systems to compose, the gateways are, in general, not uniquely determined. In order to produce gateways out of interfaces we need to decide which connection model we wish to take into account and how the interfaces do actually interact "complying" with the connection model. Once a connection model is selected, the forwarding strategy of the gateway is still not uniquely determined if the connection model is not strong. The reason is that in the case of at least two connectors with the same source or the same target, like $(\mathtt{k},\mathtt{b},\mathtt{v})$ and $(\mathtt{w},\mathtt{b},\mathtt{v})$ in Example 4.5, the gateway for $\mathtt{v}$ has a dynamic choice when to accept message $\mathtt{b}$ from $\mathtt{k}$ and when from $\mathtt{w}$. Therefore we need further (dynamic) information which will be provided by *connection policies*. A connection policy is itself a communicating system which describes the dynamic choice of partners among the possible gateways by respecting the constraints of (that is, complying with) the connection model. Technically, we first associate a set of CFSMs (the "local connection policy set") to each interface. Any element of this set specifies which communications to the "outside" are allowed in which state. Technically these communications are dual to the communications of its corresponding interface.

Definition 4.6 (Local Connection Policy Set)**.** *Let* CM *be a connection model for a set of interfaces H and let $\mathtt{h} \in H$ with CFSM $M_\mathtt{h} = (Q, q_0, \mathbb{A}, \delta)$.*
The local connection policy set *of $M_\mathtt{h}$ w.r.t.* CM *is the set of CFSMs $\mathsf{LCPS}(M_\mathtt{h}, \mathrm{CM})$ defined as follows:*

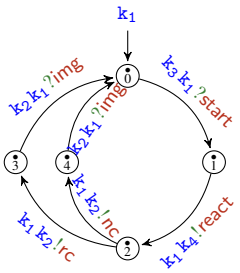
$$\mathsf{LCPS}(M_\mathtt{h}, \mathrm{CM}) = \{(\dot{Q}, \dot{q}_0, \mathbb{A}, \dot{\delta}) \mid \quad \dot{\delta} \text{ is a minimal relation s.t. } (*) \text{ and } (**)\}$$

where $\dot{Q} = \{\dot{q} \mid q \in Q\}$ and
$(*) = q \xrightarrow{\mathtt{rh?a}} q' \in \delta$ *implies* $\exists \mathtt{p} \in H \setminus \{\mathtt{h}\}$ *s.t.* $\dot{q} \xrightarrow{\dot{\mathtt{h}}\dot{\mathtt{p}}!\mathtt{a}} \dot{q}' \in \dot{\delta}$ *and* $(\mathtt{h},\mathtt{a},\mathtt{p}) \in$ CM,
$(**) = q \xrightarrow{\mathtt{hr!a}} q' \in \delta$ *implies* $\exists \mathtt{p} \in H \setminus \{\mathtt{h}\}$ *s.t.* $\dot{q} \xrightarrow{\dot{\mathtt{p}}\dot{\mathtt{h}}?\mathtt{a}} \dot{q}' \in \dot{\delta}$ *and* $(\mathtt{p},\mathtt{a},\mathtt{h}) \in$ CM.

Notice that, in the above definition, each CFSM in $\mathsf{LCPS}(M_{\mathtt{h}},\mathrm{CM})$ has name $\dot{\mathtt{h}}$. Moreover, $\dot{q}$ (resp. $\dot{\mathtt{h}}$) is to be looked at as a "decoration" of the state q (resp. the name $\mathtt{h}$). This will enable us to immediately retrieve q (resp. $\mathtt{h}$) out of $\dot{q}$ (resp. $\dot{\mathtt{h}}$).

Notation: In the following, for the sake of readability, we shall write $\mathtt{k}$ (resp. $\mathtt{k}_i$) for $\dot{\mathtt{h}}$ (resp. $\dot{\mathtt{h}}_i$).

Local connection policy sets are finite, since they contain CFSMs which only differ in the names of participants and these names belong to a finite set. Any element $(\dot{Q},\dot{q}_0,\mathbb{A},\dot{\delta})$ of $\mathsf{LCPS}(M_{\mathtt{h}},\mathrm{CM})$ does comply with the connection model CM, since it can only have transitions $\dot{q} \xrightarrow{\dot{\mathtt{h}}\mathtt{p}!\mathtt{a}} \dot{q}' \in \dot{\delta}$ with $(\mathtt{h},\mathtt{a},\mathtt{p}) \in \mathrm{CM}$ and transitions $\dot{q} \xrightarrow{\dot{\mathtt{p}}\mathtt{h}?\mathtt{a}} \dot{q}' \in \dot{\delta}$ with $(\mathtt{p},\mathtt{a},\mathtt{h}) \in \mathrm{CM}$, Moreover, $\mathsf{LCPS}(M_{\mathtt{h}},\mathrm{CM})$ is a singleton if the connection model CM is strong.



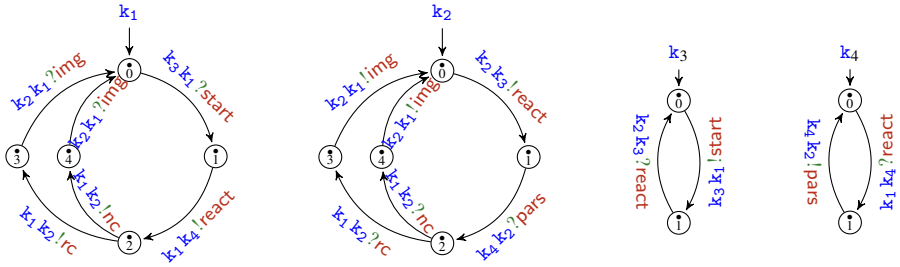
Example 4.7 (An element of a local connection policy set)**.** Let $M_{\mathtt{h}_1}$ be the CFSM for the participant $\mathtt{h}_1$ of Example 4.1 and let CM be the strong connection model for $H = \{\mathtt{h}_i\}_{i\in\{1,2,3,4\}}$ of Example 4.5. The CFSM on the right is the unique element of $\mathsf{LCPS}(M_{\mathtt{h}_1},\mathrm{CM})$. ◇

Given a connection model, a connection policy is obtained by choosing, for each interface, an element of its local connection policy set.

Definition 4.8 (Connection policy)**.** *Let $\{S_i\}_{i\in I}$ be a set of communicating systems such that, for each $i \in I$, $S_i = (M_{\mathtt{x}})_{\mathtt{x}\in\mathbf{P}_i}$, and let* CM *be a connection model for a set of interfaces $H = \{\mathtt{h}_i\}_{i\in I}$. A* connection policy (for H) complying with CM *is a communicating system $\mathbb{K} = (M_{\mathtt{k}_i})_{i\in I}$ such that, for each $i \in I$, $M_{\mathtt{k}_i} \in \mathsf{LCPS}(M_{\mathtt{h}_i},\mathrm{CM})$.*

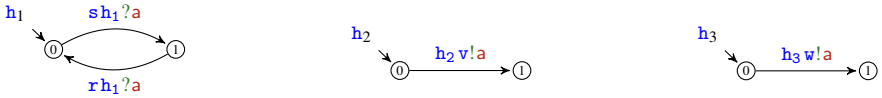
Connection policies are made of local connection policies which, due to the conditions $(*)$ and $(**)$ in Definition 4.6, fit to the given communication model CM. Consequently, in the above definition, the connection policy is said to be compliant with CM. If we dropped the two requirements $(*)$ and $(**)$ in Definition 4.6 we would get non-compliant connection policies.

Example 4.9 (A connection policy)**.** The following four CFSMs constitute a connection policy for $H = \{\mathtt{h}_i\}_{i\in I}$ complying with CM, where the $M_{\mathtt{h}_i}$'s are as in Figure 5 and CM is the connection model of Example 4.5.

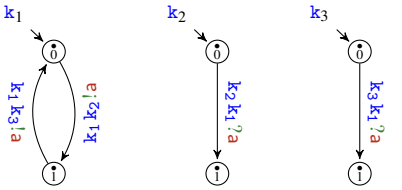


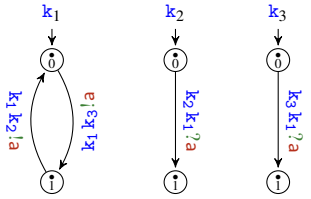
◇

Remark 4.10. A connection model can be looked at as a static and abstract description of connection policies. In particular a connection model abstracts from the order of exchanged messages. As already pointed out above there may be several connection policies complying with a given connection model CM if CM is not strong. As an example assume given three systems with the following interfaces:



We can now consider the following (non-strong) connection model: $\text{CM} = \{(\mathtt{h}_1, \mathtt{a}, \mathtt{h}_2), (\mathtt{h}_1, \mathtt{a}, \mathtt{h}_3)\}$.
It is easy to check that the connection policies $\mathbb{K}_1$ and $\mathbb{K}_2$ below do both comply with CM.

$\mathbb{K}_1 =$ 

$\mathbb{K}_2 =$ 

◇

By now we have almost all the necessary notions to formally define the PaI multicomposition of systems of communicating systems. The only missing piece is that of building the gateways using a connection policy.

We get a gateway essentially by transforming an interface $M_\mathtt{h}$ by inserting a fresh state in between any transition. Any input transition $q \xrightarrow{\mathtt{sh?a}} q'$ (resp. output transition $q \xrightarrow{\mathtt{hs!a}} q'$) of $M_\mathtt{h}$ is then transformed into two consecutive transitions

$$q \xrightarrow{\mathtt{sh?a}} \hat{q} \xrightarrow{\mathtt{hh'!a}} q' \qquad (\text{resp. } q \xrightarrow{\mathtt{h'h?a}} \hat{q} \xrightarrow{\mathtt{hs!a}} q')$$

where $\hat{q}$ is a fresh state and $\dot{q} \xrightarrow{\mathtt{kk'!a}} \dot{q}'$ (resp. $q \xrightarrow{\mathtt{k'k?a}} \dot{q}'$) belonging to the connection policy taken into account. In the formal definition below we distinguish the fresh states by superscripting them by the transition they are "inserted in between".

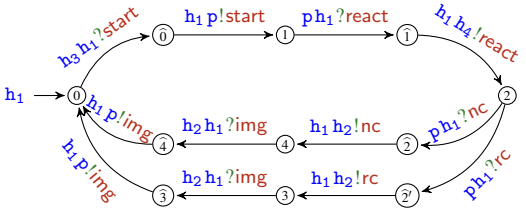
Definition 4.11 (Gateway)**.**
Assume given a connection model CM *and two CFSMs* $M_\mathtt{h}$ *and* $M_\mathtt{k}$ *such that* $M_\mathtt{h} = (Q, q_0, \mathbb{A}, \delta)$ *and* $M_\mathtt{k} = (\dot{Q}, \dot{q}_0, \mathbb{A}, \dot{\delta}) \in \mathsf{LCPS}(M_\mathtt{h}, \text{CM})$. *The* gateway $M_\mathtt{h} \leftarrow\!\!\!\rho M_\mathtt{k}$ *obtained out of* $M_\mathtt{h}$ *and* $M_\mathtt{k}$ *is defined by*

$$M_\mathtt{h} \leftarrow\!\!\!\rho M_\mathtt{k} = (Q \cup \widehat{Q}, q_0, \mathbb{A}, \hat{\delta})$$

where
– $\widehat{Q} = \bigcup_{q \in Q} \{q^{(q,l,q')} \mid (q,l,q') \in \delta\}$,
– $\hat{\delta} = \{(q, \mathtt{rh?a}, \widehat{q}), (\widehat{q}, \mathtt{hs!a}, q') \mid (q, \mathtt{hs!a}, q') \in \delta, (\dot{q}, \dot{\mathtt{r}}\mathtt{k?a}, \dot{q}') \in \dot{\delta},\ \widehat{q} = q^{(q,\mathtt{hs!a},q')}\}$
$\quad \cup \{(q, \mathtt{sh?a}, \widehat{q}), (\widehat{q}, \mathtt{hr!a}, q') \mid (q, \mathtt{sh?a}, q') \in \delta,\ (\dot{q}, \mathtt{k}\dot{\mathtt{r}}\mathtt{!a}, \dot{q}') \in \dot{\delta},\ \widehat{q} = q^{(q,\mathtt{sh?a},q')}\}$.

We refer to $\hat{\delta}$ *as* $\hat{\delta}_\mathtt{h}$ *whenever* h *is not clear from the context; similarly for* $\widehat{Q}$.

Example 4.12 (A gateway)**.** Let $M_{\mathtt{h}_1}$ be as in Example 4.1, and let $M_{\mathtt{k}_1}$ be as in the connection policy of Example 4.9. The gateway $M_{\mathtt{h}_1} \leftarrow\!\!\!\rho M_{\mathtt{k}_1}$ is as follows.



◇

Definition 4.13 (Composability)**.** *Let* $\{S_i\}_{i \in I}$ *be a set of communicating systems such that, for each* $i \in I$, $S_i = (M_\mathtt{x})_{\mathtt{x} \in \mathbf{P}_i}$. *Moreover, let* $H = \{\mathtt{h}_i\}_{i \in I}$ *be a set of interfaces for it. We say that* $\{S_i\}_{i \in I}$ *is* composable with respect to H *whenever the sets* $\mathbf{P}_i$*'s are pairwise disjoint.*

Let us now describe how systems are composed on the basis of a given connection policy.

Definition 4.14 (Multicomposition of communicating systems)**.** *Let $\{S_i\}_{i\in I}$ be a set of communicating systems composable with respect to $H = \{\mathtt{h}_i\}_{i\in I}$ and let $\mathbb{K} = (M_{\mathtt{k}_i})_{i\in I}$ be a connection policy complying with a connection model* CM *for H. The* multicomposition of $\{S_i\}_{i\in I}$ with respect to $\mathbb{K}$ *is the communicating system*

$$\mathscr{MC}(\{S_i\}_{i\in I}, \mathbb{K}) = (M'_{\mathtt{p}})_{\mathtt{p}\in\bigcup_{i\in I}\mathbf{P}_i}$$

where

$$M'_{\mathtt{p}} = \begin{cases} M_{\mathtt{p}} & \text{if } \mathtt{p} \notin \{\mathtt{h}_i\}_{i\in I} \\ M_{\mathtt{h}_i} \leftarrowtail M_{\mathtt{k}_i} & \text{if } \mathtt{p} = \mathtt{h}_i \text{ with } i \in I \end{cases}$$

Note that the CFSMs of a composition are CFSMs over $\mathbf{P} = \bigcup_{i\in I}\mathbf{P}_i$ and $\mathbb{A} = \bigcup_{i\in I}\mathbb{A}_i$. Graphically, the architectural structure of a multicomposition via gateways can be shown as in Fig. 4.

5 On the Preservation of Communication Properties

The main result of the present paper is the safety of PaI multicomposition of CFSM systems for all communication properties of Definition 3.4 but lock-freeness. Apart from orphan-message-freeness we need the no-mixed-state assumption for interfaces to obtain the preservation results.

Theorem 5.1 (Safety of PaI multicomposition of CFSM systems)**.** *Let $\{S_i\}_{i\in I}$ be a set of communicating systems composable with respect to a set $H = \{\mathtt{h}_i\}_{i\in I}$ of interfaces with no mixed states (cf. Definition 4.2) and let $\mathbb{K}$ be a connection policy for H. Let $\mathscr{P}$ be either the property of* deadlock-freeness *or* reception-error-freeness *or* progress*. If $\mathscr{P}$ holds for each S_i with $i \in I$ and for $\mathbb{K}$, then $\mathscr{P}$ holds for $S = \mathscr{MC}(\{S_i\}_{i\in I}, \mathbb{K})$. Moreover, the above holds also if the no-mixed-state condition is removed and $\mathscr{P}$ is* orphan-message-freeness*.*

Remark 5.2. The above result about safety of multicomposition is actually independent of a concrete connection model. Considering connection policies which comply with a connection model is, however, helpful at the design stage of the multicomposition and enhances the possibility of getting connection policies which satisfy communication properties and hence support the preservation of communication properties of the composed systems. ◇

Theorem 5.1 can be proved for each property $\mathscr{P}$ separately by contradiction. In particular by showing that $\mathscr{P}$ does not hold for S implies that it does not hold either for one of the S_i's or for $\mathbb{K}$.

A key notion for the proofs is that of *projection* of a reachable configuration of the composed system to configurations of each of the single systems S_i and also of the connection policy $\mathbb{K}$. On this basis, the most important tool to get contradictions is the subsequent Proposition 5.4 which essentially shows that projections of reachable configurations involving no intermediate gateway states are reachable configurations again. The complete proofs of property preservations are provided in [5]. They are independent of the communication model $\mathbb{K}$ complies with.

Definition 5.3 (Configuration projections)**.** *Let $S = \mathscr{MC}(\{S_i\}_{i\in I}, \mathbb{K})$ be as in Theorem 5.1 (but without no-mixed-state assumption). Let $s = (\vec{q}, \vec{w}) \in \mathsf{RC}(S)$ where $\vec{q} = (q_{\mathtt{p}})_{\mathtt{p}\in\mathbf{P}}$ and $\vec{w} = (w_{\mathtt{pq}})_{\mathtt{pq}\in C_{\mathbf{P}}}$. For each $i \in I$, the projection $s_{|_{\mathbf{i}}}$ of s to S_i is defined by*

$$s_{|_{\mathbf{i}}} = (\vec{q}_{|_{\mathbf{i}}}, \vec{w}_{|_{\mathbf{i}}})$$

where $\vec{q}_{|_{\mathbf{i}}} = (q_{\mathtt{p}})_{\mathtt{p}\in\mathbf{P}_i}$ and $\vec{w}_{|_{\mathbf{i}}} = (w_{\mathtt{pq}})_{\mathtt{pq}\in C_{\mathbf{P}_i}}$.
The projection $s_{|_{\mathbb{K}}}$ of $s = (\vec{q}, \vec{w})$ to $\mathbb{K}$ is defined if $q_{\mathtt{h}_i} \notin \widehat{Q}_{\mathtt{h}_i}$ for each $i \in I$ and then

$$s_{|_{\mathbb{K}}} = (\vec{q}_{|_{\mathbb{K}}}, \vec{w}_{|_{\mathbb{K}}})$$

where $\vec{q}_{|\mathbb{K}} = (p_{\mathtt{k}_i})_{i\in I}$ *is such that, for each* $i \in I$, $p_{\mathtt{k}_i} = \dot{q}_{\mathtt{h}_i}$ *(with* $\dot{q}_{\mathtt{h}_i}$ *being the "dotted decoration" of the local state* $q_{\mathtt{h}_i}$*) and where* $\vec{w}_{|\mathbb{K}} = (w'_{\mathtt{pq}})_{\mathtt{p},\mathtt{q}\in\{\mathtt{k}_i\}_{i\in I},\mathtt{p}\neq\mathtt{q}}$ *is such that, for each pair* $i,j \in I$ *with* $i \neq j$, $w'_{\mathtt{k}_i\mathtt{k}_j} = w_{\mathtt{h}_i\mathtt{h}_j}$.

Proposition 5.4 (On reachability of projections)**.** *Let* $s = (\vec{q},\vec{w}) \in \mathsf{RC}(S)$.

i) For each $i \in I$, $(q_{\mathtt{h}_i} \notin \widehat{Q_{\mathtt{h}_i}} \implies s_{|_{\mathtt{i}}} \in \mathsf{RC}(S_i))$;

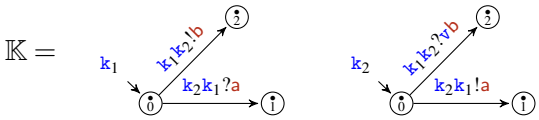
ii) $(q_{\mathtt{h}_i} \notin \widehat{Q_{\mathtt{h}_i}}$ *for each* $i \in I) \implies s_{|_{\mathbb{K}}} \in \mathsf{RC}(\mathbb{K})$.

The connection policy of Example 4.9 does enjoy all the properties of Definition 3.4. Moreover, the interfaces of the four systems of Example 4.1 are all with no mixed state. Hence Theorem 5.1 guarantees that any property (among those of Definition 3.4, but lock-freedom) enjoyed by the systems is also enjoyed by their PaI multicomposition.

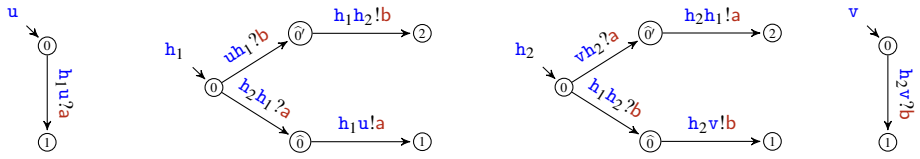
Now we provide some examples for cases in which communication properties are not preserved. First we show that all the three properties for which we have assumed the no-mixed-state condition in Theorem 5.1 would, in general, not be preserved by composition if the condition is dropped. In the counterexamples, the receiving states introduced by the gateway construction cause the breaking of the property taken into account.

Example 5.5 (No-mixed-state counterexample for deadlock-freeness and progress preservation)**.** Let us consider the two following systems S_1 and S_2 with interfaces, respectively, $\mathtt{h}_1$ and $\mathtt{h}_2$ containing mixed states.

S_1 and S_2 are both deadlock free and both enjoy the progress property. There is a unique communication model for their composition: $\textsc{cm} = \{(\mathtt{h}_2,\mathtt{a},\mathtt{h}_1),(\mathtt{h}_1,\mathtt{b},\mathtt{h}_2)\}$ The unique communication policy complying with CM is the following one.



Also $\mathbb{K}$ is deadlock free and enjoys the progress property. The system $\mathscr{MC}(\{S_1,S_2\},\mathbb{K})$ is the following one.



The initial configuration is actually a deadlock, and hence the system does also not enjoy progress. ◇

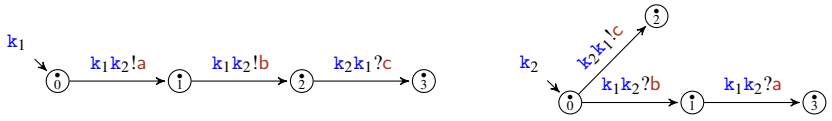
Example 5.6 (No mixed-state counterexample for reception-error-freeness preservation)**.** Let us consider the two following systems S_1 and S_2 with interfaces, respectively, $\mathtt{h}_1$ and $\mathtt{h}_2$ containing mixed states.

S_1 and S_2 are both reception-error free. The unique communication model for their composition is

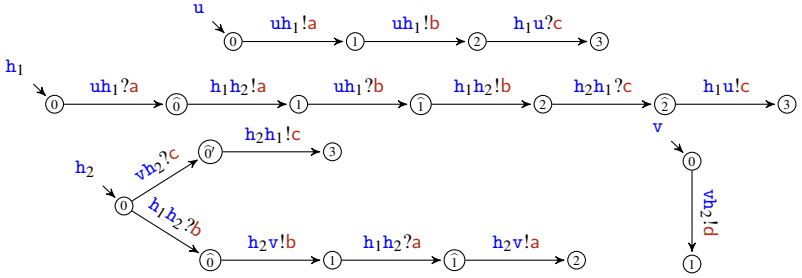
$$\text{CM} = \{(\mathtt{h}_1, \mathtt{a}, \mathtt{h}_2), (\mathtt{h}_1, \mathtt{b}, \mathtt{h}_2), (\mathtt{h}_2, \mathtt{c}, \mathtt{h}_1)\}$$

The unique communication policy complying with CM is

$\mathbb{K} =$



Also $\mathbb{K}$ is reception-error free. The system $\mathscr{MC}(\{S_1, S_2\}, \mathbb{K})$ is the following one.



This communication system, however, is not reception-error free, since it is possible to reach the configuration $s = (\vec{q}, \vec{w})$ where

$$\vec{q} = (2_{\mathtt{u}}, 2_{\mathtt{h}_1}, 0_{\mathtt{h}_2}, 1_{\mathtt{v}}), \qquad w_{\mathtt{h}_1\mathtt{h}_2} = \langle \mathtt{a} \cdot \mathtt{b} \rangle, \qquad w_{\mathtt{vh}_2} = \langle \mathtt{d} \rangle, \qquad w_c = \varepsilon \ \ (\forall c \notin \{\mathtt{h}_1\mathtt{h}_2, \mathtt{vh}_2\})$$

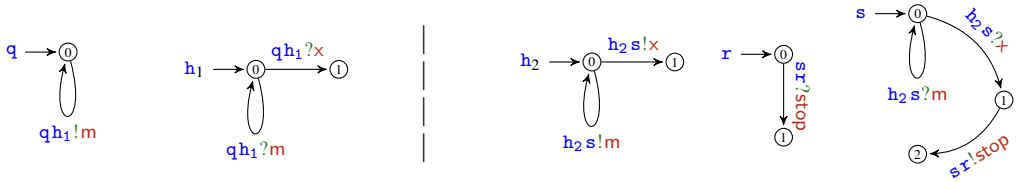
In the configuration s, the CFSM $\mathtt{h}_2$ is in a receiving state, namely 0, from which there are two transitions, namely $(0, \mathtt{vh}_2?\mathtt{c}, \hat{0}')$ and $(0, \mathtt{h}_1\mathtt{h}_2?\mathtt{b}, \hat{0})$. Moreover, the channels $\mathtt{vh}_2$ and $\mathtt{h}_1\mathtt{h}_2$ are both not empty and their first element is different from both b and c. The above configuration is hence an unspecified reception configuration. ◇

Notice that in case we dropped the requirement that $\mathbb{K}$ has to comply with a communication model, the interfaces $\mathtt{h}_1$ and $\mathtt{h}_2$ of Example 5.6 could be simplified to get the counterexample. In particular, they could have just, respectively, two and three states. The use of communication models hence limits the possibility of getting systems whose properties are not preserved by composition. This is an indication that connection models increase the possibility of getting safe compositions.

Let us now turn to the last communication property stated in Definition 3.4 which is lock-freeness. This property is also meaningful in the context of synchronous communication.

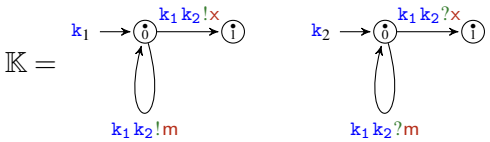
In [8, Example 6.7] a counterexample is provided, showing that in the formalism of *synchronous* CFSMs the properties of (synchronous) lock-freeness and deadlock-freeness are, in general, not preserved. As a matter of fact, lock-freeness is problematic also for the case of asynchronous communications and no mixed states, as shown in the following example, adapted from [8].

Example 5.7 (Lock-freeness is not preserved by composition)**.** Let us consider the following communicating systems S_1 and S_2.

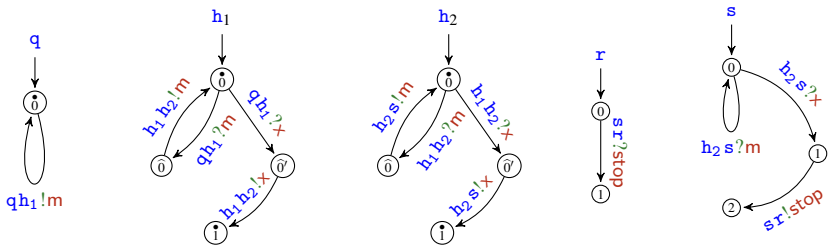


Note that both S_1 and S_2 are lock-free and their respective interfaces $\mathtt{h}_1$ and $\mathtt{h}_2$ have no mixed states.

Let us now consider the (unique) connection policy $\mathbb{K} = (M_{k_i})_{i\in\{1,2\}}$ where $M_{k_1} \in \mathsf{LCPS}(M_{h_1}, \mathrm{CM})$ and $M_{k_2} \in \mathsf{LCPS}(M_{h_2}, \mathrm{CM})$ with connection model $\mathrm{CM} = \{(h_1, x, h_2), (h_1, m, h_2)\}$.



It is easy to see that $\mathbb{K}$ is lock-free. The multicomposition $\mathscr{MC}(\{S_i\}_{i\in\{1,2\}}, \mathbb{K})$ is the following communicating system:



The initial configuration s_0 of $\mathscr{MC}(\{S_i\}_{i\in\{1,2\}}, \mathbb{K})$ is an $\mathtt{r}$-lock, since the transition $\mathtt{q}\,\mathtt{h}_1?\mathtt{x}$ of $\mathtt{h}_1$ can never be fired, so implying, in turn, that also $\mathtt{h}_1\,\mathtt{h}_2!\mathtt{x}$ of $\mathtt{h}_1$, $\mathtt{h}_1\,\mathtt{h}_2?\mathtt{x}$ of $\mathtt{h}_2$, $\mathtt{h}_2\,\mathtt{s}!\mathtt{x}$ of $\mathtt{h}_2$, $\mathtt{h}_2\,\mathtt{s}?\mathtt{x}$ of $\mathtt{s}$ and $\mathtt{s}\,\mathtt{r}!\mathtt{stop}$ of $\mathtt{s}$ can never be fired. Hence, no transition sequence out of s_0 will ever involve the participant $\mathtt{r}$. Thus $\mathscr{MC}(\{S_i\}_{i\in\{1,2\}}, \mathbb{K})$ is not lock-free. ◇

Remark 5.8. It is worth noticing that, in Examples 5.5, 5.6 and 5.7 above, the interfaces of the systems we compose do have unreachable states. It is hence natural to wonder whether it is the presence of unreachable states in interfaces that entails the possibility of getting counterexamples for the properties taken into account. ◇

6 Conclusions

The necessity of supporting the modular development of concurrent/distributed systems, as well as the need to extend/modify/adapt/upgrade them, urged the investigation of composition methods. Focusing on such investigations in the setting of abstract formalisms for the description and verification of systems enables to get general and formal guarantees of relevant features of the composition methods.

An investigation of composition in a formalism for choreographic programming was carried out in [25]. In [23] a modular technique was developed for the verification of aspect-oriented programs expressed as state machines. Team Automata is another formalism in which compositionality issues have been addressed [10, 9], as well as in assembly theories considered in [20]. Composition for protocols described via a process algebra has been investigated in [11]. In [14, 26] a technique for modular design in the setting of reactive programming is proposed. A possible approach to composition for a MultiParty Session Type (MPST) formalism is developed in [27]. The mentioned papers provide just a glimpse of the variety of approaches to system composition in the literature.

Papers dealing with the (binary) composition of systems on the basis of the *participants-as-interfaces* (PaI) approach have been pointed out already in Section 1 and the idea of PaI for multicomposition of systems has been explained in Section 2. In the present paper we study the PaI approach to multicomposition for systems of asynchronously communicating finite state machines (CFSMs). We show that

(under mild assumptions) important communication properties relevant in the context of asynchronous communication, like freeness of orphan messages and unspecified receptions, are preserved by composition (a feature dubbed *safety* in [3]). For this we assume that for each single system one participant is chosen as an interface. A key role in our work, inspired by [3], is played by *connection policies*, which are CFSM systems which determine the ways how interfaces can interact when they are replaced by gateways (forwarders) in system compositions.

For an "unstructured" formalism like CFSM, the natural generalisation from multicomposition with single interfaces to multicomposition with multiple interfaces (per system) is not trouble-free, as discussed in [1, Sect.6] for binary composition. This is mainly due to the possible indirect interactions which could occur among the interfaces inside the single systems. In more structured formalisms, however, such possible interactions can be controlled. This is the case, for instance, in MPST formalisms. In fact, in [18] the authors devise a direct composition mechanism without using gateways for MPST systems. Such a mechanism allows for the presence of multiple interfaces thanks to an hybridisation with local and external information of the standard notion of global type. A combination of global and local constructs in order to get flexible specifications (uniformly describing both the internal and the interface behavior of systems) is also present in [13].

There are several directions to be pursued in future work starting from our results. On the first place, we want to generalise the notion of connection policy such that PaI multicomposition could actually be obtained by replacing interfaces by gateways which, instead of interacting directly with each other, can interact through an "interfacing infrastructure" represented via a system of CFSMs. Such a generalisation would be equivalent to multicomposition where exactly one system can have multiple interfaces. Let us consider a possible application of the above idea. In Example 4.1, in the resulting composed system, both participants `p` and `q` do emit a `react` message. It would be more natural to have only one of them producing such a message, e.g., to have `p` be the sole sensor registering reactions which then passes that information to both `r` and `s`. This would not be possible by our composition mechanism and we cannot but make the best of the fact that we are dealing with two sensors. One could think, instead, about using an "interfacing infrastructure" containing some further participant enabling to ignore the messages from one sensor and properly duplicating the messages from the other.

We are also planning to consider further communication properties, like strong lock-freeness (any participant can eventually progress in any continuation of any reachable configuration), as well as to investigate conditions to get lock-freeness preservation, not guaranteed yet.

Unlike the present paper, in [1] safety is ensured for the binary case by assuming compatibility of interfaces and an extra condition (called ?!-determinism) on them. We are currently considering a generalisation of the binary compatibility relation. Such generalisation should imply relevant communication properties for the communication policy it depends on.

Finally, we are interested in considering "partial" gateways, where only some communications of an interface are interpreted as communications with the environment. Such an idea was actually implemented in [2] in a MPTS setting for a restricted client-multiserver composition with synchronous communications.

Acknowledgements We warmly thank the ICE'24 reviewers for their careful reading, their thoughtful comments/suggestions and the helpful discussion in the forum. We also thank Emilio Tuosto for his nice tikz style for automata.

References

[1] Franco Barbanera, Ugo de'Liguoro & Rolf Hennicker (2019): *Connecting open systems of communicating finite state machines*. *J. Log. Algebraic Methods Program.* 109, article 100476, doi:10.1016/J.JLAMP.2019.07.004.

[2] Franco Barbanera, Mariangiola Dezani-Ciancaglini & Ugo de'Liguoro (2022): *Open compliance in multiparty sessions*. In S. Lizeth Tapia Tarifa & José Proença, editors: *Proc. FACS 2022*, *LNCS* 13712, Springer, pp. 222–243, doi:10.1007/978-3-031-20872-0_13. Extended version at http://www.di.unito.it/~dezani/papers/bd23b.pdf.

[3] Franco Barbanera, Mariangiola Dezani-Ciancaglini, Lorenzo Gheri & Nobuko Yoshida (2023): *Multicompatibility for Multiparty-Session Composition*. In Santiago Escobar & Vasco T. Vasconcelos, editors: *Proc. PPDP 2023*, ACM, pp. 2:1–2:15, doi:10.1145/3610612.3610614.

[4] Franco Barbanera, Mariangiola Dezani-Ciancaglini, Ivan Lanese & Emilio Tuosto (2021): *Composition and decomposition of multiparty sessions*. *J. Log. Algebraic Methods Program.* 119, article 100620, doi:10.1016/j.jlamp.2020.100620.

[5] Franco Barbanera & Rolf Hennicker: *Safe Composition of Systems of Communicating Finite State Machines (Full Version)*. Available at https://github.com/francobarbanera/Safe-Composition-of-CFSM-Systems-Full-version-/blob/35387fa10401b774c4a30ad2a4a2b677ee957a00/CFSM-multicomposition-Full.pdf.

[6] Franco Barbanera, Ivan Lanese & Emilio Tuosto (2020): *Composing communicating systems, synchronously*. In Tiziana Margaria & Bernhard Steffen, editors: *Proc. ISoLA 2020*, *LNCS* 12476, Springer, pp. 39–59, doi:10.1007/978-3-030-61362-4_3.

[7] Franco Barbanera, Ivan Lanese & Emilio Tuosto (2022): *On composing communicating systems*. In Clément Aubert, Cinzia Di Giusto, Larisa Safina & Alceste Scalas, editors: *Proc. ICE 2022*, *EPTCS* 365, Open Publishing Association, pp. 53–68, doi:10.4204/EPTCS.365.4.

[8] Franco Barbanera, Ivan Lanese & Emilio Tuosto (2023): *Composition of synchronous communicating systems*. *J. Log. Algebraic Methods Program.* 135, article 100890, doi:10.1016/J.JLAMP.2023.100890.

[9] Maurice H. ter Beek, Rolf Hennicker & Jetty Kleijn (2020): *Compositionality of Safe Communication in Systems of Team Automata*. In Violet Ka I Pun, Volker Stolz & Adenilso Simão, editors: *Proc. ICTAC 2020*, *LNCS* 12545, Springer, pp. 200–220, doi:10.1007/978-3-030-64276-1_11.

[10] Maurice H. ter Beek & Jetty Kleijn (2003): *Team Automata Satisfying Compositionality*. In Keijiro Araki, Stefania Gnesi & Dino Mandrioli, editors: *Proc. FME 2003*, *LNCS* 2805, Springer, pp. 381–400, doi:10.1007/978-3-540-45236-2_22.

[11] Laura Bocchi, Dominic Orchard & A. Laura Voinea (2023): *A Theory of Composing Protocols*. *Art Sci. Eng. Program.* 7(2), doi:10.22152/PROGRAMMING-JOURNAL.ORG/2023/7/6. Article 6.

[12] Daniel Brand & Pitro Zafiropulo (1983): *On Communicating Finite-State Machines*. *J. ACM* 30(2), pp. 323–342, doi:10.1145/322374.322380.

[13] Luís Caires & Hugo Torres Vieira (2010): *Conversation types*. *Theor. Comput. Sci.* 411(51-52), pp. 4399–4440, doi:10.1016/J.TCS.2010.09.010.

[14] Marco Carbone, Fabrizio Montesi & Hugo Torres Vieira (2018): *Choreographies for Reactive Programming*. *CoRR* abs/1801.08107. arXiv:1801.08107.

[15] Gérard Cécé & Alain Finkel (2005): *Verification of programs with half-duplex communication*. *Inf. Comput.* 202(2), pp. 166–190, doi:10.1016/j.ic.2005.05.006.

[16] Lorenzo Clemente, Frédéric Herbreteau & Grégoire Sutre (2014): *Decidable Topologies for Communicating Automata with FIFO and Bag Channels*. In Paolo Baldan & Daniele Gorla, editors: *Proc. CONCUR 2014*, *LNCS* 8704, Springer, pp. 281–296, doi:10.1007/978-3-662-44584-6_20.

[17] Pierre-Malo Deniélou & Nobuko Yoshida (2012): *Multiparty Session Types Meet Communicating Automata*. In Helmut Seidl, editor: *Proc. ESOP 2012*, pp. 194–213, doi:10.1007/978-3-642-28869-2_10.

[18] Lorenzo Gheri & Nobuko Yoshida (2023): *Hybrid Multiparty Session Types: Compositionality for Protocol Specification through Endpoint Projection*. *Proc. ACM Program. Lang.* 7(OOPSLA1), pp. 112–142, doi:10.1145/3586031.

[19] Rolf Hennicker & Michel Bidoit (2018): *Compatibility Properties of Synchronously and Asynchronously Communicating Components*. *Log. Meth. in Comp. Sci.* 14(1), pp. 1–31, doi:10.23638/LMCS-14(1:1)2018.

[20] Rolf Hennicker & Alexander Knapp (2015): *Moving from interface theories to assembly theories*. *Acta Informatica* 52(2-3), pp. 235–268, doi:10.1007/S00236-015-0220-7.

[21] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In George C. Necula & Philip Wadler, editors: *Proc. POPL 2008*, ACM, pp. 273–284, doi:10.1145/1328438.1328472.

[22] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty asynchronous session types*. *J. ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695.

[23] Shriram Krishnamurthi, Kathi Fisler & Michael Greenberg (2004): *Verifying aspect advice modularly*. In Richard N. Taylor & Matthew B. Dwyer, editors: *Proc. SIGSOFT 2004*, ACM, pp. 137–146, doi:10.1145/1029894.1029916.

[24] Julien Lange, Emilio Tuosto & Nobuko Yoshida (2015): *From Communicating Machines to Graphical Choreographies*. In Sriram K. Rajamani & David Walker, editors: *Proc. POPL 2015*, ACM, pp. 221–232, doi:10.1145/2676726.2676964.

[25] Fabrizio Montesi & Nobuko Yoshida (2013): *Compositional Choreographies*. In Pedro R. D'Argenio & Hernán C. Melgratti, editors: *Proc. CONCUR 2013*, *LNCS* 8052, Springer, pp. 425–439, doi:10.1007/978-3-642-40184-8_30.

[26] Zorica Savanovic, Letterio Galletta & Hugo Torres Vieira (2020): *A type language for message passing component-based systems*. In Julien Lange, Anastasia Mavridou, Larisa Safina & Alceste Scalas, editors: *Proc. ICE 2020*, *EPTCS* 324, pp. 3–24, doi:10.4204/EPTCS.324.3.

[27] Claude Stolze, Marino Miculan & Pietro Di Gianantonio (2023): *Composable partial multiparty session types for open systems*. *Softw. Syst. Model.* 22(2), pp. 473–494, doi:10.1007/S10270-022-01040-X.

[28] Emilio Tuosto & Roberto Guanciale (2018): *Semantics of global view of choreographies*. *J. Log. Algebr. Meth. Program.* 95, pp. 17–40, doi:10.1016/j.jlamp.2017.11.002.

The B2Scala Tool: Integrating Bach in Scala with Security in Mind

Doha Ouardi
Nadi Research Institute
Faculty of Computer Science
University of Namur
Namur, Belgium
doha.ouardi@unamur.be

Manel Barkallah
Nadi Research Institute
Faculty of Computer Science
University of Namur
Namur, Belgium
manel.barkallah@unamur.be

Jean-Marie Jacquet
Nadi Research Institute
Faculty of Computer Science
University of Namur
Namur, Belgium
jean-marie.jacquet@unamur.be

Process algebras have been widely used to verify security protocols in a formal manner. However they mostly focus on synchronous communication based on the exchange of messages. We present an alternative approach relying on asynchronous communication obtained through information available on a shared space. More precisely this paper first proposes an embedding in Scala of a Linda-like language, called Bach. It consists of a Domain Specific Language, internal to Scala, that allows us to experiment programs developed in Bach while benefiting from the Scala eco-system, in particular from its type system as well as program fragments developed in Scala. Moreover, we introduce a logic that allows to restrict the executions of programs to those meeting logic formulae. Our work is illustrated on the Needham-Schroeder security protocol, for which we manage to automatically rediscover the man-in-the-middle attack first put in evidence by G. Lowe.

1 Introduction

Besides the use of theorem provers, process algebras have been widely used to verify security protocols in a formal manner. A seminal effort in this direction is reported in [19]. There the author illustrates how modeling in CSP [12] and utilizing the FDR tool [10] can be used to produce an attack on the Needham-Schroeder protocol. As another example, the article [2] demonstrates how state reduction techniques can be applied to analyze a model of the Bilateral Key Exchange protocol written in mCRL [6]. In these two cases the models rely on synchronous communication obtained by the exchange of messages. Although this type of communication has been fundamental in the theory of concurrency and has consequently benefited from extensive research support, it is not necessarily intuitive for analyzing security protocols. Indeed, the idea of exchanging messages in a synchronous manner between partners rests on the assumption that the communication takes place instantaneously on agreed actions and thus does not naturally leave room for an intruder to intercept messages. As an evidence at the programming level, in the above two pieces of work, this has lead the authors to duplicate the exchange of messages in their model.

Another path of research has been initiated by Gelernter and Carriero, who advocated in [9] that a clear separation between the interactional and the computational aspects of software components has to take place in order to build interactive distributed systems. Their claim has been supported by the design of a model, Linda [3], originally presented as a set of inter-agent communication primitives which may be added to almost any programming language. Besides process creation, this set includes primitives for adding, deleting, and testing the presence/absence of data in a shared dataspace. In doing so they proposed a new form of synchronization of processes, occurring asynchronously, through the availability or absence of pieces of information on a shared space. A number of other models, now referred to as

C. Aubert, C. Di Giusto, S. Fowler & V. K. I Pun (Eds.):
17th Interaction and Concurrency Experience (ICE 2024)
EPTCS 414, 2024, pp. 58–76, doi:10.4204/EPTCS.414.4

coordination models, have been proposed afterwards. These models seem highly attractive to us because, in practice, message exchanges do not occur atomically through the synchronous communication of actors. Instead, they must happen through a medium – such as a network – which can be easily modeled as a shared space.

The aim of this paper is to explore how coordination models can be used to analyze security protocols. More concretely, we will focus on a specific coordination model, named Bach, will derive a tool, named B2Scala, and will employ it to produce the attack on the Needham-Schroeder protocol [20] first put in evidence by G. Lowe (see [19]).

Implementing coordination models can be done in three different ways. First, as illustrated by Tucson [7], one may provide an implementation as a stand alone language. This has the advantage of offering support for a complete algebra-like incarnation of Linda but at the expense of having to re-implement classical programming constructs that are proposed in conventional languages (like variables, loops, lists, ...). The second approach, illustrated by pSpaces [18] is to provide a set of APIs in a conventional language in order to access the shared space through dedicated functions or methods. This approach benefits from the converse characteristics of the first one: it is easy to access classical programming constructs but the abstract control flow that is offered at a process algebraic level, like non-deterministic choice and parallel composition, is to be coded in an ad hoc manner. Finally, a third approach consists in using a domain specific language embedded inside an existing language. We will turn to this approach since, in principle, it enjoys the benefits of the first two approaches. More specifically, this paper proposes to embody the Bach coordination language inside Scala. In doing so we will profit from the Scala ecosystem while benefiting from all the abstractions offered by the Bach coordination language. A key feature is that we will interpret control flow structures, which we put in good use to restrict computations to those verifying logic formulae. As an interesting consequence, we shall be able to produce the man-in-the-middle attack of the Needham-Schroeder security protocol first put in evidence by G. Lowe.

The rest of the paper is structured as follows. Section 2 presents the Needham-Schroeder use-case as well as the Bach and Scala languages. Section 3 describes the B2Scala tool, both from the point of view of its usage by programmers and from the implementation point of view. A logic is proposed in Section 4 together with its effect on reducing executions. Section 5 illustrates how B2Scala coupled to constraint executions can be used to analyze the Needham-Schroeder protocol. Finally Section 6 draws our conclusions and compares our work with related work.

2 Background

2.1 Use-case: the Needham-Schroeder Protocol

The Needham-Schroeder protocol, developed by Roger Needham and Michael Schroeder in 1978 [20], is a pioneering cryptographic solution aimed at ensuring secure authentication and key distribution within network environments. Its primary objective is to establish a shared session key between two parties, typically referred to as the principal entities, facilitating encrypted communication to safeguard data confidentiality and integrity. The protocol unfolds in a series of steps: initialization, where a client (A) requests access to another client (B) from a trusted server (S), followed by the server's response, which involves authentication, session key generation, and ticket encryption. Subsequently, communication with party B ensues, facilitated by the transmission of the encrypted ticket, along with nonces to ensure freshness. Parties exchange messages encrypted with the session key and incorporate nonces to prevent replay attacks. Mutual authentication is achieved through encrypted messages exchanged between A and B, leveraging the established session key and nonces. Despite its early contributions, the original protocol

exhibited vulnerabilities, notably the reflection attack. In response, refined versions have emerged, such as the Needham-Schroeder-Lowe [19] and Otway-Rees protocols [17].

The description of the Needham-Schroeder public key protocol is often slimmed down to the three following actions:

$$\begin{array}{lcl} Alice \longrightarrow Bob & : & message(na : a)_{pkb} \\ Bob \longrightarrow Alice & : & message(na : nb)_{pka} \\ Alice \longrightarrow Bob & : & message(nb)_{pkb} \end{array}$$

where each transition of the form $X \to Y : m$ represents message m being sent from X to Y. Moreover, the notation m_k represents message m being encrypted with the public key k.

This version assumes that the public keys of Alice and Bob (resp. *pka* and *pkb*) are already known to each other. The full version also involves communication between the parties and a trusted server to obtain the public keys.

In this model, Alice initiates the protocol by sending to Bob her nonce *na* together with her identity *a*, the whole message being encrypted with Bob's public key *pkb*. Bob responds by sending to Alice her nonce *na* together with his nonce *nb*, the whole message being encrypted this time with Alice's public key *pka*. Finally Alice sends to Bob his nonce *nb*, as a proof that a session has been safely made between them. The message is this time encrypted with Bob's public key.

It is worth stressing that, although public keys are known publicly (as the noun suggests), it is only the owners of the corresponding private keys that can decrypt encrypted messages. For instance, the first message sent to Bob can only be decrypted by him.

It is also worth noting that, although sending messages appears as an atomic action in the above description, this is in fact not the case. Messages are transmitted through some medium, say the network, and thus are subject to be read or picked up by opponents. This will be illustrated in Section 5 where a more detailed model will be examined.

2.2 The Bach Coordination Language

Bach [8, 15] is a Linda dialect developped at the University of Namur by the authors. It borrows from Linda the idea of a shared space and reformulates data and the primitives according to the constraint logic programming setting [24]. The following presentation is based on the one of article [1].

2.2.1 Definition of data

According to the logic programming setting, we assume a non-empty set of function names, each one associated with an arity, which indicates the number of arguments the function takes. We assume a non-empty subset of function names associated with an arity 0, namely taking no argument. Such function names are subsequently referred to as *tokens*. Based on their existence, so-called structured pieces of information are introduced inductively as expressions of the form $f(a_1, \cdots, a_n)$ where f is a function name associated with arity n and where arguments $a_1, \ldots, a_n$ are structured pieces of information, understood either as tokens or in the structured form under description. Note that, as the special case where $n = 0$, tokens are considered as being structures information terms. The set of structured pieces of information is subsequently denoted by $\mathscr{I}$. For short, *si-term* is used later to denote a structured piece of information.

Example 1 *The nounces used by Alice and Bob in the Needham-Schroeder protocol are coded by the tokens* `na` *and* `nb`*, respectively. Similarly, their public keys are coded by the tokens* `pka` *and* `pkb`*. A*

$$(\mathbf{T}) \quad \langle\, tell(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t\} \,\rangle \qquad (\mathbf{G}) \quad \langle\, get(t) \mid \sigma \cup \{t\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle$$

$$(\mathbf{A}) \quad \langle\, ask(t) \mid \sigma \cup \{t\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{t\} \,\rangle \qquad (\mathbf{N}) \quad \frac{t \notin \sigma}{\langle\, nask(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 1: Transition rules for the primitives (taken from [1])

$$(\mathbf{S}) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A \,;\, B \mid \sigma \rangle \longrightarrow \langle A' \,;\, B \mid \sigma' \rangle} \qquad (\mathbf{C}) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{c}\langle A \,+\, B \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle \\ \langle B \,+\, A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle\end{array}}$$

$$(\mathbf{P}) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{c}\langle A \,||\, B \mid \sigma \rangle \longrightarrow \langle A' \,||\, B \mid \sigma' \rangle \\ \langle B \,||\, A \mid \sigma \rangle \longrightarrow \langle B \,||\, A' \mid \sigma' \rangle\end{array}} \qquad (\mathbf{Pc}) \quad \frac{P(\overline{x}) = A, \langle A[\overline{x}/\overline{u} \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle P(\overline{u}) \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}$$

Figure 2: Transition rules for the operators (taken from [1])

message encrypted by Alice with Bob's public key and providing Alice's nounce with her identity 'a' is encoded as the following structured piece of information `encrypt(na, a, pkb)`.

2.2.2 Agents

Following the concurrent constraint setting, Linda primitives `out`, `rd` and `in` respectively used to output a tuple, check its presence and consume one occurrence are reformulated as `tell`, `ask`, `get`, acting on si-terms. We add to them a negative counterpart, `nask` checking the absence of a si-term. The execution of these primitives is described by the transition relation defined in Figure 1. The configurations to be considered are pairs of instructions, for the moment reduced to simple primitives, coupled to the contents of the shared space. Following the concurrent constraint setting, the shared space is referred to as the *store*. It is taken as a multiset of si-terms. Moreover, the E symbol is used to denote a terminated computation. Consequently, rule (T) expresses that the execution of the $tell(t)$ primitive always succeeds and add an occurrence of t to the store. Rule (A) requires $ask(t)$ to succeed that t is present on the store. As this primitive just makes a test, the contents of the store is unchanged. According to rule (G), the $get(t)$ primitive acts similarly but remove one occurrence of t. Finally, as specified by rule (N), the primitive $nask(t)$ succeeds in case t is absent from the store.

Primitives are combined to form more complex agents by means of traditional operators from concurrency theory: sequential composition, denoted by the ; symbol, parallel composition, denoted by the || symbol, and non-deterministic choice, denoted by the + symbol.

Procedures are defined by associating an agent with a procedure name possibly coupled to parameters. As usual, we shall assume that the associated agents are guarded, in the sense that the execution of a primitive preceeds any call or can be rewritten in such a form. Procedures are subsequently declared after the `proc` keyword.

The execution of complex agents is defined by the transition rules of Figure 2. Sequential, parallel and choice composition operators are given the convention semantics in rules (S), (P) and (C), respec-

tively. Rule (Pc) dictates that the procedure call $P(\overline{u})$ operates as the agent A that defines P with the formal arguments $\overline{x}$ replaced by the actual ones $\overline{u}$. It is important to note that, in these rules, agents of the form $(E;A)$, $(E \,||\, A)$ and $(A \,||\, E)$ are rewritten as A.

Example 2 *As an example, the behavior of Alice and Bob can be coded as follows:*

```
proc Alice = tell(encrypt(na,a,pkb)); get(encrypt(na,nb,pka));
             tell(encrypt(nb,pkb)).

     Bob = get(encrypt(na,a,pkb)); tell(encrypt(na,nb,pka));
           get(encrypt(nb,pkb)).
```

Note that Alice and Bob only tell messages encrypted with the public key of the other and only get messages encrypted with their public key, which simulates their sole use of their private key.

It is also worth stressing that we will present a model of the Needham-Schroeder protocol and not a concrete implementation. Hence the above tokens (na, nb, ...) are to be understood as globally defined and not as a form of local variables.

2.3 The Scala Programming Language

Scala is a statically typed language known for its concise syntax and seamless fusion of object-oriented and functional programming. Variables can be declared as immutable or mutable, as illustrated by the following code snippet.

```
val immutableVariable: Int = 42
var mutableVariable: String = "Hello, Scala!"
```

Methods are introduced with the *def* keyword, can be generic (with type parameters specified in square brackets), can be written in curried form (with multiple parameter lists) and have a return type which is specified at the end of the signature. Here is a simple example for adding two integers.

```
def add(x: Int, y: Int): Int = x + y
```

Methods are typically included in the definition of objects, classes and traits, which act as interfaces in Java. Of particular interest for the implementation of B2Scala is the definition of *case classes* which are classes that automatically define setter, getter, hash and equal methods.

Two main additional features of Scala are worth stressing.

2.3.1 Functions and objects

Functions may be coded by defining objects with an apply function. For instance, if we define

```
object tell {
    def apply(siterm: SI_Term) = TellAgent(siterm)
}

object Agent {
        def apply(agent: BSC_Agent) = CalledAgent(agent)
}
```

then the evaluation of

```
val P = Agent { tell(f(1,2)) }
```

consists first in evaluating *tell* on the si-term $f(1,2)$, which results in the structure $TellAgent(f(1,2))$, and then in evaluating the function *Agent* on this value, which results in the structure $CalledAgent(TellAgent(f(1,2)))$. It is that result which is assigned to *P*.

2.3.2 Strictness and lazyness

Scala is a strict language that eagerly evaluates expressions. However there are cases in which it is desirable to postpone the evaluation of expressions, for instance to handle recursive definitions of agents. To that end, Scala proposes two basic mechanisms: call-by-name of arguments of functions and so-called thunks. To understand these two concepts, let us modify the `add` function so that it returns the double of its first argument, regardless of the value of the second one:

```
def doubleAdd(x: Int, y: => Int) = x + x
```

The first argument is passed using the call-by-value strategy. It is evaluated whenever the function is called. In contrast, the second argument is passed using the call-by-name strategy. Accordingly, it is evaluated when needed and thus in our example not evaluated at all. However one step further needs to be made to handle recursive expressions that we want to evaluate step by step. In that case, so-called thunks are used. They amount to consider functions requiring no arguments, as in the following definition

```
def myIf[A](cond: Boolean, onTrue: () => A,
                           onFalse: () => A): A = {
    if (cond) onTrue() else onFalse()
}
```

Note that the arguments *onTrue* and *onFalse* are functions taking no arguments and leading to expressions rather than simply expressions.

To conclude this point, it is possible to delay the evaluation of val-declared expression by using the *lazy* keyword, such as in

```
lazy val recursiveExpression = (1+recursiveExpression)*2
```

3 The B2Scala Tool

3.1 Programming interface

To embed Bach in Scala, two main issues must be tackled: on the one hand, how is data declared, and, on the other hand, how are agents declared.

3.1.1 Data

As regards data, the trait *SI_Term* is defined to capture si-terms. Concrete si-terms are then defined as case classes of this trait. For instance in order to manipulate $f(1,2)$ in one of the primitives (tell, ask, ...) the following declaration has to be made:

```
case class f(x:Int, y: Int) extends SI_Term
```

Similarly, tokens can be declared as in

```
case class a() extends SI_Term
```

However that leads to duplicate parentheses everywhere as in $tell(a())$. To avoid that a *Token* class has been defined as a case class of *SI_Term*. It takes as argument a string so that token *a* can be declared as

```
val a = Token(``a'')
```

Accordingly, *a* may now be used without parentheses, as in $tell(a)$.

Example 3 *As examples, the public keys and nonces used in the Needham-Schroeder protocol are declared as the following tokens:*

```
val pka = Token(``pka'')
val pkb = Token(``pkb'')
val na = Token(``na'')
val nb = Token(``nb'')
```

Encrypted messages are coded by the following si-terms:

```
case class encrypt2(n: SI_Term,k: SI_Term) extends SI_Term
case class encrypt3(n: SI_Term,x: SI_Term,k: SI_Term) extends SI_Term
```

Note that Scala does not allow the same name to be used for different case classes. We have thus renamed them according to the number of arguments.

3.1.2 Agents

The main idea for programming agents is to employ constructs of the form

```
val P = Agent { (tell(f(1,2))+tell(g(3))) || (tell(a)+tell(b)) }
```

which encapsulate a Bach agent inside Scala definitions. The *Agent* object is the main ingredient to do so. It is defined as an object with an apply method as follows

```
object Agent {
   def apply(agent: BSC_Agent) = CalledAgent(() => agent)
}
```

It thus consists of a function mapping a *BSC_Agent* into the Scala structure *CalledAgent* taking a thunk, which consists of a function taking no argument and returning an agent. As we saw above, this is needed to treat in a lazy way recursively defined agent.

The *BSC_Agent* type is in fact a trait equipped with the methods needed to parse Bach composed agents. Technically it is defined as follows:

```
trait BSC_Agent { this: BSC_Agent =>
  def *(other: => BSC_Agent) =
          ConcatenationAgent( () => this, other _)
  def ||(other: => BSC_Agent) =
          ParallelAgent( () => this, other _)
  def +(other: => BSC_Agent) =
          ChoiceAgent( () => this, other _)
}
```

As ; is a reserved symbol in Scala, sequential composition is rewritten with the $*$ symbol.

The definition of the composition symbol $*$, $||$ and $+$ employs Scala facility to postfix operations. Using the above definitions, a construct of the form $tell(t)+tell(u)$ is interpreted as the call of method $+$ to $tell(t)$ with argument $tell(u)$.

It is worth observing that the composition operators take agent arguments with call-by-name and deliver structures using thunks, namely functions without arguments to agents.

It will be useful later to generalize choices such that they offer more than two alternatives according to an index ranging over a set, such as in $\sum_{x \in L} ag(x)$ where $ag(x)$ is an agent parameterized by x. This is obtained in B2Scala by the following construct

```
GSum(L, x => ag(x))
```

where `L` is a list.

3.2 Implementation of the Domain Specific Language

The implementation of the domain specific language is based on the same ingredients as those employed in the Scan and Anemone workbenches [13, 14]. They address two main concerns: how is the store implemented and how are agents interpreted.

3.2.1 The store

The store is implemented as a mutable map in Scala. Initially empty, it is enriched for each told structured piece of information by an association of it to a number representing the number of its occurrences on the store. The implementation of the primitives follows directly from this intuition. For instance, the execution of a tell primitive, say `tell(t)`, consists in checking whether `t` is already in the map. If it is then the number of occurrences associated with it is simply incremented by one. Otherwise a new association `(t,1)` is added to the map. Dually, the execution of `get(t)` consists in checking whether `t` is in the map and, in this case, in decrementing by one the number of occurrences. In case one of these two conditions is not met then the get primitive cannot be executed.

3.2.2 Interpretation of agents

Agents are interpreted by repeatedly executing transition steps. This boils down to the definition of function `run_one`, which assumes given an agent in an internal form, namely as a subtype of *BSC_Agent*, and which returns a pair composed of a boolean and an agent in internal form. The boolean aims at specifying whether a transition step has taken place. In this case, the associated agent consists of the agent obtained by the transition step. Otherwise, failure is reported with the given agent as associated agent.

The function is defined inductively on the structure of its argument, say `ag`. If `ag` is a primitive, then the `run_one` function simply consists in executing the primitive on the store. If `ag` is a sequentially composed agent ag_i ; ag_{ii}, then the transition step proceeds by trying to execute the first subagent ag_i. Assume this succeeds and delivers ag' as resulting agent. Then the agent returned is ag' ; ag_{ii} in case ag' is not empty or more simply ag_{ii} in case ag' is empty. Of course, the whole computation fails in case ag_i cannot perform a transition step, namely in case `run_one` applied to ag_i fails.

The case of an agent composed by a parallel or choice operator is more subtle. Indeed for both cases one should not always favor the first or second subagent. To avoid that behavior, we use a boolean variable, randomly assigned to 0 or 1, and depending upon this value we start by evaluating the first or second subagent. In case of failure, we then evaluate the other one and if both fails we report a failure. In case of success for the parallel composition we determine the resulting agent in a similar way to what we did for the sequentially composed agent. For a composition by the choice operator the tried alternative is simply selected.

The computation of a procedure call is performed by computing the defining agent.

4 Constrained executions

The fact that Bach agents are interpreted in the B2Scala tool opens the door to select computations of interest. This is obtained by stating logic formulae to be met.

Two main approaches have been used in concurrency theory to describe properties by means of logic formulae. One approach, exemplified by Linear Temporal Logic (LTL) [22], is based on Kripke structures. In two words, LTL extends classical propositional logic by introducing temporal operators that allow to describe how properties evolve over time. For instance, $X\,\Phi$ means that Φ holds in the next state while $\Phi U \Psi$ specifies that Φ holds until Ψ holds. Central to this approach are, on the one hand, a transition relation between states, indicating which states can be reached from which states, and, on the other hand, a labelling function that assigns to each state a set of atomic propositions that are true in that state.

The other approach is based on labelled transition systems. It is exemplified by the Hennessy-Milner logic (HML) [11]. This logic provides a way to specify properties in terms of actions and capabilities. The two following modalities are the key concepts of HML:

- $<a>\Psi$ means that, by following the labelled transition system, it is possible to make a transition by a such that the resulting process satisfies Ψ
- $[a]\Psi$ means that, whenever a is performed the resulting process satisfies Ψ.

However, since they are finite HML formulae can only describe properties with a finite depth of reasoning. A way to circumvent this problem is to use a generalisation called the μ-calculus [16]. It extends HML with fixed-point operators, such as in $\mu X.(\Phi \vee <a>X)$ which states that there is a path where Φ holds directly or after having repeatedly taken a-transitions.

The logic we use is inspired by these three logics. It is subsequently presented in two steps by describing so-called basic formulae and the bsL-calculus. The effect on computations is then specified. This yields so-called constrainted computations.

4.1 Basic formulae

Similarly to LTL logic, we first specify formulae that are true on states. Obviously, a key concept in our coordination setting is whether a si-term is present on the store under consideration. This is specified by a construct of the form $bf(t)$ which requires that the si-term t is present on the current store. The formal definition is as follows.

Definition 1 *For any si-term t, the formula $bf(t)$ holds on store σ iff $t \in \sigma$. This is subsequently denoted as $\sigma \models bf(t)$. Such formulae are subsequently called bf-formulae.*

As expected, bf-formulae can be combined with the classical logic operators. Formulae built in this way are called *basic formulae*. The formal definition is as follows.

Definition 2 *Basic formulae are the formulae meeting the following grammar:*

$$b ::= bf(t) \mid !b \mid b_1 \vee b_2 \mid b_1 \wedge b_2$$

where $bf(t)$ denotes a bf-formula, b, b_1, b_2 denote basic formulae and the symbols $!$, $\vee$, $\wedge$ respectively express the negation, the disjunction and the conjuction of basic formulae.

The fact that a basic formula f holds on the store σ is defined from the relation $\models$ on bf-formulae according to the traditional truth tables of propositional logic. By extension, this will be subsequently denoted by $\sigma \models f$.

Example 4 *As an example, $bf(i\_running(Alice,Bob))$ is a bf-formula that states that the si-term $i\_running(Alice,Bob)$ is on the store, which can be used to specify that Alice and Bob have initiated a session.*

4.2 The bsL calculus

Similarly to Hennessy-Milner logic and the mu-calculus, we now turn to specify sequences of properties that have to hold on the sequences of stores produced by computations. Obviously, as we want to restrict computations, we have to discard the $[\ldots]$ modality. However we can use the $<\ldots>$ modality in the following manner. Remember that in HML the formula $<a><b>F$ expresses that it is possible to do an a step followed by a b step and reach a process in which F holds. In a similar way, we will express by $bf(a);bf(b)$ the property that it is possible to do a step which leads to $bf(a)$ being true followed by a step after which $bf(b)$ is true. This is for instance performed by the Bach agent $tell(a);tell(b)$. Note that as a reminder of the sequential composition of agents in Bach, we have used the ";" to compose sequentially bf-formulae. As noticed in the above mu-calculus formula, besides sequential composition, we shall also use disjunction to allow the choice between several paths. This leads us to the following grammar where, by analogy to Bach operators, the "+" symbol is used to indicate disjunction.

Definition 3 *BsL-formulae are the formula defined by the following grammar:*

$$f \quad ::= \quad b \mid P \mid f_1 \; + \; f_2 \mid f_1 \; ; \; f_2$$

where b denotes a basic formula, f_1 and f_2 are bsL-formulae and P a variable to be defined by an equation of the form $P = f'$ with f' being a bsL-formula. As usual in concurrency theory, we assume that f' is guarded in the sense that a bf-formula is requested before variable P is called recursively.

Example 5 *As an example, the attack on the Needham-Schroeder protocol may be discovered by finding a computation that obeys the bsL-formula X defined by*

$$X \quad = \quad (\,not(i\_running(Alice,Bob))\;;\;X\,) + r\_commit(Alice,Bob)$$

that is by a computation that does not produce the si-term $i\_running(Alice,Bob)$ and that ends when $r\_commit(Alice,Bob)$ appears on the store. Restated in other terms such a computation never includes the start of a session between Alice and Bob but terminates with Alice and Bob ending the session by committing together.

4.3 Constrained computations

We are now in a position to detail how computations may be constrained by bsL-formulae. Intuitively, if f is a bsL-formula composed of a sequence of basic formulae, a computation c is considered to be constrained by f if the sequence of stores involved in c successively obeys the successive basic formulae in f. This is defined by means of the auxiliary $\vdash$ relation, itself defined by the rules of Figure 3. Intuitively, the notation $\sigma \vdash f\ [f']$ states that a first basic formula of f is satisfied on the store σ and that the remaining formulae of f' need to be satisfied. Accordingly rule (BF) asserts that if the basic formula b

$$(\mathbf{BF}) \quad \frac{\sigma \models b}{\sigma \vdash b\,[\varepsilon]} \qquad (\mathbf{PF}) \quad \frac{P = f, \quad \sigma \vdash f\,[f']}{\sigma \vdash P\,[f']}$$

$$(\mathbf{CF}) \quad \frac{\sigma \vdash f_1\,[f_3]}{\begin{array}{c}\sigma \vdash (f_1 + f_2)\,[f_3] \\ \sigma \vdash (f_2 + f_1)\,[f_3]\end{array}} \qquad (\mathbf{SF}) \quad \frac{\sigma \vdash f_1\,[f_3]}{\sigma \vdash (f_1\,;\,f_2)\,[(f_3\,;\,f_2)]}$$

Figure 3: Transition rules for the $\vdash$ relation

$$(\mathbf{ET}) \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle, \quad \sigma' \vdash f\,[f']}{\langle A@f \mid \sigma\rangle \hookrightarrow \langle A'@f' \mid \sigma'\rangle}$$

Figure 4: Extended transition rule

is satisfied by the store σ then it is also the first formula to be satisfied and nothing remains to be established. The symbol ε is used there to denote an empty sequence of basic formulae. Rule (PF) states that if formula P is defined as f and if a first bf-formula of f is satisfied by σ yielding f' to be satisfied next then so does P with f' to be satisfied next. Finally rules (CF) and (SF) specify the choice and sequential composition of bsL-formulae as one may expect.

Given the $\vdash$ relation, we can define constrained computations by extending the $\rightarrow$ transition relation as the $\hookrightarrow$ relation specified by rule (ET) of Figure 4. Informally this rule states that if, on the one hand, agent A can do a transition from the store σ yielding a new agent A' and a new store σ' and if, on the other hand, a first formula of f is met by σ' yielding f' as a remaining bsL-formula to be established, then agent A can make a constrained transition from store σ and bHM-formula f to agent A' to be computed on store σ' and with respect to bHM-formula f'.

It is worth noting that the encoding in B2Scala is quite easy. On the one hand, bf-formulae are defined similarly to Bach primitives through the `bf` function and are combined as primitives are. On the other hand, bsL formulae are defined by the `bsL` function and recursive definitions are handled in the same way as recursive agents.

The interpretation of agents is then made with respect to a bsL-formula. Basically, a step is allowed by the `run_one` function if one step can be made according to the bsL-formula, as specified by the $\hookrightarrow$ transition relation. This results in a new agent to be solved together with the continuation of the bsL-formula to be satisfied.

5 The Needham-Schroeder protocol in B2Scala

As an application of the B2Scala tool, let us now code the Needham-Schroeder protocol and exhibit a computation that reflects G. Lowe's attack. The interested reader will find the code, the tool and a video of its usage under the web pages of the authors at the addresses mentioned in [21].

Allowing for an attack requires us to introduce an intruder. It is subsequently named Mallory. This being said, the first point to address is to declare nonces and public keys for all the participants of the

protocol, namely Alice, Bob and Mallory. This is achieved by the following token declarations:

```
val na = Token("Alice_nonce")
val nb = Token("Bob_nonce")
val nm = Token("Mallory_nonce")

val pka = Token("Alice_public_key")
val pkb = Token("Bob_public_key")
val pkm = Token("Mallory_public_key")
```

It will also be useful later to refer to the three participants, which can be achieved by means of the following token declarations:

```
val alice   = Token("Alice_as_agent")
val bob     = Token("Bob_as_agent")
val mallory = Token("Mallory_as_intruder")
```

To better view who takes which message produced by whom, encrypted messages introduced in Section 3, are slightly extended as si-terms of the form *message*(*Sender*,*Receiver*,*Encryted_Message*). Moreover, to highlight which message is used in the protocol, we shall subsequently rename encrypted messages as *encrypt_n*, with *n* the number in the sequence of messages. This has the additional advantage of avoiding to overload case classes, which is forbidden in Scala. The following declarations follow.

```
case class encrypt_i(vNonce: SI_Term, vAg: SI_Term,
                     vKey: SI_Term) extends SI_Term
case class encrypt_ii(vNonce: SI_Term, wNonce: SI_Term,
                     vKey: SI_Term) extends SI_Term
case class encrypt_iii(vNonce: SI_Term,
                     vKey: SI_Term) extends SI_Term
case class message(agS: SI_Term, agR: SI_Term,
                     encM: SI_Term) extends SI_Term
```

Finally, si-terms are introduced to indicate with whom Alice and Bob start and close their sessions. They are declared as follows:

```
case class a_running(vAg: SI_Term) extends SI_Term
case class b_running(vAg: SI_Term) extends SI_Term
case class a_commit(vAg: SI_Term) extends SI_Term
case class b_commit(vAg: SI_Term) extends SI_Term
```

We are now in a position to code the behavior of Alice, Bob and Mallory. Coding Alice's behavior follows the description we gave in Example 2 in Section 2. The code is provided in Figure 5. Although Alice wants to send a first encrypted message to Bob, she can just put her message on the network, hoping that it will reach Bob. The network is simulated here by the store, which leaves room to Mallory to intercept it. As a result, the first action is for Alice to start of a session. Hopefully it is with Bob but, to test for a possible attack, we have to take into account the fact that Mallory can take Bob's place. This is coded by offering a choice between Bob and Mallory by the `GSum([bob,mallory], ...)` construct. Calling this actor *Y*, Alice's first action is to tell the initialization of the session with *Y*, thanks to the `a_running(Y)` si-term being told and then to tell the first encrypted message with her nonce, her identity and the public key of Y. The sender and receiver of this message are respectively *Alice* and *Y*. Then Alice waits for a second encrypted message with her nonce and what she hopes to be Bob's nonce, this message being encrypted by her public key. As the second nonce is unknown a new choice is offered with the *WNonce* si-term. Finally, Alice sends the third encrypted message with this nonce, encoded with

```
val Alice = Agent {
  GSum( List(bob,mallory), Y => {
      tell(a_running(Y)) *
      tell( message(alice, Y, encrypt_i(na, alice, public_key(Y))) ) *
      GSum( List(na,nb,nm), WNonce => {
          get( message(Y, alice, encrypt_ii(na,WNonce,pka)) ) *
          tell( message(alice,Y,encrypt_iii(WNonce,public_key(Y))) ) *
          tell( a_commit(Y) )
      })
  })
}
```

Figure 5: Coding of Alice in B2Scala

```
val Bob = Agent {
   GSum( List(alice,mallory), Y => {
       tell(b_running(Y)) *
       GSum( List(alice,mallory), VAg => {
           get( message(Y,bob,encrypt_i(na,VAg,pkb)) ) *
           tell( message(bob,Y,encrypt_ii(na,nb,public_key(VAg))) ) *
           get( message(Y,bob,encrypt_iii(nb,pkb)) ) *
           tell( b_commit(VAg) )
       })
   })
}
```

Figure 6: Coding of Bob in B2Scala

the public key of Y and terminates the session by telling the `a_commit(Y)` si-term. It is worth noting that `public_key(Y)` consists of a call to a Scala function that returns the public key corresponding to the Y argument.

Coding Bob's behavior proceeds in a dual manner. This time the coding has to take into account that Mallory can have taken Alice's place. Hence the first choice `GSum([alice,mallory], ...)` with Y denoting the sender of the message. Moreover, the identity of the agent in the first message being got can be different from Y. A second choice `GSum([alice,mallory], ...)` results from that. The whole agent is given in Figure 6.

As an intruder, Mallory gets and tells messages from Alice and Bob, possibly modifying some parts in case the messages are encrypted with his public key. This applies for the three kinds of message sent/received by Alice and Bob. Figure 7 provides the code for the first message. It presents three `GSum` choices resulting from the three unknown arguments `VNonce`, `VAg`, `VPK` of the message. In all the cases, Bob's attitude is to get the message and to resend it, by modifying the public key if he can decrypt the message, namely if `VPK` is his public key.

To conclude the encoding of the protocol in B2Scala, a bsL-formula F is specified, on the one hand, by excluding a session starting between Bob and Alice and, on the other hand, by requiring the end of the session by Bob with Alice. These two requirements are obtained through the basic formulae `inproper_init` and `end_session`, as specified below:

```
val inproper_init = not( bf(a_running(bob)) or bf(b_running(alice)) )
```

```
lazy val Mallory:BSC_Agent = Agent {

  ( GSum( List(na,nb,nm), VNonce => {
     GSum( List(alice,bob), VAg => {
      GSum( List(pka,pkb,pkm), VPK => {
        get( message(alice,mallory,encrypt_i(VNonce,VAg,VPK)) ) *
        ( if ( VPK == pkm) {
            tell( message(mallory,bob,encrypt_i(VNonce,VAg,pkb)) )
          } else {
            tell( message(mallory,bob,encrypt_i(VNonce,VAg,VPK)) )
          } ) * Mallory
       })
      })
     }) )     + ...
```

Figure 7: Coding of Mallory in B2Scala

```
val end_session = bf(b_commit(alice))
```

Formula F is then coded recursively by requiring F after a step meeting `inproper_init` and by stopping the computation once a step is done that makes `end_session` holds. This is specified as follows.

```
val F = bsL { (inproper_init * F) + end_session }
```

Computations are started by invoking the following Scala instructions

```
val Protocol = Agent { Alice || Bob || Mallory }

val bsc_executor = new BSC_Runner
bsc_executor.execute(Protocol,F)
```

The result is given in Figure 8 in a verbose form in which all the primitives are displayed as Scala objects. As we shall see in a few lines, it produces G. Lowe's attack. To view that, let us reformulate the Scala objects *TellAgent*, *GetAgent* and *BSC_Token* in their corresponding Bach counterparts. The listing of Figure 8 then becomes as follows, where numbers are introduce to facilitate the explanation:

```
(1)  tell(a_running(mallory))
(2)  tell(b_running(mallory))
(3)  tell(message(alice,mallory,encrypt_i(na,alice,pkm)))
(4)  get(message(alice,mallory,encrypt_i(na,alice,pkm)))
(5)  tell(message(mallory,bob,encrypt_i(na,alice,pkb)))
(6)  get(message(mallory,bob,encrypt_i(na,alice,pkb)))
(7)  tell(message(bob,mallory,encrypt_ii(na,nb,pka)))
(8)  get(message(bob,mallory,encrypt_ii(na,nb,pka)))
(9)  tell(message(mallory,alice,encrypt_ii(na,nb,pka)))
(10) get(message(mallory,alice,encrypt_ii(na,nb,pka)))
(11) tell(message(alice,mallory,encrypt_iii(nb,pkm)))
(12) get(message(alice,mallory,encrypt_iii(nb,pkm)))
(13) tell(a_commit(mallory))
(14) tell(message(mallory,bob,encrypt_iii(nb,pkb)))
(15) get(message(mallory,bob,encrypt_iii(nb,pkb)))
(16) tell(b_commit(alice))
```

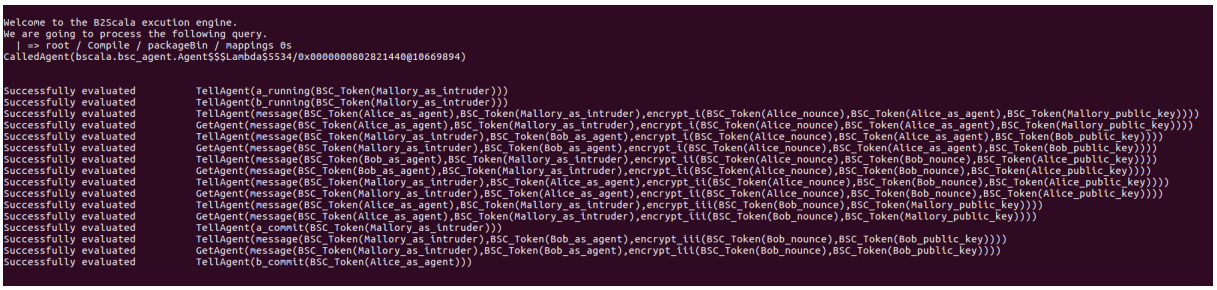


Figure 8: Screenshot of the computation

Lines (1), (2), (13) and (16) evidence that Alice and Bob have actually exchanged messages with Mallory whereas they thought they would speak to each other. In fact Mallory manages to make himself appear as Bob to Alice and as Alice to Bob. Let us abstract from these lines. It is then worth observing that the above listing makes appear tell and get in pairs employing the same message. This corresponds to one actor sending the message to another actor, which is translated in our framework as the first actor telling the message and the second one getting it. By reusing the description of Section 2.1, the listing can then be summed up as follows:

$$
\begin{array}{rcll}
Alice \longrightarrow Mallory & : & message(na : alice)_{pkm} & \text{(lines 3 and 4)} \\
Mallory \longrightarrow Bob & : & message(na : alice)_{pkm} & \text{(lines 5 and 6)} \\
Bob \longrightarrow Mallory & : & message(na : nb)_{pka} & \text{(lines 7 and 8)} \\
Mallory \longrightarrow Alice & : & message(na : nb)_{pka} & \text{(lines 9 and 10)} \\
Alice \longrightarrow Mallory & : & message(nb)_{pkm} & \text{(lines 11 and 12)} \\
Mallory \longrightarrow Bob & : & message(nb)_{pkb} & \text{(lines 14 and 15)}
\end{array}
$$

This is in fact the attack identified by G. Lowe in [19]. It consists essentially in placing Mallory in between Alice and Bob, in having him forward Alice's first message, by changing the public key encrypting the message, in getting Bob's reply and transfer it as such, and finally in forwarding Alice's reply to Bob, again by changing the public key encrypting the message.

Note that a key ingredient for producing the above computation is that imposing `inproper_init` to hold forces the first choice in Alice's code and Bob's code to be made such that Y takes Mallory as value.

6 Conclusion

In the aim of formally verifying security protocols, this paper has proposed an embedding of the coordination language Bach in Scala, in the form of an internal Domain Specific Language, named B2Scala. It has also proposed a logic that allows for constraining executions. The Needham-Schroeder protocol has been modeled with our proposal to illustrate its interest in practice.

The choice for an internal Domain Specific Language has been motivated by the possibility of taking profit from the Scala eco-system, notably its type system, while benefiting from all the abstractions offered by the Bach coordination language. We hope to have convinced the reader of these two features through the coding of the Needham-Schroeder protocol. Indeed, on the one hand, the *BSC_Agents* coding Alice, Bob and Mallory mimick the procedures that would have been written directly in Bach. Moreover the sequential composition operator, the parallel composition operator and the non-deterministic choice

operator have been used as one would have used them in Bach. This feature allows to embed the Bach control flow operators in B2Scala. It is here also worth observing that a similar description could have been written in a pure process algebra setting like the one used in the workbenches Scan and Anemone. However type checking is not supported by these workbenches but is given for free in B2Scala. Moreover, auxiliary concepts like *public_key*(Y) would have been rewritten as mapping functions, with care for completeness of the code left to the programmer while it is provided for free in B2Scala (through completeness verification done by Scala for the match operation).

On the other hand, the code to be written is a real Scala code. Examples of that are the definitions of tokens or si-terms, which are Scala case classes. In that respect, it is worth stressing that arguments of si-terms need to be declared with a type, which is verified at compilation time. Moreover, they can be obtained as the result of a Scala function, as exemplified by the use of *public_key*(Y) in the coding of Alice and Bob (see Figures 5 and 6). It is also to be noted that the *GSum* construct offers a form of local variable, binding the Scala and Bach worlds. Take for instance the first GSum of Figure 5 :

```
GSum( List(bob,mallory), Y => {
      tell(a_running(Y)) * ...
```

There Y plays the role of a local variable which has to be bound to *bob* or *mallory*. Once the value has been decided (by the `run_one` function through the alternative selected for the choice, see Section 3.2.2), it can be used later in the code. Similarly, the second *GSum* construct allows to bind *WNonce* to the value selected by the *get* primitive:

```
GSum( List(na,nb,nm), WNonce => {
      get( message(Y, alice, encrypt_ii(na,WNonce,pka)) ) * ...
```

This being said, our main goal in this paper is to offer a modelling language to describe and reason on systems, such as the Needham-Schroeder protocol, rather than a programming language to code the implementation of the protocol. In these lines, it is worth observing that a direct modelling for analysis purposes would not have been possible in (pure) Scala since we would lack the abstraction offered by process algebras like Bach.

As reported in [5], many coordination languages have been implemented, in some cases as stand alone languages, like Tucson [7], but mostly as API's of conventional languages, accessing tuple spaces through dedicated functions or methods, as in pSpaces [18]. To the best or our knowledge, B2Scala is the first implementation of a coordination language as a Domain Specific Language. We are also not aware of an implementation done in Scala. However, our work is linked to the work on Caos [23], which provides, by using Scala, a generic tool to implement structured operational semantics and to generate intuitive and interactive websites. In practice, one has however to define the semantics of the language under consideration by using Scala. In contrast, we take an opposite approach which already offers an implementation of the Bach constructs and in which programmers need to code Bach-like programs in a Scala manner. Moreover we propose a logic to constraint executions, which is not proposed in [23].

Scafi [4] is another research effort to integrate a coordination language in Scala. It targets a different line of research in the coordination community by being focussed on aggregate computing. Moreover, to the best of our knowledge, no support for constrained executions is proposed.

This work is a continuation of previous work on the Scan and Anemone workbenches [13, 14]. It differs by the fact that both Scan and Anemone interpret directly Bach programs. Moreover the PLTL logic they use is different from the logic proposed in this paper.

As regards the Needham-Schroeder protocol, to our best knowledge, it has been never been modeled in a coordination language, most probably because the Coordination community and the one on security are quite different. Nevertheless it has been modeled in more classical process algebras. In [19] the

author uses CSP and its associated FDR tool to produce an attack on the protocol. This analysis has been complemented in [2] by using the mCRL process algebra and its associated model checker. Our work differs by using a process algebra of a different nature. Indeed the Bach coordination language rests on asynchronous communication which happens by information being available or not on a shared space. This allows to naturally model messages being put on the network as si-terms told on the store. Similarly the action of an intruder is very intuitively modeled by getting si-terms. In contrast, [2] and [19] use synchronous communication which does not naturally introduce the network as a communication medium and which technically forces them to model the intruder by duplicating Alice and Bob's send and receive actions by intercept and fake messages.

Our work open several paths for future research. First the synergy with Scala given by B2Scala offers a natural way of making interfaces much more user friendly than the one of Figure 8. Second we have only investigated the use of B2Scala to analyze the Needham-Schroeder protocol. Our current research aims at exploring the security of other protocols, such as the Quic protocol. Finally, our logic is used to restrict computations at run-time without lookahead strategies, which could lead to select computations that fail later to meet the remaining logic formulae. As a solution to that problem, we are investigating how statistical model checking can be married with B2Scala.

7 Acknowledgment

The authors warmly thank the anonymous reviewers for their insightful comments, which greatly contributed to the improvement of this article. They also thank the University of Namur for its support as well as the Walloon Region for partial support through the Ariac project (convention 210235) and the CyberExcellence project (convention 2110186).

References

[1] M. Barkallah & J.-M. Jacquet (2023): *On the Introduction of Guarded Lists in Bach: Expressiveness, Correctness, and Efficiency Issues*. In C. Aubert, C. Di Giusto, S. Fowler & L. Safina, editors: *Proceedings 16th Interaction and Concurrency Experience (ICE) 2023*, *EPTCS* 383, pp. 55–72, doi:10.4204/EPTCS.383.4.

[2] S. Blom, J.F. Groote, S. Mauw & A. Serebrenik (2004): *Analysing the BKE-security Protocol with μCRL*. In I. Ulidowski, editor: *Proceedings of the 6th AMAST Workshop on Real-Time Systems*, *Electronic Notes in Theoretical Computer Science* 139, Elsevier, pp. 49–90, doi:10.1016/J.ENTCS.2005.09.005.

[3] N. Carriero & D. Gelernter (1989): *Linda in Context*. *Communications of the ACM* 32(4), pp. 444–458, doi:10.1145/63334.63337.

[4] R. Casadei, M. Viroli, G. Aguzzi & D. Pianini (2022): *ScaFi: A Scala DSL and Toolkit for Aggregate Programming*. *SoftwareX* 20, p. 101248, doi:10.1016/j.softx.2022.101248.

[5] G. Ciatto, S. Mariani, G. Di Marzo Serugendo, M. Louvel, A. Omicini & F. Zambonelli (2020): *Twenty Years of Coordination Technologies: COORDINATION Contribution to the State of Art*. *Journal of Logical and Algebraic Methods in Programming* 113, p. 100531, doi:10.1016/j.jlamp.2020.100531.

[6] S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. de Vink, W. Wesselink & T.A.C. Willemse (2013): *An Overview of the mCRL2 Toolset and Its Recent Advances*. In N. Piterman & S.A. Smolka, editors: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 7795, Springer, pp. 199–213, doi:10.1007/978-3-642-36742-7_15.

[7] M. Cremonini, A. Omicini & F. Zambonelli (2000): *Coordination and Access Control in Open Distributed Agent Systems: The TuCSoN Approach*. In A. Porto & G.-C. Roman, editors: *Proceedings of 4th International*

Conference on Coordination Languages and Models, *Lecture Notes in Computer Science* 1906, Springer, pp. 99–114, doi:10.1007/3-540-45263-X_7.

[8] D. Darquennes, J.-M. Jacquet & I. Linden (2018): *On Multiplicities in Tuple-Based Coordination Languages: The Bach Family of Languages and Its Expressiveness Study*. In G. Di Marzo Serugendo & M. Loreti, editors: *Proceedings of the 20th International Conference on Coordination Models and Languages*, *Lecture Notes in Computer Science* 10852, Springer, pp. 81–109, doi:10.1007/978-3-319-92408-3_4.

[9] D. Gelernter & N. Carriero (1992): *Coordination Languages and Their Significance*. *Communications of the ACM* 35(2), pp. 97–107, doi:10.1145/129630.129635.

[10] T. Gibson-Robinson, P. Armstrong, A. Boulgakov & A.W. Roscoe (2014): *FDR3 — A Modern Refinement Checker for CSP*. In E. Abraham & K. Havelund, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 8413, pp. 187–201, doi:10.1007/978-3-642-54862-8_13.

[11] M. Hennessy & R. Milner (1980): *On Observing Nondeterminism and Concurrency*. In J.W. de Bakker & J. van Leeuwen, editors: *Proceedings of the International Conference on Automata, Languages and Programming*, *Lecture Notes in Computer Science* 85, Springer, pp. 299–309, doi:10.1007/3-540-10003-2_79.

[12] C.A.R. Hoare (1985): *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, Prentice Hall.

[13] J.-M. Jacquet & M. Barkallah (2019): *Scan: A Simple Coordination Workbench*. In H. Riis Nielson & E. Tuosto, editors: *Proceedings of the 21st International Conference on Coordination Models and Languages*, *Lecture Notes in Computer Science* 11533, Springer, pp. 75–91, doi:10.1007/978-3-030-22397-7_5.

[14] J.-M. Jacquet & M. Barkallah (2021): *Anemone: A workbench for the Multi-Bach Coordination Language*. *Science of Computer Programming* 202, p. 102579, doi:10.1016/j.scico.2020.102579.

[15] J.-M. Jacquet & I. Linden (2007): *Coordinating Context-aware Applications in Mobile Ad-hoc Networks*. In T. Braun, D. Konstantas, S. Mascolo & M. Wulff, editors: *Proceedings of the first ERCIM workshop on eMobility*, The University of Bern, pp. 107–118.

[16] D. Kozen (1983): *Results on the Propositional mu-Calculus*. *Theoretical Computer Science* 27, pp. 333–354, doi:10.1016/0304-3975(82)90125-6.

[17] K. Liu, J. Ye & Y. Wang (2012): *The Security Analysis on Otway-Rees Protocol Based on BAN Logic*. In: *Proceedings of the Fourth International Conference on Computational and Information Sciences*, pp. 341–344, doi:10.1109/ICCIS.2012.349.

[18] M. Loreti & A. Lluch Lafuente (2024): *Programming with Spaces*. Available at `https://github.com/pSpaces/Programming-with-Spaces/blob/master/README.md`.

[19] G. Lowe (1996): *Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR*. In T. Margaria & B. Steffen, editors: *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, *Lecture Notes in Computer Science* 1055, Springer, pp. 147–166, doi:10.1007/3-540-61042-1_43.

[20] R.M. Needham & M.D. Schroeder (1978): *Using Encryption for Authentication in Large Networks of Computers*. *Communication of the ACM* 21, pp. 993–999, doi:10.1145/359657.359659.

[21] D. Ouardi, M. Barkallah & J.-M. Jacquet (2024): *Coding and Breaking the Needham-Schroeder Protocol using B2Scala*. Available at `https://github.com/UNamurCSFaculty/B2Scala`. Created on February 26th, 2024.

[22] A. Pnueli (1977): *The Temporal Logic of Programs*. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, pp. 46–57, doi:10.1109/SFCS.1977.32.

[23] J. Proença & L. Edixhoven (2023): *Caos: A Reusable Scala Web Animator of Operational Semantics*. In S.-S. Jongmans & A.Lopes, editors: *Proceedings of the 25th International Conference on Coordination Models and Languages*, *Lecture Notes in Computer Science* 13908, Springer, pp. 163–171, doi:10.1007/978-3-031-35361-1_9.

[24] V. Saraswat (1993): *Concurrent Constraint Programming*. ACM Doctoral dissertation awards, MIT Press, doi:10.7551/mitpress/2086.001.0001.