PLACES 2015

Programming Language Approaches to
Communication- and Concurrency-cEntric Software
(8th International Workshop)

An ETAPS Satellite Event

17th April 2015
London, UK

Preliminary Proceedings

Editors: Simon Gay and Jade Alglave

Preface

PLACES 2015 (full title: Programming Language Approaches to Concurrency- and Communication-Centric Software) is the eighth edition of the PLACES workshop series. After the first PLACES, which was affiliated to DisCoTec in 2008, the workshop has been part of ETAPS every year since 2009 and is now an established part of the ETAPS satellite events.

The workshop series was started in order to promote the application of novel programming language ideas to the increasingly important problem of developing software for systems in which concurrency and communication are intrinsic aspects. This includes software for both multi-core systems and large-scale distributed and/or service-oriented systems. The scope of PLACES includes new programming language features, whole new programming language designs, new type systems, new semantic approaches, new program analysis techniques, and new implementation mechanisms.

This year's call for papers attracted 13 submissions of abstracts, of which 9 became submissions of full papers. Each paper was reviewed by three PC members. After discussion, the PC decided to accept all 9 papers. We used EasyChair for the whole process, which, as always, made everything very straightforward.

We are pleased to be able to offer an invited lecture from Martin Vechev from ETH Zurich. The invited talk and the contributed talks together make up an intersting and attractive programme for this year's PLACES, and we are looking forward to a lively and productive meeting.

Finally, we would like to thank the programme committee members for their hard work, and the ETAPS workshop chairs and local organizers for their help.

Simon Gay
Jade Alglave

Programme Committee Co-Chairs

Organisation

Programme Committee Co-Chairs

Simon Gay	University of Glasgow, UK
Jade Alglave	University College London, UK

Programme Committee

Josh Berdine	Microsoft Research Cambridge, UK
Stefan Blom	University of Twente, Netherlands
Nathan Chong	University College London, UK
Ornela Dardha	University of Glasgow, UK
Alexey Gotsman	IMDEA Software Institute, Spain
Hans Hüttel	Aalborg University, Denmark
Paul Keir	Codeplay Software Ltd, UK
Fabrizio Montesi	University of Southern Denmark, Denmark
David Pearce	Victoria University of Wellington, New Zealand
Pierre-Yves Strub	IMDEA Software Institute, Spain
Jules Villard	Imperial College London, UK

Organising Committee

Alastair Beresford	University of Cambridge, UK
Simon Gay	University of Glasgow, UK
Alan Mycroft	University of Cambridge, UK
Vasco Vasconcelos	University of Lisbon, Portugal
Nobuko Yoshida	Imperial College London, UK

Contents

Commutativity Race Detection: Concepts, Algorithms and Open Problems (Invited Lecture)

Martin Vechev

Software Reliability Lab, ETH Zurich, Switzerland

Abstract

In this talk I will introduce the notion of a commutativity race. A commutativity race occurs when two method invocations happen concurrently yet they do not commute (according to a logical specification). Commutativity races are an elegant concept which generalize classic data races and enable reasoning about concurrent interaction at the library interface. I will then present an approach which takes as input a logical specification that captures commutativity and automatically synthesizes an efficient concurrency analyzer for that specification. The resulting analyzers have been used to find concurrency errors in large scale production applications. Finally, generalization of classic data race detection leads to many fundamental research questions, which I will discuss, including: black box specification learning, impossibility of simulating race detectors, connections between logical fragments and asymptotic complexity of the analysis, as well as various practical applications.

Using session types as an effect system

Dominic Orchard and Nobuko Yoshida

Imperial College London

Abstract

Side effects are a core part of practical programming. However, they are often hard to reason about, particularly in a concurrent setting. We propose a foundation for reasoning about concurrent side effects using *sessions*. We show that *session types* are expressive enough to encode an *effect system* for stateful processes. This is formalised via an effect-preserving encoding of a simple imperative language with an effect system into the π-calculus with sessions and session types (into which we encode effect specifications). We demonstrate how this technique can be used to reason about, specify, and control the scope of concurrent side effects.

1 Introduction

Side effects such as input-output and mutation of memory are important features of practical programming. However, effects are often difficult to reason about due to their implicit impact. Reasoning about effects is even more difficult in a concurrent setting, where interference may cause unintended non-determinism. For example, consider a parallel program: $\mathbf{put}\, x\, ((\mathbf{get}\, x) + 2) \mid \mathbf{put}\, x\, ((\mathbf{get}\, x) + 1)$ where x is a mutable memory cell. Given an initial assignment $x \mapsto 0$, the final value stored at x may be any of 3, 2, or 1 since calls to **get** and **put** may be interleaved.

Many approaches to reasoning, specifying, and controlling the scope of effects have therefore been proposed. Seemingly orthogonally, various approaches for specifying and reasoning about concurrent interactions have also been developed. In this paper, we show that two particular approaches for reasoning about effects and concurrency are in fact *non*-orthogonal; one can be embedded into the other. We show that *session types* [9] for concurrent processes are expressive enough to encode *effect systems* [3] for state. We formalise this ability by embedding/encoding a simple imperative language with an effect system into the π-calculus with session types: sessions simulate state and session types become effect annotations. Formally, our embedding maps type-and-effect judgements to session type judgements:

$$\Gamma \vdash M : \tau, F \quad \xrightarrow{embedding} \quad [\![\Gamma]\!];\ \mathit{res} : ![\![\tau]\!].\mathbf{end}, \mathit{eff} : [\![F]\!] \vdash [\![M]\!] \tag{1}$$

That is, an expression M of type τ in context Γ, performing effects F is mapped to a process $[\![M]\!]$ which sends its result over session channel *res* and simulates effects by interactions $[\![F]\!]$ (defined by an interpretation of the effect annotation) over session channel *eff*.

We start with the traditional encoding of a mutable store into the π-calculus (Section 2) and show how its session types provide a kind of effect system. Section 2 introduces a simple imperative language, which we embed into the π-calculus with sessions (sometimes called the *session calculus*) (Section 3). The embedding is shown sound with respect to an equational theory for effects. Section 4 demonstrates how to build upon the encoding to guard against effect interference in a concurrent setting and to safely introduce implicit parallelism.

Our embedding has been formalised in Agda to account for all details and is available at `https://github.com/dorchard/effects-as-sessions` (Appendix B gives a brief description).

The main result of this paper is technical, about the expressive power of the π-calculus with session primitives and session types. This technical result has a number of possible uses:

- *Effects systems for the π-calculus*: rather than adding an additional effect system on top of the π-calculus, we show that existing work on session types can be reused for this purpose.
- *Semantics of concurrency and effects*: our approach provides an intermediate language for the semantics of effects in a concurrent setting. We demonstrate this in Section 4, with a semantics for parallel composition in the source language which avoids race conditions.
- *Compilation*: Related to the above, the session calculus can be used as a typed intermediate language for compilation, where our embedding provides the translation. Section 4 demonstrates an optimisation step where safe implicit parallelism is introduced based on effect information and soundness results of our embedding.

Effect systems have been used before to reason about effects in concurrent programs. For example, Deterministic Parallel Java uses an effect system to check that parallel processes can safely commute without memory races, and otherwise schedules processes to ensure determinism [1]. Our approach allows state effects to be incorporated directly into concurrent protocol descriptions, reusing session types, without requiring interaction between two distinct systems.

2 Simulating state with sessions

2.1 State via processes A well-known way to implement state in a process algebra is to represent a mutable store as a server-like process (often called a *variable agent*) that offers two modes of interaction (get and put). In the get mode, the agent waits to receive a value on its channel which is then "stored"; in the put it sends the stored value. This can be implemented in the π-calculus with branching and recursive definitions as follows (Figure 1 describes the syntax), where *Store* is parameterised by a session channel c and the stored value x:

$$\textbf{def}\ \mathit{Store}(c,x) = c \triangleright \{\mathsf{get} : c!\langle x\rangle.\mathit{Store}\langle c,x\rangle,\ \mathsf{put} : c?(y).\mathit{Store}\langle c,y\rangle,\ \mathsf{stop} : \mathbf{0}\}\ \textbf{in}\ \mathit{Store}\langle \mathit{eff}, i\rangle \quad (2)$$

That is, *Store* provides a choice (by $\triangleright$) over channel c between three behaviours labelled get, put, and stop. The get branch sends the state x on c and then recurses with the same parameters, preserving the stored value. The put branch receives y which then becomes the new state by continuing with recursive call $\mathit{Store}\langle c,y\rangle$. The stop branch provides finite interaction by terminating the agent. The store agent is initialised with channel *eff* and initial value i.

The following parameterised operations *get* and *put* then provide interaction with the store:

$$\mathit{get}(c)(x).P = \overline{c} \triangleleft \mathsf{get}\,.\,\overline{c}?(x).P \qquad \mathit{put}(c)\langle V\rangle.P = \overline{c} \triangleleft \mathsf{put}\,.\,\overline{c}!\langle V\rangle.P \quad (3)$$

where $\overline{c}$ is the opposite endpoint of a channel, and *get* selects (by the $\triangleleft$ operator) the get branch then receives a value which is bound to x in the scope of P, and *put* selects its relevant branch then sends a value V before continuing as P.

A process can then use *get* and *put* for stateful computation by parallel composition with *Store*, *e.g.* $\mathit{get}(\mathit{eff})(x).\mathit{put}(\mathit{eff})\langle x+1\rangle.\textbf{end} \mid \mathit{Store}\langle \mathit{eff}, i\rangle$ increments the initial value.

2.2 Session types Session types provide descriptions (and restrictions) of the interactions that take place over channels [9]. Session types record sequences of typed *send* ($![\tau]$) and *receive* ($?[\tau]$) interactions, terminated by the **end** marker, branched by *select* ($\oplus$) and *choice* ($\&$) interactions, with cycles provided by a fixed point $\mu\alpha$ and session variables α:

$$S,T ::= ![\tau].S \mid ?[\tau].S \mid \oplus[l_1 : S_1,\ldots,l_n : S_n] \mid \&[l_1 : S_1,\ldots,l_n : S_n] \mid \mu\alpha.S \mid \alpha \mid \textbf{end}$$

where τ ranges over value types **nat**, **unit** and session channels S, and l ranges over labels.

$$
\begin{array}{llll}
(\textit{value variables}) & v ::= x, y, z & (\textit{session channel variables}) & c, d, \overline{c}, \overline{d} \\
(\textit{values}) & V ::= C \mid v & & \textit{constants / variables} \\
(\textit{processes}) & P, Q ::= c?(x).P & \mid c!\langle V\rangle.P & \textit{receive / send} \\
& \quad \mid c?(d).P & \mid c!\langle d\rangle.P & \textit{channel receive / send} \\
& \quad \mid c \rhd \{\tilde{l} : \tilde{P}\} & \mid c \lhd l.P & \textit{branching / selection} \\
& \quad \mid \mathbf{def}\, X(\tilde{x}, \tilde{c}) = P \;\mathbf{in}\; Q & \mid X\langle \tilde{V}, \tilde{c}\rangle & \textit{recursive definition / use} \\
& \quad \mid \nu c.P & & \textit{channel restriction} \\
& \quad \mid (P \mid Q) & & \textit{parallel composition} \\
& \quad \mid \mathbf{0} & & \textit{nil process} \\
(\textit{value-types}) & \tau ::= \mathbf{unit} \mid \mathbf{nat} \mid S & (\textit{contexts}) & \Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, X : (\tilde{\tau}, \tilde{S})
\end{array}
$$

(l ranges over labels, $\tilde{l} : \tilde{P}$ over sequences of label-process pairs, and $\tilde{e}$ over syntax sequences)

Figure 1: Syntax of π-calculus with recursion and sessions

Figure 4 (p. 11) gives the rules of the session typing system (based on that in [9]). Session typing judgements for processes have the form $\Gamma; \Delta \vdash P$ meaning a process P has value variables $\Gamma = x_1 : \tau_1 \ldots x_n : \tau_n$ and session-typed channels $\Delta = c_1 : S_1, \ldots c_n : S_n$.

For some state type τ and initial value $i : \tau$, the *Store* process (2) has session judgement:

$$\Gamma; \mathit{eff} : \mu\alpha.\,\&[\mathsf{get} : ![\tau].\alpha, \mathsf{put} : ?[\tau].\alpha, \mathsf{stop} : \mathbf{end}] \vdash \mathit{Store}\langle \mathit{eff}, i\rangle$$

That is, *eff* is a channel over which there is an sequence of offered choice between the get branch, which sends a value, put branch which receives a value, and terminated by the stop branch. The session judgements for *get* and *put* (3) are then:

$$\frac{\Gamma, x : \tau;\ \Delta, \overline{\mathit{eff}} : S \vdash P}{\Gamma; \Delta, \overline{\mathit{eff}} : \oplus[\mathsf{get} : ?[\tau].S] \vdash \mathit{get}(\mathit{eff})(x).P} \qquad \frac{\Gamma; \emptyset \vdash V : \tau \quad \Gamma; \Delta, \overline{\mathit{eff}} : S \vdash P}{\Gamma; \Delta, \overline{\mathit{eff}} : \oplus[\mathsf{put} : ![\tau].S] \vdash \mathit{put}(\mathit{eff})\langle V\rangle.P} \quad (4)$$

We use a variant of session typing where selection $\rhd$, used by *get* and *put*, has a selection session type $\oplus$ with only the selected label (seen above), and not the full range of labels offered by its dual branching process, which would be $\oplus[\mathsf{get} : ?[\tau].S, \mathsf{put} : ![\tau].S, \mathsf{stop} : \mathbf{end}]$ for both. Duality of select and branch types is achieved by using session subtyping to extend select types with extra labels [2] (see Appendix A.1 for details).

2.3 Effect systems Effect systems are a class of static analyses for effects, such as state [3]. Traditionally, effect systems are described as syntax-directed analyses by augmenting typing rules with effect judgements, *i.e.*, $\Gamma \vdash M : \tau, F$ where F describes the effects of M – usually a set of effect tokens. We define the *effect calculus*, a simple imperative language with effectful operations and a type-and-effect system defined in terms of an abstract effect algebra.

Terms comprise variables, **let**-binding, operations, and constants, and types comprise value types for natural numbers and unit:

$$M, N ::= x \mid \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N \mid \mathit{op}\, M \mid c \qquad\qquad \tau, \sigma ::= \mathbf{unit} \mid \mathbf{nat}$$

where x ranges over variables, *op* over unary operations, and c over constants. We do not include function types as there is no abstraction (higher-order calculi are discussed in Section 5). Constants and operations can be effectful and are instantiated to provide application-specific effectful operations in the calculus. As defaults we add zero and unit constants $0, \mathsf{unit} \in c$ and a pure successor operation for natural numbers $\mathsf{suc} \in \mathit{op}$.

$$\text{(var)}\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau, I} \quad \text{(let)}\frac{\Gamma \vdash M : \sigma, F \quad \Gamma, x : \sigma \vdash N : \tau, G}{\Gamma \vdash \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N : \tau, F \bullet G} \quad \text{(const)}\frac{}{\Gamma \vdash c : C_\tau, C_F} \quad \text{(op)}\frac{\Gamma \vdash M : Op_\sigma, I}{\Gamma \vdash op\, M : Op_\tau, Op_F}$$

Figure 2: Type-and-effect system for the effect calculus

Definition 1 (Effect system)**.** Let F be a set of effect annotations with a monoid structure $(F, \bullet, I)$ where $\bullet$ combines effects (corresponding to sequential composition) and I is the trivial effect (for pure computation). Throughout F, G, H will range over effect annotations.

Figure 2 defines the type-and-effect relation. The (var) rule marks variable use as pure (with I). In (let), the left-to-right evaluation order of **let**-binding is exposed by the composition order of the effect F of the bound term M followed by effect G of the **let**-body N. The (const) rule introduces a constant of type C_τ with effects C_F, and (op) applies an operation to its pure argument of type Op_σ, returning a result of type Op_τ with effect Op_F.

2.4 State effects The effect calculus can be instantiated with different notions of effect. For state, we use the effect monoid $(\mathsf{List}\,\{\mathbf{G}\,\tau, \mathbf{P}\,\tau\}, ++, [\,])$ of lists of effect tokens, where $\mathbf{G}$ and $\mathbf{P}$ represent *get* and *put* effects parameterised by a type τ, $++$ concatenates lists and $[\,]$ is the empty list. Many early effect systems annotated terms with sets of effects. Here we use lists to give a more precise account of state which includes the order in which effects occur.

Terms are extended with constant get and unary operation put where $\emptyset \vdash \mathsf{get} : \tau, [\mathbf{G}\,\tau]$ and $\Gamma \vdash \mathsf{put}\, M : \mathbf{unit}, [\mathbf{P}\,\tau]$ for $\Gamma \vdash M : \tau, I$. For example, the following is a valid judgement:

$$\emptyset \vdash \mathbf{let}\, x \leftarrow \mathsf{get}\ \mathbf{in}\ \mathsf{put}\,(\mathsf{suc}\ x)\ : \mathbf{nat}, [\mathbf{G}\,\mathbf{nat}, \mathbf{P}\,\mathbf{nat}] \tag{5}$$

Type safety of the store is enforced by requiring that any *get* effects must have the same type as their nearest preceding *put* effect. We implicitly apply this condition throughout.

2.5 Sessions as effects The session types of processes interacting with *Store* provide the same information as the state effect system. Indeed, we can define a bijection between state effect annotations and the session types of *get* and *put* (4):

$$[\![[\,]]\!] = \mathbf{end} \qquad [\![(\mathbf{G}\,\tau) :: F]\!] = \oplus[\mathsf{get} :?[\tau].[\![F]\!]] \qquad [\![(\mathbf{P}\,\tau) :: F]\!] = \oplus[\mathsf{put} : ![\tau].[\![F]\!]] \tag{6}$$

where :: is the *cons* operator for lists. Thus processes interacting with *Store* have session types corresponding to effect annotations. For example, the following has the same state semantics as (5) and isomorphic session types:

$$\emptyset; \overline{\mathit{eff}} : [\![[\mathbf{G}\,\mathbf{nat}, \mathbf{P}\,\mathbf{nat}]]\!] \vdash \mathit{get}(\mathit{eff})(x).\mathit{put}(\mathit{eff})\langle \mathbf{suc}\, x\rangle \tag{7}$$

3 Embedding the effect calculus into the π-calculus

Our embedding is based on the embedding of the call-by-value λ-calculus (without effects) into the π-calculus [5, 8] taking $\mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N = (\lambda x.N)\, M$. Since effect calculus terms return a result and π-calculus processes do not, the embedding is parameterised by a *result channel* r over which the return value is sent, written $[\![-]\!]_r$. Variables and pure **let**-binding are embedded:

$$[\![\mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N]\!]_r = \ \nu q.\ ([\![M]\!]_q \mid \overline{q}?(x).[\![N]\!]_r) \qquad [\![x]\!]_r = \ r!\langle x\rangle \tag{8}$$

Variables are simply sent over the result channel. For **let**, an intermediate channel q is created over which the result of the bound term M is sent by the left-hand parallel process $\llbracket M \rrbracket_q$ and received and bound to x by the right-hand process before continuing with $\llbracket N \rrbracket_r$. This enforces a left-to-right, CBV evaluation order (despite the parallel composition).

Pure constants and unary operations can be embedded similarly to variables and **let** given suitable value operations in the π-calculus. For example, successor and zero are embedded as:

$$\llbracket \mathsf{suc}\, M \rrbracket_r = \nu q.\ (\llbracket M \rrbracket_q \mid \overline{q}?(x).r!\langle \mathbf{suc}\ x \rangle) \qquad \llbracket \mathsf{zero} \rrbracket_r = r!\langle \mathbf{zero} \rangle \tag{9}$$

Given a mapping $\llbracket - \rrbracket$ from effect calculus types to corresponding value types in the π-calculus, the above embedding of terms (8),(9) can be extended to typing judgements as follows (where $\llbracket \Gamma \rrbracket$ interprets the type of each free-variable assumption, preserving the structure of Γ):

$$\llbracket \Gamma \vdash M : \tau \rrbracket_r = \llbracket \Gamma \rrbracket; r : !\llbracket \tau \rrbracket.\mathbf{end} \vdash \llbracket M \rrbracket_r \tag{10}$$

With effects Effectful computations are embedded by interactions with a side-effect handling agent over a session channel. The embedding, written $\llbracket - \rrbracket_r^{\mathit{eff}}$, maps a judgement $\Gamma \vdash M : \tau,\, F$ to a session type judgement with channels $\Delta = (r : !\llbracket \tau \rrbracket.\mathbf{end},\ \mathit{eff} : \llbracket F \rrbracket)$ *i.e.*, the effect annotation F is interpreted as the session type of channel *eff*. For state, this interpretation is defined as in eq. (6). The embedding first requires an intermediate step, written $\llparenthesis - \rrparenthesis_r^{\mathsf{ei},\mathsf{eo}}$

$$\llparenthesis \Gamma \vdash M : \tau, F \rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} = \forall g.\quad \llbracket \Gamma \rrbracket;\ r : !\llbracket \tau \rrbracket,\ \mathsf{ei} : ?\llbracket F \bullet g \rrbracket,\ \overline{\mathsf{eo}} :!\llbracket g \rrbracket\ \vdash \llparenthesis M \rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} \tag{11}$$

where ei and eo are session channels over which session channels for simulating effects are communicated: ei receives a channel of session type $\llbracket F \bullet g \rrbracket$ (*i.e.*, capable of carrying out effects $F \bullet g$) and $\overline{\mathsf{eo}}$ sends a channel of session type $\llbracket g \rrbracket$ (capable of carrying out effects g). Here the effect g is universally quantified at the meta level. This provides a way to thread a channel for effect interactions through a computation, such as in the case of **let**-binding.

The interpretation is then defined:

$$\begin{aligned}
\llparenthesis \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N \rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} &= \nu q, \mathsf{ea}\,.\ (\llparenthesis M \rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).\llparenthesis N \rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) \\
\llparenthesis x \rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} &= \mathsf{ei}?(c).r!\langle x \rangle.\overline{\mathsf{eo}}!\langle c \rangle \\
\llparenthesis C \rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} &= \mathsf{ei}?(c).\llbracket C \rrbracket_r.\overline{\mathsf{eo}}!\langle c \rangle && \textit{(when C is pure)} \\
\llparenthesis op\, M \rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} &= \mathsf{ei}?(c).\llbracket op\, M \rrbracket_r.\overline{\mathsf{eo}}!\langle c \rangle && \textit{(when op is pure)}
\end{aligned} \tag{12}$$

The embedding of variables is straightforward, where the channel for simulating effects is received on ei and then sent without use on $\overline{\mathsf{eo}}$. Embedding pure operations/constants is similar, reusing the pure embedding defined in equation (9).

The **let** case resembles the pure embedding of **let** but threads through an effect-carrying session channel. Intermediate channel ea is introduced over which the effect channel is passed from the embedding of M to N. Let $\Gamma \vdash M : \tau, F$ and $\Gamma, x : \sigma \vdash N : \tau, G$ then the universally quantified effect variable $\forall g$ for $\llparenthesis M \rrparenthesis_q^{\mathsf{ei},\mathsf{ea}}$ is instantiated to $G \bullet g$. The following partial session-type derivation for the **let** encoding shows the propagation of effects via session types:

$$\dfrac{\dfrac{q :!\llbracket \sigma \rrbracket, \mathsf{ei} :?\llbracket F \bullet G \bullet g \rrbracket, \overline{\mathsf{ea}} :!\llbracket G \bullet g \rrbracket \vdash \llparenthesis M \rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \qquad \overline{q} :?\llbracket \sigma \rrbracket, r :!\llbracket \tau \rrbracket, \mathsf{ea} :?\llbracket G \bullet g \rrbracket, \overline{\mathsf{eo}} :!\llbracket g \rrbracket \vdash \overline{q}?(x).\llparenthesis N \rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}}{r :!\llbracket \tau \rrbracket, \mathsf{ei} :?\llbracket F \bullet G \bullet g \rrbracket, \overline{\mathsf{eo}} :!\llbracket g \rrbracket, q :!\llbracket \sigma \rrbracket, \overline{q} :?\llbracket \sigma \rrbracket, \overline{\mathsf{ea}} :!\llbracket G \bullet g \rrbracket, \mathsf{ea} :?\llbracket G \bullet g \rrbracket \vdash \llparenthesis M \rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).\llparenthesis N \rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}}}{r :!\llbracket \tau \rrbracket, \mathsf{ei} :?\llbracket F \bullet G \bullet g \rrbracket, \overline{\mathsf{eo}} :!\llbracket g \rrbracket \vdash \nu q, \mathsf{ea}.\ (\llparenthesis M \rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).\llparenthesis N \rrparenthesis_r^{\mathsf{ea},\mathsf{eo}})}$$

The *get* and *put* operations of our state effects are embedded similarly to equation (3) (page 4), but with the receiving and sending of the session channel which interacts with the store:

$$\begin{aligned} (\!|\, \mathsf{get} \,|\!)^{\mathsf{ei},\mathsf{eo}}_r &= \mathsf{ei}?(c).c \triangleleft \mathsf{get} \,.\, c?(x).r!\langle x\rangle.\overline{\mathsf{eo}}!\langle c\rangle \\ (\!|\, \mathsf{put}\, M \,|\!)^{\mathsf{ei},\mathsf{eo}}_r &= \nu q.\ (\llbracket M \rrbracket_q \mid\ \mathsf{ei}?(c).\overline{q}?(x).c \triangleleft \mathsf{put} \,.\, c!\langle x\rangle.r!\langle \mathbf{unit}\rangle.\overline{\mathsf{eo}}!\langle c\rangle) \end{aligned} \tag{13}$$

The embedding of get receives channel c over which it performs its effect by selecting the get branch and receiving x which is sent as the result on r before sending c on $\overline{\mathsf{eo}}$. The put embedding is similar to get and **let**, but using the pure embedding $\llbracket M \rrbracket_q$ since M is pure.

The full embedding is then defined in terms of the intermediate as follows:

$$\llbracket \Gamma \vdash M : \tau, F \rrbracket^{\mathit{eff}}_r = \llbracket \Gamma \rrbracket;\ r : !\llbracket \tau \rrbracket, \mathit{eff} : \llbracket F \rrbracket\ \vdash\ \nu \mathsf{ei},\mathsf{eo}.\ ((\!|\, \Gamma \vdash M : \tau, F \,|\!)^{\mathsf{ei},\mathsf{eo}}_r \mid\ \overline{\mathsf{ei}}!\langle \mathit{eff}\rangle.\mathsf{eo}?(c)) \tag{14}$$

where *eff* is the free session channel over which effects are performed.

Finally, the embedded program is composed in parallel with the variable agent, for example:

$$\mathbf{def}\ \mathit{Store}(c,x) = \ldots (\text{see } (2))\ \mathbf{in}\ \mathit{Store}\langle \overline{\mathit{eff}}, 0\rangle \mid \llbracket \mathbf{let}\, x \leftarrow \mathsf{get}\, \mathbf{in}\, \mathsf{put}\,(\mathsf{suc}\, x) \rrbracket^{\mathit{eff}}_r \tag{15}$$

3.1 Soundness The effect calculus exhibits the equational theory defined by the relation $\equiv$ in Figure 3, which enforces monoidal properties on effects and the effect algebra (assoc),(unitL),(unitR), and which allows pure computations to commute with effectful ones (comm). Our embedding is sound with respect to these equations and the weak bisimulation relation of the π-calculus with sessions (see [4] for more on weak bisimulation).

Theorem 1 (Soundness)**.** *If* $\Gamma \vdash M \equiv N : \tau, F$ *then* $\llbracket \Gamma \rrbracket; (r : !\llbracket \tau \rrbracket.\mathbf{end}, e : \llbracket F \rrbracket) \vdash \llbracket M \rrbracket^e_r \approx \llbracket N \rrbracket^e_r$

Appendix C provides the proof. The proof of soundness for (comm) uses the following lemma on the encoding of pure terms (those annotated with I), which requires an additional restriction on the effect algebra.

Lemma 1 (Pure encoding)**.** *An effect system where* $\forall F, G.(F \bullet G \equiv I) \Rightarrow (F \equiv G \equiv I)$ *has the following property of the intermediate encoding, for all* M, Γ, τ*:*

$$(\!|\, \Gamma \vdash M : \tau, I \,|\!)^{\mathsf{ei},\mathsf{eo}}_r\ \approx\ \mathsf{ei}?(c).\overline{\mathsf{eo}}!\langle c\rangle.\llbracket M \rrbracket_r$$

That is, the intermediate encoding of a pure term is bisimilar to a pure encoding (without effect simulation) prefixed by receiving an effect-simulating channel c on ei and sending it on $\overline{\mathsf{eo}}$ without any use. Appendix C provides the proof. The state effect system described here satisfies this additional condition on the effect algebra since any two lists whose concatenation is the empty list implies that both lists are themselves the empty list.

$$(\text{assoc})\ \frac{\Gamma \vdash M : \sigma, F \quad \Gamma, x : \sigma \vdash N : \tau', G \quad \Gamma, y : \tau' \vdash P : \tau, H \quad x \notin FV(P)}{(\mathbf{let}\, y \leftarrow (\mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N)\, \mathbf{in}\, P) \equiv (\mathbf{let}\, x \leftarrow M\, \mathbf{in}\, (\mathbf{let}\, y \leftarrow N\, \mathbf{in}\, P)) : \tau, F \bullet G \bullet H}$$

$$(\text{unitL})\ \frac{\Gamma \vdash x : \sigma, I \quad \Gamma, y : \sigma \vdash M : \tau, F}{\Gamma \vdash (\mathbf{let}\, y \leftarrow x\, \mathbf{in}\, M) \equiv M[x/y] : \tau, F} \qquad (\text{unitR})\ \frac{\Gamma \vdash M : \tau, F}{\Gamma \vdash (\mathbf{let}\, x \leftarrow M\, \mathbf{in}\, x) \equiv M : \tau, F}$$

$$(\text{comm})\ \frac{\Gamma \vdash M : \tau_1, I \qquad \Gamma \vdash N : \tau_2, F \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash P : \tau, G \quad x \notin FV(N) \quad y \notin FV(M)}{\Gamma \vdash \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, (\mathbf{let}\, y \leftarrow N\, \mathbf{in}\, P) \equiv \mathbf{let}\, y \leftarrow N\, \mathbf{in}\, (\mathbf{let}\, x \leftarrow M\, \mathbf{in}\, P) : \tau, F \bullet G}$$

Figure 3: Equations of the effect calculus

4 Discussion

Concurrent effects In a concurrent setting, side effects can lead to non-determinism and race conditions. For example, the program in the introduction $\mathbf{put}\,x\,((\mathbf{get}\,x){+}2)\,|\,\mathbf{put}\,x\,((\mathbf{get}\,x){+}1)$ has four possible final values for x due to arbitrarily interleaved **get** and **put** operations.

Consider an extension to the source language which adds a binary operator for parallel composition $\mid$ (we elide details of the type-and-effect rule, but an additional effect operator, describing parallel effects, might be included). We might then consider the following encoding using the parallel composition of the π-calculus:

$$[\![M \mid N]\!]^{\mathit{eff}}_r = \nu q_1, q_2\,.\,([\![M]\!]^{\mathit{eff}}_{q_1} \mid [\![N]\!]^{\mathit{eff}}_{q_2} \mid \overline{q_1}?(x).\overline{q_2}?(y).r!\langle(x,y)\rangle)$$

where q_1 and q_2 are the result channels for each term, from which the results are paired and sent over r. This encoding is not well-typed under the session typing scheme: the (par) rule (see Figure 4, p. 11) requires that the session channel environments of each process be disjoint but *eff* appears on both sides. Thus, session types naturally prevent effect interference.

Concurrent effects can be allowed by introducing *shared channels*, over which sessions can be initiated [9]. One possible semantics for parallel composition in the source language might be that the store is "locked" by each process, providing atomicity. This can be described by the following redefinition of *Store* and the encoding of parallel composition:

$$\mathbf{def}\;\mathit{Store}(c,x) = \ldots(\mathit{see}(2))\;\mathbf{in\;accept}\;k(c).\mathit{Store}\langle c, 0\rangle$$

$$[\![M \,|\, N]\!]^{k}_r = \nu q_1, q_2.\,(\mathbf{request}\;k(c).[\![M]\!]^{c}_{q_1} \mid \mathbf{request}\;k(d).[\![N]\!]^{d}_{q_2} \mid \overline{q_1}?(x).\overline{q_2}?(y).r!\langle(x,y)\rangle)$$

where k is a shared channel and **request**/**accept** initiate two separate binary sessions between the store process and the client processes, ensuring atomicity of side effects within each process.

Various other kinds of concurrent effect interaction can be described using the rich language of the session calculus and variations of our embedding.

Compiling to the session calculus One use for our embedding is as a typed intermediate language for a compiler since the π-calculus with sessions provides an expressive language for concurrency. For example, even without explicit concurrency in the source language our encoding can be used to introduce implicit parallelism as part of a compilation step via the session calculus. In the case of compiling a term which matches either side of the (comm) rule above, a pure term M can be computed in parallel with N, *i.e.*, given terms $\Gamma \vdash M : \tau_1$, I and $\Gamma \vdash N : \tau_2$, F and $\Gamma, x : \tau_1, y : \tau_2 \vdash P : \tau$, G where $x \notin FV(N), y \notin FV(M)$ then the following specialised encodings can be given:

$$(\!|\,\mathbf{let}\,y \leftarrow N\,\mathbf{in}\,(\mathbf{let}\,x \leftarrow M\,\mathbf{in}\,P)\,|\!)^{\mathsf{ei,eo}}_r =$$
$$(\!|\,\mathbf{let}\,x \leftarrow M\,\mathbf{in}\,(\mathbf{let}\,y \leftarrow N\,\mathbf{in}\,P)\,|\!)^{\mathsf{ei,eo}}_r = \nu\,q, s, \mathsf{eb}.\,([\![M]\!]_q \mid (\!|\,N\,|\!)^{\mathsf{ei,eb}}_s \mid \overline{q}?(x).\overline{s}?(y).(\!|\,P\,|\!)^{\mathsf{eb,eo}}_r)$$

This alternate encoding introduces the opportunity for parallel evaluation of M and N. It is enabled by the effect system (which annotates M with I) and it is sound: it is weakly bisimilar to the usual encoding (which follows from the soundness proof of (comm) in Appendix C and the pure encoding lemma 1).

5 Summary and further work

This paper showed that sessions and session types are expressive enough to encode stateful computations with an effect system. We formalised this via a sound embedding of a simple, and general, effect calculus into the session calculus. Whilst we have focussed on causal state effects, our effect calculus and embedding can also be instantiated for I/O effects, where *input*/*output* operations and effects have a similar form to *get*/*put*. We considered only state effects on a single store, but traditional effect systems account for multiple stores via *regions*. Our approach could be extended with a store and session channel per region. Other instantiations of our effect calculus/embedding are further work, for example, for set-based effects.

Effect reasoning is difficult in higher-order settings as the effects of abstracted computations are locally unknown. Effect systems account for this by annotating function types with the *latent effects* of a function which are delayed till application. A possible encoding of a function type with latent effects into a session type could be following:

$$[\![\sigma \xrightarrow{F} \tau]\!] = ![\![\sigma]\!] \,.\, ![?[\![F \bullet G]\!]] \,.\, ![![\![G]\!]] \,.\, ![![\![\tau]\!]]$$

i.e., a channel over which four things can be sent: a $[\![\sigma]\!]$ value for the function argument, a channel which can receive a further channel capable of simulating effects $F \bullet G$, a channel which can send a channel capable of simulating effects G, and a channel which can send a $[\![\tau]\!]$ for the result. Thus, the encoding of a function receives effect handling channels which have the same form as the effect channels for first-order term encodings. A full, formal treatment of effects in a higher-order setting is forthcoming work.

Effects systems also commonly include a (partial) ordering on effects, which describes how effects can be overapproximated [3]. For example, causal state effects are ordered by prefix inclusion, thus an expression M with judgement $\Gamma \vdash M : \tau$, $[\mathbf{G}\,\tau]$ might have its effects overapproximated (via a subsumption rule) to $\Gamma \vdash M : \tau$, $[\mathbf{G}\,\tau, \mathbf{P}\,\tau']$. It is possible to account for (some) subeffecting using subtyping of sessions. Formalising this is further work.

Whilst we have embedded effects into sessions, the converse seems possible: to embed sessions into effects. Nielson and Nielson previously defined an effect system for higher-order concurrent programs which resembles some aspects of session types [6]. Future work is to explore mutually inverse embeddings of sessions and effects. Relatedly, further work is to explore whether various kinds of *coeffect system* (which dualise effect systems, analysing context and resource use [7]) such as bounded linear logics, can also be embedded into session types.

Acknowledgements Thanks to Tiago Cogumbreiro and the anonymous reviewers for their feedback. The work has been partially sponsored by EPSRC EP/K011715/1, EP/K034413/1, and EP/L00058X/1, and EU project FP7-612985 UpScale.

References

[1] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. *In Proocedings of OOPSLA 2009*, pages 97–116, 2009.

[2] T.-C. Chen, M. Dezani-Ciancaglini, and N. Yoshida. On the preciseness of subtyping in session types. In *PPDP 2014*, pages 146–135. ACM Press, 2014.

[3] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of Conference on LISP and func. prog.*, LFP '86, 1986.

[4] D. Kouzapas, N. Yoshida, R. Hu, and K. Honda. On asynchronous eventful session semantics. *MSCS*. To appear (2015).

[5] R. Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.

[6] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of the symposium on Principles of programming languages*, pages 84–97. ACM, 1994.

[7] T. Petricek, D. A. Orchard, and A. Mycroft. Coeffects: a calculus of context-dependent computation. In *Proceedings of ICFP*, pages 123–135, 2014.

[8] D. Sangiorgi and D. Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[9] N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

A Session types

Figure 4 gives the full session typing system used in this work which is close to that of Yoshida and Vasconcelos [9]. For a session S, its *dual* $\overline{S}$ is defined in the usual way [9]. Throughout we used the usual convention of eliding a trailing **0**, *e.g.*, writing $r!\langle x\rangle$ instead of $r!\langle x\rangle.\mathbf{0}$, and likewise for session types, *e.g.*, $![\tau]$ instead of $![\tau].\mathbf{end}$.

$$\boxed{\Gamma;\Delta \vdash V : \tau} \quad (\textit{value typing}) \quad (\text{const})\ \frac{C : C_\tau}{\Gamma;\emptyset \vdash C : C_\tau} \quad (\text{var})\ \frac{v : \tau \in \Gamma}{\Gamma;\emptyset \vdash v : \tau} \quad (\text{suc})\ \frac{\Gamma \vdash V : \mathbf{nat}}{\Gamma \vdash \mathbf{suc}\, V : \mathbf{nat}}$$

$$\boxed{\Gamma;\Delta \vdash P} \quad (\textit{process typing})$$

$$(\text{end})\ \Gamma;\tilde{c} : \mathbf{end} \vdash \mathbf{0} \quad (\text{par})\ \frac{\Gamma;\Delta_1 \vdash P \quad \Gamma;\Delta_2 \vdash Q}{\Gamma;\Delta_1,\Delta_2 \vdash P \mid Q} \quad (\text{restrict})\ \frac{\Gamma;\Delta, c : S, \overline{c} : \overline{S} \vdash P}{\Gamma;\Delta \vdash \nu c.P}$$

$$(\text{def})\ \frac{\begin{array}{c}\Gamma, X : (\tilde{\tau},\tilde{S}), \tilde{x} : \tilde{\tau};\ \tilde{c} : \tilde{S} \vdash P \\ \Gamma, X : (\tilde{\tau},\tilde{S});\Delta \vdash Q\end{array}}{\Gamma;\Delta \vdash \mathbf{def}\, X(\tilde{x},\tilde{c}) = P\ \mathbf{in}\ Q} \quad (\text{dvar})\ \frac{\Gamma;\tilde{d} : \mathbf{end} \vdash \tilde{e} : \tilde{\tau}}{\Gamma, X : (\tilde{\tau},\tilde{S});\tilde{c} : \tilde{S}, \tilde{d} : \mathbf{end} \vdash X\langle\tilde{e},\tilde{c}\rangle}$$

$$(\text{chan-recv})\ \frac{\Gamma;\Delta, c : T, d : S \vdash P}{\Gamma;\Delta, c : ?[S].T \vdash c?(d).P} \quad (\text{chan-send})\ \frac{\Gamma;\Delta, c : T \vdash P}{\Gamma;\Delta, c : ![S].T, d : S \vdash c!\langle d\rangle.P}$$

$$(\text{recv})\ \frac{\Gamma, x : \tau;\Delta, c : S \vdash P}{\Gamma;\Delta, c : ?[\tau].S \vdash c?(x).P} \quad (\text{send})\ \frac{\Gamma;\emptyset \vdash e : \tau \quad \Gamma;\Delta, c : S \vdash P}{\Gamma;\Delta, c : ![\tau].S \vdash c!\langle e\rangle.P}$$

$$(\text{branch})\ \frac{\Gamma;\Delta, c : S_i \vdash P_i}{\Gamma;\Delta, c : \&[\tilde{l} : \tilde{S}] \vdash c \triangleright \{\tilde{l} : \tilde{P}\}} \quad (\text{select})\ \frac{\Gamma;\Delta, c : S \ \vdash P}{\Gamma;\Delta, c : \oplus[l : S] \vdash c \triangleleft l.P}$$

where $\tilde{x} : \tilde{\tau}$ is shorthand for a sequence of variable-type pairs, and similarly $\tilde{c} : \tilde{S}$ for channels, $\tilde{l} : \tilde{S}$ for labels and sessions, and $\tilde{e}$ for a sequence of expressions.

Figure 4: Session typing relation over the π-calculus with recursion and sessions [9].

A.1 Subtyping and selection

Our session typing system assigns selection types that include only the label l being selected ((select) in Figure 4). Duality with branch types is provided by subtyping on selection types:

$$(\text{sel}) \qquad \oplus[\tilde{l} : \tilde{S}] \prec \oplus[\tilde{l} : \tilde{S}, \tilde{l}' : \tilde{S}']$$

(this is a special case of the usual full subtyping rule for selection, see [2, [SUB-SEL], Table 5, p. 4]). Therefore, for example, the *get* process could be typed:

$$\text{(sub)}\ \dfrac{\dfrac{\Gamma, x : \tau; \Delta; \overline{c} : S \vdash P}{\Gamma; \Delta, \overline{c} : \oplus[\mathsf{get} : ?[\tau].S] \vdash \mathit{get}(c)(x).P} \quad \text{(sel)}\ \dfrac{}{\oplus[\mathsf{get} : ?[\tau].S] \prec \oplus[\mathsf{get} : ?[\tau].S, \mathsf{put} : ![\tau].S]}}{\Gamma; \Delta, \overline{c} : \oplus[\mathsf{get} : ?[\tau].S, \mathsf{put} : ![\tau].S] \vdash \mathit{get}(c)(x).P}$$

However, such subtyping need only be applied when duality is being checked, that is, when opposing endpoints of a channel are bound by channel restriction, $\nu c.P$. We take this approach, thus subtyping is only used with channel restriction such that, prior to restriction, session types can be interpreted as effect annotations with selection types identifying effectful operations.

B Agda encoding

The Agda formalisation of our embedding defines data types of typed terms for the effect calculus `_,_⊢_,_` and session calculus `_*_⊢_`, indexed by the terms' effects, types, and contexts:

```
data_,_⊢_,_ (eff : Effect) : (Gam : Context Type) -> Type -> (Carrier eff) -> Set where ...
data _*_⊢_: (Γ : Context VType) -> (Σ : Context SType) -> (t : PType) -> Set where ...
```

These type constructors are multi-arity infix operators. For the effect calculus type, the first index `eff : Effect` is a record providing the effect algebra, operations, and constants, of which the `Carrier` field holds the type for effect annotations. The embedding is then a function:

```
embed : forall {Γ τ F} -> (e : stEff , Γ ⊢ τ , F)
                       -> (map interpT Γ) * ((Em , [ interpT τ ]!·end) , interpEff F) ⊢ proc
```

where `interpT : Type -> VType` maps types of the effect calculus to value types for sessions, and `interpEff : List StateEff -> SType` maps state effect annotations to session types `SType`. Here the constructor `[_]!·_` is a binary data constructor representing the session type for send. The intermediate embedding has the type (which also uses the receive session type `[_]?·_`):

```
embedInterm : forall {Γ τ F G}
  -> (M : stEff , Γ ⊢ τ , F)
  -> (map interpT Γ * (((Em , [ interpT τ ]!· end),
                          , [ sess (interpEff (F ++ G)) ]?· end)
                          , [ sess (interpEff G) ]!·end) ⊢ proc
```

C Soundness proof of embedding, wrt. Figure 3 equations

Theorem (Soundness). If $\Gamma \vdash M \equiv N : \tau, F$ then $[\![\Gamma]\!]; (r : ![\![\tau]\!].\mathbf{end}, e : [\![F]\!]) \vdash [\![M]\!]^e_r \approx [\![N]\!]^e_r$

Proof Since $[\![M]\!]^e_r = \nu\mathsf{ei},\mathsf{eo}.\ ((\!|\, M\,|\!)^{\mathsf{ei},\mathsf{eo}}_r \mid \overline{\mathsf{ei}}!\langle e\rangle.\mathsf{eo}?(c))$ and $[\![N]\!]^e_r = \nu\mathsf{ei},\mathsf{eo}.\ ((\!|\, N\,|\!)^{\mathsf{ei},\mathsf{eo}}_r \mid \overline{\mathsf{ei}}!\langle e\rangle.\mathsf{eo}?(c))$ (eq. 14) we need only consider $(\!|\, M\,|\!)^{\mathsf{ei},\mathsf{eo}}_r \approx (\!|\, N\,|\!)^{\mathsf{ei},\mathsf{eo}}_r$, *i.e.*, bisimilarity of the intermediate embeddings. We address each equation in turn. The relation $\overset{\text{def}}{=}$ denotes definitional equality based on $(\!|\, - \,|\!)^{\mathsf{ei},\mathsf{eo}}_r$.

(unitR)

$$\begin{aligned}
&(\!|\, \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, x \,|\!)^{\mathsf{ei},\mathsf{eo}}_r \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.\, ((\!|\, M\,|\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(x).\mathsf{ea}?(c).r!\langle x\rangle.\overline{\mathsf{eo}}!\langle c\rangle) \\
&\approx \nu\, \mathsf{ea}.\, ((\!|\, M\,|\!)^{\mathsf{ei},\mathsf{ea}}_r \mid \mathsf{ea}?(c).\overline{\mathsf{eo}}!\langle c\rangle) && \{\text{forwarding } q \to r,\ \beta\text{-reduction}\} \\
&\approx (\!|\, M\,|\!)^{\mathsf{ei},\mathsf{eo}}_r \quad \square && \{\text{forwarding } \mathsf{ea} \to \mathsf{eo},\ \beta\text{-reduction}\}
\end{aligned}$$

(unitL)

$$\begin{aligned}
&\llparenthesis\, \mathbf{let}\, y \leftarrow x \,\mathbf{in}\, M \,\rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.(\llparenthesis\, x \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(y).\llparenthesis\, M \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.(\mathsf{ei}?(c).q!\langle x\rangle.\overline{\mathsf{ea}}!\langle c\rangle \mid \overline{q}?(y).\llparenthesis\, M \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) \\
&\approx \nu\, \mathsf{ea}.(\mathsf{ei}?(c).\overline{\mathsf{ea}}!\langle c\rangle \mid \llparenthesis\, M \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}[x/y]) && \{\beta, \text{structural congruence}\} \\
&\approx \llparenthesis\, M \,\rrparenthesis_r^{\mathsf{ei},\mathsf{eo}}[x/y] && \{\text{forwarding } \mathsf{ei} \to \mathsf{ea}\} \\
&\approx \llparenthesis\, M[x/y] \,\rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} \quad \square && \{\text{var substitution preserved by } \llparenthesis - \rrparenthesis\}
\end{aligned}$$

(assoc)

$$\begin{aligned}
&\llparenthesis\, \mathbf{let}\, y \leftarrow (\mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N) \,\mathbf{in}\, P \,\rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.\,(\llparenthesis\, \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(y).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.\,(\nu\, q1, \mathsf{eb}.\,(\llparenthesis\, M \,\rrparenthesis_{q1}^{\mathsf{ei},\mathsf{eb}} \mid \overline{q1}?(x).\llparenthesis\, N \,\rrparenthesis_q^{\mathsf{eb},\mathsf{ea}}) \mid \overline{q}?(y).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) \\
(*) &\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\,(\llparenthesis\, M \,\rrparenthesis_{q1}^{\mathsf{ei},\mathsf{eb}} \mid \overline{q1}?(x).\llparenthesis\, N \,\rrparenthesis_q^{\mathsf{eb},\mathsf{ea}} \mid \overline{q}?(y).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) && \{\text{structural congruence}\} \\
&\llparenthesis\, \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, (\mathbf{let}\, y \leftarrow N \,\mathbf{in}\, P) \,\rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.\,(\llparenthesis\, M \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).\llparenthesis\, \mathbf{let}\, y \leftarrow N \,\mathbf{in}\, P \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.\,(\llparenthesis\, M \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).\nu\, q1, \mathsf{eb}.\,(\llparenthesis\, N \,\rrparenthesis_{q1}^{\mathsf{ea},\mathsf{eb}} \mid \overline{q1}?(y).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{eb},\mathsf{eo}})) \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\,(\llparenthesis\, M \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).\llparenthesis\, N \,\rrparenthesis_{q1}^{\mathsf{ea},\mathsf{eb}} \mid \overline{q1}?(y).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{eb},\mathsf{eo}}) && \{\text{sequentiality}, x \notin fv(P)\} \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\,(\llparenthesis\, M \,\rrparenthesis_{q1}^{\mathsf{ei},\mathsf{eb}} \mid \overline{q1}?(x).\llparenthesis\, N \,\rrparenthesis_q^{\mathsf{eb},\mathsf{ea}} \mid \overline{q}?(y).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) && \{\alpha, \mathsf{ea} \leftrightarrow \mathsf{eb}, q \leftrightarrow q1\} \\
&\approx (*) \quad \square
\end{aligned}$$

(comm)

$$\begin{aligned}
&\llparenthesis\, \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, (\mathbf{let}\, y \leftarrow N \,\mathbf{in}\, P) \,\rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.\,(\llparenthesis\, M \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).\llparenthesis\, \mathbf{let}\, y \leftarrow N \,\mathbf{in}\, P \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.\,(\llparenthesis\, M \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).\nu\, q1, \mathsf{eb}.\,(\llparenthesis\, N \,\rrparenthesis_{q1}^{\mathsf{ea},\mathsf{eb}} \mid \overline{q1}?(y).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{eb},\mathsf{eo}})) \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\,(\llparenthesis\, M \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \llparenthesis\, N \,\rrparenthesis_{q1}^{\mathsf{ea},\mathsf{eb}} \mid \overline{q}?(x).\overline{q1}?(y).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{eb},\mathsf{eo}}) && \{\text{sequentiality}, x \notin fv(N)\} \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\,(\mathsf{ei}?(c).\overline{\mathsf{ea}}!\langle c\rangle.[\![M]\!]_q \mid \llparenthesis\, N \,\rrparenthesis_{q1}^{\mathsf{ea},\mathsf{eb}} \mid \overline{q}?(x).\overline{q1}?(y).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{eb},\mathsf{eo}}) && \{\text{purity lemma on } M\} \\
(*) &\approx \nu\, q, q1, \mathsf{eb}.\,([\![M]\!]_q \mid \llparenthesis\, N \,\rrparenthesis_{q1}^{\mathsf{ei},\mathsf{eb}} \mid \overline{q}?(x).\overline{q1}?(y).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{eb},\mathsf{eo}}) && \{\text{forwarding } \mathsf{ei} \to \mathsf{ea}\} \\
&\llparenthesis\, \mathbf{let}\, y \leftarrow N \,\mathbf{in}\, (\mathbf{let}\, x \leftarrow M \,\mathbf{in}\, P) \,\rrparenthesis_r^{\mathsf{ei},\mathsf{eo}} \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.\,(\llparenthesis\, N \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(y).\llparenthesis\, \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, P \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.\,(\llparenthesis\, N \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(y).\nu\, q1, \mathsf{eb}.\,(\llparenthesis\, M \,\rrparenthesis_{q1}^{\mathsf{ea},\mathsf{eb}} \mid \overline{q1}?(x).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{eb},\mathsf{eo}})) \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\,(\llparenthesis\, N \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \llparenthesis\, M \,\rrparenthesis_{q1}^{\mathsf{ea},\mathsf{eb}} \mid \overline{q}?(y).\overline{q1}?(x).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{eb},\mathsf{eo}}) && \{\text{sequentiality}, y \notin fv(M)\} \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\,(\llparenthesis\, N \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \mathsf{ea}?(c).\overline{\mathsf{eb}}!\langle c\rangle.[\![M]\!]_{q1} \mid \overline{q}?(y).\overline{q1}?(x).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{eb},\mathsf{eo}}) && \{\text{purity lemma on } M\} \\
&\approx \nu\, q, q1, \mathsf{ea}.\,(\llparenthesis\, N \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid [\![M]\!]_{q1} \mid \overline{q}?(y).\overline{q1}?(x).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) && \{\text{forwarding } \mathsf{ea} \to \mathsf{eb}\} \\
&\approx \nu\, q, q1, \mathsf{ea}.\,([\![M]\!]_{q1} \mid \llparenthesis\, N \,\rrparenthesis_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(y).\overline{q1}?(x).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{ea},\mathsf{eo}}) && \{\text{structural congruence}\} \\
&\approx \nu\, q, q1, \mathsf{eb}.\,([\![M]\!]_q \mid \llparenthesis\, N \,\rrparenthesis_{q1}^{\mathsf{ei},\mathsf{eb}} \mid \overline{q1}?(y).\overline{q}?(x).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{eb},\mathsf{eo}}) && \{\alpha, q \leftrightarrow q1, \mathsf{ea} \leftrightarrow \mathsf{eb}\} \\
&\approx \nu\, q, q1, \mathsf{eb}.\,([\![M]\!]_q \mid \llparenthesis\, N \,\rrparenthesis_{q1}^{\mathsf{ei},\mathsf{eb}} \mid \overline{q}?(x).\overline{q1}?(y).\llparenthesis\, P \,\rrparenthesis_r^{\mathsf{eb},\mathsf{eo}}) && \{\text{reorder recv.}\} \\
&\approx (*) \quad \square
\end{aligned}$$

Lemma (Pure encoding) If an effect system has the property that $\forall F, G.(F \bullet G \equiv I) \Rightarrow (F \equiv G \equiv I)$ then, for all M, Γ, τ it follows that:

$$(\!| \Gamma \vdash M : \tau, I |\!)_r^{\mathsf{ei},\mathsf{eo}} \;\approx\; \mathsf{ei}?(c).\overline{\mathsf{eo}}!\langle c\rangle.[\![M]\!]_r$$

Proof. By induction on the derivation of type-and-effect judgments with a pure effect.

- (var) $\Gamma \vdash v : I$, trivial by the definition of $(\!| - |\!)$.
- (let) $\Gamma \vdash \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N : \tau, I$, the embedding is:

$$(\!| \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N |\!)_r^{\mathsf{ei},\mathsf{eo}} = \nu q, \mathsf{ea}.((\!| M |\!)_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).(\!| N |\!)_r^{\mathsf{ea},\mathsf{eo}})$$

The condition of the lemma on effect systems means $\Gamma \vdash M : \sigma, I$ and $\Gamma, x : \sigma \vdash N : \tau, I$, therefore the inductive hypotheses are that:

$$(\!| M |\!)_q^{\mathsf{ei},\mathsf{ea}} \approx \mathsf{ei}?(c).\overline{\mathsf{ea}}!\langle c\rangle.[\![M]\!]_q \qquad (\!| N |\!)_r^{\mathsf{ea},\mathsf{eo}} \approx \mathsf{ea}?(c).\overline{\mathsf{eo}}!\langle c\rangle.[\![N]\!]_r$$

Therefore:

$$\begin{aligned}
(\!| \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N |\!)_r^{\mathsf{ei},\mathsf{eo}} &= \nu q, \mathsf{ea}.((\!| M |\!)_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).(\!| N |\!)_r^{\mathsf{ea},\mathsf{eo}}) \\
&= \nu q, \mathsf{ea}.(\mathsf{ei}?(c).\overline{\mathsf{ea}}!\langle c\rangle.[\![M]\!]_q \mid \overline{q}?(x).\mathsf{ea}?(c).\overline{\mathsf{eo}}!\langle c\rangle.[\![N]\!]_r) \\
&\approx \nu q.(\mathsf{ei}?(c).([\![M]\!]_q \mid \overline{q}?(x).\overline{\mathsf{eo}}!\langle c\rangle.[\![N]\!]_r)) \\
&\approx \nu q.(\mathsf{ei}?(c).\overline{\mathsf{eo}}!\langle c\rangle.([\![M]\!]_q \mid \overline{q}?(x).[\![N]\!]_r)) \\
&\approx \mathsf{ei}?(c).\overline{\mathsf{eo}}!\langle c\rangle.[\![\mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N]\!]_r
\end{aligned}$$

- (constant) $\Gamma \vdash C : \tau, I$, trivial by definition of $(\!| - |\!)$.
- (op) $\Gamma \vdash op\, M : \tau, I$, where $\Gamma \vdash M : \sigma, I$, trivial by definition of $(\!| - |\!)$.

□

Precise subtyping for synchronous multiparty sessions[*]

Mariangiola Dezani-Ciancaglini[1][†], Silvia Ghilezan[2], Svetlana Jakšić[2], Jovanka Pantović[2], and Nobuko Yoshida[3] [‡]

[1] Università di Torino, Italy
[2] Univerzitet u Novom Sadu, Serbia
[3] Imperial College London

1 Introduction

The notion of subtyping has gained an important role both in theoretical and applicative domains: in lambda and concurrent calculi as well as in programming languages. The soundness and the completeness, together referred to as the preciseness of subtyping, can be considered from two different points of view: denotational and operational. The former preciseness is based on the denotation of a type which is a mathematical object that describes the meaning of the type in accordance with the denotations of other expressions from the language. The latter preciseness has been recently developed with respect to type safety, i.e. the safe replacement of a term of a smaller type when a term of a bigger type is expected. Operational preciseness has been first introduced in [10] for a call-by-value λ-calculus with sum, product and recursive types. Both operational and denotational preciseness have been studied in [5] for a λ-calculus with choice and parallel constructors and in [2] for binary sessions.

Subtyping for session calculi can be defined to assure safety of substitutability of either channels [6] or processes [4]. We claim that substitutability of processes better fits the notion of preciseness.

This paper shows the operational and denotational preciseness of the subtyping introduced in [4] for a synchronous multiparty session calculus [9]. For the denotational preciseness we interpret a type as the set of processes having that type. For the operational preciseness we take the view that well-typed sessions never get stuck.

The most technical challenge is the operational completeness, which requires a non trivial extension of the method used in the case of binary sessions. The core of this extension is the construction of *characteristic global types.* Given a session type T and a session participant p which does not occur in T, the associated characteristic global type expresses the communications prescribed by T between p and the participants in T. After each communication involving p, the characteristic global type creates a cyclic communication between all participants in T. Such a cyclic communication is essential to project the characteristic global type and to generate deadlock when the the subtyping relation is extended.

2 Synchronous Multiparty Session Calculus

This section introduces syntax and semantics of a synchronous multiparty session calculus. Since our focus is on subtyping, we simplify the calculus in [9] eliminating both shared channels

[*]Partly supported by COST IC1201 BETTY and DART bilateral project between Italy and Serbia.

[†]Partly supported by MIUR PRIN Project CINA Prot. 2010LHT4KM and Torino University/Compagnia San Paolo Project SALT.

[‡]Partly supported by EPSRC EP/K011715/1, EP/K034413/1, and EP/L00058X/1, and EU Project FP7-612985 UpScale.

$$\mathsf{succ}(\mathsf{n}) \downarrow (\mathsf{n}+1) \quad \mathtt{neg}(\mathsf{i}) \downarrow (-\mathsf{i}) \quad \neg\mathsf{true} \downarrow \mathsf{false} \quad \neg\mathsf{false} \downarrow \mathsf{true} \quad \mathsf{v} \downarrow \mathsf{v}$$

$$(\mathsf{i}_1 > \mathsf{i}_2) \downarrow \begin{cases} \mathsf{true} & \text{if } \mathsf{i}_1 > \mathsf{i}_2, \\ \mathsf{false} & \text{otherwise} \end{cases} \qquad \frac{\mathsf{e}_1 \downarrow \mathsf{v} \text{ or } \mathsf{e}_2 \downarrow \mathsf{v}}{\mathsf{e}_1 \oplus \mathsf{e}_2 \downarrow \mathsf{v}} \qquad \frac{\mathsf{e} \downarrow \mathsf{v} \quad \mathcal{E}(\mathsf{v}) \downarrow \mathsf{v}'}{\mathcal{E}(\mathsf{e}) \downarrow \mathsf{v}'}$$

Table 1: Expression evaluation.

for session initiations and session channels for communications inside sessions. We conjecture the preciseness of the subtyping in [4] also for the full calculus, but we could not use the present approach for the proof, since well-typed interleaved sessions can be stuck [3].

Syntax A *multiparty session* is a series of interactions between a fixed number of participants, possibly with branching and recursion, and serves as a unit of abstraction for describing communication protocols.

We use the following base sets: *values*, ranged over by $\mathsf{v}, \mathsf{v}', \ldots$; *expressions*, ranged over by $\mathsf{e}, \mathsf{e}', \ldots$; *expression variables*, ranged over by $x, y, z \ldots$; *labels*, ranged over by $\ell, \ell', \ldots$; *session participants*, ranged over by $\mathsf{p}, \mathsf{q}, \ldots$; *process variables*, ranged over by $X, Y, \ldots$; *processes*, ranged over by $P, Q, \ldots$; and *multiparty sessions*, ranged over by $\mathcal{M}, \mathcal{M}', \ldots$.

The values are natural numbers n, integers i, and boolean values true and false. The expressions e are variables or values or expressions built from expressions by applying the operators $\mathtt{succ}, \mathtt{neg}, \neg, \oplus$, or the relation $>$. An *evaluation context* $\mathcal{E}$ is an expression with exactly one hole, built in the same manner from expressions and the hole.

Processes P are defined by:

$$P \quad ::= \quad \mathsf{p}?\ell(x).P \;\mid\; \mathsf{p}!\ell(\mathsf{e}).P \;\mid\; P+P \;\mid\; \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ P \;\mid\; \mu X.P \;\mid\; X \;\mid\; \mathbf{0}$$

The input process $\mathsf{p}?\ell(x).P$ waits for an expression with label ℓ from participant p and the output process $\mathsf{q}!\ell(\mathsf{e}).Q$ sends the value of expression e with label ℓ to participant q. The external choice $P+Q$ offers to choose either P or Q. The process $\mu X.P$ is a recursive process. We take an equi-recursive view, not distinguishing between a process $\mu X.P$ and its unfolding $P\{\mu X.P/X\}$. We assume that the recursive processes are guarded, i.e. $\mu X.X$ is not a process.

A multiparty session $\mathcal{M}$ is a parallel composition of pairs (denoted by $\mathsf{p} \triangleleft P$) of participants and processes:

$$\mathcal{M} \quad ::= \quad \mathsf{p} \triangleleft P \;\mid\; \mathcal{M} \mid \mathcal{M}$$

We will use $\sum_{i \in I} P_i$ as short for $P_1 + \ldots + P_n$, and $\prod_{i \in I} \mathsf{p}_i \triangleleft P_i$ as short for $\mathsf{p}_1 \triangleleft P_1 \mid \ldots \mid \mathsf{p}_n \triangleleft P_n$, where $I = \{1, \ldots, n\}$.

If $\mathsf{p} \triangleleft P$ is well typed (see Table 8), then participant p does not occur in process P, since we do not allow self-communications.

Operational semantics *The value* v *of expression* e (notation $\mathsf{e} \downarrow \mathsf{v}$) is as expected, see Table 1. The successor operation $\mathtt{succ}$ is defined only on natural numbers, the negation $\mathtt{neg}$ is defined on integers (and then also on natural numbers), and $\neg$ is defined only on boolean values. The internal choice $\mathsf{e}_1 \oplus \mathsf{e}_2$ evaluates either to the value of e_1 or to the value of e_2.

The *computational rules of multiparty sessions* (Table 3) are closed with respect to the structural congruence defined in Table 2 and the following reduction contexts:

$$\mathcal{C}[\cdot] ::= [\cdot] \;\mid\; \mathcal{C}[\cdot] \mid \mathcal{M}$$

In rule [R-COMM] participant q sends the value v choosing label ℓ_j to participant p which offers inputs on all labels ℓ_i with $i \in I$.

$$\begin{array}{lll} [\textsc{s-extch 1}] & [\textsc{s-extch 2}] & [\textsc{s-multi}] \\ P+Q \equiv Q+P & (P+Q)+R \equiv P+(Q+R) & P \equiv Q \Rightarrow \mathsf{p} \triangleleft P \equiv \mathsf{p} \triangleleft Q \end{array}$$

$$\begin{array}{lll} [\textsc{s-par 1}] & [\textsc{s-par 2}] & [\textsc{s-par 3}] \\ \mathsf{p} \triangleleft \mathbf{0} \mid \mathcal{M} \equiv \mathcal{M} & \mathcal{M} \mid \mathcal{M}' \equiv \mathcal{M}' \mid \mathcal{M} & (\mathcal{M} \mid \mathcal{M}') \mid \mathcal{M}'' \equiv \mathcal{M} \mid (\mathcal{M}' \mid \mathcal{M}'') \end{array}$$

Table 2: Structural congruence.

[R-COMM]
$$\frac{j \in I \qquad \mathsf{e} \downarrow \mathsf{v}}{\mathsf{p} \triangleleft \sum_{i \in I} \mathsf{q}?\ell_i(x).P_i \;\mid\; \mathsf{q} \triangleleft \mathsf{p}!\ell_j(\mathsf{e}).Q \longrightarrow \mathsf{p} \triangleleft P_j\{\mathsf{v}/x\} \;\mid\; \mathsf{q} \triangleleft Q}$$

[T-CONDITIONAL]
$$\frac{\mathsf{e} \downarrow \mathsf{true}}{\mathsf{p} \triangleleft \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow \mathsf{p} \triangleleft P}$$

[F-CONDITIONAL]
$$\frac{\mathsf{e} \downarrow \mathsf{false}}{\mathsf{p} \triangleleft \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow \mathsf{p} \triangleleft Q}$$

[R-CONTEXT]
$$\frac{\mathcal{M} \longrightarrow \mathcal{M}'}{\mathcal{C}[\mathcal{M}] \longrightarrow \mathcal{C}[\mathcal{M}']}$$

[R-STRUCT]
$$\frac{\mathcal{M}_1' \equiv \mathcal{M}_1 \quad \mathcal{M}_1 \longrightarrow \mathcal{M}_2 \quad \mathcal{M}_2 \equiv \mathcal{M}_2'}{\mathcal{M}_1' \longrightarrow \mathcal{M}_2'}$$

Table 3: Reduction rules.

In order to define the operational preciseness of subtyping it is crucial to formalise when a multiparty session contains communications that will never be executed.

Definition 2.1. *A multiparty session $\mathcal{M}$ is* stuck *if $\mathcal{M} \not\equiv \mathsf{p} \triangleleft \mathbf{0}$ and there is no multiparty session $\mathcal{M}'$ such that $\mathcal{M} \longrightarrow \mathcal{M}'$. A multiparty session $\mathcal{M}$ gets* stuck*, notation* $\mathtt{stuck}(\mathcal{M})$*, if it reduces to a stuck multiparty session.*

3 Type System

This section introduces the type system, which is a simplification of that in [9] due to the new formulation of the calculus.

Types *Sorts* are ranged over by S and defined by: $S \;::=\; \mathtt{nat} \mid \mathtt{int} \mid \mathtt{bool}$
Global types generated by:

$$\mathsf{G} \;::=\; \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i \in I} \quad\mid\quad \mu\mathbf{t}.\mathsf{G} \quad\mid\quad \mathbf{t} \quad\mid\quad \mathtt{end}$$

describe the whole conversation scenarios of multiparty sessions. *Session types* correspond to projections of global types on the individual participants. Inspired by [11], we use intersection and union types instead of standard branching and selection [8] to take advantage from the subtyping induced by subset inclusion. The grammar of session types, ranged over by T, is then

$$\mathsf{T} \;::=\; \bigwedge_{i \in I} \mathsf{p}?\ell_i(S_i).\mathsf{T}_i \quad\mid\quad \bigvee_{i \in I} \mathsf{q}!\ell_i(S_i).\mathsf{T}_i \quad\mid\quad \mu\mathbf{t}.\mathsf{T} \quad\mid\quad \mathbf{t} \quad\mid\quad \mathtt{end}$$

We require that $\ell_i \neq \ell_j$ with $i \neq j$ and $i, j \in I$ and recursion to be guarded in both global and session types. Recursive types are considered modulo fold/unfold. In writing types we omit unnecessary brackets, intersections, unions and $\mathtt{end}$.

We extend the original definition of projection of global types onto participants [8] in the line of [13], but keeping the definition simpler than that of [13]. This generalisation is enough to project the characteristic global types of next Section. We use the partial operator $\bigwedge\!\!\!\bigwedge$ on session types. This operator applied to two identical types gives one of them, applied to two intersection types with same sender and different labels gives their intersection and it is

$$\mathsf{T} \curlywedge \mathsf{T}' = \begin{cases} \mathsf{T} & \text{if } \mathsf{T} = \mathsf{T}', \\ \mathsf{T} \wedge \mathsf{T}' & \text{if } \mathsf{T} = \bigwedge_{i\in I} \mathsf{p}?\ell_i(S_i).\mathsf{T}_i \text{ and } \mathsf{T}' = \bigwedge_{j\in J} \mathsf{p}?\ell'_j(S'_j).\mathsf{T}'_j \\ & \text{ and } \ell_i \neq \ell'_j \text{ for all } i \in I, j \in J \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I} \upharpoonright \mathsf{r} = \begin{cases} \bigvee_{i\in I} \mathsf{q}!\ell_i(S_i).\mathsf{G}_i \upharpoonright \mathsf{r} & \text{if } \mathsf{r} = \mathsf{p}, \\ \bigwedge_{i\in I} \mathsf{p}?\ell_i(S_i).\mathsf{G}_i \upharpoonright \mathsf{r} & \text{if } \mathsf{r} = \mathsf{q}, \\ \curlywedge_{i\in I} \mathsf{G}_i \upharpoonright \mathsf{r} & \text{if } \mathsf{r} \neq \mathsf{p},\ \mathsf{r} \neq \mathsf{q} \text{ and } \curlywedge_{i\in I} \mathsf{G}_i \upharpoonright \mathsf{r} \text{ is defined.} \end{cases}$$

$$\mu\mathbf{t}.\mathsf{G} \upharpoonright \mathsf{r} = \begin{cases} \mathsf{G} \upharpoonright \mathsf{r} & \text{if } \mathsf{r} \text{ occurs in } \mathsf{G}, \\ \mathtt{end} & \text{otherwise.} \end{cases} \qquad \mathbf{t} \upharpoonright \mathsf{r} = \mathbf{t} \qquad \mathtt{end} \upharpoonright \mathsf{r} = \mathtt{end}$$

Table 4: Projection of global types onto participants.

undefined otherwise, see Table 4. The same table gives the *projection* of the global type G onto the participant r, notation $\mathsf{G} \upharpoonright \mathsf{r}$. This projection allows participants to receive different messages in different branches of global types.

Example 3.1. *If* $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\ell_1(\mathtt{nat}).\mathsf{G}_1, \ell_2(\mathtt{bool}).\mathsf{G}_2\}$*, where* $\mathsf{G}_1 = \mathsf{q} \to \mathsf{r} : \ell_3(\mathtt{int})$ *and* $\mathsf{G}_2 = \mathsf{q} \to \mathsf{r} : \ell_4(\mathtt{nat})$*, then*

$$\mathsf{G} \upharpoonright \mathsf{r} = \mathsf{G}_1 \upharpoonright \mathsf{r} \curlywedge \mathsf{G}_2 \upharpoonright \mathsf{r} = \mathsf{q}?\ell_3(\mathtt{int}) \curlywedge \mathsf{q}?\ell_4(\mathtt{nat}) = \mathsf{q}?\ell_3(\mathtt{int}) \wedge \mathsf{q}?\ell_4(\mathtt{nat}).$$

$$[\text{SUB-END}]\ \ \mathtt{end} \leqslant \mathtt{end} \qquad [\text{SUB-IN}]\ \ \frac{\forall i \in I: \quad S'_i \leq: S_i \quad \mathsf{T}_i \leqslant \mathsf{T}'_i}{\bigwedge_{i\in I\cup J} \mathsf{p}?\ell_i(S_i).\mathsf{T}_i \leqslant \bigwedge_{i\in I} \mathsf{p}?\ell_i(S'_i).\mathsf{T}'_i} \qquad [\text{SUB-OUT}]\ \ \frac{\forall i \in I: \quad S_i \leq: S'_i \quad \mathsf{T}_i \leqslant \mathsf{T}'_i}{\bigvee_{i\in I} \mathsf{p}!\ell_i(S_i).\mathsf{T}_i \leqslant \bigvee_{i\in I\cup J} \mathsf{p}!\ell_i(S'_i).\mathsf{T}'_i}$$

Table 5: Subtyping rules.

Subtyping *Subsorting* $\leq:$ on sorts is the minimal reflexive and transitive closure of the relation induced by the rule: $\mathtt{nat} \leq: \mathtt{int}$. *Subtyping* $\leqslant$ on session types takes into account the contra-variance of inputs, the covariance of outputs, and the standard rules for intersection and union. Table 5 gives the subtyping rules: the double line in rules indicates that the rules are interpreted *coinductively* [12, 21.1]. Subtyping can be easily decided, see for example [6]. For reader convenience Table 6 gives the procedure $\mathcal{S}(\Theta, \mathsf{T}, \mathsf{T}')$, where Θ is a set of subtyping judgments. This procedure terminates since unfolding of session types generates regular trees, so Θ cannot grow indefinitely and we have only a finite number of subtyping judgments to consider. Clearly $\mathcal{S}(\emptyset, \mathsf{T}, \mathsf{T}')$ is equivalent to $\mathsf{T} \leqslant \mathsf{T}'$.

$$\mathcal{S}(\Theta, \mathsf{T}, \mathsf{T}') = \begin{cases} \mathsf{true} & \text{if } \mathsf{T} \leqslant \mathsf{T}' \in \Theta \text{ or } \mathsf{T} = \mathsf{T}' \\ \&_{i\in I}\mathcal{S}(\Theta \cup \{\mathsf{T} \leqslant \mathsf{T}'\}, \mathsf{T}_i, \mathsf{T}'_i) & \text{if } (\mathsf{T} = \bigwedge_{i\in I\cup J} \mathsf{p}?\ell_i(S_i).\mathsf{T}_i \text{ and } \mathsf{T}' = \bigwedge_{i\in I} \mathsf{p}?\ell_i(S'_i).\mathsf{T}'_i \\ & \quad \text{and } \forall i \in I : S'_i \leq: S_i) \text{ or} \\ & (\mathsf{T} = \bigvee_{i\in I} \mathsf{p}!\ell_i(S_i).\mathsf{T}_i \text{ and } \mathsf{T}' = \bigvee_{i\in I\cup J} \mathsf{p}!\ell_i(S'_i).\mathsf{T}'_i \\ & \quad \text{and } \forall i \in I : S_i \leq: S'_i) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Table 6: The procedure $\mathcal{S}(\Theta, \mathsf{T}, \mathsf{T}')$.

$$\Gamma \vdash \mathsf{n} : \mathtt{nat} \qquad \Gamma \vdash \mathsf{i} : \mathtt{int} \qquad \Gamma \vdash \mathsf{true} : \mathtt{bool} \qquad \Gamma \vdash \mathsf{false} : \mathtt{bool} \qquad \Gamma, x : S \vdash x : S$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathtt{nat}}{\Gamma \vdash \mathtt{succ}(\mathsf{e}) : \mathtt{nat}} \qquad \frac{\Gamma \vdash \mathsf{e} : \mathtt{int}}{\Gamma \vdash \mathtt{neg}(\mathsf{e}) : \mathtt{int}} \qquad \frac{\Gamma \vdash \mathsf{e} : \mathtt{bool}}{\Gamma \vdash \neg\mathsf{e} : \mathtt{bool}}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : S \quad \Gamma \vdash \mathsf{e}_2 : S}{\Gamma \vdash \mathsf{e}_1 \oplus \mathsf{e}_2 : S} \qquad \frac{\Gamma \vdash \mathsf{e}_1 : \mathtt{int} \quad \Gamma \vdash \mathsf{e}_2 : \mathtt{int}}{\Gamma \vdash \mathsf{e}_1 > \mathsf{e}_2 : \mathtt{bool}} \qquad \frac{\Gamma \vdash \mathsf{e} : S \quad S \leq: S'}{\Gamma \vdash \mathsf{e} : S'}$$

Table 7: Typing rules for expressions.

$$\frac{\Gamma, x : S \vdash P : \mathsf{T}}{\Gamma \vdash \mathsf{q}?\ell(x).P : \mathsf{q}?\ell(S).\mathsf{T}}\,[\text{T-IN}] \qquad \Gamma \vdash \mathbf{0} : \mathtt{end}\ [\text{T-}\mathbf{0}] \qquad \frac{\Gamma \vdash \mathsf{e} : S \quad \Gamma \vdash P : \mathsf{T}}{\Gamma \vdash \mathsf{q}!\ell(\mathsf{e}).P : \mathsf{q}!\ell(S).\mathsf{T}}\,[\text{T-OUT}]$$

$$\frac{\Gamma \vdash P_1 : \mathsf{T}_1 \quad \Gamma \vdash P_2 : \mathsf{T}_2}{\Gamma \vdash P_1 + P_2 : \mathsf{T}_1 \wedge \mathsf{T}_2}\,[\text{T-CHOICE}] \qquad \frac{\Gamma \vdash \mathsf{e} : \mathtt{bool} \quad \Gamma \vdash P_1 : \mathsf{T}_1 \quad \Gamma \vdash P_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : \mathsf{T}_1 \vee \mathsf{T}_2}\,[\text{T-COND}]$$

$$\frac{\Gamma, X : \mathsf{T} \vdash P : \mathsf{T}}{\Gamma \vdash \mu X.P : \mathsf{T}}\,[\text{T-REC}] \qquad \Gamma, X : \mathsf{T} \vdash X : \mathsf{T}\ [\text{T-VAR}] \qquad \frac{\Gamma \vdash P : \mathsf{T} \qquad \mathsf{T} \leqslant \mathsf{T}'}{\Gamma \vdash P : \mathsf{T}'}\,[\text{T-SUB}]$$

$$\frac{\vdash P_1 : \mathsf{T}_1 \ldots \vdash P_n : \mathsf{T}_n \quad \mathtt{coherent}\{(\mathsf{T}_1, \mathsf{p}_1), \ldots, (\mathsf{T}_n, \mathsf{p}_n)\}}{\vdash \mathsf{p}_1 \triangleleft P_1 \ \mid\ \ldots \mid \mathsf{p}_n \triangleleft P_n}\,[\text{T-SESS}]$$

Table 8: Typing rules for processes and sessions.

Typing system We distinguish three kinds of typing judgments

$$\Gamma \vdash \mathsf{e} : S \qquad\qquad \Gamma \vdash P : \mathsf{T} \qquad\qquad \vdash \mathcal{M},$$

where Γ is the environment $\Gamma ::= \emptyset \mid \Gamma, x : S \mid \Gamma, X : \mathsf{T}$ that associates expression variables with sorts and process variables with session types. The typing rules for expressions are standard, see Table 7. Table 8 gives the typing rules for processes and multiparty sessions. Processes are typed as expected, we only notice that the syntax of session types only allows input processes in external choices and output processes in the branches of conditionals. In order to type a session, rule [T-SESS] requires that the processes in parallel can play as participants of a whole communication protocol or the terminated process, i.e. their types are projections of a unique global type. This is assured by the condition $\mathtt{coherent}\{(\mathsf{T}_1, \mathsf{p}_1), \ldots, (\mathsf{T}_n, \mathsf{p}_n)\}$. More precisely we define the set $\mathtt{pt}\{\mathsf{G}\}$ of participants of a global type G as follows:

$$\mathtt{pt}\{\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I}\} = \{\mathsf{p}, \mathsf{q}\} \cup \mathtt{pt}\{\mathsf{G}_i\}\ (i \in I)^1$$
$$\mathtt{pt}\{\mu\mathbf{t}.\mathsf{G}\} = \mathtt{pt}\{\mathsf{G}\} \qquad \mathtt{pt}\{\mathbf{t}\} = \emptyset \qquad \mathtt{pt}\{\mathtt{end}\} = \emptyset$$

and we say that $\{(\mathsf{T}_1, \mathsf{p}_1), \ldots, (\mathsf{T}_n, \mathsf{p}_n)\}$ is *coherent* (notation $\mathtt{coherent}\{(\mathsf{T}_1, \mathsf{p}_1), \ldots, (\mathsf{T}_n, \mathsf{p}_n)\}$) if there is a global type G with $\mathtt{pt}\{\mathsf{G}\} \subseteq \{\mathsf{p}_1, \mathsf{p}_2, \ldots, \mathsf{p}_n\}$ and $\mathsf{T}_i = \mathsf{G} \upharpoonright \mathsf{p}_i$, $i = 1, \ldots, n$. Notice that in this way we can also type sessions containing $\mathsf{p}\triangleleft\mathbf{0}$, a property needed to assure invariance of types under structural congruence.

The proposed type system for multiparty sessions enjoys type preservation under reduction (subject reduction) and the safety property that a typed multiparty session will never get stuck. The remaining of this section is devoted to the proof of these properties.

[1]The projectability of G assures $\mathtt{pt}\{\mathsf{G}_i\} = \mathtt{pt}\{\mathsf{G}_j\}$ for all $i, j \in I$.

As usual we start with an inversion and a substitution lemmas.

Lemma 3.2. (Inversion lemma)

1. *Let* $\Gamma \vdash P : \mathsf{T}$.
   (a) *If* $P = \mathsf{p}?\ell(x).Q$, *then* $\mathsf{p}?\ell(S').\mathsf{T}' \leqslant \mathsf{T}$ *and* $\Gamma, x : S' \vdash Q : \mathsf{T}'$.
   (b) *If* $P = \mathsf{p}!\ell(\mathsf{e}).Q$, *then* $\mathsf{p}!\ell(S').\mathsf{T}' \leqslant \mathsf{T}$ *and* $\Gamma \vdash \mathsf{e} : S'$ *and* $\Gamma \vdash Q : \mathsf{T}'$.
   (c) *If* $P = P_1 + P_2$, *then* $\mathsf{T}_1 \wedge \mathsf{T}_2 \leqslant \mathsf{T}$, $\Gamma \vdash P_1 : \mathsf{T}_1$ *and* $\Gamma \vdash P_2 : \mathsf{T}_2$.
   (d) *If* $P = \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$, *then* $\mathsf{T}_1 \vee \mathsf{T}_2 \leqslant \mathsf{T}$ *and* $\Gamma \vdash P_1 : \mathsf{T}_1$ *and* $\Gamma \vdash P_2 : \mathsf{T}_2$.
   (e) *If* $P = \mu X.Q$, *then* $\Gamma, X : \mathsf{T} \vdash Q : \mathsf{T}$.
   (f) *If* $P = X$, *then* $\Gamma = \Gamma', X : \mathsf{T}'$ *and* $\mathsf{T}' \leqslant \mathsf{T}$.
   (g) *If* $P = \mathbf{0}$, *then* $\mathsf{T} = \mathsf{end}$.
2. *If* $\vdash \mathcal{M}$, *then* $\mathcal{M} = \mathsf{p}_1 \triangleleft P_1 \ \mid\ \dots\ \mid\ \mathsf{p}_n \triangleleft P_n$, *where* $n \geq 0$ *and* $\vdash P_1 : \mathsf{T}_1 \dots \vdash P_n : \mathsf{T}_n$ *and* $\mathtt{coherent}\{(\mathsf{T}_1, \mathsf{p}_1), \dots, (\mathsf{T}_n, \mathsf{p}_n)\}$.

Proof. By induction on type derivations. □

Lemma 3.3. (Substitution lemma) *If* $\Gamma, x : S \vdash P : \mathsf{T}$ *and* $\Gamma \vdash \mathsf{v} : S$, *then* $\Gamma \vdash P\{\mathsf{v}/x\} : \mathsf{T}$.

Proof. By structural induction on P. □

The coherence of a set of pairs (session type, participant) requires that if participant p waits for inputs from participant q, and participant q is ready to send to participant p, then the session types of p and q start with dual communications. This is the content of the following proposition.

Proposition 3.4. *If* $\mathtt{coherent}\{(\bigwedge_{i\in I} \mathsf{q}?\ell_i(S_i).\mathsf{T}_i, \mathsf{p}), (\mathsf{T}, \mathsf{q}), (\mathsf{T}_1, \mathsf{p}_1), \dots, (\mathsf{T}_n, \mathsf{p}_n)\}$ *and* $\mathsf{p}!\ell_j(S).\mathsf{T}' \leqslant \mathsf{T}$, *then* $j \in I$ *and* $\mathsf{T} = \bigvee_{i\in I} \mathsf{p}!\ell_i(S_i).\mathsf{T}'_i$ *and*

$$\mathtt{coherent}\{(\mathsf{T}_i, \mathsf{p}), (\mathsf{T}'_i, \mathsf{q}), (\mathsf{T}_1, \mathsf{p}_1), \dots, (\mathsf{T}_n, \mathsf{p}_n)\}$$

for all $i \in I$.

Proof. By definitions of coherence and projection. □

We can now show subject reduction.

Theorem 3.5. (Subject reduction) *If* $\vdash \mathcal{M}$ *and* $\mathcal{M} \longrightarrow \mathcal{M}'$, *then* $\vdash \mathcal{M}'$.

Proof. By induction on the multiparty session reduction. We only consider the case of rule [R-COMM] as premise of rule [R-CONTEXT]. In this case

$$\mathcal{M} \equiv \mathsf{p} \triangleleft \sum_{i\in I} \mathsf{q}?\ell_i(x).P_i \ \mid\ \mathsf{q} \triangleleft \mathsf{p}!\ell_j(\mathsf{e}).P \mid \prod_{l\in L} \mathsf{p}_l \triangleleft Q_l$$

and
$$\mathcal{M}' \equiv \mathsf{p} \triangleleft P_j\{\mathsf{v}/x\} \ \mid\ \mathsf{q} \triangleleft P \mid \prod_{l\in L} \mathsf{p}_l \triangleleft Q_l,$$

where $j \in I$, $\mathsf{e} \downarrow \mathsf{v}$. By Lemma 3.2(2) $\vdash \mathcal{M}$ implies $\vdash \sum_{i\in I} \mathsf{q}?\ell_i(x).P_i : \mathsf{T}$, and $\vdash \mathsf{p}!\ell_j(\mathsf{e}).P : \mathsf{T}'$, and $\vdash Q_l : \mathsf{T}_l$ for $l \in L$ with $\mathtt{coherent}\{(\mathsf{T}, \mathsf{p}), (\mathsf{T}', \mathsf{q}), (\mathsf{T}_l, \mathsf{p}_l) \mid l \in L\}$. By Lemma 3.2(1c) and (1a) $\bigwedge_{i\in I} \mathsf{q}?\ell_i(S_i).\mathsf{T}'_i \leqslant \mathsf{T}$, which implies $\mathsf{T} = \bigwedge_{i\in I'} \mathsf{q}?\ell_i(S'_i).\mathsf{T}''_i$ with $S'_i \leq: S_i$, $\mathsf{T}'_i \leqslant \mathsf{T}''_i$ for $i \in I'$ and $I' \subseteq I$. By Lemma 3.2(1b) $\mathsf{p}!\ell_j(S').\mathsf{T}'' \leqslant \mathsf{T}'$. Proposition 3.4 implies $j \in I'$ and $\mathsf{T}' = \bigvee_{i\in I'} \mathsf{p}!\ell_i(S'_i).\mathsf{T}'''_i$ and $\mathtt{coherent}\{(\mathsf{T}''_j, \mathsf{p}), (\mathsf{T}'''_j, \mathsf{q}), (\mathsf{T}_l, \mathsf{p}_l) \mid l \in L\}$. By Lemma 3.2(1c) and (1a) we get $x : S_j \vdash P_j : \mathsf{T}'_j$, which implies $x : S'_j \vdash P_j : \mathsf{T}''_j$ by rule [T-SUB] and the last rule of Table 7. From $\mathsf{p}!\ell_j(S').\mathsf{T}'' \leqslant \bigvee_{i\in I'} \mathsf{p}!\ell_i(S'_i).\mathsf{T}'''_i$ we get $S' \leq: S'_j$ and $\mathsf{T}'' \leqslant \mathsf{T}'''_j$. By Lemma 3.2(1b) we get $\vdash \mathsf{e} : S'$ and $\vdash P : \mathsf{T}''$, which implies $\vdash \mathsf{e} : S'_j$ by the last rule of Table 7 and $\vdash P : \mathsf{T}'''_j$ by rule [T-SUB]. Lastly Lemma 3.3 gives $\vdash P_j\{\mathsf{v}/x\} : \mathsf{T}''_j$. We can then derive $\vdash \mathcal{M}'$. □

The safety property that a typed multiparty session will never get stuck is a consequence of subject reduction.

Theorem 3.6. (Safety) *If* $\vdash \mathcal{M}$*, then it does not hold* $\mathtt{stuck}(\mathcal{M})$.

4 Operational Preciseness

We adapt the notion of operational preciseness [10, 2, 5] to our calculus.

Definition 4.1. *A subtyping relation is* operationally precise *if for any two types* T *and* T' *the following equivalence holds:*

$$\mathsf{T} \leqslant \mathsf{T}' \textit{ if and only if there are no } P, \mathsf{p}, \mathcal{M} \textit{ such that:}$$

(1) $\vdash P : \mathsf{T}$*; and* (2) $\vdash Q : \mathsf{T}'$ *implies* $\vdash \mathsf{p} \triangleleft Q \mid \mathcal{M}$*; and* (3) $\mathtt{stuck}(\mathsf{p} \triangleleft P \mid \mathcal{M})$.

The *operational soundness*, i.e. if for all Q such that $\vdash Q : \mathsf{T}'$ implies $\vdash \mathsf{p} \triangleleft Q \mid \mathcal{M}$, then $\mathsf{p} \triangleleft P \mid \mathcal{M}$ is not stuck, follows from the subsumption rule [T-SUB] and the safety theorem, Theorem 3.6.

To show the vice versa, it is handy to define the set $\mathtt{pt}\{\mathsf{T}\}$ of participants of a session type T as follows

$$\mathtt{pt}\{\textstyle\bigwedge_{i\in I} \mathsf{p}?\ell_i(S_i).\mathsf{T}_i\} = \mathtt{pt}\{\bigvee_{i\in I} \mathsf{p}!\ell_i(S_i).\mathsf{T}_i\} = \{\mathsf{p}\} \cup \bigcup_{i\in I} \mathtt{pt}\{\mathsf{T}_i\}$$
$$\mathtt{pt}\{\mu\mathbf{t}.\mathsf{T}\} = \mathtt{pt}\{\mathsf{T}\} \qquad \mathtt{pt}\{\mathbf{t}\} = \mathtt{pt}\{\mathtt{end}\} = \emptyset$$

The proof of *operational completeness* comes in four steps.

- **[Step 1]** We characterise the negation of the subtyping relation by inductive rules (notation $\not\trianglelefteq$).
- **[Step 2]** For each type T and participant $\mathsf{p} \notin \mathtt{pt}\{\mathsf{T}\}$, we define a *characteristic global type* $\mathcal{G}(\mathsf{T}, \mathsf{p})$ such that $\mathcal{G}(\mathsf{T}, \mathsf{p}) \upharpoonright \mathsf{p} = \mathsf{T}$.
- **[Step 3]** For each type T, we define a *characteristic process* $\mathcal{P}(\mathsf{T})$ typed by T, which offers the series of interactions described by T.
- **[Step 4]** We prove that if $\mathsf{T} \not\trianglelefteq \mathsf{T}'$, then $\mathtt{stuck}(\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \prod_{1\leq i\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i))$, where $\mathsf{G} = \mathcal{G}(\mathsf{T}', \mathsf{p})$ and $\mathtt{pt}\{\mathsf{T}'\} = \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$, and $\mathsf{T}_i = \mathsf{G} \upharpoonright \mathsf{p}_i$ for $1 \leq i \leq n$. Hence we achieve completeness by choosing $P = \mathcal{P}(\mathsf{T})$ and $\mathcal{M} = \prod_{1\leq i\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i)$ in the definition of preciseness (Definition 4.1).

Negation of subtyping Table 9 gives the negation of subtyping, which uses the negation of subsorting $\not\leq:$ defined as expected. These rules say that a type different from $\mathtt{end}$ cannot be compared to $\mathtt{end}$, two input or output types with different participants, or different labels, sorts or continuations which do not match cannot be compared. The rules in the last line just take into account the set theoretic properties of intersection and union. One can show that either $\mathsf{T} \leqslant \mathsf{T}'$ or $\mathsf{T} \not\trianglelefteq \mathsf{T}'$ holds for two arbitrary types T, T' by giving a decision algorithm.

$$[\text{NSUB-END-L}]\ \frac{\mathsf{T} \neq \mathtt{end}}{\mathsf{T} \ntrianglelefteq \mathtt{end}} \qquad [\text{NSUB-END-R}]\ \frac{\mathsf{T} \neq \mathtt{end}}{\mathtt{end} \ntrianglelefteq \mathsf{T}} \qquad [\text{NSUB-DIFF-PART}]\ \frac{\mathsf{p} \neq \mathsf{q} \qquad \dagger, \ddagger \in \{?, !\}}{\mathsf{p} \dagger \ell_1(S_1).\mathsf{T}_1 \ntrianglelefteq \mathsf{q} \ddagger \ell_2(S_2).\mathsf{T}_2}$$

$$[\text{NSUB-OUT-IN}]\ \mathsf{p}!\ell_1(S_1).\mathsf{T}_1 \ntrianglelefteq \mathsf{p}?\ell_2(S_2).\mathsf{T}_2 \qquad [\text{NSUB-IN-OUT}]\ \mathsf{p}?\ell_1(S_1).\mathsf{T}_1 \ntrianglelefteq \mathsf{p}!\ell_2(S_2).\mathsf{T}_2$$

$$[\text{NSUB-IN-IN}]\ \frac{\ell_1 \neq \ell_2 \ \text{ or } S_2 \not\leq: S_1 \ \text{ or } \ \mathsf{T}_1 \ntrianglelefteq \mathsf{T}_2}{\mathsf{p}?\ell_1(S_1).\mathsf{T}_1 \ntrianglelefteq \mathsf{p}?\ell_2(S_2).\mathsf{T}_2} \qquad [\text{NSUB-OUT-OUT}]\ \frac{\ell_1 \neq \ell_2 \ \text{ or } S_1 \not\leq: S_2 \ \text{ or } \ \mathsf{T}_1 \ntrianglelefteq \mathsf{T}_2}{\mathsf{p}!\ell_1(S_1).\mathsf{T}_1 \ntrianglelefteq \mathsf{p}!\ell_2(S_2).\mathsf{T}_2}$$

$$[\text{NSUB-INTR}]\ \frac{\mathsf{T} \ntrianglelefteq \mathsf{T}_1 \text{ or } \mathsf{T} \ntrianglelefteq \mathsf{T}_2}{\mathsf{T} \ntrianglelefteq \mathsf{T}_1 \wedge \mathsf{T}_2} \qquad [\text{NSUB-UNIL}]\ \frac{\mathsf{T}_1 \ntrianglelefteq \mathsf{T} \text{ or } \mathsf{T}_2 \ntrianglelefteq \mathsf{T}}{\mathsf{T}_1 \vee \mathsf{T}_2 \ntrianglelefteq \mathsf{T}} \qquad [\text{NSUB-INTR-UNIL}]\ \frac{\forall i \in I\ \forall j \in J\ \mathsf{T}_i \ntrianglelefteq \mathsf{T}_j}{\bigwedge_{i \in I} \mathsf{T}_i \ntrianglelefteq \bigvee_{j \in J} \mathsf{T}_j}$$

Table 9: Negation of subtyping

$$\begin{array}{c}
\mathcal{G}_0(\bigwedge_{i\in I} \mathsf{p}_{j_0}?\ell_i(S_i).\mathsf{T}_i, \mathsf{p}, \{\mathsf{p}_j\}_{1\leq j\leq n}) = \mathsf{p}_{j_0} \to \mathsf{p} : \{\ell_i(S_i).\mathsf{G}_i^{j_0}\}_{i\in I} \\
\mathcal{G}_0(\bigvee_{i\in I} \mathsf{p}_{j_0}!\ell_i(S_i).\mathsf{T}_i, \mathsf{p}, \{\mathsf{p}_j\}_{1\leq j\leq n}) = \mathsf{p} \to \mathsf{p}_{j_0} : \{\ell_i(S_i).\mathsf{G}_i^{j_0}\}_{i\in I} \\
\mathcal{G}_0(\mu \mathbf{t}.\mathsf{T}, \mathsf{p}, \{\mathsf{p}_j\}_{1\leq j\leq n}) = \mu \mathbf{t}.\mathcal{G}_0(\mathsf{T}, \mathsf{p}, \{\mathsf{p}_j\}_{1\leq j\leq n}) \\
\mathcal{G}_0(\mathbf{t}, \mathsf{p}, \{\mathsf{p}_j\}_{1\leq j\leq n}) = \mathbf{t} \qquad \mathcal{G}_0(\mathtt{end}, \mathsf{p}, \{\mathsf{p}_j\}_{1\leq j\leq n}) = \mathtt{end}
\end{array}$$

$$\begin{array}{rcl}
\mathsf{G}_i^{j_0} & = & \mathsf{p}_{j_0} \to \mathsf{p}_{j_0+1} : \ell_i(\mathtt{bool}).\ldots\mathsf{p}_{n-1} \to \mathsf{p}_n : \ell_i(\mathtt{bool}).\mathsf{p}_n \to \mathsf{p}_1 : \ell_i(\mathtt{bool}). \\
 & & \mathsf{p}_1 \to \mathsf{p}_2 : \ell_i(\mathtt{bool}).\ldots.\mathsf{p}_{j_0-1} \to \mathsf{p}_{j_0} : \ell_i(\mathtt{bool}).\mathcal{G}_0(\mathsf{T}_i, \mathsf{p}, \{\mathsf{p}_j\}_{1\leq j\leq n})
\end{array}$$

Table 10: The function $\mathcal{G}_0(\mathsf{T}, \mathsf{p}, \{\mathsf{p}_j\}_{1\leq j\leq n})$.

Lemma 4.2. $\mathsf{T} \ntrianglelefteq \mathsf{T}'$ *is the negation of* $\mathsf{T} \leqslant \mathsf{T}'$.

Proof. We show that either $\mathsf{T} \leqslant \mathsf{T}'$ or $\mathsf{T} \ntrianglelefteq \mathsf{T}'$ holds for two arbitrary types T, T' by giving a decision algorithm. The case in which T, T' are both $\mathtt{end}$ types is immediate. In order to deal with recursion we unfold types every time we reach a μ-binding.
If $\mathsf{T} = \mathsf{p}\dagger\ell(S).\mathsf{T}_0$ and $\mathsf{T}' = \mathsf{q}\ddagger\ell'(S').\mathsf{T}_0'$, then $\mathsf{T} \leqslant \mathsf{T}'$ only if $\mathsf{T}_0 \leqslant \mathsf{T}_0'$ follows from the assumption $\mathsf{T} \leqslant \mathsf{T}'$ and $\mathsf{p} = \mathsf{q}$ and $\ell = \ell'$ and either $\dagger = \ddagger =?$ and $S' \leq: S$, or $\dagger = \ddagger =!$ and $S \leq: S'$. If $\mathsf{p} \neq \mathsf{q}$, then $\mathsf{T} \ntrianglelefteq \mathsf{T}'$ by rule [NSUB-DIFF-PART]. If $\dagger \neq \ddagger$, then $\mathsf{T} \ntrianglelefteq \mathsf{T}'$ by rule [NSUB-OUT-IN] or rule [NSUB-IN-OUT]. If $\ell \neq \ell'$, then $\mathsf{T} \ntrianglelefteq \mathsf{T}'$ by rule [NSUB-IN-IN] or rule [NSUB-OUT-OUT]. If $\dagger = \ddagger =?$ and $S' \not\leq: S$, then $\mathsf{T} \ntrianglelefteq \mathsf{T}'$ by rule [NSUB-IN-IN]. If $\dagger = \ddagger =!$ and $S \not\leq: S'$, then $\mathsf{T} \ntrianglelefteq \mathsf{T}'$ by rule [NSUB-OUT-OUT]. If $\mathsf{T}_0 \ntrianglelefteq \mathsf{T}_0'$, then $\mathsf{T} \ntrianglelefteq \mathsf{T}'$ by rule [NSUB-IN-IN] or rule [NSUB-OUT-OUT].
If $\mathsf{T}' = \mathsf{T}_1 \wedge \mathsf{T}_2$, then $\mathsf{T} \leqslant \mathsf{T}'$ only if $\mathsf{T} \leqslant \mathsf{T}_1$ and $\mathsf{T} \leqslant \mathsf{T}_2$ follow from the assumption $\mathsf{T} \leqslant \mathsf{T}'$. Otherwise $\mathsf{T} \ntrianglelefteq \mathsf{T}_1$ or $\mathsf{T} \ntrianglelefteq \mathsf{T}_2$, then $\mathsf{T} \ntrianglelefteq \mathsf{T}'$ by rule [NSUB-INTR].
If $\mathsf{T} = \mathsf{T}_1 \vee \mathsf{T}_2$, then $\mathsf{T} \leqslant \mathsf{T}'$ only if $\mathsf{T}_1 \leqslant \mathsf{T}'$ and $\mathsf{T}_2 \leqslant \mathsf{T}'$ follow from the assumption $\mathsf{T} \leqslant \mathsf{T}'$. Otherwise $\mathsf{T}_1 \ntrianglelefteq \mathsf{T}$ or $\mathsf{T}_2 \ntrianglelefteq \mathsf{T}$, then $\mathsf{T} \ntrianglelefteq \mathsf{T}'$ by rule [NSUB-UNIL].
If $\mathsf{T} = \bigwedge_{i\in I} \mathsf{T}_i$ and $\mathsf{T}' = \bigvee_{j\in J} \mathsf{T}_j$, then $\mathsf{T} \leqslant \mathsf{T}'$ only if there are $i \in I$ and $j \in J$ such that $\mathsf{T}_i \leq \mathsf{T}_j$ follows from the assumption $\mathsf{T} \leqslant \mathsf{T}'$. This implies that at least one between I and J must be a singleton set, by the syntax of session types. Otherwise $\mathsf{T}_i \ntrianglelefteq \mathsf{T}_j$ for all $i \in I$ and all $j \in J$, then $\mathsf{T} \ntrianglelefteq \mathsf{T}'$ by rule [NSUB-INTR-UNIL]. ☐

Characteristic global types The characteristic global type $\mathcal{G}(\mathsf{T}, \mathsf{p})$ of the type T for the participant p describes the communications between p and all participants in $\mathtt{pt}\{\mathsf{T}\}$ following

$$\mathcal{P}(\mathsf{T}) = \begin{cases} \mathsf{p}?\ell(x).\mathsf{if}\ \mathtt{succ}(x) > 0\ \mathsf{then}\ \mathcal{P}(\mathsf{T}')\ \mathsf{else}\ \mathcal{P}(\mathsf{T}') & \text{if } \mathsf{T} = \mathsf{p}?\ell(\mathtt{nat}).\mathsf{T}', \\ \mathsf{p}?\ell(x).\mathsf{if}\ \mathtt{neg}(x) > 0\ \mathsf{then}\ \mathcal{P}(\mathsf{T}')\ \mathsf{else}\ \mathcal{P}(\mathsf{T}') & \text{if } \mathsf{T} = \mathsf{p}?\ell(\mathtt{int}).\mathsf{T}', \\ \mathsf{p}?\ell(x).\mathsf{if}\ \neg x\ \mathsf{then}\ \mathcal{P}(\mathsf{T}')\ \mathsf{else}\ \mathcal{P}(\mathsf{T}') & \text{if } \mathsf{T} = \mathsf{p}?\ell(\mathtt{bool}).\mathsf{T}', \\ \mathsf{p}!\ell(5).\mathcal{P}(\mathsf{T}') & \text{if } \mathsf{T} = \mathsf{p}!\ell(\mathtt{nat}).\mathsf{T}', \\ \mathsf{p}!\ell(-5).\mathcal{P}(\mathsf{T}') & \text{if } \mathsf{T} = \mathsf{p}!\ell(\mathtt{int}).\mathsf{T}', \\ \mathsf{p}!\ell(\mathsf{true}).\mathcal{P}(\mathsf{T}') & \text{if } \mathsf{T} = \mathsf{p}!\ell(\mathtt{bool}).\mathsf{T}', \\ \mathcal{P}(\mathsf{T}_1) + \mathcal{P}(\mathsf{T}_2) & \text{if } \mathsf{T} = \mathsf{T}_1 \wedge \mathsf{T}_2 \\ \mathsf{if}\ \mathsf{true} \oplus \mathsf{false}\ \mathsf{then}\ \mathcal{P}(\mathsf{T}_1)\ \mathsf{else}\ \mathcal{P}(\mathsf{T}_2) & \text{if } \mathsf{T} = \mathsf{T}_1 \vee \mathsf{T}_2, \\ \mu X_{\mathbf{t}}.\mathcal{P}(\mathsf{T}') & \text{if } \mathsf{T} = \mu\mathbf{t}.\mathsf{T}', \\ X_{\mathbf{t}} & \text{if } \mathsf{T} = \mathbf{t}, \\ \mathbf{0} & \text{if } \mathsf{T} = \mathsf{end}. \end{cases}$$

Table 11: Characteristic processes

T. In fact after each communication involving p and some $\mathsf{q} \in \mathtt{pt}\{\mathsf{T}\}$, q starts a cyclic communication involving all participants in $\mathtt{pt}\{\mathsf{T}\}$ both as receivers and senders. This is needed for getting both a projectable global type and a stuck session, see the proof of Theorem 4.5 and Examples 4.3 and 4.6. More precisely, we define the characteristic global type $\mathcal{G}(\mathsf{T},\mathsf{p})$ of the type T for the participant $\mathsf{p} \notin \mathtt{pt}\{\mathsf{T}\}$ as $\mathcal{G}(\mathsf{T},\mathsf{p}) = \mathcal{G}_0(\mathsf{T},\mathsf{p},\mathtt{pt}\{\mathsf{T}\})$, where $\mathcal{G}_0(\mathsf{T},\mathsf{p},\{\mathsf{p}_j\}_{1\le j\le n})$ is defined in Table 10.

Example 4.3. *Some characteristic global types are projectable thanks to the cyclic communication. Take for example* $\mathsf{T} = \mathsf{q}!\ell_1(\mathtt{nat}).\mathsf{r}?\ell_2(\mathtt{int}).\mathsf{end} \vee \mathsf{q}!\ell_3(\mathtt{int}).\mathsf{end}$. *Without the cyclic communication we would get the global type* $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\ell_1(\mathtt{nat}).\mathsf{r} \to \mathsf{p} : \ell_2(\mathtt{int}).\mathsf{end}, \ell_3(\mathtt{int}).\mathsf{end}\}$ *and* $\mathsf{G} \upharpoonright \mathsf{r} = \mathsf{p}!\ell_2(\mathtt{int}).\mathsf{end} \bigwedge\!\!\!\!\bigwedge \mathsf{end}$ *is undefined. Instead*

$$\begin{aligned} \mathcal{G}(\mathsf{T},\mathsf{p}) = \mathsf{p} \to \mathsf{q} : \{&\ell_1(\mathtt{nat}).\mathsf{q} \to \mathsf{r} : \ell_1(\mathtt{bool}).\mathsf{r} \to \mathsf{q} : \ell_1(\mathtt{bool}). \\ &\quad \mathsf{r} \to \mathsf{p} : \ell_2(\mathtt{int}).\mathsf{r} \to \mathsf{q} : \ell_2(\mathtt{bool}).\mathsf{q} \to \mathsf{r} : \ell_2(\mathtt{bool}).\mathsf{end}, \\ &\ell_3(\mathtt{int}).\mathsf{q} \to \mathsf{r} : \ell_3(\mathtt{bool}).\mathsf{r} \to \mathsf{q} : \ell_3(\mathtt{bool}).\mathsf{end}\} \end{aligned}$$

$$\begin{aligned} \mathcal{G}(\mathsf{T},\mathsf{p}) \upharpoonright \mathsf{r} = &(\mathsf{q}?\ell_1(\mathtt{bool}).\mathsf{q}!\ell_1(\mathtt{bool}).\mathsf{p}!\ell_2(\mathtt{int}).\mathsf{q}!\ell_2(\mathtt{bool}).\mathsf{q}?\ell_2(\mathtt{bool}).\mathsf{end}) \wedge \\ &(\mathsf{q}?\ell_3(\mathtt{bool}).\mathsf{q}!\ell_3(\mathtt{bool}).\mathsf{end}) \end{aligned}$$

It is easy to verify that $\mathcal{G}(\mathsf{T},\mathsf{p}) \upharpoonright \mathsf{p} = \mathsf{T}$ and $\mathcal{G}(\mathsf{T},\mathsf{p}) \upharpoonright \mathsf{q}$ is defined for all $\mathsf{q} \in \mathtt{pt}\{\mathsf{T}\}$ by induction on the definition of characteristic global types. Since $\mathtt{pt}\{\mathsf{G}\} = \{\mathsf{p}\} \cup \{\mathsf{p}_j\}_{1\le j\le n}$ we get the following lemma.

Lemma 4.4. *If* $\mathsf{G} = \mathcal{G}(\mathsf{T},\mathsf{p})$ *and* $\mathtt{pt}\{\mathsf{T}\} = \{\mathsf{p}_j\}_{1\le j\le n}$ *and* $\mathsf{T}_j = \mathsf{G} \upharpoonright \mathsf{p}_j$ *($1 \le j \le n$), then* $\mathtt{coherent}\{(\mathsf{T},\mathsf{p}),(\mathsf{T}_1,\mathsf{p}_1),\ldots,(\mathsf{T}_n,\mathsf{p}_n)\}$.

Characteristic processes We define the characteristic process $\mathcal{P}(\mathsf{T})$ of the type T by using the operators $\mathtt{succ}$, $\mathtt{neg}$, and $\neg$ to check if the received values are of the right sort and exploiting the correspondence between external choices and intersections, conditionals and unions. Conditionals also allow the evaluation of expressions which can be stuck. The definition of $\mathcal{P}(\mathsf{T})$ by induction on T is given in Table 11. By induction on the structure of $\mathcal{P}(\mathsf{T})$ it is easy to verify that $\vdash \mathcal{P}(\mathsf{T}) : \mathsf{T}$.

We have now all the necessary machinery to show operational preciseness of subtyping.

Theorem 4.5. (Preciseness) *The synchronous multiparty session subtyping is operationally precise.*

Proof. We only need to show completeness of the synchronous multiparty session subtyping. Let $\mathsf{T} \leqslant \mathsf{T}'$ and $\mathsf{p} \notin \mathtt{pt}\{\mathsf{T}'\} = \{\mathsf{p}_i\}_{1\le i\le n}$ and $\mathsf{G} = \mathcal{G}(\mathsf{T}',\mathsf{p})$ and $\mathsf{T}_i = \mathsf{G} \upharpoonright \mathsf{p}_i$ for $1 \le i \le n$. Then

$\vdash Q : \mathsf{T}'$ implies $\vdash \mathsf{p} \triangleleft Q \mid \prod_{1\leq i\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i)$ by Lemma 4.4 and [T-SESS]. We show that

$$\mathtt{stuck}(\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \prod_{1\leq i\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i)).$$

The proof is by induction on the definition of $\not\trianglelefteq$. We only consider some interesting cases.

[NSUB-DIFF-PART]

$$\frac{\mathsf{q} \neq \mathsf{p}_h \qquad \dagger,\ddagger \in \{?,!\}}{\mathsf{q} \dagger \ell(S).\mathsf{T}_0 \not\trianglelefteq \mathsf{p}_h \ddagger \ell'(S').\mathsf{T}'_0}$$

By definition $\mathcal{P}(\mathsf{T}) = \mathsf{q}\dagger\ell(\mathsf{e}).P$ for suitable e, P. If $\mathsf{q} \not\in \{\mathsf{p}_i\}_{1\leq i\leq n}$, then $\mathtt{stuck}(\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \prod_{1\leq i\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i))$, since $\mathcal{P}(\mathsf{T})$ will never communicate.

Otherwise let $\mathsf{q} = \mathsf{p}_j$ with $1 \leq j \leq n$ and $j \neq h$. By construction $\mathcal{P}(\mathsf{T}_h) = \mathsf{p}\bar{\ddagger}\ell'(\mathsf{e}_h).P_h$, where $\bar{\ddagger} = \begin{cases} ? & \text{if } \ddagger = ! \\ ! & \text{if } \ddagger = ? \end{cases}$, and $\mathcal{P}(\mathsf{T}_k) = \mathsf{p}_{f(k)}?\ell'(x).P_k$, where $f(k) = \begin{cases} k-1 & \text{if } k > 1 \\ n & \text{if } k = 1 \end{cases}$ for $1 \leq k \leq n$ and $k \neq h$. Therefore $\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \prod_{1\leq i\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i)$ cannot reduce.

[NSUB-IN-IN]

$$\frac{\ell_1 \neq \ell_2 \text{ or } S_2 \not\leq: S_1 \text{ or } \mathsf{T}_1 \not\trianglelefteq \mathsf{T}_2}{\mathsf{p}_h?\ell_1(S_1).\mathsf{T}_1 \not\trianglelefteq \mathsf{p}_h?\ell_2(S_2).\mathsf{T}_2}$$

A paradigmatic case is $\ell_1 = \ell_2 = \ell$, $S_1 = \mathtt{nat}$, $S_2 = \mathtt{int}$, $\mathsf{T}_1 = \mathsf{T}_2 = \mathtt{end}$. By definition $\mathtt{pt}\{\mathsf{T}'\} = \{\mathsf{p}_h\}$ and $\mathcal{P}(\mathsf{T}) = \mathsf{p}_h?\ell(x).\mathsf{if}\ \mathtt{succ}(x) > 0\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}$ and $\mathcal{P}(\mathsf{T}_h) = \mathsf{p}!\ell(-5).\mathbf{0}$. Therefore $\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \mathcal{P}(\mathsf{T}_h)$ reduces to $\mathsf{p} \triangleleft \mathsf{if}\ \mathtt{succ}(-5) > 0\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}$, which is stuck.

[NSUB-INTR]

$$\frac{\mathsf{T} \not\trianglelefteq \mathsf{T}'_1 \text{ or } \mathsf{T} \not\trianglelefteq \mathsf{T}'_2}{\mathsf{T} \not\trianglelefteq \mathsf{T}'_1 \wedge \mathsf{T}'_2}$$

By definition T'_1 and T'_2 must be intersections of inputs with the same sender, let it be p_h. Let $\mathsf{G}_1 = \mathcal{G}(\mathsf{T}'_1, \mathsf{p})$, $\mathsf{G}_2 = \mathcal{G}(\mathsf{T}'_2, \mathsf{p})$, $P_h^{(1)} = \mathcal{P}(\mathsf{G}_1 \upharpoonright \mathsf{p}_h)$, $P_h^{(2)} = \mathcal{P}(\mathsf{G}_2 \upharpoonright \mathsf{p}_h)$. Then by construction

$$P_h = \mathcal{P}(\mathcal{G}(\mathsf{T}'_1 \wedge \mathsf{T}'_2, \mathsf{p}) \upharpoonright \mathsf{p}_h) = \mathsf{if}\ \mathsf{true} \oplus \mathsf{false}\ \mathsf{then}\ P_h^{(1)}\ \mathsf{else}\ P_h^{(2)}.$$

This implies that $\mathsf{p}\triangleleft\mathcal{P}(\mathsf{T}) \mid \prod_{1\leq i\leq n} \mathsf{p}_i\triangleleft\mathcal{P}(\mathsf{T}_i)$ reduces to $\mathsf{p}\triangleleft\mathcal{P}(\mathsf{T}) \mid \mathsf{p}_h\triangleleft P_h^{(1)} \mid \prod_{1\leq i\neq h\leq n} \mathsf{p}_i\triangleleft\mathcal{P}(\mathsf{T}_i)$ and $\mathsf{p}\triangleleft\mathcal{P}(\mathsf{T}) \mid \mathsf{p}_h\triangleleft P_h^{(2)} \mid \prod_{1\leq i\neq h\leq n} \mathsf{p}_i\triangleleft\mathcal{P}(\mathsf{T}_i)$. By induction either $\mathsf{p}\triangleleft\mathcal{P}(\mathsf{T}) \mid \mathsf{p}_h\triangleleft P_h^{(1)} \mid \prod_{1\leq i\neq h\leq n} \mathsf{p}_i\triangleleft\mathcal{P}(\mathsf{T}_i)$ or $\mathsf{p}\triangleleft\mathcal{P}(\mathsf{T}) \mid \mathsf{p}_h\triangleleft P_h^{(2)} \mid \prod_{1\leq i\neq h\leq n} \mathsf{p}_i\triangleleft\mathcal{P}(\mathsf{T}_i)$ is stuck, and therefore also $\mathsf{p}\triangleleft\mathcal{P}(\mathsf{T}) \mid \prod_{1\leq i\leq n} \mathsf{p}_i\triangleleft\mathcal{P}(\mathsf{T}_i)$ is stuck.

[NSUB-UNIL]

$$\frac{\mathsf{T}'_1 \not\trianglelefteq \mathsf{T} \text{ or } \mathsf{T}'_2 \not\trianglelefteq \mathsf{T}}{\mathsf{T}'_1 \vee \mathsf{T}'_2 \not\trianglelefteq \mathsf{T}}$$

By definition T'_1 and T'_2 must be unions of outputs with the same receiver, let it be p_h. By definition $\mathcal{P}(\mathsf{T}'_1 \vee \mathsf{T}'_2) = \mathsf{if}\ \mathsf{true}\oplus\mathsf{false}\ \mathsf{then}\ \mathcal{P}(\mathsf{T}'_1)\ \mathsf{else}\ \mathcal{P}(\mathsf{T}'_2)$. Then $\mathsf{p}\triangleleft\mathcal{P}(\mathsf{T}'_1\vee\mathsf{T}'_2) \mid \prod_{1\leq i\leq n} \mathsf{p}_i\triangleleft\mathcal{P}(\mathsf{T}_i)$ reduces to $\mathsf{p}\triangleleft\mathcal{P}(\mathsf{T}'_1) \mid \prod_{1\leq i\leq n} \mathsf{p}_i\triangleleft\mathcal{P}(\mathsf{T}_i)$ and $\mathsf{p}\triangleleft\mathcal{P}(\mathsf{T}'_2) \mid \prod_{1\leq i\leq n} \mathsf{p}_i\triangleleft\mathcal{P}(\mathsf{T}_i)$. By induction

either $\mathsf{p}\triangleleft\mathcal{P}(\mathsf{T}'_1) \mid \prod_{1\leq i\leq n} \mathsf{p}_i\triangleleft\mathcal{P}(\mathsf{T}_i)$ or $\mathsf{p}\triangleleft\mathcal{P}(\mathsf{T}'_2) \mid \prod_{1\leq i\leq n} \mathsf{p}_i\triangleleft\mathcal{P}(\mathsf{T}_i)$ is stuck, and therefore $\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}'_1 \vee \mathsf{T}'_2) \mid \prod_{1\leq i\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i)$ is stuck too.

[NSUB-INTR-UNIL]

$$\frac{\forall l \in L\ \forall j \in J\ \mathsf{T}'_l \not\trianglelefteq \mathsf{T}''_j}{\bigwedge_{l\in L} \mathsf{T}'_l \not\trianglelefteq \bigvee_{j\in J} \mathsf{T}''_j}$$

If L and J are both singleton sets it is immediate by induction.
If L and J both contain more than one index, then by definition T'_i must be intersections of inputs with the same sender, let it be p_h, and T''_j must be unions of outputs with the same receiver, let it be p_k. By definition $\mathcal{P}(\mathsf{T}) = \sum_{l\in L} \mathsf{p}_h?\ell_l(x).P'_l$, and $\mathcal{P}(\mathsf{T}_k) = \sum_{j\in J} \mathsf{p}?\ell_j(x).P''_j$ and $\mathcal{P}(\mathsf{T}_u) = \mathsf{p}_{f(u)}?\ell_j(x).P_u$, where f is as in the case of rule [NSUB-DIFF-PART], for $1 \leq u \leq n$ and $u \neq k$. Therefore $\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \prod_{1\leq i\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i)$ cannot reduce.
If L contains more than one index and J is a singleton set, then by definition T'_j must be an intersection of inputs. By definition $\mathcal{P}(\mathsf{T}) = \sum_{l\in L} P'_l$, where $P'_l = \mathcal{P}(\mathsf{T}'_l)$ for $l \in L$. Let us assume ad absurdum that $\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \prod_{1\leq i\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i)$ is not stuck. Then there must be $l_0 \in L$ such that $\mathsf{p} \triangleleft P'_{l_0} \mid \prod_{1\leq i\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i)$ is not stuck, contradicting the hypothesis.
If L is a singleton set and J contains more than one index, T''_j must be a union of outputs with the same receiver, let it be p_h. Let $\mathsf{G}_j = \mathcal{G}(\mathsf{T}''_j, \mathsf{p})$ and $P^{(j)}_h = \mathcal{P}(\mathsf{G}_j \upharpoonright \mathsf{p}_h)$. Then $P_h = \mathcal{P}(\mathcal{G}(\bigvee_{j\in J} \mathsf{T}''_j, \mathsf{p}) \upharpoonright \mathsf{p}_h) = \sum_{j\in J} P^{(j)}_h$. Let us assume ad absurdum that $\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \prod_{1\leq i\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i)$ is not stuck. In this case there must be $j_0 \in J$ such that $\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \mathsf{p}_h \triangleleft P^{(j_0)}_h \mid \prod_{1\leq i\neq h\leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i)$ is not stuck, contradicting the hypothesis. □

Example 4.6. *An example showing the utility of the cyclic communication in the definition of characteristic global types is* $\mathsf{T} = \mathsf{p}_1!\ell_1(\mathtt{nat}).\mathsf{p}_2!\ell_2(\mathtt{nat})$ *and* $\mathsf{T}' = \mathsf{p}_2!\ell_2(\mathtt{nat}).\mathsf{p}_1!\ell_1(\mathtt{nat})$. *In fact without the cyclic communication the characteristic global type of* T' *would be*

$$\mathsf{G} = \mathsf{p} \to \mathsf{p}_2 : \ell_2(\mathtt{nat}).\mathsf{p} \to \mathsf{p}_1 : \ell_1(\mathtt{nat})$$

and then $\mathcal{M} = \mathsf{p}_1 \triangleleft \mathcal{P}(\mathsf{G} \upharpoonright \mathsf{p}_1) \mid \mathsf{p}_2 \triangleleft \mathcal{P}(\mathsf{G} \upharpoonright \mathsf{p}_2) = \mathsf{p}_1 \triangleleft \mathsf{p}?\ell_1(x).\mathbf{0} \mid \mathsf{p}_2 \triangleleft \mathsf{p}?\ell_2(x).\mathbf{0}$. *Being* $\mathcal{P}(\mathsf{T}) = \mathsf{p}_1!\ell_1(5).\mathsf{p}_2!\ell_2(5).\mathbf{0}$, *the session* $\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \mathcal{M}$ *reduces to* $\mathsf{p} \triangleleft \mathbf{0}$. *Instead*

$$\begin{array}{rcl}\mathcal{G}(\mathsf{T}', \mathsf{p}) & = & \mathsf{p} \to \mathsf{p}_2 : \ell_2(\mathtt{nat}).\mathsf{p}_2 \to \mathsf{p}_1 : \ell_2(\mathtt{bool}).\mathsf{p}_1 \to \mathsf{p}_2 : \ell_2(\mathtt{bool}).\\ & & \mathsf{p} \to \mathsf{p}_1 : \ell_1(\mathtt{nat}).\mathsf{p}_1 \to \mathsf{p}_2 : \ell_1(\mathtt{bool}).\mathsf{p}_2 \to \mathsf{p}_1 : \ell_1(\mathtt{bool}),\end{array}$$

which implies $\mathcal{P}(\mathcal{G}(\mathsf{T}', \mathsf{p}) \upharpoonright \mathsf{p}_1) = \mathsf{p}_2?\ell_2(x).\ldots$ *and* $\mathcal{P}(\mathcal{G}(\mathsf{T}', \mathsf{p}) \upharpoonright \mathsf{p}_2) = \mathsf{p}?\ell_2(x).\ldots$. *It is then easy to verify that* $\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \mathsf{p}_1 \triangleleft \mathcal{P}(\mathcal{G}(\mathsf{T}', \mathsf{p}) \upharpoonright \mathsf{p}_1) \mid \mathsf{p}_2 \triangleleft \mathcal{P}(\mathcal{G}(\mathsf{T}', \mathsf{p}) \upharpoonright \mathsf{p}_2)$ *is stuck.*

5 Denotational Preciseness

In λ-calculus types are usually interpreted as subsets of the domains of λ-models [1, 7]. *Denotational preciseness* of subtyping is then:

$$\mathsf{T} \leqslant \mathsf{T}' \text{ if and only if } [\![\mathsf{T}]\!] \subseteq [\![\mathsf{T}']\!],$$

using $[\![\ ]\!]$ to denote type interpretation.

In the present context let us interpret a session type T as the set of closed processes typed by T, i.e.

$$[\![\mathsf{T}]\!] = \{P \mid \vdash P : \mathsf{T}\}$$

We can then show that the subtyping is denotationally precise. The subsumption rule [T-SUB] gives the denotational soundness. Denotational completeness follows from the following key property of characteristic processes:

$$\vdash \mathcal{P}(\mathsf{T}) : \mathsf{T}' \text{ implies } \mathsf{T} \leqslant \mathsf{T}'.$$

If we could derive $\vdash \mathcal{P}(\mathsf{T}) : \mathsf{T}'$ with $\mathsf{T} \not\leqslant \mathsf{T}'$, then the multiparty session

$$\mathsf{p} \triangleleft \mathcal{P}(\mathsf{T}) \mid \prod_{1 \leq i \leq n} \mathsf{p}_i \triangleleft \mathcal{P}(\mathsf{T}_i),$$

where $\mathtt{pt}\{\mathsf{T}'\} = \{\mathsf{p}_i\}_{1 \leq i \leq n}$ and $\mathsf{G} = \mathcal{G}(\mathsf{T}', \mathsf{p})$ and $\mathsf{T}_i = \mathsf{G} \upharpoonright \mathsf{p}_i$ for $1 \leq i \leq n$, could be typed. Theorem 4.5 shows that this process is stuck, and this contradicts the soundness of the type system. We get the desired property, which implies denotational completeness, since if $\mathsf{T} \not\leqslant \mathsf{T}'$, then $\mathcal{P}(\mathsf{T}) \in [\![\mathsf{T}]\!]$, but $\mathcal{P}(\mathsf{T}) \notin [\![\mathsf{T}']\!]$.

Theorem 5.1. (Denotational preciseness) *The subtyping relations is denotationally precise.*

Acknowledgments. We are grateful to the anonymous reviewers for their useful remarks.

References

[1] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[2] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the preciseness of subtyping in session types. In *PPDP*, pages 135–146. ACM Press, 2014.

[3] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 2015. to appear.

[4] Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR*, volume 6901 of *LNCS*, pages 280–296. Springer, 2011.

[5] Mariangiola Dezani-Ciancaglini and Silvia Ghilezan. Preciseness of subtyping on intersection and union types. In *RTATLCA*, volume 8560 of *LNCS*, pages 194–207. Springer, 2014.

[6] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[7] J. Roger Hindley. The completeness theorem for typing lambda-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[8] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM Press, 2008.

[9] Dimitrios Kouzapas and Nobuko Yoshida. Globally governed session semantics. In *CONCUR*, volume 8052 of *LNCS*, pages 395–409. Springer, 2013.

[10] Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. On subtyping-relation completeness, with an application to iso-recursive types. Technical report, University of South Florida, 2014.

[11] Luca Padovani. Session types = intersection types + union types. In *ITRS*, volume 45 of *EPTCS*, pages 71–89. Open Publishing Association, 2011.

[12] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In *FOSSACS*, volume 6014 of *LNCS*, pages 128–145. Springer, 2010.

A Typed Model for Dynamic Authorizations

Silvia Ghilezan[1], Svetlana Jakšić[1], Jovanka Pantović[1], Jorge A. Pérez[2], and Hugo Torres Vieira[3]

[1] University of Novi Sad, Serbia
[2] University of Groningen, The Netherlands
[3] IMT Institute for Advanced Studies Lucca, Italy

Abstract

Security requirements in distributed software systems are inherently dynamic. In the case of authorization policies, resources are meant to be accessed only by authorized parties, but the authorization to access a resource may be dynamically granted/yielded. We describe ongoing work on a model for specifying communication and dynamic authorization handling. We build upon the π-calculus so as to enrich communication-based systems with authorization specification and delegation; here authorizations regard channel usage and delegation refers to the act of yielding an authorization to another party. Our model includes: (i) a novel scoping construct for authorization, which allows to specify authorization boundaries, and (ii) communication primitives for authorizations, which allow to pass around authorizations to act on a given channel. An authorization error may consist in, e.g., performing an action along a name which is not under an appropriate authorization scope. We introduce a typing discipline that ensures that processes never reduce to authorization errors, even when authorizations are dynamically delegated.

1 Introduction

Nowadays, computing systems operate in distributed environments, which may be highly heterogeneous, including at the level of trustworthiness. It is often the case that collaborating systems need to protect themselves from malicious entities by enforcing authorization policies that ensure actions are carried out by properly authorized parties. Such *authorizations* to act upon a resource may be statically prescribed — for instance, a determined party is known to have a determined authorization — but may also be dynamically established — for example, when a server delegates a task to a slave it may be sensible to pass along the appropriate authorization to carry out the delegated task.

As a motivating example, consider the message sequence chart given in Figure 1 describing a scenario where a client interacts with a bank portal in order to request a credit. After the client submits the request, the bank portal asks a teller to approve the request, allowing him/her to join the ongoing interaction. Apart from some rating that could be automatically calculated by the bank portal, it is the teller who ultimately approves/declines the request. It then seems reasonable that the teller *impersonates* the bank when informing the client about the outcome of the request. At this point we may ask: is the teller authorized to act on behalf of the bank portal in this structured interaction? Even if the teller gained access to the communication medium when joining the interaction, the authorization to act on behalf of the bank portal may not be necessarily granted; in such cases an explicit mechanism that dynamically grants such an authorization is required. To account for this kind of scenarios, in previous work [3] we explored the idea of role authorizations. It appears to us that the key notions underlying this idea can be well explored in a more general setting; here we aim at distilling such notions in a simple setting.

We distinguish an authorization from the resource itself: a system may already know the identity of the resource (say, an email address or a file name) but may not be authorized to act upon it (e.g., is not able to send an email on behalf of a given address or to write on a file). Also, it might be the case that the system acquires knowledge about the resource (for instance, by receiving an email address or a file) but not necessarily is immediately granted access to act upon it. We focus on communication-centered systems in which authorizations are a first-class notion modeled in a dedicated way, minimally

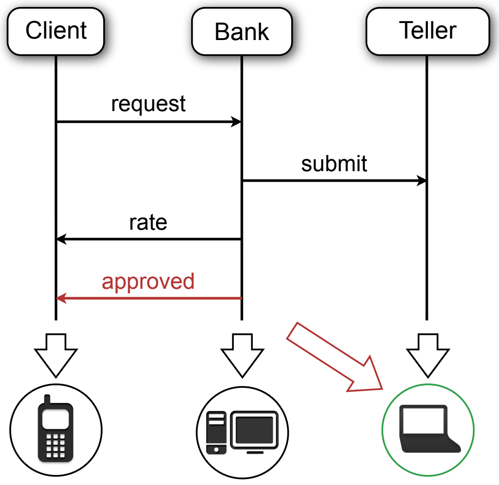


Figure 1: Credit request scenario.

extending the π-calculus [7] to capture dynamic authorization handling. As such, the resources that we consider are communication channels; authorizations regard the ability to communicate on channels.

Authorizations may be associated with a *spatial* connotation, as it seems fairly natural that a determined part of the system has access to a resource while the rest of the system does not. To this end, we introduce a *scoping operator* to specify delimited authorizations: we write $(a)P$ to specify that process P is authorized to act upon the resource a. For example, by $(a)a!b.Q$ we specify a process that is authorized on channel a and that is willing to use it to send b after which it behaves as Q.

Also, since we are interested in addressing systems in which authorizations are dynamically passed around, we model authorization communication in a distinguished way by means of dedicated communication actions: we write $a\langle b\rangle.P$ to specify the action of sending an authorization to act upon b (and proceeding as P) and $a(b).P$ to specify the action of receiving an authorization to act upon b (and proceeding as P), where in both cases channel a is used as the underlying communication medium. We remark that both in the authorization scoping $(a)P$ and in the authorization reception $b(a).P$ the name a is not bound so as to capture the notion that authorizations are handled at the level of known identities.

Given the sensitive nature of an authorization, we believe it is natural to enforce a *specialized* discipline regarding authorization manipulation. Namely, we consider that the act of passing along an authorization —*authorization delegation*— entails the yielding of the communicated authorization. That is, a party willing to communicate an authorization loses it after synchronization. Consider, for example, a process

$$S = (a)(a(b).P \mid (b)a\langle b\rangle.Q)$$

while the process on the left-hand side of the parallel composition (|) is awaiting an authorization to act on b (via a synchronization on channel a), the process on the right-hand side is willing to delegate authorization to act on b. In one reduction step, process S evolves to $(a)(((b)P) \mid Q)$, thus capturing the fact that the process on the right (Q) has now lost the authorization to act on b. Notice that this authorization transfer may have influence on the resources already known to the receiving party (i.e., process P may specify communications on channel b).

The fact that authorizations are yielded when communicated allows us to model a form of authorization accounting, in the sense that authorizations are viewed as a “countable” resource. As such, in

P, Q	$::=$	0	(Inaction)	α	$::=$	$a!b$	(Output)
	$\mid$	$P \mid Q$	(Parallel)		$\mid$	$a?x$	(Input)
	$\mid$	$(\nu a)P$	(Restriction)		$\mid$	$a\langle b\rangle$	(Send authorization)
	$\mid$	$(a)P$	(Authorization)		$\mid$	$a(b)$	(Receive authorization)
	$\mid$	$\alpha.P$	(Prefix)				

Table 1: Syntax of processes.

general we would expect $(a)(a)P$ to differ from $(a)P$. However, since we intuitively interpret $(a)P$ as "the whole of P is authorized to interact on a", it does not seem sensible that part of P can completely yield the authorization. Consider, e.g., process $(a)(b\langle a\rangle.P \mid Q)$ where it does not seem reasonable that the authorization delegation expressed by prefix $b\langle a\rangle$ interferes with the authorization on a already held by Q which is (concurrently) active in the authorization scope. Hence, a system cannot create/discard valid authorizations (that are scoping active processes); authorizations can only float around. It is also reasonable to allow that a party that delegates an authorization may get it back via another synchronization step. This way, our model allows for reasoning about authorization ownership and lending. Finally, we envisage (in our untyped model) a possibility for sharing a given authorization scope with a specified number of parties. This may be represented by specifying multiple copies of the same authorization scope. For example, process $(a)(a)(b)b\langle a\rangle.P$ (or $(a)(b)b\langle a\rangle.(a)P$) will retain authorization scope for a and reduce to $(a)(b)P$, after communication with $(b)b(a).Q$

Although some previous works have explored dedicated scoping operators with security motivations (see, e.g., [9, 4]), to our knowledge the particular combination of a (non binding) scoping construct with π-calculus name passing present in our model seems to be new. The syntactic elements of our process model, together with the dynamic nature of authorizations, pose challenges at the level of statically identifying processes that act only upon resources for which they are properly authorized. In this paper we start exploring a typing discipline for authorization manipulation that allows to statically ensure that processes never incur in authorization errors, essentially by accounting process authorization requirements. In the remaining, we formally present the language and type system, and state our results.

2 Process Model

We introduce our process calculus with authorization scoping and authorization delegation. Let $\mathcal{N}$ be a countable set of *names*, ranged over by a, b, $c, \ldots$, x, y, z. The syntax of processes is given in Table 1. Processes 0, $P \mid Q$, $(\nu a)P$, $a!b.P$ and $a?x.P$ comprise the usual π-calculus operators for specifying inaction, parallel composition, name restriction, and output and input communication actions, respectively. We introduce three novel operators, motivated earlier:

1. $a\langle b\rangle.P$ sends an authorization for the name b on a and proceeds as P;
2. $a(b).P$ receives an authorization for the name b on a and proceeds as P;
3. $(a)P$ authorizes all actions on the channel a in P.

We remark on the novel reasoning regarding scope authorization $(a)P$ in combination with π-calculus-like name passing, since all actions on channel a in process P are authorized, including actions originally specified for received names. For example, consider a process $P = (a)b?x.x!c.0$ that interacts in a context that sends name a on b. Then P may evolve to $P' = (a)a!c.0$, which is authorization safe. Still, authorizations may be "revoked" via authorization delegations.

We introduce some auxiliary notions and abbreviations, useful for the remaining formal presentation. The set of free names of a process P, denoted $\mathsf{fn}(P)$, accounts for authorization constructs in the

$$P \mid 0 \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad (\nu a)0 \equiv 0$$
$$(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P \qquad P \mid (\nu a)Q \equiv (\nu a)(P \mid Q) \quad \text{if } a \notin \mathsf{fn}(P) \qquad P \equiv_\alpha Q \implies P \equiv Q$$
$$(a)(b)P \equiv (b)(a)P \qquad (a)0 \equiv 0 \qquad (a)(P \mid Q) \equiv (a)P \mid (a)Q \qquad (a)(\nu b)P \equiv (\nu b)(a)P \quad \text{if } a \neq b$$

Table 2: Structural congruence.

following way:

$$\begin{array}{rcl} \mathsf{fn}((a)P) & \triangleq & \{a\} \cup \mathsf{fn}(P) \\ \mathsf{fn}(a\langle b\rangle.P) = \mathsf{fn}(a(b).P) & \triangleq & \{a,b\} \cup \mathsf{fn}(P) \end{array}$$

Given a name a, we use α_a to refer to either $a!b$, $a?x$, $a\langle b\rangle$, or $a(b)$. We abbreviate $(\nu a_1)(\nu a_2)\dots(\nu a_k)P$ by $(\nu\vec{a})P$ and likewise $(a_1)(a_2)\dots(a_k)P$ by $(\vec{a})P$.

Structural congruence expresses basic identities on the structure of processes, defined as the least equivalence relation between processes that satisfies the rules given in Table 2. Apart from the usual identities for the static fragment of the π-calculus (cf. first seven rules in Table 2), structural congruence gives basic principle for the novel authorization scope: (i) authorizations can be swapped around; (ii) authorizations can be discarded/created only for the inactive process; (iii) authorizations distributes over parallel composition; and (iv) authorizations and name restrictions can be swapped if the corresponding names differ. We remark that, differently from name restrictions, authorization scopes can neither be extruded/confined: since authorizations are specified for free names (that cannot be α-converted), extruding/confining authorizations actually changes the meaning of processes. For example, processes $(b)b?x.x!b.0$ and $(a)(b)b?x.x!b.0$ are not considered as structurally congruent, as the latter one authorizes the action on a in case it receives a through b, while the former one does not. Another distinctive property comes from the significance of multiplicity of authorization scopes. We do not adopt $(a)P \equiv (a)(a)P$ for $P \neq 0$, for the sake of authorization accounting. Before presenting the operational semantics of the language, we ensure that the rewriting supported by structural congruence is enough to isolate top level communication actions together with their respective authorization scopes.

Proposition 1 (Normal Form). *For any process Q we have that there are $P_1,\dots,P_k$, $\alpha_1,\dots,\alpha_k$, $\vec{c}$, and $\vec{a}_1,\dots,\vec{a}_k$, where $(\nu\vec{c})$ and $(\vec{a}_i)$ for $i \in 1,\dots,k$ can be empty sequences, such that*

$$Q \equiv (\nu\vec{c})((\vec{a}_1)\alpha_1.P_1 \mid (\vec{a}_2)\alpha_2.P_2 \mid \dots \mid (\vec{a}_k)\alpha_k.P_k) \tag{1}$$

Proof. (by induction on the structure of Q)
$Q \equiv 0$: It is in the form (1).
$Q \equiv Q' \mid Q''$: By induction hypothesis, we have that

$$Q' \equiv (\nu\vec{c})((\vec{a}_1)\alpha_1.P_1 \mid \dots \mid (\vec{a}_k)\alpha_k.P_k) \qquad Q'' \equiv (\nu\vec{d})((\vec{b}_1)\beta_1.Q_1 \mid \dots \mid (\vec{b}_l)\beta_l.Q_l)$$

and we can assume, by application of α-conversion, that $\vec{d} \cap \mathsf{fn}(Q') = \emptyset$. Therefore,

$$Q \equiv (\nu\vec{c})(\nu\vec{d})((\vec{a}_1)\alpha_1.P_1 \mid \dots \mid (\vec{a}_k)\alpha_k.P_k \mid (\vec{b}_1)\beta_1.Q_1 \mid \dots \mid (\vec{b}_l)\beta_l.Q_l).$$

$Q \equiv (\nu a)P$: Applying the induction hypothesis on P, we have that

$$Q \equiv (\nu a)(\nu\vec{c})((\vec{a_1})\alpha_1.P_1 \mid (\vec{a_2})\alpha_2.P_2 \mid \dots \mid (\vec{a_k})\alpha_k.P_k).$$

$$\text{(STRU)}\ \frac{P \equiv P' \to Q' \equiv Q}{P \to Q} \qquad \text{(PARC)}\ \frac{P \to Q}{P \mid R \to Q \mid R} \qquad \text{(NEWC)}\ \frac{P \to Q}{(\nu a)P \to (\nu a)Q} \qquad \text{(AUTC)}\ \frac{P \to Q}{(a)P \to (a)Q}$$

(COMM)
$$(\vec{a}_1)(b)b!c.P \mid (\vec{a}_2)(b)b?x.Q \to (\vec{a}_1)(b)P \mid (\vec{a}_2)(b)Q\{c/x\}$$

(AUTH)
$$(\vec{a}_1)(b)(c)b\langle c\rangle.P \mid (\vec{a}_2)(b)b(c).Q \to (\vec{a}_1)(b)P \mid (\vec{a}_2)(b)(c)Q$$

Table 3: Reduction rules.

$Q \equiv (a)P$: By induction hypothesis and α-conversion, we have that

$$Q \equiv (a)(\nu\vec{c})((\vec{a}_1)\alpha_1.P_1 \mid (\vec{a}_2)\alpha_2.P_2 \mid \ \dots \ \mid (\vec{a}_k)\alpha_k.P_k),$$

where $a \notin \{c_1, \dots, c_k\}$. Hence,

$$Q \equiv (\nu\vec{c})((a)(\vec{a}_1)\alpha_1.P_1 \mid \ \dots \ \mid (a)(\vec{a}_k)\alpha_k.P_k).$$

$Q \equiv \alpha.P$: It is in the form (1). □

We may then characterize the evolution of systems via a reduction relation, denoted by $\to$, defined as the least relation that satisfies the rules given in Table 3, focusing on the representative cases for synchronization and closing the relation under structural congruence (STRU) and static contexts (PARC), (NEWC), and (AUTC). Rule (COMM) formalizes communication of names, stating that it can be performed only via authorized channel names — notice we single out authorization scopes on channel b both for output and input. Authorization delegation is formalized by rule (AUTH). It meets the following requirements: synchronization is realized via an authorized channel and the emitting process must have the authorization in order to delegate it away (names b and c in the rule, respectively); after sending an authorization for a name the emitting process proceeds (P) falling outside of authorization scope of that name (losing authorization), and after receiving an authorization for a name the receiving process proceeds (Q) under the scope of the received authorization (acquiring authorization). Notice rules (COMM) and (AUTH) address action prefixes up to the relevant authorizations (cf. Proposition 1). We denote by $\to^\star$ the reflexive and transitive closure of $\to$.

We introduce some auxiliary notions in order to syntactically characterize *authorization errors* in our setting. First of all, we define the usual notion of *active contexts* for our calculus:

Definition 1 (Active Context)**.** $\mathcal{C}[\cdot] \ ::= \ \cdot \quad \mid \quad P \mid \mathcal{C}[\cdot] \quad \mid \quad (\nu a)\mathcal{C}[\cdot] \quad \mid \quad (a)\mathcal{C}[\cdot]$

Active contexts allow us to talk about any active communication prefixes of a process. Also, we may introduce a predicate that states that an active context authorizes actions on a given channel. More precisely, for a given context and a channel, when the hole of the context is filled with an action on the channel, it is authorised for that action.

Definition 2 (Context Authorization)**.** *For an active context $\mathcal{C}[\cdot]$ and a channel a, the context authorization predicate, denoted $auth(\mathcal{C}[\cdot], a)$, is defined inductively on the structure of $\mathcal{C}[\cdot]$ as*

$$auth(\mathcal{C}[\cdot], a) \triangleq \begin{cases} false & \text{if } \mathcal{C}[\cdot] = \cdot \\ true & \text{if } \mathcal{C}[\cdot] = (a)\mathcal{C}'[\cdot] \\ auth(\mathcal{C}'[\cdot], a) & \text{if } \mathcal{C}[\cdot] = (b)\mathcal{C}'[\cdot] \text{ and } a \neq b \\ auth(\mathcal{C}'[\cdot], a) & \text{if } \mathcal{C}[\cdot] = P \mid \mathcal{C}'[\cdot] \\ auth(\mathcal{C}'[\cdot], a) & \text{if } \mathcal{C}[\cdot] = (\nu b)\mathcal{C}'[\cdot] \end{cases}$$

$$\text{(TSTOP)}\quad \emptyset \vdash 0 \qquad \text{(TPAR)}\quad \frac{\rho_1 \vdash P \quad \rho_2 \vdash Q}{\rho_1 \cup \rho_2 \vdash P \mid Q} \qquad \text{(TNEW)}\quad \frac{\rho \vdash P \quad a \notin \rho}{\rho \vdash (\nu a)P} \qquad \text{(TAUTH)}\quad \frac{\rho \vdash P}{\rho \setminus \{a\} \vdash (a)P}$$

$$\text{(TSEND)}\quad \frac{\rho \vdash P}{\rho \cup \{a\} \vdash a!b.P} \qquad \text{(TRECV)}\quad \frac{\rho \vdash P \quad x \notin \rho}{\rho \cup \{a\} \vdash a?x.P} \qquad \text{(TDELEG)}\quad \frac{\rho \vdash P \quad b \notin \rho}{\rho \cup \{a,b\} \vdash a\langle b\rangle.P} \qquad \text{(TRECP)}\quad \frac{\rho \vdash P}{(\rho \setminus \{b\}) \cup \{a\} \vdash a(b).P}$$

Table 4: Typing rules.

We may then use active contexts and context authorization to precisely characterize errors in our model, since active contexts allow us to talk about any communication prefix in the process and the context authorization predicate allows to account for the authorizations granted by the context: processes that have active communication prefixes which are not authorized are errors.

Definition 3 (Error)**.** *We say process P is an error if $P \equiv \mathcal{C}[\alpha_a.Q]$ where*

1. *$auth(\mathcal{C}[\cdot], a) = false$ or*
2. *$\alpha_a = a\langle b\rangle$ and $auth(\mathcal{C}[\cdot], b) = false$.*

Notice that by α_a we refer to any communication action on channel a, so essentially any communication action that may cause a stuck configuration according to our semantics (where synchronizations only occur when processes hold the proper authorizations) is seen as an error.

3 Type System

In order to statically single out the processes that can never evolve into authorization errors, we introduce a suitable type system that accounts for the authorizations required by the processes.

Typing Judgment and Typing Rules. The typing judgment $\rho \vdash P$, where ρ denotes a set of names, states that process P is typed if the context provides authorizations for names ρ; hence the process performs actions in the (unauthorized) names ρ (i.e., actions along names not under respective authorization scopes). Thus, $\emptyset \vdash P$ says that all communication actions prescribed by process P are authorized, i.e., occur within the scope of the appropriate authorizations. We say that process P is *well typed* if $\emptyset \vdash P$.

Typing rules are given in Table 4. The inactive process contains no actions along unauthorized channel names (TSTOP). If two processes act along unauthorized channel names ρ_1 and ρ_2, their parallel composition performs actions along the union $\rho_1 \cup \rho_2$ of unauthorized names (TPAR). If a typed process P does not perform actions along a channel a, the process where name a is restricted is typed under the same set of names as P (TNEW). If P acts under a set of unauthorized names ρ, then $(a)P$ authorizes a in P and thus performs actions under the set of unauthorized names $\rho \setminus \{a\}$ (TAUTH). Sending a name along a channel a extends the set of unauthorized names with the name a (TSEND). Receiving a name x along a channel a extends the set of unauthorized names with the name a and it is typed only if there is no unauthorized actions along x within P (TRECV). Sending authorization for a name b along a channel a extends the set of unauthorized names with both names a and b (TDELEG). Receiving authorization for a name b along a name a is typed under the set of unauthorized names that is extended with a and does not contain b (since the reception authorizes b in P).

Main Results. Our main result is *type safety*: well-typed processes never evolve into an (authorization) error, in the sense of Definition 3. This is stated as Corollary 1; before giving the main statement we show its supporting results. We first state a basic property of typing derivations: unauthorized names must be included in the free names of the process.

Proposition 2. *If $\rho \vdash P$ then $\rho \subseteq \mathsf{fn}(P)$.*

Proof. (by induction on the depth of the derivation of $\rho \vdash P$)
If $\emptyset \vdash 0$ then $\mathsf{fn}(0) = \emptyset$.
The following cases follow by definition of free names and the induction hypothesis.
Case $\rho_1 \cup \rho_2 \vdash P \mid Q$ is derived from $\rho_1 \vdash P$ and $\rho_2 \vdash Q$: $\mathsf{fn}(P \mid Q) = \mathsf{fn}(P) \cup \mathsf{fn}(Q) \supseteq \rho_1 \cup \rho_2$.
Case $\rho \vdash (\nu a)P$ is derived from $\rho \vdash P$ and $a \notin \rho$: $\mathsf{fn}((\nu a)P) = \mathsf{fn}(P) \setminus \{a\} \supseteq \rho \setminus \{a\} = \rho$.
Case $\rho \setminus \{a\} \vdash (a)P$ is derived from $\rho \vdash P$: $\mathsf{fn}((a)P) = \mathsf{fn}(P) \cup \{a\} \supseteq \rho \cup \{a\} \supseteq \rho \setminus \{a\}$.
Case $\rho \cup \{a\} \vdash a!b.P$ is derived from $\rho \vdash P$: $\mathsf{fn}(a!b.P) = \mathsf{fn}(P) \cup \{a\} \supseteq \rho \cup \{a\}$.
Case $\rho \cup \{a\} \vdash a?x.P$ is derived from $\rho \vdash P$ and $x \notin \rho$: $\mathsf{fn}(a?x.P) = (\mathsf{fn}(P) \setminus \{x\}) \cup \{a\} \supseteq (\rho \setminus \{x\}) \cup \{a\} = \rho \cup \{a\}$.
Case $\rho \cup \{a, b\} \vdash a\langle b\rangle.P$ is derived from $\rho \vdash P$ and $b \notin \rho$: $\mathsf{fn}(a\langle b\rangle.P) = \mathsf{fn}(P) \cup \{a, b\} \supseteq \rho \cup \{a, b\}$.
Case $(\rho \setminus \{b\}) \cup \{a\} \vdash a(b).P$ is derived from $\rho \vdash P$: $\mathsf{fn}(a(b).P) = \mathsf{fn}(P) \cup \{a, b\} \supseteq \rho \cup \{a, b\} \supseteq (\rho \setminus \{b\}) \cup \{a\}$. □

We now state results used to prove that typing is preserved under system evolution, namely that (i) typing is preserved under structural congruence, as reduction is closed under structural congruence, and that (ii) typing is preserved under name substitution, since channel passing involves name substitution.

Lemma 1 (Inversion Lemma)**.** *1. If $\rho \vdash 0$ then $\rho = \emptyset$.*

2. *If $\rho \vdash P \mid Q$ then there are ρ_1 and ρ_2 such that $\rho = \rho_1 \cup \rho_2$ and $\rho_1 \vdash P$ and $\rho_2 \vdash Q$.*
3. *If $\rho \vdash (\nu a)P$ then $\rho \vdash P$ and $a \notin \rho$.*
4. *If $\rho \vdash (a)P$ then there is ρ' such that $\rho = \rho' \setminus \{a\}$ and $\rho' \vdash P$.*
5. *If $\rho \vdash a!b.P$ then there is ρ' such that $\rho = \rho' \cup \{a\}$ and $\rho' \vdash P$.*
6. *If $\rho \vdash a?x.P$ then there is ρ' such that $\rho = \rho' \cup \{a\}$ and $x \notin \rho'$ and $\rho' \vdash P$.*
7. *If $\rho \vdash a\langle b\rangle.P$ then there is ρ' such that $\rho = \rho' \cup \{a, b\}$ and $b \notin \rho'$ and $\rho' \vdash P$.*
8. *If $\rho \vdash a(b).P$ then there is ρ' such that $\rho = (\rho' \setminus \{b\}) \cup \{a\}$ and $\rho' \vdash P$.*

Lemma 2 (Subject Congruence)**.** *If $\rho \vdash P$ and $P \equiv Q$ then $\rho \vdash Q$.*

Proof. (by induction on the depth of the derivation of $P \equiv Q$)

We only write the following two interesting cases, and other cases can be obtained by similar reasoning.

Case $P \mid (\nu a)Q \equiv (\nu a)(P \mid Q)$ and $a \notin \mathsf{fn}(P)$:

From $\rho \vdash P \mid (\nu a)Q$, by Lemma 1. 2, there are ρ_1 and ρ_2 such that $\rho_1 \vdash P$ and $\rho_2 \vdash (\nu a)Q$. Therefore, by Lemma 1. 3, $\rho_2 \vdash Q$ and $a \notin \rho_2$. Since $a \notin \mathsf{fn}(P)$ we conclude by Proposition 2 that $a \notin \rho_1$. By the rule (TPAR) we get that $\rho_1 \cup \rho_2 \vdash P \mid Q$, and from $a \notin \rho_1 \cup \rho_2$, by (TNEW) we derive $\rho \vdash (\nu a)(P \mid Q)$.

Case $(a)(\nu b)P \equiv (\nu b)(a)P$ and $a \neq b$:

If $\rho \vdash (a)(\nu b)P$ then by Lemma 1. 4 there is ρ' such that $\rho' \vdash (\nu b)P$ and $\rho = \rho' \setminus \{a\}$. By Lemma 1. 3, $\rho' \vdash P$ and $b \notin \rho'$ (and so $b \notin \rho' \setminus \{a\}$). Hence, by (TAUTH) and (TNEW), we get $\rho \vdash (\nu b)(a)P$. □

Lemma 3 (Substitution)**.** *If* $\rho \vdash P$ *then* $\rho\{a/b\} \vdash P\{a/b\}$.

Proof. (by induction on the depth of the derivation of $\rho \vdash P$)

We give only one interesting case. If $\rho \cup \{b,c\} \vdash c\langle b\rangle.P$ is derived from $\rho \vdash P$ and $b \notin \rho$. It holds that $(\rho \cup \{b,c\})\{a/b\} = \rho \cup \{a,c\}$ and $(c\langle b\rangle.P)\{a/b\} = c\langle a\rangle.P\{a/b\}$. By induction hypothesis, $\rho\{a/b\} \vdash P\{a/b\}$, and by the rule (TDELEG), $\rho \cup \{a,c\} \vdash c\langle a\rangle.P\{a/b\}$. □

We may now state our soundness result which ensures typing is preserved under reduction.

Theorem 1 (Subject reduction)**.** *If* $\rho \vdash P$ *and* $P \to Q$ *then* $\rho \vdash Q$.

Proof. (by induction on the depth of the derivation of $P \to Q$)

Base case 1: Assume that $\rho \vdash (\vec{a}_1)(b)b!c.P \mid (\vec{a}_2)(b)b?x.Q$ and

$$\text{(COMM)} \qquad (\vec{a}_1)(b)b!c.P \mid (\vec{a}_2)(b)b?x.Q \to (\vec{a}_1)(b)P \mid (\vec{a}_2)(b)Q\{c/x\}.$$

By Lemma 1. 2, there are ρ_1 and ρ_2 such that $\rho = \rho_1 \cup \rho_2$ and

$$\rho_1 \vdash (\vec{a}_1)(b)b!c.P \quad \text{and} \quad \rho_2 \vdash (\vec{a}_2)(b)b?x.Q.$$

By consecutive application of Lemma 1. 4, there are ρ_1' and ρ_2' such that $\rho_1 = \rho_1' \setminus \{\vec{a_1}, b\}$ and $\rho_2 = \rho_2' \setminus \{\vec{a_2}, b\}$ and

$$\rho_1' \vdash b!c.P \quad \text{and} \quad \rho_2' \vdash b?x.Q.$$

By Lemma 1. 5-6, there are ρ_1'' and ρ_2'' such that $\rho_1' = \rho_1'' \cup \{b\}$ and $\rho_2' = \rho_2'' \cup \{b\}$ and $x \notin \rho_2''$ and

$$\rho_1'' \vdash P \quad \text{and} \quad \rho_2'' \vdash Q.$$

One should notice that $\rho_1 = (\rho_1'' \cup \{b\}) \setminus \{\vec{a_1}, b\} = \rho_1'' \setminus \{\vec{a_1}, b\}$ and $\rho_2 = (\rho_2'' \cup \{b\}) \setminus \{\vec{a_2}, b\} = \rho_2'' \setminus \{\vec{a_2}, b\}$ and $\rho_2''\{c/x\} = \rho_2''$ (since $x \notin \rho_2''$). By Lemma 3 and consecutive application of the typing rules (TAUTH) we get

$$\rho_1 \vdash (\vec{a}_1)(b)P \quad \text{and} \quad \rho_2 \vdash (\vec{a}_2)(b).Q\{c/x\}.$$

and finally, by (TPAR), we have $\rho \vdash (\vec{a}_1)(b)P \mid (\vec{a}_2)(b).Q\{c/x\}$.

Base case 2: Assume that $\rho \vdash (\vec{a}_1)(b)(c)b\langle c\rangle.P \mid (\vec{a}_2)(b)b(c).Q$ and

$$\text{(AUTH)} \qquad (\vec{a}_1)(b)(c)b\langle c\rangle.P \mid (\vec{a}_2)(b)b(c).Q \to (\vec{a}_1)(b)P \mid (\vec{a}_2)(b)(c)Q$$

By Lemma 1. 2, there are ρ_1 and ρ_2 such that $\rho = \rho_1 \cup \rho_2$ and

$$\rho_1 \vdash (\vec{a}_1)(b)(c)b\langle c\rangle.P \quad \text{and} \quad \rho_2 \vdash (\vec{a}_2)(b)b(c).Q$$

By consecutive application of Lemma 1. 4, there are ρ_1' and ρ_2' such that $\rho_1 = \rho_1' \setminus \{\vec{a_1}, b, c\}$ and $\rho_2 = \rho_2' \setminus \{\vec{a_2}, b\}$ and

$$\rho_1' \vdash b\langle c\rangle.P \quad \text{and} \quad \rho_2' \vdash b(c).Q$$

By Lemma 1. 7-8, there are ρ_1'' and ρ_2'' such that $\rho_1' = \rho_1'' \cup \{b,c\}$ and $c \notin \rho_1''$ and $\rho_2' = (\rho_2'' \setminus \{c\}) \cup \{b\}$ and

$$\rho_1'' \vdash P \quad \text{and} \quad \rho_2'' \vdash Q.$$

We conclude that $\rho_1 = (\rho_1'' \cup \{b, c\}) \setminus \{\vec{a_1}, b, c\} = \rho_1'' \setminus \{\vec{a_1}, b\}$ (since $c \notin \rho_1''$) and $\rho_2 = \rho_2'' \setminus \{\vec{a_2}, b, c\}$. By consecutive application of the typing rule (TAUTH), we get

$$\rho_1 \vdash (\vec{a}_1)(b)P \quad \text{and} \quad \rho_2 \vdash (\vec{a}_2)(b)(c)Q$$

and by (TPAR)

$$\rho \vdash (\vec{a}_1)(b)P \mid (\vec{a}_2)(b)(c)Q.$$

Case 3: Assume that $\rho \vdash P \mid R$ and $P \mid R \to Q \mid R$ is derived from $P \to Q$. By Lemma 1. 2, there are ρ_1 and ρ_2 such that $\rho = \rho_1 \cup \rho_2$ and

$$\rho_1 \vdash P \quad \text{and} \quad \rho_2 \vdash R.$$

By induction hypothesis, it holds that $\rho_1 \vdash Q$ and therefore we have, by (TPAR), that $\rho \vdash Q \mid R$.

Case 4: Assume that $\rho \vdash (\nu a)P$ and $(\nu a)P \to (\nu a)Q$ is derived from $P \to Q$. By Lemma 1. 3, it holds that $a \notin \rho$ and $\rho \vdash P$. By induction hypothesis, it holds that $\rho \vdash Q$ and therefore, by (TNEW), we get $\rho \vdash (\nu a)Q$.

Case 5: Assume that $\rho \vdash (a)P$ and $(a)P \to (a)Q$ is derived from $P \to Q$. By Lemma 1. 4, it holds that there is ρ' such that $\rho = \rho' \setminus \{a\}$ and $\rho' \vdash P$. By induction hypothesis, it holds that $\rho' \vdash Q$ and therefore, by (TAUTH), we get $\rho \vdash (a)Q$.

Case 6: Assume that $\rho \vdash P$ and $P \to Q$ is derived from $P' \to Q'$, where $P \equiv P'$ and $Q' \equiv Q$. We conclude by Lemma 2 that $\rho \vdash P'$. Than, by induction hypothesis, we get $\rho \vdash Q'$, and applying again Lemma 2, we have that $\rho \vdash Q$. □

Theorem 1 ensures, considering $\rho = \emptyset$, that well-typed processes always reduce to well-typed processes. We now express the basic property for well-typed systems, namely that they do not expose any authorization errors up to the ones granted by pending authorizations ρ.

Proposition 3 (Error Free)**.** *If $\rho \vdash \mathcal{C}[\alpha_a.Q]$ and $a \notin \rho$ then $auth(\mathcal{C}[\cdot], a)$.*

Proof. (by induction on the structure of $\mathcal{C}[\cdot]$)

Case $\mathcal{C}[\cdot] = [\cdot]$: If $\rho \vdash \alpha_a.Q$ we conclude that $a \in \rho$, by Lemma 1.
Case $\mathcal{C}[\cdot] = P \mid \mathcal{C}'[\cdot]$: If $\rho \vdash P \mid \mathcal{C}'[\alpha_a.Q]$, by Lemma 1. 2, there are ρ_1 and ρ_2 such that $\rho = \rho_1 \cup \rho_2$ and $\rho_1 \vdash P$ and $\rho_2 \vdash \mathcal{C}'[\alpha_a.Q]$. By induction hypothesis, $auth(\mathcal{C}'[\cdot], a)$, while by definition $auth(P \mid \mathcal{C}'[\cdot], a) = auth(\mathcal{C}'[\cdot], a)$.
Case $\mathcal{C}[\cdot] = (\nu b)\mathcal{C}'[\cdot]$: If $a \notin \rho$ and $\rho \vdash (\nu b)\mathcal{C}'[\alpha_a.Q]$, by Lemma 1. 3, $\rho \vdash \mathcal{C}'[\alpha_a.Q]$ and $b \notin \rho$. By induction hypothesis, $auth(\mathcal{C}'[\cdot], a)$. By definition, $auth((\nu b)\mathcal{C}'[\cdot], a) = auth(\mathcal{C}'[\cdot], a)$.
Case $\mathcal{C}[\cdot] = (b)\mathcal{C}'[\cdot]$ and $a \neq b$: If $a \notin \rho$ and $\rho \vdash (b)\mathcal{C}'[\alpha_a.Q]$, by Lemma 1. 4, there is ρ' such that $\rho = \rho' \setminus \{b\}$ and $\rho' \vdash \mathcal{C}'[\alpha_a.Q]$. If $a \neq b$ and $a \notin \rho$ then $a \notin \rho'$. By induction hypothesis, $auth(\mathcal{C}'[\cdot], a)$. By definition, $auth((b)\mathcal{C}'[\cdot], a) = auth(\mathcal{C}'[\cdot], a)$.
Case $\mathcal{C}[\cdot] = (a)\mathcal{C}'[\cdot]$: By definition $auth((a)\mathcal{C}'[\cdot], a) = true$. □

Proposition 3 thus ensures that active communication prefixes that do not involve a pending authorization (outside of ρ) are not errors. Considering $\rho = \emptyset$ we thus have that well-typed processes do not have any unauthorized prefixes and thus are not errors. Along with Theorem 1 we may then state our safety result which says well-typed processes never evolve into an error.

Corollary 1 (Type Safety). *If $\emptyset \vdash P$ and $P \rightarrow^{\star} Q$ then Q is not an error.*

Proof. Immediate from Theorem 1 and Proposition 3. □

Corollary 1 attests that well-typed systems never reduce to authorization errors, including when authorizations are dynamically delegated. The presented type system allows for a streamlined analysis on process authorization requirements, which we intend to exploit as the building block for richer analysis.

4 Concluding Remarks

The work presented here builds on our previous work [3], in which we explored authorization passing in the context of communication-centered systems. In [3], the analysis addressed not only authorization passing but also role-based protocol specification, building on the conversation type analysis presented in [1]. Here we focussed exclusively on the authorization problem, obtaining a simple model which paves the way for further investigation, since the challenges involved may now be highlighted in a crisper way. As usual, there are non typable processes that are authorized for all the actions and reduce to 0. This is unsurprising, given the simplicity of the analysis. An example is

$$(a)(b)(b?x.x!b.0 \mid b!a.a?x.0).$$

For the same reason, even though our untyped model enables to keep existing copies of delegated authorization scopes, the type system restricts the usage of their scopes. For example, the current discipline can not type the process

$$(b)(a)(a)b\langle a\rangle a!b.0 \mid (b)b(a)a?x.0$$

even though it safely reduces to 0. We believe it would be interesting to enrich the typing analysis so that it encompasses the contextual information (authorizations already held by the process) so as to address name reception and authorization delegation in a different way. We also believe it would be interesting to discipline authorization usage so as to ensure absence of double authorizations for the sake of authorization accountability, so as to ensure only the strictly necessary authorizations are specified.

Naturally, it would also be interesting to integrate the analysis presented here in richer settings, for instance (i) considering the need to ensure protocol fidelity using session types [6], or (ii) ensuring liveness properties so that security critical events are guaranteed to take place, or (iii) exploring an ontology on names so that authorizations to act upon higher ranked names automatically yield authorization for lower-level ones. While relevant, these extensions appear as orthogonal developments to the analysis presented here and therefore should be studied in depth in a dedicated way.

We briefly review some related works. Scoping operators have been widely used for the purpose of modeling security aspects (e.g., [4]) but typically they use bound names (e.g., to model secrets). With the aim of representing secrecy and confidentiality requirements in process specifications, an alternative scoping operator called *hide* is investigated in [4]. The hide operator is embedded in the so-called secrecy π-calculus, tailored to program secrecy in communications. The expressiveness of the hide operator is investigated in the context of a behavioral theory, by means of an Spy agent. In contrast, our (free name) scoping operator focuses on authorization, a different security concern. In [9], a scoping operator (called *filter*) is proposed for dynamic channel screening. In a different setting (higher-order communication) and with similar properties, the filter operator blocks all the actions that are not contained in the corresponding filter (which contains polarised channel names). Contrary to the authorization scope, filters are statically assigned to processes, while the authorization scope assigned to a process may be dynamically changed. To the best of our knowledge, the authorization scoping proposed here has not been explored before for the specification of communication-centered systems. There are high-level similarities between our work and the concept of *ownership types*, as well studied for object-oriented languages [2]. Although in principle ownership types focus on static ownership structures, assessing their use for disciplining dynamic authorizations is interesting future work. We also plan to compare our approach with different forms of authorization handling, such as those defined in [5] and [8].

Acknowledgments. We thank the anonymous referees for their insightful and useful remarks. This work was supported by COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems (BETTY) via Short-Term Scientific Mission grants (to Pantović and Vieira).

References

[1] P. Baltazar, L. Caires, V. T. Vasconcelos, and H. T. Vieira. A type system for flexible role assignment in multiparty communicating systems. In *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Revised Selected Papers*, volume 8191 of *LNCS*, pages 82–96. Springer, 2012.

[2] D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In B. N. Freeman-Benson and C. Chambers, editors, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998.*, pages 48–64. ACM, 1998.

[3] S. Ghilezan, S. Jaksic, J. Pantovic, J. A. Pérez, and H. T. Vieira. Dynamic role authorization in multiparty conversations. In *Proc. of BEAT 2014*, volume 162 of *EPTCS*, pages 1–8, 2014.

[4] M. Giunti, C. Palamidessi, and F. D. Valencia. Hide and new in the pi-calculus. In *Proceedings Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, EXPRESS/SOS 2012*, volume 89 of *EPTCS*, pages 65–79, 2012.

[5] D. Gorla and R. Pugliese. Dynamic management of capabilities in a network aware coordination language. *J. Log. Algebr. Program.*, 78(8):665–689, 2009.

[6] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, 1998, Proceedings*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[7] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes.* CUP, 2001.

[8] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Proceedings*, volume 6012 of *LNCS*, pages 529–549. Springer, 2010.

[9] J. Vivas and N. Yoshida. Dynamic channel screening in the higher order pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 66(3):170–184, 2002.

Behavioural types for non-uniform memory accesses

Juliana Franco and Sophia Drossopoulou

Imperial College London, United Kingdom

Abstract

Concurrent programs executing on NUMA architectures consist of concurrent entities (e.g. threads, actors) and data placed on different nodes. Execution of these concurrent entities often reads or updates states from remote nodes. The performance of such systems depends on the extent to which the concurrent entities can be executing in parallel, and on the amount of the remote reads and writes.

We consider an actor-based object oriented language, and propose a type system which expresses the topology of the program (the placement of the actors and data on the nodes), and an effect system which characterises remote reads and writes (in terms of which node reads/writes from which other nodes). We use a variant of ownership types for the topology, and a combination of behavioural and ownership types for the effect system.

1 Introduction

A prevalent paradigm in high performance machines is NUMA (non uniform memory access) systems, e.g., the AMD Bulldozer server[1]. NUMA systems have many *nodes* which contain processors and memory; Figure 1 shows the common NUMA structure. The nodes are connected with the other nodes through a system bus that allows processes running on a specific node to access the memory of the other nodes.

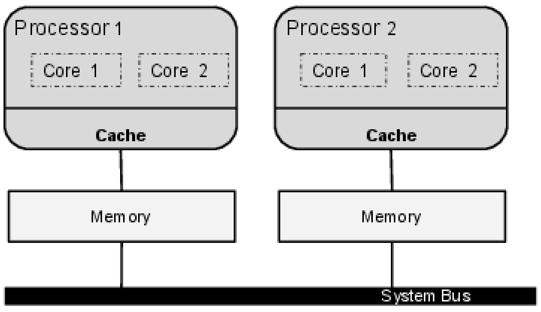


Figure 1: NUMA system [13].

Memory access is either local, i.e. accessing memory in the local node, or remote, i.e. accessing memory of remote nodes. Remote accesses require requests to the system bus, and are thus more expensive than local accesses. Moreover, different remote accesses do not necessarily have the same cost (the time to obtain/write data in memory). Therefore, to characterize the communication (read/write) costs of a concurrent program, we need to know its topology (the placement of the actors and data on the nodes), and a characterisation of the reads and writes across nodes.

In this work we consider a concurrent language based on actors (or active objects) and objects [5], which we call $\mathcal{L}_{numa}$, a language where, for the sake of simplicity, mutually recursive (synchronous and asynchronous) method invocations with communication are assumed to be not allowed and all the active objects must be created in the main class.

We develop a variant of ownership types [6] to express the location of actors and of data. In particular, we propose two levels of abstraction: classes have ownership (location) parameters, the main program defines the abstract locations and creates objects in these abstract locations; and at runtime the abstract locations are mapped to nodes (cf. Appendix C). We also propose a combination of behavioural and ownership types to characterise the interactions (reads, writes and messages sent) among objects located in different nodes.

Ownership types [4, 6, 12] were first introduced to statically describe the heap topology. Here we introduce ownership-like annotations to describe the system topology, that is, its nodes and where threads are running and data is allocated. Behavioural types [2, 8, 9, 14, 18] are usually used to describe and statically, or dynamically, verify patterns of interaction between processes/threads/participants of concurrent and parallel computations. Here we present a type system that allows the programmer to specify the interactions among objects located in different nodes, and therefore we abstract the communication made through the system bus.

Outline. This paper is organised as follows: Section 2 introduces the syntax of $\mathcal{L}_{numa}$, Section 3 gives the operational semantics, Section 4 presents the typing rules, and Section 4 shows properties of $\mathcal{L}_{numa}$, and finally Section 6 concludes. Several definitions are given in the appendix.

2 Syntax

Figure 2 presents the syntax of $\mathcal{L}_{numa}$. A program consists of a set of class declarations representing actors and passive objects. The use of the keyword active in a class declaration indicates that the class represents actors. Passive objects are similar to ordinary Java objects while actors have all the properties of passive objects, but in addition also have their own execution thread and may send messages to other actors. As in actor-based languages, messages are stored in private queues. A more detailed definition can be found in [5].

$$
\begin{array}{rcl}
P \in \mathit{Program} & ::= & \mathit{Cd}^* \\
\mathit{Cd} \in \mathit{ClassDecl} & ::= & [\mathsf{active}]\ \mathsf{class}\ C\langle \overline{p}^{+} \rangle\ \overline{\mathit{Fd}}\ \overline{\mathit{Md}} \\
\mathit{Fd} \in \mathit{FieldDecl} & ::= & f : T \\
\mathit{Md} \in \mathit{MethodDecl} & ::= & \mathsf{def}\ m(x : T) :\ T\ \mathsf{as}\ b\ \{e\} \\
ot \in \mathit{OwnershipType} & ::= & C\langle \overline{l}^{+} \rangle \\
T \in \mathit{Type} & ::= & \mathsf{bool} \mid \mathsf{nil} \mid \boxed{\mathsf{int}} \mid ot \\
l \in \mathit{Location} & ::= & p \mid L \\
val \in \mathit{Value} & ::= & \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} \\
var \in \mathit{Variable} & ::= & x \mid \mathsf{this}
\end{array}
$$

$$
\begin{array}{rcl}
e \in \mathit{Expr} & ::= & var \mid val \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& & \mid e.m(e) \mid e!m(e) \mid e.f \\
& & \mid e.f = e \mid \mathsf{new}\ ot \\
& & \mid \mathsf{for}\ i\ \mathsf{in}\ n_1..n_2\ \mathsf{do}\ e \\
& & \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \boxed{\mathsf{return}\ e} \\
\pi \in \mathit{RemAccess} & ::= & \mathsf{rd}(l,l) \mid \mathsf{wrt}(l,l) \mid \mathsf{msg}(l,l,m) \\
bop & ::= & \pi \mid \{b\ \mathsf{or}\ b\} \mid \mathsf{Loop}(n\colon b) \\
b \in \mathit{Behaviour} & ::= & \varepsilon \mid bop.b \mid [b, b]
\end{array}
$$

Figure 2: Syntax of classes and (behavioural) types. The boxed constructs are not user syntax. In the class declaration Cd the use of [ ] means that the keyword active is optional.

Each class, active or passive, is annotated with a set of location parameters $p_1, \ldots, p_n$ where p_1 represents the place where the instance of the class is allocated and $p_2, \ldots, p_n$ locations that can be used in the types of the rest of the class. The location parameters of the main class, $L_1, ..., L_n$, are abstractions of the concrete nodes, and at runtime will be mapped to concrete node identifiers.

A class declaration might have field and method declarations. A field declaration consists of a field identifier and its type; a method declaration consists of a method identifier, one parameter (variable and type), return type, behavioural type and an expression (method body). $\mathcal{L}_{numa}$ has the types bool, nil, and an ownership type $C\langle l_1, ..., l_n\rangle$ which represents objects located in l_1 that may contain references to objects in locations $l_2, ..., l_n$. The syntax of expressions is similar to other OO programming languages; note only the asynchronous method call (message sending), $e!m(e)$.

The most interesting part of the syntax is our treatment of behavioural types. We have basic operations, π, which are reading from a remote node ($\mathsf{rd}(l,l)$), writing to a remote node ($\mathsf{wrt}(l,l)$), and message sending ($\mathsf{msg}(l,l,m)$)—this has to be reflected in the behaviour, as it adds messages to queues in remote memory. For all of them the first location is where the expression is running and the second is the location where a read/write is made or a message sent. We also have types to describe conditional expressions, $\{b \text{ or } b\}$, (the two branches in the expression imply two branches in the type), and for-loops, $\mathsf{Loop}(n\colon b)$. A behavioural type, b, may be empty, ε, meaning that there is no "communication" across different nodes, the sequence of operations, $bop.b$, and two types in parallel, $[b, b]$, introduced by message sendings.

3 Semantics

We now describe the dynamic semantics of $\mathcal{L}_{numa}$. Nodes, $\mathcal{N}$, defined in Figure 3, aim to reflect NUMA nodes. Namely, a node in our formalism has an identifier, a heap with all the data allocated in it, and several execution threads *Ethread*. An execution thread belongs to an actor, and has a stack and an expression being executed. A heap is a mapping from addresses

$$\begin{array}{rl}
\mathcal{N} \in \mathit{Node} = & \mathit{NodeId} \times \mathit{Heap} \times \overline{\mathit{EThread}} \\
\mathcal{T} \in \mathit{EThread} = & \mathit{Stack} \times \mathit{Expr} \\
h \in \mathit{Heap} = & \mathit{Addr} \rightarrow \mathit{Object} \\
\sigma \in \mathit{Stack} = & \mathit{Addr} \times \overline{\mathit{Frame}} \\
\varphi \in \mathit{Frame} = & \mathit{var} \rightarrow \mathit{value} \\
Q \in \mathit{Queue} ::= & \bullet \mid \emptyset \mid m(v) :: Q \\
\mathcal{L} \in \mathit{LocsMap} = & \mathit{LocId} \rightarrow \mathit{NodeId} \\
\kappa \in \mathit{NodeId} = & \mathbb{N}
\end{array}
\qquad
\begin{array}{rl}
o \in \mathit{Object} = & \mathit{ClassId} \times \overline{\mathit{NodeId}} \times \\
& (\mathit{FieldId} \rightarrow \mathit{value}) \times \mathit{Queue} \\
\alpha \in \mathit{Addr} = & \mathit{NodeId} \times \mathbb{N} \\
v \in \mathit{value} = & \mathit{val} \mid \mathit{Addr} \mid \mathsf{skip} \mid \mathsf{NPE} \\
E[] ::= & [\cdot] \mid [\cdot].m(e) \mid \alpha.m([\cdot]) \mid [\cdot]!m(e) \mid \alpha!m([\cdot]) \\
& \mid [\cdot].f \mid [\cdot].f = e \mid \mathsf{let}\ x = [\cdot]\ \mathsf{in}\ e \mid \alpha.f = [\cdot] \\
& \mid \mathsf{if}\ [\cdot]\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{let}\ x = [\cdot]\ \mathsf{in}\ e \mid \mathsf{return}\ [\cdot] \\
l \in \mathit{Location} ::= & \text{as before} \mid \kappa
\end{array}$$

Figure 3: Dynamic Entities. We assume the existence of a map $\mathcal{L}$ that maps abstract locations (declared by the programmer in the main class) to NUMA node identifiers.

to (passive and active) objects. An object consists of a class identifier C, a sequence of node identifiers representing the actual location parameters, a mapping from field identifiers to their values, and a message queue, where the queue of a passive object is $\bullet$. An address, α, consists of a node identifier, $\kappa \in \mathit{NodeId}$, and an offset, $n \in \mathbb{N}$.

In our system, a configuration $\overline{\mathcal{N}}$ can be reduced to another configuration $\overline{\mathcal{N}}'$ either without any communication or implying a remote access from one of the nodes to another node. In the first case, the rule **[GsExec1]** should be applied, where only one node is involved in the reduction. In the second case the rule **[GsExec2]** should be used, where two nodes are involved in the reduction, as shown in Figure 4.

In the same way, expression reduction may result in accessing remote memory or not; therefore we divide the operational semantics rules as follows:

1. Expressions that do not access memory or send messages. These are defined in Figure 5.
2. Expressions that result in accesses to memory. These are defined in Figure 6 and are further divided in:
   (a) The access happens locally—only one node required.

[GsExec1]

$$\frac{\kappa, h, \sigma, e \xrightarrow{\pi} h', \sigma', e'}{\overline{\mathcal{N}}, (\kappa, h, \overline{\mathcal{T}}, \langle \sigma, e\rangle) \xrightarrow{\pi} \overline{\mathcal{N}}, (\kappa, h', \overline{\mathcal{T}}, \langle \sigma', e'\rangle)}$$

[GsExec2]

$$\frac{\kappa_1, h_1, \sigma_1, e_1 \parallel \kappa_2, h_2 \xrightarrow{\pi} h_1', \sigma_1', e_1' \parallel h_2'}{\overline{\mathcal{N}}, (\kappa_1, h_1, \overline{\mathcal{T}_1}, \langle \sigma_1, e_1\rangle), (\kappa_2, h_2, \overline{\mathcal{T}_2}) \xrightarrow{\pi} \overline{\mathcal{N}}, (\kappa_1, h_1', \overline{\mathcal{T}_1}, \langle \sigma_1', e_1'\rangle), (\kappa_2, h_2', \overline{\mathcal{T}_2})}$$

Figure 4: Global semantics

(b) The access happens remotely—two different nodes required.

Figure 5 shows shows the rules for the point 1, where no accesses to memory, eiher in the same node or not, are made. Each rule takes a node identifier, its heap, a stack and an

[SIfTrue]

$$\frac{}{\kappa, h, \sigma, \textsf{if true then } e_1 \textsf{ else } e_2 \xrightarrow{\varepsilon} h, \sigma, e_1}$$

[SIfFalse]

$$\frac{}{\kappa, h, \sigma, \textsf{if false then } e_1 \textsf{ else } e_2 \xrightarrow{\varepsilon} h, \sigma, e_2}$$

[SLet]

$$\frac{x \text{ fresh in } \varphi \quad \varphi' = \varphi[x \mapsto v]}{\kappa, h, \sigma.\varphi, \textsf{let } x = v \textsf{ in } e \xrightarrow{\varepsilon} h, \sigma.\varphi', e}$$

[SRet]

$$\frac{}{\kappa, h, \sigma.\varphi, \textsf{return } v \xrightarrow{\varepsilon} h, \sigma, v}$$

[SVar]

$$\frac{\varphi(x) = v}{\kappa, h, \sigma.\varphi, x \xrightarrow{\varepsilon} h, \sigma.\varphi, v}$$

[SFor]

$$\frac{x \text{ fresh in } \varphi \quad e' = (\textsf{let } x = e[n_1/i] \textsf{in for } i \textsf{ in } (n_1+1)..n_2 \textsf{ do } e)}{\kappa, h, \sigma.\varphi, \textsf{for } i \textsf{ in } n_1..n_2 \textsf{ do } e \xrightarrow{\varepsilon} h, \sigma.\varphi, e'}$$

[SForSkip]

$$\frac{n_2 > n_1}{\kappa, h, \sigma, \textsf{for } i \textsf{ in } n_1..n_2 \textsf{ do } e \xrightarrow{\varepsilon} h, \sigma, \textsf{skip}}$$

[SSkip]

$$\frac{}{\kappa, h, \sigma, \textsf{skip} \xrightarrow{\varepsilon} h, \sigma, \textsf{null}}$$

[SCallL]

$$\frac{\text{owners}(h, \alpha) = C\langle\overline{\kappa}\rangle \quad \varphi = \alpha \cdot (\textsf{this} \mapsto \alpha, x \mapsto v)}{\kappa, h, \sigma, \alpha.m(v) \xrightarrow{\varepsilon} h, \sigma.\varphi, \textsf{return } \mathcal{M}(C, m)\downarrow_3 [\overline{\kappa}]}$$

[SReceiveL]

$$\frac{\alpha\downarrow_1 = \kappa \quad h(\alpha) = (C, \overline{\kappa}, \_, m(v) :: Q) \quad e = \mathcal{M}(C, m)[\overline{\kappa}]}{\kappa, h, \alpha \cdot \emptyset, \textsf{null} \xrightarrow{\varepsilon} h[\alpha \mapsto Q], \alpha \cdot (\textsf{this} \mapsto \alpha, x \mapsto v), \textsf{return } e}$$

[SContextNPE]

$$\frac{}{\kappa, h, \sigma, E[\textsf{NPE}] \xrightarrow{\varepsilon} h, \sigma, \textsf{NPE}}$$

[SNPE]

$$\frac{}{\kappa, h, \sigma, e_{npe} \xrightarrow{\varepsilon} h, \sigma, \textsf{NPE}}$$

where e_{npe} can be $\textsf{null}.f$, $\textsf{null}.f = e$, $\textsf{null}.m(e)$, $\textsf{null}!m(e)$, $\textsf{null}[i]$, $\textsf{null}[i] = e'$

Figure 5: Semantic rules for expressions that do not perform remote operations. Null-pointer exceptions included.

expression, and reduces to a new heap, a new stack and a new expression. They have the form $\kappa, h, \sigma, e \xrightarrow{\pi} h', \sigma', e'$. These rules reduce without any communication among different nodes, and therefore they show reduction of expressions through ε. The intuition behind these rules is standart and similar can be found in the literature. Note only the rule for the message receiving, **[SReceiveL]**, which takes an empty stack and a null expression, meaning that the expression of the thread being executed is fully reduced, and returns a new frame and expression after taking the next message in the queue to be processed. The expression returned is the body of the method asynchronously invoked, as the new frame has the values passed as arguments.

Figure 6 shows the semantic rules for the point 2. The rules on the left belong to 2(a); they have the same form of the rules introduced in Figure 5. The rules on the right belong to 2(b); they take two node identifiers, their heaps, a stack and an expression, and reduce to two new heaps, a new stack and a new expression. They have the form $\kappa_1, h_1, \sigma, e \parallel \kappa_2, h_2 \xrightarrow{\pi} h'_1, \sigma', e' \parallel h'_2$. In both cases they reduce through an operation described by π—the remote operation made or empty, ε (in the case of the absence of a remote operation). For instance,

[SMsgL]
$$\frac{h' = h[\langle\kappa.n\rangle :: m(v)]}{\kappa, h, \sigma, \langle\kappa.n\rangle!m(v) \xrightarrow{\varepsilon} h', \sigma, \mathsf{null}}$$

[SMsgR]
$$\frac{\pi = \mathsf{msg}(\kappa_1, \kappa_2, m) \quad h'_2 = h_2[\langle\kappa_2.n\rangle :: m(v)]}{\kappa_1, h_1, \sigma, \langle\kappa_2.n\rangle!m(v) \parallel \kappa_2, h_2 \xrightarrow{\pi} h_1, \sigma, \mathsf{null} \parallel h'_2}$$

[SFReadL]
$$\frac{}{\kappa, h, \sigma, \langle\kappa.n\rangle.f \xrightarrow{\varepsilon} h, \sigma, h(\kappa.n)(f)}$$

[SFReadR]
$$\frac{\pi = \mathsf{rd}(\kappa_1, \kappa_2) \quad v = h_2(\langle\kappa_2.n\rangle)(f)}{\kappa_1, h_1, \sigma, \langle\kappa_2.n\rangle.f \parallel \kappa_2, h_2 \xrightarrow{\pi} h_1, \sigma, v \parallel h_2}$$

[SFWriteL]
$$\frac{}{\kappa, h, \sigma, \langle\kappa.n\rangle.f = v \xrightarrow{\varepsilon} h[\langle\kappa.n\rangle, f \mapsto v], \sigma, v}$$

[SFWriteR]
$$\frac{\pi = \mathsf{wrt}(\kappa_1, \kappa_2) \quad h'_2 = h_2[\langle\kappa_2.n\rangle, f \mapsto v]}{\kappa_1, h_1, \sigma, \langle\kappa_2.n\rangle.f = v \parallel \kappa_2, h_2 \xrightarrow{\pi} h_1, \sigma, v \parallel h'_2}$$

[SNewL]
$$\frac{\kappa = \mathcal{L}(L_1) \quad \langle\kappa.n\rangle \notin \mathrm{dom}(h) \qquad h' = h[\langle\kappa.n\rangle \mapsto \mathrm{initObj}(C\langle\overline{L}\rangle)]}{\kappa, h, \sigma, \mathsf{new}\ C\langle\overline{L}\rangle \xrightarrow{\varepsilon} h', \sigma, \langle\kappa.n\rangle}$$

[SNewR]
$$\frac{\kappa_2 = \mathcal{L}(L_1) \quad \langle\kappa_2.n\rangle \notin \mathrm{dom}(h_2) \quad \pi = \mathsf{wrt}(\kappa_1, \kappa_2) \qquad h'_2 = h_2[\langle\kappa_2.n\rangle \mapsto \mathrm{initObj}(C\langle\overline{L}\rangle)]}{\kappa_1, h_1, \sigma, \mathsf{new}\ C\langle\overline{L}\rangle \parallel \kappa_2, h_2 \xrightarrow{\pi} h_1, \sigma, \langle\kappa_2.n\rangle \parallel h'_2}$$

[SContextL]
$$\frac{\kappa, h, \sigma, e \xrightarrow{\pi} h', \sigma', e'}{\kappa, h, \sigma, E[e] \xrightarrow{\pi} h', \sigma', E[e']}$$

[SContextR]
$$\frac{\kappa_1, h_1, \sigma, e \parallel \kappa_2, h_2 \xrightarrow{\pi} h'_1, \sigma', e' \parallel h'_2}{\kappa_1, h_1, \sigma, E[e] \parallel \kappa_2, h_2 \xrightarrow{\pi} h'_1, \sigma', E[e'] \parallel h'_2}$$

Figure 6: Set of semantic rules described in 2. The left rules show the reduction of expressions that execute locally (a) and the right rules, expressions that interact with remote objects (b).

message sending in rule **[SMsgL]** adds a message to the queue of an actor in the same node as this, while **[SMsgR]** adds the message to the queue of an object in a different node. In the first case π is empty and in the second case it is $\mathsf{msg}(\kappa_1, \kappa_2, m)$. In both cases, the stack remains unchanged and the returned expression is null; namely execution is asynchronous. All the other rules, except the context rules, on the left show, as expected, reads and writes to the local heap and on the right present reads and writes to a remote heap.

4 Type Checking

Figure 7 shows the typing rules of $\mathcal{L}_{numa}$. They have the form $\overline{\Gamma} \vdash e \triangleright T, b$ where an expression e is verified against a sequence of typing contexts $\overline{\Gamma}$ resulting in a type T and an effect b. A typing context is a mapping from variables and addresses to types:

$$\Gamma \in \mathit{TypingContext} = (\mathit{var}\ \cup\ \mathit{Addr}) \rightarrow \mathit{Type}$$

The effect b describes the behaviour of e, that is, the memory accesses and messages sent to remote locations. Effects are concatenated via the function $\circ$ as defined below.

$$\varepsilon \circ b = b \qquad (\mathit{bop}.b_1) \circ b_2 = \mathit{bop}.(b_1 \circ b_2) \qquad [b_1, b_2] \circ b_3 = [b_1 \circ b_3, b_2]$$

The type T associated to an expression is found in a standard way: similar can be found

[T-Var/Addr]
$$\frac{}{\begin{array}{c}\overline{\Gamma}.\Gamma \vdash var \triangleright \Gamma(var), \varepsilon \\ \overline{\Gamma}.\Gamma \vdash \alpha \triangleright \Gamma(\alpha), \varepsilon\end{array}}$$

[T-True/False]
$$\frac{}{\begin{array}{c}\overline{\Gamma} \vdash \mathsf{true} \triangleright \mathsf{bool}, \varepsilon \\ \overline{\Gamma} \vdash \mathsf{false} \triangleright \mathsf{bool}, \varepsilon\end{array}}$$

[T-Skip/Null]
$$\frac{}{\begin{array}{c}\overline{\Gamma} \vdash \mathsf{skip} \triangleright \mathsf{nil}, \varepsilon \\ \overline{\Gamma} \vdash \mathsf{null} \triangleright \mathsf{nil}, \varepsilon\end{array}}$$

[T-Let]
$$\frac{\begin{array}{c}\overline{\Gamma}.\Gamma \vdash e_1 \triangleright T_1, b_1 \quad x \notin \mathrm{dom}(\Gamma) \\ \overline{\Gamma}.\Gamma[x \mapsto T_1] \vdash e_2 \triangleright T_2, b_2\end{array}}{\overline{\Gamma}.\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \triangleright T_2, b_1 \circ b_2}$$

[T-Cond]
$$\frac{\overline{\Gamma} \vdash e_1 \triangleright \mathsf{bool}, b_1 \quad \overline{\Gamma} \vdash e_2 \triangleright T, b_2 \quad \overline{\Gamma} \vdash e_3 \triangleright T, b_3}{\overline{\Gamma} \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \triangleright T, b_1 \circ \{b_2\ \mathsf{or}\ b_3\}}$$

[T-For]
$$\frac{k > j \quad \overline{\Gamma} = \overline{\Gamma}'.\Gamma \quad \overline{\Gamma}'.\Gamma[i \mapsto \mathsf{int}] \vdash e \triangleright T, b}{\overline{\Gamma} \vdash \mathsf{for}\ i\ \mathsf{in}\ j..k\ \mathsf{do}\ e \triangleright T, \mathsf{Loop}(k - j + 1\colon b)}$$

[T-Ret]
$$\frac{\overline{\Gamma} \vdash e \triangleright T, b}{\overline{\Gamma}.\Gamma \vdash \mathsf{return}\ e \triangleright T, b}$$

[T-NewO]
$$\frac{\mathrm{isActive}(C) \implies \mathrm{isMain}(\overline{\Gamma}, \mathsf{this}) \quad ot = C\langle l_1, ..., l_n\rangle \quad l_1 \neq ... \neq l_n}{\overline{\Gamma} \vdash \mathsf{new}\ ot \triangleright ot, \mathsf{wrt}(\ell(\overline{\Gamma}), l_1)}$$

[T-FWrite]
$$\frac{\overline{\Gamma} \vdash e \triangleright C\langle\bar{l}\rangle, b_1 \quad \mathcal{F}(C, f)[\bar{l}] = T \quad \overline{\Gamma} \vdash e' \triangleright T, b_2}{\overline{\Gamma} \vdash e.f = e' \triangleright T, b_1 \circ b_2 \circ \mathsf{wrt}(\ell(\overline{\Gamma}), l_1)}$$

[T-FRead]
$$\frac{\overline{\Gamma} \vdash e \triangleright C\langle\bar{l}\rangle, b_1 \quad \mathcal{F}(C, f)[\bar{l}] = T}{\overline{\Gamma} \vdash e.f \triangleright T, b_1 \circ \mathsf{rd}(\ell(\overline{\Gamma}), l_1)}$$

[T-Call]
$$\frac{\begin{array}{c}\overline{\Gamma} \vdash e_1 \triangleright C\langle\bar{l}\rangle, b_1 \quad \overline{\Gamma} \vdash e_2 \triangleright T', b_2 \\ \ell(\overline{\Gamma}) = l_1 \quad \mathcal{M}(C, m)[\bar{l}] = (T, T', e_3, b_3)\end{array}}{\overline{\Gamma} \vdash e_1.m(e_2) \triangleright T, b_1 \circ b_2 \circ b_3}$$

[T-Message]
$$\frac{\begin{array}{c}\overline{\Gamma} \vdash e_1 \triangleright C\langle\bar{l}\rangle, b_1 \quad \overline{\Gamma} \vdash e_2 \triangleright T', b_2 \\ \ell(\overline{\Gamma}) = l_0 \quad \mathcal{M}(C, m)[\bar{l}] = (\mathsf{nil}, T', e_3, b)\end{array}}{\overline{\Gamma} \vdash e_1!m(e_2) \triangleright \mathsf{nil}, b_1 \circ b_2 \circ \mathsf{msg}(l_0, l_1, m).[\varepsilon, b]}$$

Figure 7: Typing rules

in [4], therefore we focus only in the behaviour produced. The rules for variables and values, [**T-Var/Addr**], [**T-True/False**], [**T-Skip/Null**] result in empty effects, ε, because they do not represent any communication. The typing rule [**T-Let**] results in the concatenation of the behaviour of both expressions. The resulting behaviour of the rule [**T-Cond**] is the behaviour of the predicate concatenated with a choice type which describes the behaviour of both branches. The rule [**T-For**] returns a loop type $\mathsf{Loop}(n\colon b)$, where n is the number of iterations of the loop and b is the behavioural type of its body.

The behaviour of the creation of an object, with [**T-NewO**], is a write behaviour, from the location of this to the location of the new object, as new data is written to memory. The predicate isActive(C) is true if the class of the object being created in annotated as active and the predicate isMain($\overline{\Gamma}$, this) is true if the class being verified is the main class. The field write is also represented by the write behaviour, given that it changes data already in memory. Typing the expression $e.f = e'$ with the rule [**T-FWrite**] returns the concatenation of the behaviour of e, the behaviour of e' and the write from the location of this to the location of the object changed. Following the same idea, the field read, $e.f$, is represented by the read behaviour and therefore the rule [**T-FRead**] gives the concatenation of the behaviour of e with a read type from the location of this and to the location of the object read. The typing rule, [**T-Call**], describes synchronous method invocation which is only allowed if the receiver is in the same location as the this object. Its behaviour is the behaviour of the receiver concatenated with the behaviour of the expression passed as argument and the behavioural type annotated in the body of the invoked method. The typing rule for the message send, [**T-Message**], is similar. However, it is possible to send a message to a different location and moreover it introduces parallelism in our types: the receiving of the message should be executed in parallel with the continuation of the message sending—the resulting behaviour has the continuation type, which in this case is ε, in parallel with the expression to be executed due the message received.

5 The global behaviour

We define a global behaviour, Σ, as a sequence of behavioural types

$$\Sigma \in \overline{Behaviour}$$

The behaviour of a node $\mathcal{N}$ describes the remote reads, writes and message sends to be executed by the node; it is obtained from the behaviour of the execution threads and message queues of all actors in that node. The global behaviour of a runtime configuration, $\overline{\mathcal{N}}$, describes the remote reads, writes and message sends to be executed by all nodes; it is the parallel combination of the behaviours of each the nodes $\mathcal{N}_i$. Both definitions, the behaviour of a node and the global behaviour of a configuration, are below.

Definition 1 (The global behaviour)**.**

(1) $\mathcal{N}_1, \ldots, \mathcal{N}_n \blacktriangleright \overline{b}_1, \ldots, \overline{b}_n$ ***iff*** $\forall i \in 1..n : \mathcal{N}_i \blacktriangleright \overline{b}_i$

(2) $\kappa, h, \langle\sigma_1, e_1\rangle, \ldots, \langle\sigma_n, e_n\rangle \blacktriangleright b_1, \ldots, b_n$ ***iff*** $\forall i \in 1..n : h, \sigma_i, e_i \blacktriangleright b_i$

(3) $h, \sigma, e \blacktriangleright \text{filter}(b \circ b_1 \circ ... \circ b_n)$ ***iff*** $\exists T : h, \sigma \vdash e \triangleright T, b\ \wedge$

$(h(\sigma{\downarrow_1}) = (C, \kappa^+, \_, m_1(v_1) :: ... :: m_n(v_n) :: \emptyset)\ \wedge\ \forall j \in 1..n : \exists T_i : h, \sigma \vdash \mathcal{M}(C, n)[\kappa^+] \triangleright T_i, b_i)$

Using this notion of global behaviour, we implicitly assume a well-formed program and we state soundness of our typing, which says that if a well-formed configuration, $\overline{\mathcal{N}}$, with a global behaviour Σ, reduces to another configuration $\overline{\mathcal{N}}'$ through a communication step π then the resulting configuration $\overline{\mathcal{N}}'$ will have behaviour Σ' which is a reduction of Σ through π.

Theorem 1. *If* $\vdash \overline{\mathcal{N}}\ \wedge\ \overline{\mathcal{N}} \blacktriangleright \Sigma\ \wedge\ \overline{\mathcal{N}} \xrightarrow{\pi} \overline{\mathcal{N}}'$ *then* $\exists \Sigma' : \overline{\mathcal{N}}' \blacktriangleright \Sigma'\ \wedge\ \Sigma \sqsubseteq_\pi \Sigma'$

The definitions of well-formed configuration (including well-formed heap and well-formed stack) and (global) behaviour reduction are defined below:

Definition 2 (Well-formed (1) configuration, (2) node, (3) heap, (4) stack and (5) stack frame)**.**

(1) $\vdash \overline{\mathcal{N}}$ ***iff*** $\forall i, j : \mathcal{N}_i{\downarrow_1} = \mathcal{N}_j{\downarrow_1} \implies i = j\ \wedge\ \forall \mathcal{N}' : \overline{\mathcal{N}} \vdash \mathcal{N}'$

(2) $\overline{\mathcal{N}} \vdash \kappa, h, (\langle\sigma_1, e_1\rangle, ..., \langle\sigma_n, e_n\rangle)$ ***iff***

$\forall \alpha \in \text{dom}(h) : \alpha{\downarrow_1} = \kappa\ \wedge\ h(\alpha){\downarrow_2} = \kappa, \_\ \wedge\ \overline{\mathcal{N}} \vdash h$

$\wedge\ \forall i \in \{1..n\} : \text{heaps}(\overline{\mathcal{N}}) \vdash \sigma_i\ \wedge\ \exists T_i, b_i : h, \sigma_i \vdash e_i \triangleright T_i, b_i$

(3) $\overline{\mathcal{N}} \vdash h$ ***iff*** $\forall \alpha \in \text{dom}(h) : \text{heaps}(\overline{\mathcal{N}}) \vdash \alpha : \text{owners}(h, \alpha)$

(4) $h \vdash \alpha \cdot \varphi_1, ..., \varphi_n$ ***iff*** $\forall i \in \{1..n\} : h \vdash \varphi_i$

(5) $h \vdash (\mathsf{this} \mapsto \alpha, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n)$ ***iff*** $\{\alpha, v_1 ... v_n\} \subseteq \{\mathsf{true}, \mathsf{false}, \mathsf{null}\} \cup \text{dom}(h)$

Definition 3 (Global behaviour reduction)**.**

$$\Sigma \sqsubseteq_\pi \Sigma'\ \textbf{\textit{iff}}\ \Sigma = \overline{b}_1, b, \overline{b}_2\ \wedge\ \Sigma' = \overline{b}'_1, b', \overline{b}'_2\ \wedge\ b \sqsubseteq_\pi b'\ \wedge$$
$$(b = [b_1, b_2] \implies b' = b_1\ \wedge\ \exists b_j \in \Sigma, b'_j \in \Sigma' : b'_j = b_j \circ b_2)$$

Definition 4 (Behaviour reduction)**.**

$$b_1 \sqsubseteq_\pi b_2\ \textbf{\textit{iff}}\ b_1 = \pi.b_2$$
$$b_1 \sqsubseteq_\varepsilon b_2\ \textbf{\textit{iff}}\ b_1 = b_2\ \vee\ b_1 = \{b_2\ \mathsf{or}\ \_\}\ \vee\ b_1 = \{\_\ \mathsf{or}\ b_2\}\ \vee$$
$$(b_1 = \mathsf{Loop}(n\!: b).b'\ \wedge\ b_2 = b.\mathsf{Loop}(n-1\!: b).b')\ \vee\ b_1 = [b_2, \_]$$

Theorem 1 is a corollary of Lemmas 1 and 2.

Lemma 1. *If* $\overline{\mathcal{N}} \vdash h \ \wedge\ h \vdash \sigma \ \wedge\ \kappa, h, \sigma, e \xrightarrow{\pi} h', \sigma', e' \ \wedge\ h, \sigma \vdash e \triangleright T, b \ \wedge\ \neg(\sigma{\downarrow_2} = \emptyset \ \wedge\ e = \mathsf{null})$ *then* $\exists b' : h', \sigma' \vdash e' \triangleright T, b' \ \wedge\ \text{filter}(b) \sqsubseteq_\pi \text{filter}(b')$

Lemma 2. *If* $\overline{\mathcal{N}} \vdash h_1 \ \wedge\ \overline{\mathcal{N}} \vdash h_2 \ \wedge\ h_1 \cup h_2 \vdash \sigma \ \wedge\ \kappa_1, h_1, \sigma, e \parallel \kappa_2, h_2 \xrightarrow{\pi} h'_1, \sigma', e' \parallel h'_2 \ \wedge\ h_1 \cup h_2, \sigma \vdash e \triangleright T, b$ *then* $\exists b' : h'_1 \cup h'_2, \sigma' \vdash e' \triangleright T, b' \ \wedge\ \text{filter}(b) \sqsubseteq_\pi \text{filter}(b')$

6 Final Remarks

Related Work. To the best of our knowledge there is no integration of behavioural types in the active/passive object paradigm, or any formalism that combines behavioural types with ownership types to describe memory accesses; however there are already a few programming languages that use session (behavioural) types in actor-based languages, namely: the integration of session types in a Featherweight Erlang introduced by Mostrous and Vasconcelos [10]; an implementation of multiparty session types in an actor library written in Python presented by Neykova and Yoshida [11]; and the behavioural type system for an actor calulus, proposed by Crafa [7].

With respect to programming languages with the notion of locations and proximity among processes and data, Rinard presented an extension of the programming language Jade (an implicitly parallel programming language designed to explore task-level concurrency [16]) that allows the execution of tasks close to the data that they will use [15]. The language has constructs to describe how the processes access to the data; this information is analysed and used to improve the communication. Given that it is more expensive to access data remotely than locally, the author introduces a locality optimization algorithm that schedules the execution of tasks on places (processors) close to the data. The programming language X10 [17], developed by IBM, also features a notion of locality/places. In X10, each object can be either assigned to a place or distributed among different places. Chandra et al. [3] presented a new dependent type system for X10 that captures information about the locality of the resources in a partitioned heap for distributed data structures, called *place types*. It provides information not only about whether a reference is local or remote, but also if two remote references point to resources in the same place or not. Therefore, the compiler may use this information to decrease the runtime overhead.

Conclusion. This paper presents the fomalisation of a small oject-oriented programming language that amalgamates behavioural types with ownership types in order to describe remote memory accesses in NUMA systems. Ownership types play a role in the representation of the topology and behavioural types in the definition of reads, writes and messages sent to remote locations. This sequence of memory access operations are annotated in the method declarations as the ownership/location parameters are annotated in class declarations. This formalisation is just the first step towards a programming language that optimises performance by moving objects to nodes where they have a cheaper cost (the cost of interacting with other objects and of doing remote accesses).

Acknowledgements. This work was funded by the EU project UpScale FP7-612985 (`http://www.upscale-project.eu/`).

References

[1] Amd bulldozer server. `http://www.amd.com/en-us/products/server`.

[2] Giuseppe Castagna and Luca Padovani. Contracts for mobile processes. In *CONCUR 2009*, volume 5710 of *LNCS*, pages 211–228. Springer, 2009.

[3] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *PPoPP '08*, pages 11–22. ACM, 2008.

[4] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *SIGPLAN Not.*, 37:292–310, 2002.

[5] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *APLAS '08*, pages 139–154. Springer, 2008.

[6] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, pages 48–64. ACM, 1998.

[7] Silvia Crafa. Behavioural types for actor systems. Technical report, 2012.

[8] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *European Symposym on Programming*, volume 1381, pages 22–138, 1998.

[9] Naoki Kobayashi. Type systems for concurrent programs. In *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 439–453. Springer, 2003.

[10] Dimitris Mostrous and Vasco Thudichum Vasconcelos. Session typing for a featherweight Erlang. In *COORDINATION 2011*, volume 6721 of *LNCS*, pages 95–109. Springer, 2011.

[11] Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *COORDINATION 2014*, volume 8459 of *LNCS*, pages 131–146. Springer, 2014.

[12] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP'98*, pages 158–185. Springer, 1998.

[13] Optimizing applications for numa. `https://software.intel.com/en-us/articles/optimizing-applications-for-numa`.

[14] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *LICS'93*, pages 376–385, 1993.

[15] Martin C. Rinard. Locality optimizations for parallel computing using data access information. *International Journal of High Speed Computing*, 9(2):161–179, 1997.

[16] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26, 1993.

[17] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: Concurrent programming for modern architectures. In *PPoPP '07*, pages 271–271. ACM, 2007.

[18] R.E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, pages 157–171, 1986.

A Identifier Conventions and Semantics

Identifier conventions.

$$n \in \mathbb{N} \quad C \in \mathit{ClassId} \quad m \in \mathit{MethId} \quad f \in \mathit{FieldId} \quad L \in \mathit{LocId} \quad p \in \mathit{OwnershipId} \quad x, i \in \mathit{varId}$$

B Auxiliary definitions, shorthands and lookup functions

Definition 5 (Well-formed program and class)**.**

$$\vdash P \equiv \forall([\mathsf{active}]\ \mathsf{class}\ C\langle ...\rangle ... \in P) : P \vdash C \qquad P \vdash C \equiv \begin{cases} \mathcal{O}(C) = \{p_1, ..., p_n\}\ \wedge \\ \forall m : \mathcal{M}(C, m) = (T, x : T', e, b)\ \wedge \\ (\mathsf{this} \mapsto C\langle p_1, ..., p_n\rangle, x \mapsto T') \vdash e \triangleright T, b' \\ \implies\ b = \text{filter}(b') \end{cases}$$

Given that the effects returned during type checking do not exclude reads and writes happening in the same node, we apply a function filter(b) in order to exclude such annotations. The function is define as follows.

$$\text{filter}(\varepsilon) = \varepsilon \qquad \text{filter}([b_1, b_2]) = [\text{filter}(b_1), \text{filter}(b_2)]$$
$$\text{filter}(\pi.b) = (\text{if source}(\pi) = \text{dest}(\pi) \text{ then } \varepsilon \text{ else } \pi).\,\text{filter}(b)$$
$$\text{filter}(\{b_1 \mathsf{\ or\ } b_2\}.b_3) = (\text{if filter}(b_1) = \varepsilon\ \wedge\ \text{filter}(b_2) = \varepsilon \text{ then } \varepsilon \text{ else } \{\text{filter}(b_1) \mathsf{\ or\ } \text{filter}(b_2)\}).\,\text{filter}(b_3)$$
$$\text{filter}(\mathsf{Loop}(n\colon b).b') = (\text{if filter}(b) = \varepsilon \text{ then } \varepsilon \text{ else } \mathsf{Loop}(n\colon \text{filter}(b))).\,\text{filter}(b')$$

Note that if the expressiosns nested in for-loops or conditional expressions have behaviour ε, then the the loop or choice types are not annotated.

Definition 6 (Value agreement)**.**

$$\textbf{[WFTrue]}\ \frac{}{h \vdash \mathsf{true} : \mathsf{bool}} \qquad \textbf{[WFFalse]}\ \frac{}{h \vdash \mathsf{false} : \mathsf{bool}} \qquad \textbf{[WFNull]}\ \frac{T = \mathsf{nil}\ \vee\ \text{isValid}(T)}{h \vdash \mathsf{null} : T}$$

$$\textbf{[WFObj]}\ \frac{\begin{array}{c} h(\alpha) = (C, (\overline{\kappa}), (f_i \mapsto v_i)_{i \in I}, \bullet) \\ \forall i \in I : h \vdash v_i : \mathcal{F}(C, f_i)[\overline{\kappa}] \end{array}}{h \vdash \alpha : C\langle\overline{\kappa}\rangle}$$

$$\textbf{[WFAObj]}\ \frac{\begin{array}{c} \textit{For } I \textit{ some index set} \quad h(\alpha) = (C, (\overline{\kappa}), (f_i \mapsto v_i)_{i \in I}, m_1(v_1) :: ... :: m_n(v_n) :: \emptyset) \\ \forall i \in I : h \vdash v_i : \mathcal{F}(C, f_i)[\overline{\kappa}] \quad h, \alpha \cdot (\mathsf{this} \mapsto \alpha, x \mapsto v_i) \vdash v_i \triangleright \mathcal{M}(C, m_i){\downarrow_2}\ [\overline{\kappa}], b \end{array}}{h \vdash \alpha : C\langle\overline{\kappa}\rangle}$$

Lookup functions Considering P, the globally accessible program, and the class declaration $\mathsf{class}\ C\langle p^+\rangle\{\overline{Fd}\ \overline{Md}\} \in P$:

$$\mathcal{O}(C) = \{p^+\} \qquad \mathcal{F}(C, f) = T \textbf{ iff } f : T \in \overline{Fd} \qquad \mathcal{F}_s(C) = \{\overline{Fd}\}$$
$$\mathcal{M}(C, m) = (T, T', e, b) \textbf{ iff } \mathsf{def}\ m(x : T') : T \mathsf{\ in\ } b\ \{e\} \in \overline{Md}$$
$$\mathcal{F}(C, f)[l_1, ..., l_n] = \mathcal{F}(C, f)[l_1/p_1, \ldots, l_n/p_n] \textbf{ where } \mathcal{O}(C) = \{p_1, \ldots, p_n\}$$

Operations on the heap

$$\begin{aligned} h[\alpha \mapsto o] = h' &\textbf{ where } h'(\alpha) = o\ \wedge\ \forall \alpha_i \in \text{dom}(h) \setminus \{\alpha\} :\ h(\alpha_i) = h'(\alpha_i) \\ h[\alpha, f \mapsto v] = h' &\textbf{ where } h'(\alpha) = h(\alpha)[f \mapsto v]\ \wedge\ \forall \alpha_i \in \text{dom}(h) \setminus \{\alpha\} :\ h(\alpha_i) = h'(\alpha_i) \\ h[\alpha :: m(v)] = h' &\textbf{ where } h(\alpha) = o\ \wedge\ o{\downarrow_4} \neq \bullet\ \wedge\ h' = h[\alpha \mapsto (o{\downarrow_1}, o{\downarrow_2}, o{\downarrow_3}, m(v) :: o{\downarrow_4})] \\ &\qquad \wedge\ \forall \alpha_i \in \text{dom}(h) \setminus \{\alpha\} :\ h(\alpha_i) = h'(\alpha_i) \\ \text{owners}(h, \alpha) = C\langle\overline{\kappa}\rangle &\textbf{ where } h(\alpha){\downarrow_1} = C\ \wedge\ h(\alpha){\downarrow_2} = \overline{\kappa} \\ h_1 \cup h_2 = h &\textbf{ where } \forall \alpha \in \text{dom}(h) : h_1(\alpha) = h(\alpha)\ \vee\ h_2(\alpha) = h(\alpha) \end{aligned}$$

Operations on objects

$$o(f) \equiv o{\downarrow_3}\,(f)$$

$$o[f \mapsto v] \equiv (o{\downarrow_1}, o{\downarrow_2}, (f \mapsto v, \overline{f_i \mapsto v_i}), o{\downarrow_4}) \qquad \textbf{where } o{\downarrow_3} = f \mapsto \_, \overline{f_i \mapsto v_i}$$

$$\text{initObj}(C\langle L_1, ..., L_m\rangle) \equiv \begin{cases} (C, \kappa_1, ..., \kappa_m, (f_i \mapsto \text{init}(T_i))_{i \in 1..n}, \emptyset) & \text{isActive}(C) \\ (C, \kappa_1, ..., \kappa_m, (f_i \mapsto \text{init}(T_i))_{i \in 1..n}, \bullet) & \text{otherwise} \end{cases}$$

$$\textbf{where } \mathcal{F}_s(C) = \{f_1 : T_1, \ldots, f_n : T_n\} \text{ and } \forall j \in \{1..m\} : \kappa_j = \mathcal{L}(L_j)$$

Operations on types

$$\text{init}(T) \equiv \text{if } T = \mathsf{bool} \text{ then } \mathsf{false} \text{ else } \mathsf{null} \qquad \ell(\overline{\Gamma}) = l \textbf{ iff } \overline{\Gamma} = \_.\Gamma \;\wedge\; \Gamma(\mathsf{this}) = C\langle l, \_\rangle$$

Other definitions

$$e[C, \kappa_1, \ldots, \kappa_n] = e[\kappa_1/p_1, \ldots, \kappa_n/p_n] \text{ where } \mathcal{O}(C) = \{p_1, \ldots, p_n\}$$

$$\text{heaps}(\mathcal{N}_1, \ldots, \mathcal{N}_n) = h_1 \cup \cdots \cup h_n \textbf{ iff } \forall i \in \{1..n\} : \mathcal{N}_i{\downarrow_2} = h_i$$

$$h, \sigma \vdash e \triangleright T, b \textbf{ iff } \text{buildContext}(h, \sigma) \vdash e \triangleright T, b$$

$$\text{typeOf}(h, v) \equiv \text{if } v = \mathsf{true} \;\vee\; v = \mathsf{false} \text{ then } \mathsf{bool} \text{ else } \text{owners}(h, v)$$

$$\frac{\begin{array}{c}\text{buildContext}(h, \varphi_1) = \Gamma_n \\ \ldots \\ \text{buildContext}(h, \varphi_n) = \Gamma_1\end{array}}{\text{buildContext}(h, \alpha \cdot \varphi_1 \ldots \varphi_n)} \qquad \frac{\begin{array}{c}T_{this} = \text{typeOf}(h, \alpha) \\ T_1 = \text{typeOf}(h, v_1) \quad \ldots \quad T_n = \text{typeOf}(h, v_n) \\ \Gamma = (\mathsf{this} \mapsto T_{this}, x_1 \mapsto T_1, \ldots, x_n \mapsto T_n)\end{array}}{\text{buildContext}(h, \mathsf{this} \mapsto \alpha, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n) = \Gamma}$$

C Topology Example

Consider the following code with three class declarations: an active class C, a passive D and the class Main. An active object, instance of C, has three fields pointing to three objects in different locations of type D. The class main creates three abstract (or symbolic) locations L1, L2, L3 and the body of the main method.

```
active class C⟨p1, p2, p3⟩
  d1: D⟨p1⟩
  d2: D⟨p2⟩
  d3: D⟨p3⟩

class D⟨p⟩
```

```
class Main⟨L1, L2, L3⟩
  def main(): nil
    as b write(L1, L2). write(L1, L3) {
    let x = new C ⟨L1, L2, L3⟩ in
    let y = (x.d1 = new D⟨L1⟩) in
    let z = (x.d2 = new D⟨L2⟩) in
                x.d3 = new D⟨L3⟩
  }
```

The topology after execution of the main method is depicted in the following figure. In the abstract

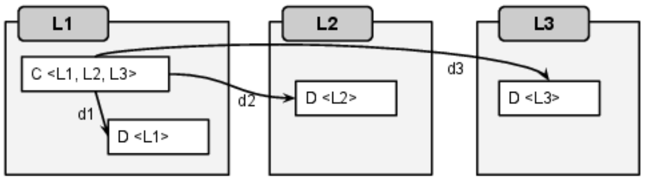


Figure 8: The ownership topology after the execution of the expression in the method main.

location L1 there is an instance of class C and an instance of class D. Abstract locations L2 and L3 have both an instance of class D. Although the programmer define 3 abstract locations, the machine might have a different number of nodes. For instance, in a system with two different nodes, we could have the mapping $(L_1 \mapsto \kappa_1, L_2 \mapsto \kappa_2, L_3 \mapsto \kappa_2)$ between abstract locations and node identifiers, which means that the objects in L1 are in the node κ_1, and objects from L2 and L3 are in the same node, as depicted in Figure 9.

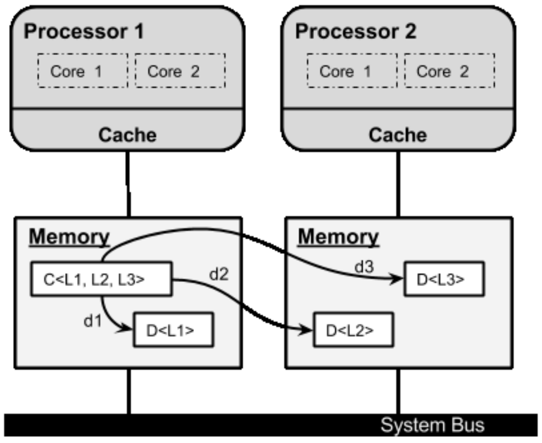


Figure 9: NUMA system with two different nodes

Broadcast and aggregation in BBC

Hans Hüttel[1] and Nuno Pratas[2]

[1] Department of Computer Science, Selma Lagerlöfs Vej 300, 9220 Aalborg Ø, Aalborg University, Denmark
hans@cs.aau.dk

[2] Department of Electronic Systems, Fredrik Bajers Vej 7, 9220 Aalborg Ø, Aalborg University, Denmark
nup@es.aau.dk

Abstract

In this paper we introduce a process calculus BBC that has both forms of communication. For both many-to-one and one-to-many communication, it is often a natural assumption that communication is bounded; this reflects two distinct aspects of the limitations of a medium. In the case of broadcast, the bound limits the number of possible recipients of a message. In the case of collection, the bound limits the number of messages that can be received. For this reason, BBC uses a notion of bounded broadcast and collection. Moreover, the syntax of the calculus introduces an explicit notion of connectivity that makes it possible to represent a communication topology directly. By using a proof technique introduced by Palamidessi we show that even a version of BBC that only uses collection is more expressive than the π-calculus.

1 Introduction

In the setting of distributed systems, interprocess communication will often be in the form of multi-party communication. Here there are two dual paradigms: that of one-to-many communication, commonly denoted as *broadcast*, and that of many-to-one, denoted as *aggregation* or *collection*.

Many-to-one communication is a commonly occurring phenomenon and often occurs in settings with intricate network topologies. As an example consider the case of a smart grid with thousands of meters (such as electricity, water, heating, etc . . .) that report their current measurements. A central server somewhere will collect these data; however, the server will usually not be directly reachable from the meters – there will be several intermediate hops, each one facilitated by a relay. If the underlying protocol of the network is correct, we would expect this complicated network to be semantically equivalent to a simple planar network, where each node is at one-hop distance from the server.

There has already been considerable interest in understanding the semantic foundations of one-to-many communication in the process calculus community, starting with the work by Prasad [6] on a broadcast version of CCS. Later, Ene and Muntean extended the notion to a broadcast version of the π-calculus [1] and proved that this notion is strictly more expressive than standard π-calculus synchronization.

Other process calculi with a notion of broadcast arise in the search for behavioral models of protocols for wireless networks. Singh et al. describe a process calculus with localities [9], and Kouzapas and Philippou [4] introduce another process calculus with a notion of localities, whose configuration can evolve dynamically.

In this paper we introduce a process calculus BBC that has both forms of communication. For both many-to-one and one-to-many communication, it is often a natural assumption that communication is *bounded*; this reflects two distinct aspects of the limitations of a medium.

$$M ::= x \mid S \mid (M_1, \ldots, M_k) \mid g(E) \mid f(M)$$

$$P ::= a\,(\lambda\vec{x}M\,)P_1 \mid a\,[\lambda\vec{x}M.S\,]P_1 \mid \overline{a}\langle M\rangle.P_1 \mid (\nu x : \beta_x)P_1 \mid [M_1 = M_2]P_1 \mid [M_1 \neq M_2]P_1 \mid (P_1 \mid P_2)$$

$$\mid (P_1 + P_2) \mid \mathbf{0} \mid A\left(\vec{M}\right)$$

$$E ::= \{M_1, \ldots, M_k\} \mid S$$

$$N ::= l[P] \mid (N_1 \mid N_2) \mid (\nu x : b)N \mid l \triangleright m$$

Table 1: Formation rules for BBC

In the case of broadcast, the bound limits the number of possible recipients of a message. In the case of collection, the bound limits the number of messages that can be received. For this reason, BBC uses a notion of bounded broadcast and collection. Moreover, the syntax of the calculus introduces an explicit notion of connectivity that makes it possible to represent a communication topology directly. By using a proof technique introduced by Palamidessi [5] we show that even a version of BBC that only uses collection is more expressive than the π-calculus.

The remainder of our paper is organized as follows. We start by introducing the syntax of BBC in Section 2, followed by Section 3 where we provide an example of a broadcast and collection protocol modeled using BBC. Then, in Section 4 we expose the reduction semantics. In Section 5 we introduce a notion of barbed bisimilarity, which is then used in Section 6 to prove the correctness of the exposed protocol example. Finally, in Section 7 we show that even a version of BBC that only uses collection is more expressive than the π-calculus.

2 The syntax of BBC

In this section we describe the syntax of BBC.

2.1 The syntactic categories

A central notion in BBC is that of *names*. Processes reside at named sites, called *locations*, and use named *channels* for communication. We assume that names are taken from a countably infinite set **Names**. In general, we denote names of channels by $a, b, c \ldots$, names of locations by $l, m, n \ldots$ and if nothing is assumed about the usage of the name we denote them by $x, y, z \ldots$.

We let $M \in \mathbf{Msg}$ range over the set of messages, let P range over the set of processes and let N range over the set of networks. Since a collecting input (defined below) can receive a multiset of messages, each coming from a distinct sender, we also consider *multiset expressions* E and multiset variables S that can be instantiated to multiset expressions. The formation rules defining the syntactic categories of BBC are given below.

2.2 Messages and patterns

For ordinary expressions we assume a collection of term constructors ranged over by f, that build messages out of other messages. Moreover, we assume the existence of a collection of multiset selectors ranged over by g; these can be used to build messages out of multisets.

If a channel is to be chosen among a collection of candidate channels, we can use a multiset selector to describe this.

Example 1. *An example of a multiset selector is the function find-a that intuitively returns the name a if this name occurs as the first component in a multiset of pairs of names and the name $k \neq a$ otherwise. This function is defined by find-a$(S) = a$ if $(a, x) \in S$ for some x, otherwise by k.*

A practical example of interest is the election of a common channel in a ad-hoc network.

An important notion is that of an *input pattern* which is of the form $(\lambda\vec{x}M)$, where the variable names in $\vec{x}$ are distinct and occur free in M. A message O matches this pattern, if it can be obtained from it through substitution.

More formally, a *term substitution* is a finite function $\theta : \textbf{Names} \to \textbf{Msg}$. The substitution can also be written as a list of bindings $\theta = [x_1 \mapsto M_1, \ldots, x_k \mapsto M_k]$. The action of θ on an arbitrary message or multiset expression is defined in the expected way. M' is said to match $(\lambda\vec{x}M)$ with θ if for a substitution θ with $\text{dom}(\theta) = \vec{x}$ if $M' = M\theta$ is true.

2.3 Processes

In a collecting communication setting, the receiver can make no assumption about the number of messages that will be received, nor on the order in which they are received. Moreover, we cannot assume that a message that has arrived will only occur once among the messages received during a single collecting communication. We shall therefore think of a collecting input as the reception of a multiset of messages.

There are two kinds of input prefixes in BBC:

- The *broadcast input* $a\,(\lambda\vec{x}M\,)P_1$ in which a single term matching the pattern $(\lambda\vec{x}M)$ is received on the channel a. The pattern variables in $\vec{x}$ are bound in P_1 and get instantiated with the appropriate subterms that correspond to the pattern.
- The *collection input* $a\,[\lambda\vec{x}M.S\,]P_1$ in which a non-empty multiset of terms $\{M_1, \ldots, M_K\}$ each of which matches the pattern $[\lambda\vec{x}M]$ is received on the channel a. Note that in this case the scope of the pattern variables in $\vec{x}$ *does not extend* to P_1. Following the input, the multiset variable S is instantiated to the multiset $\{M_1, \ldots, M_K\}$.

In a *restriction* $(\nu x : \beta_x)P_1$ the bounded capacity of the bound name x is described by the function β_x, where $\beta_x : \textbf{Names} \to \mathbb{N}$ is a function such that for any location name m we have that $\beta_x(m) = k$, if it is the case that for a process located at m there are at most k senders that are able to send a message to (in the case of collection) or receive from (in the case of broadcast) using the channel x. For free names their capacity bound is given by a function $b : \textbf{Names} \times \textbf{Names} \to \mathbb{N}$; we return to this in Section 4.

The remaining process constructs are standard. The *output* process $\overline{a}\langle M\rangle.P_1$ sends out the message M on the channel named a and then continues as P_1. *Match* $[M_1 = M_2]P_1$ and *mismatch* $[M_1 \neq M_2]P_1$ proceed as P_1 if M_1 and M_2 are equal, respectively distinct. *Parallel composition* $P_1 \mid P_2$ runs the components P_1 and P_2 in parallel. *Nondeterministic choice* $P_1 + P_2$ can proceed as either P_1 or P_2; *inaction* $\mathbf{0}$, has no behavior.

Finally, we allow agent identifiers $A(\vec{M})$ parameterized by a sequence of messages; an identifier must be defined using an equation of the form $A(\vec{x}) \overset{\text{def}}{=} P$. The only names free in P must be the parameters found in $\vec{x}$, that is $\text{fn}(P) \subseteq \vec{x}$. Definitions of this form can be recursive, with occurrences of $A(\vec{x})$ (with names in $\vec{x}$ instantiated by concrete messages) occurring within P.

Restriction and broadcast input are name binders; for a process P, the sets of *free names* $\mathrm{fn}(P)$ and *bound names* $\mathrm{bn}(P)$ of P are defined as expected. For collection input we define

$$\mathrm{bn}(a\,[\lambda\vec{x}M\;S\,]P_1) = \mathrm{bn}(P_1)$$

Replication, denoted as $!\;P$, is a derived construct in BBC; a replicated process $!\;P$ is expressed by the agent identifier A_P whose defining equation is $A_P = P \mid A_P$ and should therefore be thought of as an unbounded supply of parallel copies of P.

2.4 Networks

As in the distributed π-calculus [7] a parallel composition of located processes is called a *network*. According to the formation rules, a *network* N can be a process P running at location l, which is denoted as $l[P]$. We also allow parallel composition $P_1 \mid P_2$ and restriction $(\nu x)N$ at network level. Moreover, the *neighborhood predicate* $l \triangleright k$ denotes that location l is close to k.

For the neighbourhood predicate, parallel composition is thought of as logical conjunction. So, if l and k are close to each other, then $l \bowtie m$ can be written instead of $l \triangleright m | m \triangleright l$.

For any term M, the set of free names $\mathrm{fn}(M)$ is defined in the standard way.

The usual notions of α-conversion also apply here. We write $P_1 \equiv_\alpha P_2$ if P_1 can be obtained from P_2 by renaming bound names in P_2, and likewise we write $N_1 \equiv_\alpha N_2$ if N_1 can be obtained from N_2 by renaming bound names in N_2.

3 A hierarchical protocol

In this section we outline how one can use BBC to describe a distributed protocol that involves both collection and broadcast. The protocol itself is generic but representative of the current trend in distributed communication systems with a large number of devices and very few controlling/central entities. In essence, this protocol uses traffic collection in the upstream direction, i.e. from the leaves to the central entity, and traffic broadcast in the downstream direction, i.e. from the central entity to the leaf nodes.

Figure 1 shows the multi-level network topology assumed for this protocol; the topology is that of a tree. Each level of the network is connected to the one above through a local collection and broadcast node, which we simply denote as local central node. This node is then a leaf node of the above level, e.g. the nodes denoted as C^1 are leaf nodes of the level l_0, while being the local central nodes at each of the locations of l_1^i, which for ease of notation we denote solely as l_1. We assume that there can be at most β nodes connected to every central node, meaning that the bound of every channel should be β.

The goal of the distributed protocol is to collect information from all leaf nodes in the network; and then communicate it to a central entity, denoted as C^0. The central entity C^0, then reaches a global decision, which is then communicated to the leaf nodes. For simplicity, we assume that this decision only depends on the data collected from the leaves.

3.1 The n-level protocol in BBC

In what follows, we define the components of the network and protocol inductively. We assume the network to have n levels or depth n, where level 0 corresponds to the central entity, while level n denotes the level where all the leaf nodes are.

One of the sub-networks of the n-level protocol at level k, as depicted in Figure 1, is composed by m nodes and a local central node C_k. We denote this sub-network by the agent identifier

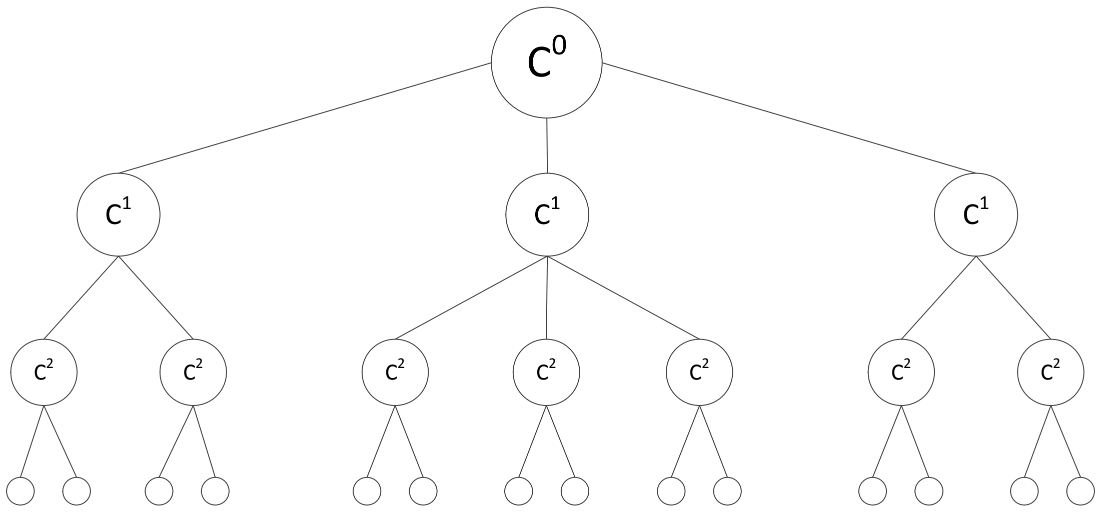


Figure 1: A multi-level network.

$D^{k,n}\,(a'_k, a''_k, b'_k, b''_k, \mathcal{L}, \ell_{k-1})$ where $\mathcal{L}$ is a set of locations and ℓ_{k-1} is the local central location at $k-1$ level. The other name parameters are

- a'_k, a''_k - names of the channels used for communications at location l_k, i.e. between the leaf nodes and the local central entity; the names transmitted are collected by the central node at level $k+1$.
- b'_k, b''_k - names of the channels that the local central entity uses to communicate to the local central entity for which it is a lead, i.e. the channel over which the local central entity C_k communicates with the local central entity C_{k-1}. The names transmitted are broadcast to the child nodes of the local central (at level $k-1$).
- $\mathcal{L}$ is the set of nodes at this level.

At the leaf level, $k = 0$ we have,

$$D^{0,n}(a'_0, a''_0, b'_0, b''_0\mathcal{L}, \ell_0) \stackrel{\text{def}}{=} \prod_{\ell_i \in \mathcal{L}} \ell_i[P_i(a'_0, a''_0)] \mid \prod_{\ell_i \in \mathcal{L}} \ell_i \triangleright \ell_1 \mid \ell_0[C^0(a'_0, a''_0, b'_0, b''_0)]$$

The subprocess P_i where $\ell_i \in \mathcal{L}$ is defined by

$$P_i(a'_0, a''_0) \stackrel{\text{def}}{=} (\nu\omega_i)\overline{a'_0}\langle\omega_i\rangle.a''_0(z).(Q_i(z) \mid P_i(a'_0, a''_0))$$

where $Q_i(z)$ depicts the computation that occurs at the leaf ℓ_i; we shall not describe this here and ω_i is the local name that is the contribution made by the process.

The local central node is defined by

$$C^{0,n}(a', a'', b', b'') \stackrel{\text{def}}{=} a'(S).\overline{b'}\langle f(S)\rangle.b''(z).\overline{a''}\langle z\rangle.C^{0,n}(a', a'', b', b'')$$

This process will receive names that are collected to the set S and then pass the processed information $f(S)$ upwards on the b channel. After this, the local central node waits for a response to the information sent and passes on the received name to its children.

The *selection function* $f : \mathcal{P}(\mathcal{N}) \to \mathcal{N}$ processes the messages received from the leaf nodes and selects a name; this could be e.g. a channel estimate. We require that selection is *idempotent* in the sense that for any family of multisets $S_1, \ldots, S_k$ we have

$$f(\{f(S_1), \ldots, f(S_k)\}) = f(\cup_{i=1}^{k} S_i)$$

If we think of names as natural numbers, the function $f(S) = \min_{x \in S} x$ is an example of an idempotent selection function.

For intermediate levels, $0 < k < n$, $D^k\,(a_k, b_k, \ell_k, \ell_{k-1})$ is defined as follows,

$$D^{k+1,n}(a'_{k+1}, a''_{k+1}, b'_{k+1}, b''_{k+1}, \mathcal{L}, \ell_{k+1}) \stackrel{\text{def}}{=} (\nu \vec{\ell}, a'_k : \beta, a''_k : \beta, b'_k : \beta, b''_k : \beta)$$
$$\left(\prod_{\mathcal{L}_i \in \mathsf{Locs}(\mathcal{L},k)} D^{k,n}(a'_k, a''_k, b'_k, b''_k, \mathcal{L}_i, \ell_i) \mid \prod_{i=1}^{m} \ell_i \triangleright \ell_k \mid \ell_{k+1}[C^{k+1}(a'_k, a''_k, b'_k, b''_k)] \right)$$

where the set of locations $\mathsf{Locs}(\mathcal{L}, k)$ can be partitioned as $\mathsf{Locs}(\mathcal{L}, k) = \bigcup_{i \in I} \mathcal{L}_i$ where I is some index set and $\mathcal{L}_i \cap \mathcal{L}_j = \emptyset$ for all $i \neq j$ and $i, j \in I$. m'_i is the number of neighbours in the i'th sub-network found at level $k + 1$ and $\ell_i \in \vec{\ell}$.

The definition of local central nodes is (for $k < n$)

$$C^{k,n}((a_k, b_k) \stackrel{\text{def}}{=} a_k(S).\overline{b_k}\langle f(S) \rangle.b_k(z).\overline{a_k}\langle z \rangle.(Q_c(z) \,|\, C^k(a_k, b_k))$$

Finally, at level n the information collected from all the leaf nodes reaches the central entity, where the final processing of the received information is performed. The description of the entire network D^n is $D^{n,n}_m(a_0, \ell_c)$, defined as

$$\begin{aligned} & D^{n,n}(a'_n, a''_n, b'_n, b''_n, \ell_n) \\ & \stackrel{\text{def}}{=} (\nu \vec{\ell}, a'_{n-1} : \beta, a''_{n-1} : \beta, b'_{n-1} : \beta, b''_{n-1} : \beta) \\ & \left(\prod_{\mathcal{L}_i \in \mathsf{Locs}(\mathcal{L},n)} D^{1,n}_{m'_i}(a'_{n-1}, a''_{n-1}, b'_{n-1}, b''_{n-1}, \mathcal{L}_i, \ell_n) \mid \right. \\ & \left. \quad \prod_{i=1}^{m} \ell_i \triangleright \ell_0 \mid \ell_0[C^n(a'_n, a''_n, b'_n, b''_n)] \right) \end{aligned}$$

Here the central entity will collect the estimates from its children and then broadcast back the value of the selection.

$$C^{n,n}(a'_n, a''_n, b'_n, b''_n) \stackrel{\text{def}}{=} a'_n(S).\overline{b'_n}\langle f(S) \rangle.C^{n,n}(a'_n, a''_n, b'_n, b''_n)) \tag{1}$$

4 The semantics of BBC

In this section we give a reduction semantics of BBC.

4.1 Evaluation of message terms

In our semantics, we rely on an evaluation relation $\rightsquigarrow$ defined for both message terms and multiset expressions.

Example 2. *Assume that our set of function symbols contains the projection function which extracts the first coordinate of a pair of names and that this is the only function symbol that can lead to evaluation. The evaluation relation can then be defined by the axiom*

$$\mathit{first}(x, y) \rightsquigarrow x$$

and the rules

$$\frac{M \rightsquigarrow M'}{\{\!|\, M, \dots \,|\!\} \rightsquigarrow \{\!|\, M', \dots \,|\!\}} \qquad \frac{M \rightsquigarrow M'}{f(M) \rightsquigarrow f(M')} \qquad \frac{E \rightsquigarrow E'}{g(E) \rightsquigarrow g(E')}$$

A message term M is *normal* if $M \not\rightsquigarrow M_1$ for any M_1.

4.2 Structural congruence and normal form

Structural congruence $\equiv$ is defined for both processes and networks; two processes (or networks) are related, if they are identical up to simple structural modifications such as the ordering of parallel components.

Most axioms defining $\equiv$ are omitted here, as they are standard and identical to those for the distributed π-calculus [7]. The most interesting axiom is

$$l[P \mid Q] \equiv l[P] \mid l[Q]$$

since this lets us fuse located processes that have the same location name.

By using the laws of structural congruence, any network can be rewritten to normal form. Informally, a network is in normal form if it consists of the total neighborhood information as one parallel component and the location information as the other. In the following, the notation $\prod_{i \in I} P_i$ is used to denote iterated parallel composition, i.e. the parallel composition of $P_1, \dots, P_k$, where $I = \{1, \dots, k\}$.

Definition 1. *A network N is in* normal form *if $N = (\nu \vec{m})(\mathcal{C} \mid \prod_{k \in K} l_k[P_k])$ where $\mathcal{C} = \prod_{i \in I} \prod_{j \in I \setminus \{i\}} l_i \triangleright m_j$.*

We write $l \triangleright m \in \mathcal{C}$ if $\mathcal{C} \equiv l \triangleright m \mid \mathcal{C}'$ for some $\mathcal{C}'$. It is easy to see that any network N can be rewritten into normal form.

Theorem 1. *For any network N, there exists a network N_1 such that $N_1 \equiv N$ and N_1 is on normal form.*

Proof. Induction in the structure of N. The proof is similar to that for the π-calculus [8]; the idea is to use the scope extension axioms to push out restrictions while α-converting bound names whenever needed. □

4.3 The reduction relation

We now present a reduction semantics of BBC. We assume that free names have their capacity found given by a function b : **Names** $\times$ **Names** $\to \mathbb{N}$ and that $b(a, l) = k$ denotes that there are at most k recipients that can receive a message broadcast on channel a. In our reduction rules, we assume that network terms are on normal form, as defined above. The rules are given in Table 2. We call a location that contains an available input a *receiving location* for channel a and a location which contains an available output a *sending location.* For broadcast, we require that if the number of receiving locations for a channel a exceeds the bound $b(a, l)$ for a (or

$$\begin{array}{ll}
 & (\nu\vec{n} : \vec{\beta})(\mathcal{C} \mid l[\overline{a}\langle M\rangle.P_k] \mid \prod_{i=1}^{k} m_i[a\,(\lambda\vec{x}M_i')\,Q_i] \mid N_1) \\
(\textsc{R-Broadcast}) & \longrightarrow \\
 & (\nu\vec{n} : \vec{\beta})(\mathcal{C} \mid l[P_k] \mid \prod_{i=k'+1}^{k} m_i[a\,(\lambda\vec{x}M_i')\,Q_i] \mid \prod_{i=1}^{k'} m_\ell[Q_i\theta_i] \mid N_1) \\
 & \text{where } l \rhd m_i \in \mathcal{C} \text{ for all } 1 \leq i \leq k \text{ and either} \\
 & \quad (1)\ a \notin \vec{n},\ k' \leq |\{m_\ell \mid l_k \rhd m_\ell \in \mathcal{C}\}| \leq b(a,l) \text{ for all } \ell \in L, \text{ or} \\
 & \quad (2)\ a \in \vec{n},\ k' \leq |\{m_\ell \mid l_k \rhd m_\ell \in \mathcal{C}\}| \leq \beta_a(l) \text{ for all } \ell \in L, \\
 & \text{and for all } 1 \leq i \leq k' \text{ we have } M = M_i'\theta_i \text{ for some } \theta_i, \\
 & (\nu\vec{n} : \vec{\beta})(\mathcal{C} \mid \mathcal{D} \mid l[\overline{a}\langle M\rangle.P \mid a\,(\lambda\vec{x}M')\,Q \mid P'] \mid N_1) \\
(\textsc{R-Local}) & \longrightarrow \\
 & (\nu\vec{n} : \vec{\beta})(\mathcal{C}_1 \mid \mathcal{D}_1 \mid l_k[P \mid Q\theta \mid P'] \mid N_1) \\
 & \text{where } M = M'\theta \text{ for some } \theta \\
 & \\
 & (\nu\vec{n} : \vec{\beta})(\mathcal{C} \mid \prod_{k\in K} l_k[\overline{a}\langle M_k\rangle.P_k \mid P_k'] \mid m[a\,[\lambda\vec{x}M.S\,]Q_\ell] \mid N_1) \\
(\textsc{R-Collect}) & \longrightarrow \\
 & (\nu\vec{n} : \vec{\beta})(\mathcal{C} \mid \prod_{k\in K} l_k[P_k \mid P_k'] \mid m[Q_\ell[S \mapsto \{\!|\, M_1, \ldots, M_{|K|} \,|\!\}] \mid N_1) \\
 & \text{where } l_j \rhd m \in \mathcal{C} \text{ for all } j \in K \text{ and either} \\
 & \quad (1)\ a \notin \vec{n}, \text{ and } 1 \leq |K| \leq b(a,m) \text{ or} \\
 & \quad (2)\ a \in \vec{n} \text{ and } 1 \leq |K| \leq \beta_a(m) \\
 & \text{and for all } j \in K \text{ we have } M_j = M\theta_\ell \text{ for some } \theta_\ell
\end{array}$$

Table 2: Reduction rules for communication in BBC networks on normal form, assuming connectivity $b(a,m)$

$\beta_a(m)$, if a is a bound name), then at most $b(a,l)$ receiving locations can receive the message. All other receiving locations do not receive anything and will still be waiting for an input. Each of the receiving locations must be connected to the sending location l. This is captured by the rule (R-Broadcast).

For collection, the number of locations that can simultaneously send messages on a channel a cannot exceed the bound $b(a,l)$ for a (or $\beta_a(m)$, if a is a bound name). Moreover, each sending location must be connected to the receiving location m. This is captured by the rule (R-Collect). Finally, the reduction rule (R-Local) describes local communication within the confines of a single location.

Example 3. *Consider the network*

$$N = l_1 \rhd l_3 \mid l_2 \mid l_3 \mid l_1[\overline{a}\langle(a,b)\rangle]paral_2[\overline{a}\langle(c,b)\rangle] \mid l_3[a\,[\lambda x(x,b).S]\,.\overline{d}\langle \textit{find-a}(S)\rangle]$$

Assume that $b(a) = 2$. This network consists of three locations. Location l_1 offers an output on the a-channel of the pair (a,b), and location l_2 offers an output on the a-channel of the pair (c,b). At location l_3 we have a process that on the channel a will receive a set of messages, all of which are of the form (x,b) for some x, and subsequently output a if one of the pairs received contained a.

We have $N \to l_1 \triangleright l_3 \mid l_2 \mid l_3 \mid l_1[\mathbf{0}] \mid l_2[\mathbf{0}] \mid l_3[\overline{d}\langle a\rangle]$.

5 Bisimilarity in BBC

5.1 Barbs

In our treatment of bisimilarity, we define an observability predicate (aka barbs) We write $P \downarrow_{a,\mathsf{B}}$ if P admits a broadcast observation on channel a, and $\vdash P \downarrow_{a,\mathsf{C}}$ if P admits a collection observation on channel a. We can define a similar predicate for networks. We write $N \downarrow_{a,d} @l$ if N allows a barb $\downarrow_{a,d}$ at location l. Table 3 contains the most interesting rules.

(Inp-B)	$a\,(\lambda\vec{x}M\,)P_1 \downarrow_{a,\mathsf{B}}$		(Inp-C)	$a\,[\lambda\vec{x}M.S\,]P_1 \downarrow_{a,\mathsf{C}}$
(Outp)	$\overline{a}\langle M\rangle.P \downarrow_{a,d}$		(Par)	$\dfrac{P_1 \downarrow_{a,d}}{P_1 \mid P_2 \downarrow_{a,d}}$
(New)	$\dfrac{P_1 \downarrow_{a,d}}{(\nu x:\beta_n)P_1 \downarrow_{a,d}}$	$x \neq a$	(Locate)	$\dfrac{P \downarrow_{a,d}}{l[P] \downarrow_{a,d} @\ell}$
(Cong)	$\dfrac{N_1 \downarrow_{a,d} @l \quad N_1 \equiv N}{N \downarrow_{a,d} @l}$			

Table 3: Selected rules for barbs

Definition 2. *A weak barbed bisimulation R is a symmetric binary relation on networks which satisfies that whenever $N_1 R N_2$ we have*

1. *If $N_1 \to N_1'$ then for some N_2' we have $N_2 \to^* N_2'$ where $N_1' R N_2'$*
2. *For every location l, if $N_1 \downarrow_{a,d} @l$ then also $N_2 \downarrow_{a,d} @l$*

We write $N_1 \stackrel{\cdot}{\approx} N_2$ if $N_1 R N_2$ for some weak barbed bisimulation R.

Theorem 2. *Weak barbed bisimilarity is an equivalence.*

As in other process calculi, the notion of barbed bisimilarity is unfortunately not a congruence, as it is not preserved under parallel composition.

Two networks are weak barbed congruent if they are barbed bisimilar in every parallel context, i.e. no surrounding network can tell them apart.

Definition 3. *We write $N_1 \approx N_2$ if for all networks N we have that $N_1 \mid N \stackrel{\cdot}{\approx} N_2 \mid N$*

6 Correctness of the hierarchical protocol

Let us now return to the hierarchical protocol from Section 3. We would like the multi-level network that serves K leaf nodes to be equivalent to a single-level network containing the same K leaf nodes but now connected to a single central entity.

To this end we define the flattened version of a hierarchical network N, denoted $\mathsf{flat}(N, \ell)$. This is precisely a network in which all nodes are connected to the same central node at location ℓ.

This operation is defined inductively as follows.

$$\mathsf{flat}(D^{0,n}(a'_0, a''_0, b'_0, b''_0, \mathcal{L}, \ell_0), \ell) \stackrel{\text{def}}{=}$$

$$D^{0,n}(a'_0, a''_0, b'_0, b''_0, \mathcal{L}, \ell_0)\mathsf{flat}(D^{k+1,n}(a'_{k+1}, a''_{k+1}, b'_{k+1}, b''_{k+1}, \mathcal{L}, \ell_{k+1}), \ell) \stackrel{\text{def}}{=}$$

$$(\nu\vec{\ell}, a'_k, a''_k, b'_k, b''_k)\left(\prod_{\mathcal{L}_i \in \mathsf{Locs}(\mathcal{L},k)}^{m} \mathsf{flat}(D^{k,n}(a'_k, a''_k, b'_k, b''_k, \mathcal{L}_i, \ell_i)) \mid \prod_{i=1}^{m} \ell_i \rhd \ell\right) \text{ where } 0 < k < n$$

Theorem 3. *For every $n \geq 0$, for every set of distinct names $\{a'_0, a''_0, b'_0, b''_0\}$ and set of locations $\mathcal{L}$ we have*

$$D^{0,n}(a'_0, a''_0.b'_0, b''_0, \mathcal{L}, \ell) \stackrel{\cdot}{\approx} \mathit{flat}(D^{0,n}(a'_0, a''_0, b'_0, b''_0, \mathcal{L}, \ell), \ell)$$

7 The expressive power of BBC

There is no compositional encoding of BBC into the π-calculus. In the presence of broadcast this follows from the negative result due to Ene and Muntean [1]. However, the result also holds if we only consider collection. The following criteria for compositionality are due to Palamidessi [5] and Ene and Muntean [1].

Definition 4. *An encoding $[\![\ ]\!]$ is* compositional *if*

1. $[\![N_1 \mid N_2]\!] = [\![N_1]\!] \mid [\![N_2]\!]$.
2. *For any substitution σ we have $[\![N\sigma]\!] = [\![N]\!]\sigma$*
3. *If $N \to^* N'$ then $[\![N]\!] \to^* [\![N']\!]$.*
4. *If $[\![N]\!] \to^* M$ then $[\![N]\!] \to^* M \to^* [\![N']\!]$ for some N'.*

A central notion in the study of process calculi is that of an *electoral system*. This is a network in which the participants perform a computation that elects a unique leader. Following [5, 1], our definition assumes a network in which the names used are the 'natural names' that represent the identity of the n processes in the network.

Definition 5 (Electoral system)**.** *A network $N = P_1 \mid \cdots \mid P_n$ is an* electoral system *if for every computation C, there exists an extension C' of C, and an index $k \in \{1, \ldots, n\}$ such that for every $i \in \{1, \ldots, n\}$ the projection C'_i contains exactly one output action of the form $\overline{k}$ and any trace of a P_i may contain at most one action of the form $\overline{l}$ with $l \in \{1, \ldots, n\}$.*

We now describe such a system in BBC. The system uses a common channel a whose bound is n, where n is the number of principals. The idea is to collect names until n sets of names have been collected. The first to do so sends out a success announcement to everyone in the form of a name $\mathsf{chosen}(m)$; the collection function $g(S)$ is defined to select the name with the minimal index among the names of S unless S contains one or more occurrences of a name of the form $\mathsf{chosen}(m')$ for some m. Thus, the first component to receive sufficiently many names from all other components chooses the leader.

Our network is of the form

$$N = \prod_{i=1}^{n} l[l_i]P_i \mid \prod_{i=1}^{n} \prod_{j=1, j \neq i}^{m} l_i \rhd l_j$$

We define the n components as follows:

$$P_i \stackrel{\mathrm{def}}{=} \overline{a}\langle i \rangle . a\,[\lambda x.S_1\,] \ldots a\,[\lambda x.S_k\,]\overline{a}\langle \mathsf{chosen}(g(\{\{g(S_1), \ldots g(S_k)\}\}))\rangle . P$$

The selection function is defined by

$$g(S) = \begin{cases} \min_{x \in S} x & \text{if } S \subseteq \{1, \ldots, n\} \\ \mathsf{chosen}(n) & \text{if } \mathsf{chosen}(n) \in S \end{cases}$$

As there is no such electoral system for the π-calculus [5], we can use the same proof as that given by Ene and Muntean to conclude that there can be no composition encoding of BBC in the π-calculus.

8 Directions for further work

In this paper we have presented the BBC calculus, which is a distributed process calculus that generalizes the π-calculus with notions of channels with bounded forms of broadcast and collection and an explicit notion of connectivity.

The present work only considers barbed bisimulation; it is well-known that this relation is not preserved by parallel composition and that the notion of barbed congruence does not lend itself well to co-inductive proof techniques. Further work includes finding a labeled characterization of barbed congruence; this requires a labeled transition semantics in the spirit of [1].

At present, we informally distinguish between channels for broadcast and collection. An interesting direction of work is to develop a type system that will allow the same channel to be used in both modes and according to a protocol. This is similar to the idea of dyadic session types [2], and a further step in this direction is to study multiparty session types [3] and how projection to dyadic session types can defined in the setting of BBC.

References

[1] Cristian Ene and Traian Muntean. A broadcast-based calculus for communicating systems. In *Proceedings of IPDPS'1*. IEEE Computer Society, 2001.

[2] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[3] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *POPL*, pages 273–284. ACM, 2008.

[4] Dimitrios Kouzapas and Anna Philippou. A process calculus for dynamic networks. In Roberto Bruni and Juergen Dingel, editors, *Formal Techniques for Distributed Systems*, volume 6722 of *Lecture Notes in Computer Science*, pages 213–227. Springer Berlin Heidelberg, 2011.

[5] Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *MSCS*, 13(5):685–719, 2003.

[6] K. V. S. Prasad. A calculus of broadcasting systems. *Sci. Comput. Prog.*, 25(2-3):285–327, 1995.

[7] James Riely and Matthew Hennessy. A typed language for distributed mobile processes (extended abstract). In *POPL*, pages 378–390, 1998.

[8] Davide Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes*, Cambridge University Press, 2001.

[9] Anu Singh, C. R. Ramakrishnan, and Scott A. Smolka. A process calculus for mobile ad hoc networks. *Science of Computer Programming*, 75(6):440–469, 2010.

Communicating machines as a dynamic binding mechanism of services*

Ignacio Vissani[1], Carlos Gustavo Lopez Pombo[1,2], and Emilio Tuosto[3]

[1] Department of computing, School of Science, Universidad de Buenos Aires
[2] Consejo Nacional de Investigaciones Científicas y Tecnolópicas
[3] Department of Computer Science, University of Leicester

Abstract

Distributed software is becoming more and more dynamic to support applications able to respond and adapt to the changes of their execution environment. For instance, *service-oriented computing* (SOC) envisages applications as services running over globally available computational resources where discovery and binding between them is transparently performed by a middleware. *Asynchronous Relational Networks* (ARNs) is a well-known formal orchestration model, based on hypergraphs, for the description of service-oriented software artefacts. Choreography and orchestration are the two main design principles for the development of distributed software. In this work, we propose *Communicating Relational Networks* (CRNs), which is a variant of ARNs, but relies on choreographies for the characterisation of the communicational aspects of a software artefact, and for making their automated analysis more efficient.

1 Introduction and motivation

Distributed software is becoming more and more dynamic to support applications able to respond and adapt to the changes of their execution environment. For instance, *service-oriented computing* (SOC) envisages applications as services running over globally available computational resources; at run-time, services search for other services to bind to and use. Software architects and programmers have no control as to the nature of the components that an application can bind to due to the fact that the discovery and binding are transparently performed by a middleware.

Choreography and orchestration are the two main design principles for the development of distributed software (see e.g., [6]). Coordination is attained in the latter case by an *orchestrator*, specifying (and possibly executing) the distributed work-flow. Choreography features the notion of *global view*, that is a holistic specification describing distributed interactions amenable of being "projected" onto the constituent pieces of software. In an orchestrated model, the distributed computational components coordinate with each other by interacting with a special component, *the orchestrator*, which at run time decides how the work-flow has to evolve. For example the orchestrator of a service offering the booking of a flight and a hotel may trigger a service for hotel and one for flight booking in parallel, wait for the answers of both sites, and then continue the execution. In a choreographed model, the distributed components autonomously execute and interact with each other on the basis of a local control flow expected to comply with their role as specified in the "global viewpoint". For example, the choreography of hotel-flight booking example above could specify that the flight service interacts with the hotel service which in turns communicates the results to the buyer.

We use Asynchronous relational networks (ARNs) [8] as the basis of our approach. In ARNs, systems are formally modelled as hypergraphs obtained by connecting hyperarcs which represent unit of

*This work has been supported by the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS)

computations and communication. More precisely, hyperarcs are interpreted as either processes (*services* or unit of computation) or as communication channels (unit of communication). The nodes can only be adjacent to: 1. 1. one process hyperarc and one communication hyperarc, meaning that the computation formalised by the process hyperarc is bound through the communication channel formalised by the communication hyperarc, 2. one process hyperarc, meaning that it is a *provides-points* through which the computation formalised by the process hyperarc can be bound to and activity that requires that particular service, or 3. one communication hyperarc, meaning that it is a *requires point* to which a given service can be bound using one of its provides-points. The rationale behind this separation is that a provides-point yields the interface through which a service exports its functionality while a requires-point is the interface through which an activity expects certain service to provide a functionality. Composition of services can then be understood as fusing a provides-point with a requires-point in a way that the service exported by the former satisfy the expectations of the latter, usually formalised as contracts in some formal language.

Hyperarcs are labelled with (Müller) automata; in the case of process hyperarcs, automata formalise the interactions carried out by that particular service while, in the case of communication hyperarcs, they represent the orchestrator coordinating the behaviour of the participants of the communication. In fact, the automaton Λ associated to a communication hyperarc coordinates the processes bound to its ports by, at each time, interacting with one of the processes and deciding, depending on the state Λ is in, what is the next interaction (if any) to execute. The global behaviour of the system is then obtained by composing the automata associated to process and communication hyperarcs. In the forthcoming sections we will introduce a running example to show how definitions work and concretely discuss the contributions of the present work.

As anticipated, the composition of ARNs yields a semantic definition of a binding mechanism of services in terms of "fusion" of provides-points and requires-points. Once coalshed, the nodes become "internal", that is they are no longer part of the interface and cannot be used for further bindings. In existing works, like [8], the binding is subject to an entailment relation between *linear temporal logic* [7] theories attached to the provides- and requires-points that can be checked by resorting to any decision procedure for LTL (for example, [4])

Although the orchestration model featured by ARNs is rather expressive and versatile, we envisage two drawbacks:

1. the binding mechanism based on LTL-entailment establishes an asymmetric relation between requires-point and provides-point as it formalises a notion of trace inclusion; also,
2. including explicit orchestrators (the automaton labelling the communication hyperarcs), in the composition, together with the computational units (the automaton labelling the process hyperarcs) increases the size of the resulting automaton making the analysis more expensive.

In the present work we propose *Communicating Relational Networks* (CRNs), a variant of ARNs relying on choreographies to overcome those issues, where provides-points are labelled with *Communicating Finite State Machines* [2] declaring the behaviour (from the communication perspective) exported by the service, and communication hyperarcs are labelled with *Global Graphs* [3] declaring the global behaviour of the communication channel. In this way, our proposal blends the orchestration framework of ARNs with a choreography model based on global graphs and communicating machines. Unlike most of the approaches in the literature (where choreography and orchestration are considered antithetical), we follow a comprehensive approach showing how choreography-based mechanisms could be useful in an orchestration model.

The present work is organised as follows; in Section 2 we provide the formal definitions of most of the concepts used along this paper. Such definitions are illustrated with a running example introduced in Section 3. In Section 4 we introduce the main contribution of this paper, being the definition of CRNs, we show how they are used to rewrite the running example and we discuss several aspects regarding the design-time checking to assert internal coherence of services, the run-time checking ruling the binding mechanism and the cost of software analysis. Finally, in Section 5 we draw some conclusions and discuss some further research directions.

2 Preliminaries

In this section we present the preliminary definitions used throughout the rest of the present work. We first summarise communicating machines and global graphs borrowing definitions from [5] and from [3]. Finally we introduce some basic definitions in order to present ARNs; the definition here are adapted from [8].

2.1 Communicating machines and global graphs

Communicating machines were introduced in [2] to model and study communication protocols in terms of finite transition systems capable of exchanging messages through some channels. We fix a finite set Msg of *messages*, a finite set P of participants

Definition 1 ([2]). *A* communicating finite state machine *on* Msg *(CFSMs, for short) is a finite transition system* $(\mathsf{Q}, \mathsf{C}, q_0, \mathsf{Msg}, \delta)$ *where*

- Q *is a finite set of states;*
- $\mathsf{C} = \{\mathsf{pq} \in \mathsf{P}^2 \mid \mathsf{p} \neq \mathsf{q}\}$ *is a set of channels;*
- $q_0 \in \mathsf{Q}$ *is an initial state;*
- $\delta \subseteq \mathsf{Q} \times (\mathsf{C} \times \{!, ?\} \times \mathsf{Msg}) \times \mathsf{Q}$ *is a finite set of* transitions.

A communicating system *is a map* S *assigning a CFSM* $S(\mathsf{p})$ *to each* $\mathsf{p} \in \mathsf{P}$*. We write* $q \in S(\mathsf{p})$ *when* q *is a state of the machine* $S(\mathsf{p})$ *and likewise and* $t \in S(\mathsf{p})$ *when* t *is a transition of* $S(\mathsf{p})$*.*

The execution of a system is defined in terms of transitions between configurations as follows:

Definition 2. *The* configuration *of communicating system* S *is a pair* $s = (\vec{q}, \vec{w})$ *where* $\vec{q} = (q_{\mathsf{p}})_{\mathsf{p} \in \mathsf{P}}$ *where* $q_{\mathsf{p}} \in S(\mathsf{p})$ *for each* $\mathsf{p} \in \mathsf{P}$ *and* $\vec{w} = (w_{\mathsf{pq}})_{\mathsf{pq} \in \mathsf{C}}$ *with* $w_{\mathsf{pq}} \in \mathsf{Msg}^{\star}$*. A configuration* $s' = (\vec{q'}, \vec{w'})$ *is* reachable *from another configuration* $s = (\vec{q}, \vec{w})$ *by the* firing of the transition t *(written* $s \xrightarrow{t} s'$*) if there exists* $\mathsf{m} \in \mathsf{Msg}$ *such that either:*

1. $t = (q_{\mathsf{p}}, \mathsf{pq}!\mathsf{m}, q'_{\mathsf{p}}) \in \delta_{\mathsf{p}}$ *and*
   (a) $q'_{\mathsf{p'}} = q_{\mathsf{p'}}$ *for all* $\mathsf{p'} \neq \mathsf{p}$*; and*
   (b) $w'_{\mathsf{pq}} = w_{\mathsf{pq}} \cdot \mathsf{m}$ *and* $w'_{\mathsf{p'q'}} = w_{\mathsf{p'q'}}$ *for all* $\mathsf{p'q'} \neq \mathsf{pq}$*; or*
2. $t = (q_{\mathsf{q}}, \mathsf{pq}?\mathsf{m}, q'_{\mathsf{q}}) \in \delta_{\mathsf{q}}$ *and*
   (a) $q'_{\mathsf{p'}} = q_{\mathsf{p'}}$ *for all* $\mathsf{p'} \neq \mathsf{q}$*; and*
   (b) $\mathsf{m} \cdot w'_{\mathsf{pq}} = w_{\mathsf{pq}}$ *and* $w'_{\mathsf{p'q'}} = w_{\mathsf{p'q'}}$ *for all* $\mathsf{p'q'} \neq \mathsf{pq}$

A *global graph* is a finite graph whose nodes are labelled over the set $\mathsf{L} = \{\bigcirc, \circledcirc, \oplus, \boxdot\} \cup \{\mathsf{s} \to \mathsf{r} : \mathsf{m} \mid \mathsf{s}, \mathsf{r} \in \mathsf{P} \wedge \mathsf{m} \in \mathsf{Msg}\}$ according to the following definition.

Definition 3. *A* global graph *(over* P *and* Msg*) is a labelled graph* $\langle V, A, \Lambda \rangle$ *with a set of* vertexes V*, a set of* edges $A \subseteq V \times V$*, and* labelling function $\Lambda : V \to \mathsf{L}$ *such that* $\Lambda^{-1}(\bigcirc)$ *is a singleton and, for each* $v \in V$

1. *if $\Lambda(v)$ is of the form* $\mathsf{s} \to \mathsf{r} : \mathsf{m}$ *then v has a unique incoming and unique outgoing edges,*
2. *if $\Lambda(v) \in \{\diamond, \boxplus\}$ then v has at least one incoming edge and one outgoing edge and,*
3. *$\Lambda(v) = \circledcirc$ then v has zero outgoing edges.*

Label $\mathsf{s} \to \mathsf{r} : \mathsf{m}$ represents an interaction where machine s sends a message m to machine r. A vertex with label $\bigcirc$ reperesents the source of the global graph, $\circledcirc$ represents the termination of a branch or of a thread, $\boxplus$ indicates forking or joining threads, and $\diamond$ marks vertexes corresponding to branch or merge points, or to entry points of loops.

In the following we use a projections algorithms that given a global graph retrieves communicating machines for each of its participants. Undestranding such algorithm is not necessary for the sake of this paper and the interested reader is referred to [5] for its definition.

2.2 Asynchronous relational networks

A Müller automaton is a finite state automaton where final states are replaced by a family of states to define an acceptance condition on infinite words.

Definition 4 (Müller automaton)**.** *A* Müller automaton *over a finite set A of* actions *is a structure of the form $\langle Q, A, \Delta, I, \mathcal{F}\rangle$, where*

1. *Q is a finite set (of* states*)*
2. *$\Delta \subseteq Q \times A \times Q$ is a* transition relation *(we write $p \xrightarrow{\iota} q$ when $(p, \iota, q) \in \Delta$),*
3. *$I \subseteq Q$ is the set of* initial states*, and*
4. *$\mathcal{F} \subseteq 2^Q$ is the set of* final-state sets.

We say that an automaton accepts *an inifinite trace $\omega = q_0 \xrightarrow{\iota_0} q_1 \xrightarrow{\iota_1} \dots$ if and only if $q_0 \in I$ and there exists $i \geq 0$ and $S \in \mathcal{F}$ such that for all $s \in S$, the set $\bigcup_{i \leq j \wedge q_j = s}\{j\}$ is infinite.*

Asynchronous relational networks are hypergraphs connecting *ports* that can be thought of as communication end-points through which messages can be sent to or received from other ports.

Definition 5 (Port)**.** *A* port *is a structure $\pi = \langle \pi^+, \pi^- \rangle$ where π^+, π^- are disjoint finite sets of messages. We say that two ports are disjoint when they are formed by componentwise disjoint sets of messages. The* actions over *π are $A_\pi = \{m! \mid m \in \pi^-\} \cup \{m¡ \mid m \in \pi^+\}$.*

The computational agents of ARNs are *processes* formalised as a set of ports togetherr with a Müller automaton describing the communication pattern of the agents.

Definition 6 (Process)**.** *A* process *$\langle \gamma, \Lambda \rangle$ consists of a set γ of pairwise disjoint ports and a Müller automaton Λ over the set of actions $A_\gamma = \bigcup_{\pi \in \gamma} A_\pi$.*

Processes are connected through *connections* whose basic role is to establish relations among the messages that processes exchange on the ports of processes and communication hyperedges. Intuitively, one can thing of the messages used by processes and communication hyperedges as 'local' messages whose 'global' meaning is established by connections.

Definition 7 (Connection)**.** *Given a set of pairwise disjoint ports γ, an* attachment injection *on γ is a pair $\langle M, \mu \rangle$ where and a finite set M of messages and $\mu = \{\mu_\pi\}_{\pi \in \gamma}$ is a family of finite partial injections $\mu_\pi : M \rightharpoonup \pi^- \cup \pi^+$. We say that $\langle M, \mu, \Lambda \rangle$ is a* connection *on γ iff $\langle M, \mu \rangle$ is an attachment injection on γ and a Müller automaton Λ over $\{m! \mid m \in M\} \cup \{m¡ \mid m \in M\}$ such that:*

$$\mu_\pi^{-1}(\pi^-) \subseteq \bigcup_{\hat{\pi} \in \gamma \setminus \{\pi\}} \mu_{\hat{\pi}}^{-1}(\hat{\pi}^+) \qquad \textit{and} \qquad \mu_\pi^{-1}(\pi^+) \subseteq \bigcup_{\hat{\pi} \in \gamma \setminus \{\pi\}} \mu_{\hat{\pi}}^{-1}(\hat{\pi}^-).$$

for each $\pi \in \gamma$.

Definition 8 (Asynchronous Relational Network [8])**.** *Let M be a finite set of messages. An* asynchronous relational net α *on M is a structure $\langle X, P, C, \{\pi_x\}_{x\in X}, \{\mu_c\}_{c\in C}, \{\gamma\}_{x\in X}, \{\Lambda_e\}_{e\in P\cup C}\rangle$ where*

- *$\langle X, P\cup C\rangle$ is a hypergraph, with X is a (finite) set of vertexes, P is a set of* hyperedges *(non-empty subsets of X)* computation hyperedges*, and C is a set of* communication hyperedges *such that X, P, and C are pairwise disjoint, no adjacent hyperedges belong to the same partition,*
- *three labelling functions that assign* (a) *a port π_x with messages in M to each point $x \in X$,* (b) *a process $\langle\gamma_p, \Lambda_p\rangle$ to each hyperedge $p \in P$ such that $\gamma_p \subseteq \{\pi_x\}_{x\in X}$, and* (c) *a connection $\langle M_c, \mu_c, \Lambda_c\rangle$ to each hyperedge $c \in C$.*

An ARN with no provides-point is called activity *and formalises the notion of a software artefact that can execute, while an ARN that has at least one provides-point is called a* service *and can only execute provided it is bound through one of them to a requires-point of an* activity.

3 The running example

The following running example will help us to present intuitions behind the definitions, and later, to introduce and motivate our contributions. Consider an application providing the service of hotel reservation and payment processing. A client activity TravelClient asks for hotel options made available by a provider HotelsService returning a list of offers. If the client accepts any of the offers, then HotelsService calls for a payment processing service PaymentProcessService which will ask the client for payment details, and notify HotelsService whether the payment was accepted or rejected. Finally, HotelsService notifies the outcome of the payment process to the client.

Figures 1, 2, and 3 show the ARNs (including the automata), for the TravelClient, HotelsService, and PaymentProcessService respectively. The ARN in Fig. 1(a) represents an activity composed with a communication channel. More precisely, TravelClient (in the solid box on the left) represents a process hyperedge whose Müller automaton is Λ_{TC} (depicted in Fig. 1(b)). The solid "y-shaped" contour embracing the three dashed boxes represents a communication hyperedge used to specify the two requires-points (i.e., HS and PPS) of the component necessary to fulfill its goals. Note that such ARN does not provide itself any service to other components and that the dashed box lists the outgoing and incoming messages expected (respectively denoted by names prefixed by '+' and '-' signs).

It is worth remarking that communication hyperarcs in ARNs yield the coordination mechanism among a number of services. In fact, a communication hyperarc enables the interaction among the services that bind to its requires-points such as TravelClient, HotelsService, and PaymentProcessService in our example. The coordination is specified through a Müller automaton associated with the communication hyperarc that acts as the orchestrator of the servises. In our running example, the communication hyperarc of Fig. 1 is labeled with the automaton Λ_{CC} of Fig. 1(c) where, for readability and conciseness, the dotted and dashed edges stand for the paths

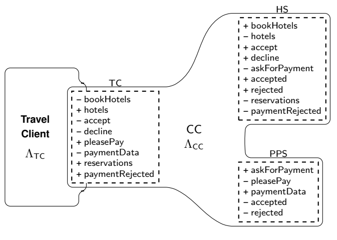
$$\xrightarrow{bookHotels!} \cdot \xrightarrow{bookHotels¡} \cdot \xrightarrow{hotels!} \cdot \xrightarrow{hotels¡}$$

and

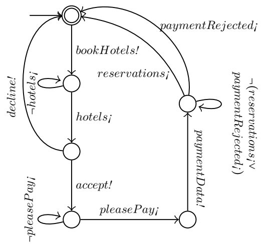
$$\xrightarrow{accept!} \cdot \xrightarrow{accept¡} \cdot \xrightarrow{askForPayment!} \cdot \xrightarrow{askForPayment¡} \cdot \xrightarrow{paymentData!} \cdot \xrightarrow{paymentData¡}$$

respectively. As we will see, such automaton corresponds to a global choreography when replacing the binding mechanism of ARNs with choreography-based mechanisms. The transitions of the automata are labelled with input/output actions; according to the usual ARNs notation, a label $m!$ stands for the ouput of message m while label $m¡$ stands for the input of message m.

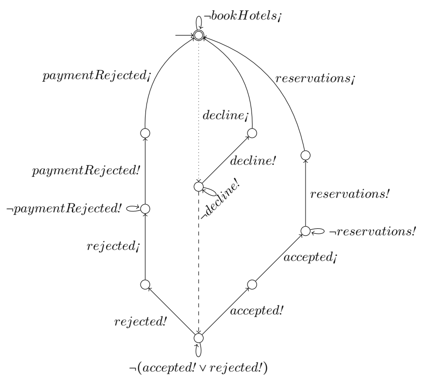
Figures 2 and 3 represent two services with their automata (resp. Λ_{HS} and Λ_{PPS}) and their provides-point (resp. HS and PPS) not bound to any communication channel yet.



(a) The TravelClient activity

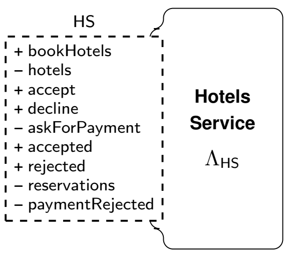


(b) Müller automaton Λ_{TC}

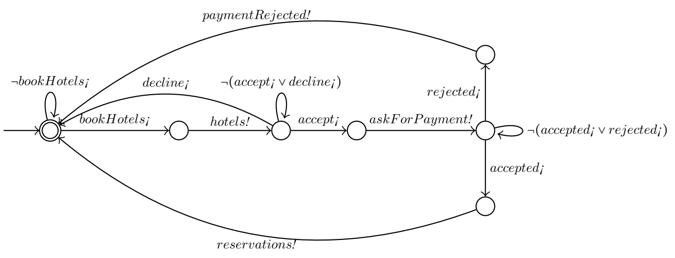


(c) Müller automaton Λ_{CC}.

Figure 1: The TravelClient activity together with the Müller automata.

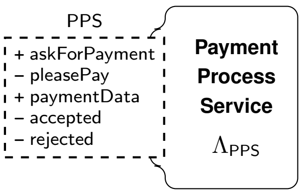


(a) The HotelsService participant

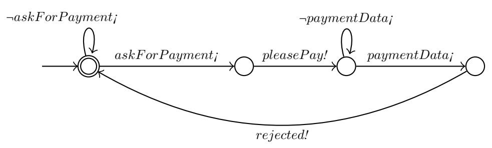


(b) Müller automaton Λ_{HS}

Figure 2: The HotelsService participant together with the machine Hs



(a) The PaymentProcessService participant



(b) Müller automaton Λ_{PPS}, that only reject paymens

Figure 3: The PaymentProcessService participant.

The composition of ARNs yields a semantic definition of a binding mechanism of services in terms of "fusion" of provides-points and requires-points. More precisely, the binding is subject to an entailment relation between *linear temporal logic* [7] theories attached to the provides- and requires-points as illustrated in the following.

4 Communicating Relational Networks

As we mentioned before, even when the orchestration model featured by ARNs is rather expressive and versatile, we envisage two drawbacks which now can be presented in more detail.

4.1 On the binding mechanism

If we consider the binding mechanism based on LTL entailment presented in previous works, the relation between requires-point and provides-point is established in an asymmetric way whose semantics is read as trace inclusion. This asymmetry leads to undesired situations. For instance, if we return to our running example, a contract stating that the outcome of an execution is either $accept$ or $reject$ of a payment could be specified by assigning the LTL formula

$$\Diamond((-accept \vee -reject) \wedge \neg(-accept \wedge -reject))$$

to the requires-point PPS of Fig. 1(a). Likewise, one could specify a contract for the provides-point PPS of the ARN in Fig. 3(b) stating that payments are always rejected by including the formula[1]

$$\Diamond(-reject \wedge \neg - accept)$$

It is easy to show that

$$\Diamond(-reject \wedge \neg - accept) \quad \vdash^{LTL} \quad \Diamond((-accept \vee -reject) \wedge \neg(-accept \wedge -reject))$$

by resorting to any decision procedure for LTL (see for instance, [4]). The intuition is that every state satisfying $-reject \wedge \neg - accept$ also satisfies $(-accept \vee -reject) \wedge \neg(-accept \wedge -reject)$ so if the former eventually happens, then also the latter.

The reader should note that this scenario leads us to accept a service provider that, even when it can appropriately ensure a subset of the expected outcomes, cannot guaranty that all possible outcomes will eventually be produced.

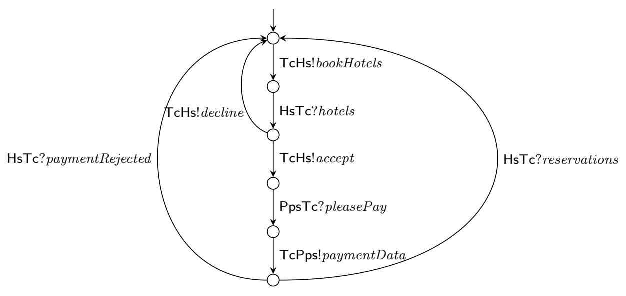
Communicating Relational Networks are defined exactly as ARNs but with definition of *Connection* based global graphs where, given a set of ports, the messages are related to the messages in the ports, and the participants are identified by the ports themselves.

Definition 9 (Connection)**.** *We say that* $\langle M, \mu, \Gamma\rangle$ *is a* connection *on* γ *iff* $\langle M, \mu\rangle$ *is an attachment injection on* γ *and* Γ *is a global graph where the set of participants is* $\{\mathsf{p}_\pi\}_{\pi \in \gamma}$ *exchanging messages in* M *such that:*

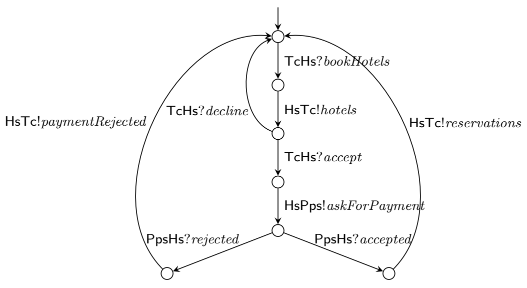
$$\mu_\pi^{-1}(\pi^-) \subseteq \bigcup_{\hat{\pi} \in \gamma \setminus \{\pi\}} \mu_{\hat{\pi}}^{-1}(\hat{\pi}^+) \qquad \text{and} \qquad \mu_\pi^{-1}(\pi^+) \subseteq \bigcup_{\hat{\pi} \in \gamma \setminus \{\pi\}} \mu_{\hat{\pi}}^{-1}(\hat{\pi}^-).$$

for each $\pi \in \gamma$.

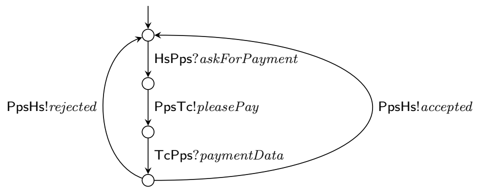
[1]In this examples we use two propositions, $accept$ and $reject$, forcing us to include in the specification their complementary behaviour, but making the formulae easier to read.



(a) Communicating machine for the port TC



(b) Communicating machine for the port HS



(c) Communicating machine for the port PPS

Figure 4: Communicating machines labelling the ports TC, HS and PPS.

Definition 10 (Communicating relational network)**.** *A* communicating relational net α *is a structure* $\langle X, P, C, \gamma, M, \mu, \Lambda \rangle$ *consisting of:*

- *a hypergraph* $\langle X, E \rangle$*, where* X *is a (finite) set of* points *and* $E = P \cup C$ *is a set of* hyperedges *(non-empty subsets of* X*) partitioned into* computation hyperedges $p \in P$ *and* communication hyperedges $c \in C$ *such that no adjacent hyperedges belong to the same partition, and*
- *three labelling functions that assign* (a) *a port* M_x *to each point* $x \in X$*,* (b) *a process* $\langle \gamma_p, \Lambda_p \rangle$ *to each hyperedge* $p \in P$*, and* (c) *a connection* $\langle M_c, \mu_c, \Lambda_c \rangle$ *to each hyperedge* $c \in C$*.*

Figures 4 and 5 show the communicating machines and global graphs that can be used to redefine of the same services of the running example presented in Sec. 2, but as CRNs.

The machine in Fig. 4(a) specifies that upon reception of a *bookHotel* message from the client, HotelsService sends back a list of *hotels*; if the client accepts then computation continues, otherwise the HotelsService returns to its initial state, etc.. Also, Figs. 4(b) and (c) depict the communicating machines associated to the provides-points of services HotelsService and PaymentProcessService, respectively. From the point of view of the requires-points, the expected behaviour of the participants of a communication is declared by means of a choreography associated to communication hyperarcs. We illustrate such graphs by discussing the choreography in Fig. 5 (corresponding to the automaton in Fig. 1(c)). The graph dictates that first client and HotelsService interact to make the request and receive a list of available hotels, then the client decides whether to accept or decline the offer, etc. Global graphs are a rather convenient formalism to express distributed choices (as well as parallel computations) of

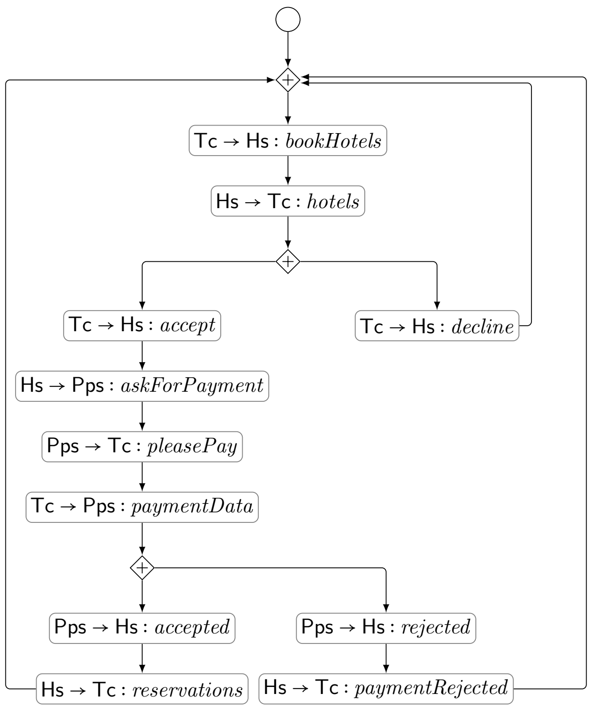


Figure 5: Global graph of the running example

work-flows. As we mentioned before, an interesting feature of global graph is that they can easily show branch/merge points of distributed choices; for instance, in the global graph of Fig. 5 branching points merge in the loop-back node underneath the initial node.

Based on Definition 10, we can define two new binding mechanisms by exploiting the "top-down" (projection) and "bottom-up" (synthesis) nature offered by choreographies.

Top-Down According to the first mechanism, provides-points are bound to require points when the projections of the global graph attached to the communication hyperarc are bisimilar to the corresponding communicating machine (exposed on the provides-points of services being evaluated for binding).

Bottom-Up The second mechanism is more flexible and it is based on a recent algorithm to synthesise choreographies out of communicating machines [5]. More precisely, one checks that the choreographies synthesised from the communicating machines, associated to the provides-points of services being evaluated for binding are isomorphic to the one labelling the communication hyperarc.

For example, the projections of the global graph of Fig. 5 with respect to the components HotelsService and PaymentProcessService yields the communicating machines in Figures 4(b) and 4(c) respectively; so, when adopting the first criterion, the binding is possible and it is guaranteed to be well-behaved (e.g., there will be no deadlocks or unspecified receptions [2]). Likewise, when adopting the second criterion, the binding is possible because the synthesis of the machines in Fig. 4 yields the global graph of Fig. 5.

In this way, our approach combines choreography and orchestration by exploiting their complementary characteristics at two different levels. On the one hand, services use global graphs to declare the behaviour expected from the composition of all the parties and use communicating machines to declare their exported behaviour. On the other hand, the algorithms available on choreographies are used for checking the run-time conditions on the dynamic binding.

The resulting choreography-based semantics of binding guarantees properties of the composition of services that are stronger than those provided by the traditional binding mechanism of ARNs, and yielding a more symmetric notion of interoperability between activities and services.

4.2 Comparison of the analysis and the binding mechanism

Among the many advantages of developing software using formal tools, is the possibility of providing analysis as a means to cope with (critical) requirements. This approach generally involves the formal description of the software artefact through some kind of contract describing its behaviour. As we mentioned before, in SOC, services are described by means of their contracts associated to their provide- and require-points, playing the role that in structured programming play post- and pre-conditions of functions, respectively. From this point of view, analysing a software artefact requires:

- the verification of the computational aspects of a service with respect to its contracts, yielding a *coherence condition*, whose checking takes place at design-time, and
- the verification of the satisfaction of a property by an activity with respect to a given service repository, yielding a *quality assessment* of the software artefact, whose checking takes place also at design-time.

On the other hand, service-oriented software artefacts require the run-time checking associated to the *binding mechanism*, in order to decide whether a given service taken from the repository provides the service required by an executing activity.

Table 1 shows a comparison of the procedures that have to be implemented for checking the coherence condition of a service, the quality assessment of a service-oriented software artefact with respect to a particular repository, and for obtaining a binding mechanism for both of the approaches, the one based on ARNs, and the one based on CRNs.

5 Concluding Remarks

We propose the use of communicating relational networks as a formal model for service-oriented software design. CRNs are a variant of ARNs that harnesses the orchestration perspective underlying ARNs with a choreography viewpoint for characterising the behaviour of participants (services) over a communication channel. The condition for binding a provides-points of services to the require points of a communication channel of an activity relies on checking the compliance of the local perspective of the process, declared as communicating machines, with the global view implicit in the choreography associated to the communication channel. The binding mechanisms of ARNs (i.e., the inclusion of the set of traces of the provides-point of the service bound in the set of traces allowed by the requires-point of the activity) yields an asymmetric acceptation condition. Our approach provides a more symmetric mechanism based on rely-guarantee types of contracts.

Our framework requires the definition of a criterion to establish the coherence among the Müller automaton Λ of a process hyperedge and the communicating machines associated to its provides-points. This criterion, checked only at design time, is the bisimilarity of the communicating machine projected from Λ and the ones associated to the provides-points. The reader familiar with Mu¨ller automata should note that defining such projection is not trivial when the automata are defined over a powerset of actions. The definition of the projection from Müller automata to communicating machine is conceptually straightforward (although technically not trivial) if the automata are defined over sets of actions (instead

Formalisation	Coherence Condition	Quality assessment	Binding Mechanism
ARNs	$\{\Delta_{\Lambda_p} \models^{\mathsf{LTL}} \Gamma_\pi\}_{\pi \in \gamma_p}$ where $p \in P$, $\langle \gamma_p, \Lambda_p \rangle$ is a process, Δ_{Λ_p} the set of traces of the Müller automaton Λ_p and Γ_π is the LTL contract associated to port π.	$\prod_{m \in P \cup C} \Lambda_m$	$\Gamma_\pi \vdash^{\mathsf{LTL}} \Gamma_\rho$ where π is a provides point of a service, ρ is a requires point of an activity, and Γ_π and Γ_ρ their LTL contract respectively.
CRNs	$\{\Lambda\|_{p_\pi} \approx \mathcal{A}_\pi\}_{\pi \in \gamma_p}$ where $\Lambda\|_{p_\pi}$ is the projection of Müller automaton Λ over the alphabet of port π, $\mathcal{A}_\pi$ is the communication machine labelling port π and $\approx$ denotes bisimilarity.	$\prod_{m \in P} \Lambda_m$	**Top-Down**: $G\|_\rho \approx \mathcal{A}_\pi$ where π is a provides point of a service, ρ is a requires point of an activity, $G_c\|_{p_\rho}$ is the projection of the global graph G_c over the language of the port ρ, $\mathcal{A}_\pi$ is the communication machine labelling port π and $\approx$ denotes bisimilarity. **Bottom-Up**: $S(\{\mathcal{A}_\pi\}_{\pi \in \Pi}) \equiv G_c$ where Π is the set of provides-points of the services to be bound, G_c is the global graph associated to $c \in C$, $S(\bullet)$ is the algorithm for synthesising choreographies from communication machines [5] and $\equiv$ denotes isomorphism.

Table 1: Comparison of the procedures for the approaches based in ARNs and CRNs

of powersets of them). Altough this is enough for the purposes of this paper, a better solution would be to extend communicating machines so to preserve the semantics of Müller automata even when they are defined on powersets of actions. This is however more challenging (as the reader familiar with Muller automata would recognise) and it is left as a future line of research.

We strived here for simplicity suggesting trivial acceptance conditions. For instance, in the "bottom-up" binding mechanism we required that the exposed global graph coincides (up to isomorphism) to the synthesised one. In general, one could extend our work with milder conditions using more sophisticated relations between choreographies. For instance, one could require that the interactions of the synthesised graph can be simulated by the ones of the declared global graph.

We also envisage benefits that the orchestration model of ARNs could bring into the choreography model we use (similarly to what suggested in [1]). In particular, we argue that the 'incremental binding' naturally featured in the ARN model could be integrated with the choreography model of global graphs and communicating machines. This would however require the modifications of algorithms based on choreography to allow incremental synthesis of choreographies.

References

[1] D. Basile, P. Degano, G. L. Ferrari, and E. Tuosto. From orchestration to choreography through contract automata. In *Proceedings 7th Interaction and Concurrency Experience, ICE 2014, Berlin, Germany, 6th June 2014.*, pages 67–85, 2014.

[2] D. Brand and P. Zafiropulo. On communicating finite-state machines. *JACM*, 30(2):323–342, Apr. 1983.

[3] P. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, pages 194–213, 2012.

[4] Y. Kesten, Z. Manna, and H. M. A. Pnueli. A decision algorithm for full propositional temporal logic. In *CAV*, pages 97–109, 1993.

[5] J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In *Principles of Programming Languages (PoPL)*, 2015. To appear.

[6] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[7] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Comput. Sci.*, 13(1):45–60, 1981.

[8] I. Ţuţu and J. L. Fiadeiro. A logic-programming semantics of services. In *CALCO*, pages 299–313, 2013.

Distributed Programming via Safe Closure Passing

Philipp Haller[1] and Heather Miller[2]

[1] KTH Royal Institute of Technology, Sweden
phaller@kth.se
[2] EPFL, Switzerland
heather.miller@epfl.ch

Abstract

Programming systems incorporating aspects of functional programming, e.g., higher-order functions, are becoming increasingly popular for large-scale distributed programming. New frameworks such as Apache Spark leverage functional techniques to provide high-level, declarative APIs for in-memory data analytics, often outperforming traditional "big data" frameworks like Hadoop MapReduce. However, widely-used programming models remain rather ad-hoc; aspects such as implementation trade-offs, static typing, and semantics are not yet well-understood. We present a new asynchronous programming model that has at its core several principles facilitating functional processing of distributed data. The emphasis of our model is on simplicity, performance, and expressiveness. The primary means of communication is by passing functions (closures) to distributed, immutable data. To ensure safe and efficient distribution of closures, our model leverages both syntactic and type-based restrictions. We report on a prototype implementation in Scala. Finally, we present preliminary experimental results evaluating the performance impact of a static, type-based optimization of serialization.

1 Introduction

Programming systems for large-scala data processing are increasingly embracing functional programming, *i.e.,* programming with first-class functions and higher-order functions. Arguably, one of the first widely-used programming models for "big data" processing making use of concepts from functional programming is Google's MapReduce [3]. Indeed, [7] shows a precise executable semantics of MapReduce in Haskell. While leveraging functional programming *concepts*, popular implementations of the MapReduce model, such as Hadoop MapReduce[1] for Java, have been developed without making use of functional *language features* such as closures. In contrast, a new generation of programming systems for large-scale data processing, such as Apache Spark [17], Twitter's Scalding[2], and Scoobi[3] build on functional language features in order to provide high-level, declarative APIs.

However, these programming systems suffer from several problems that negatively affect their usage, maintenance, and optimization:

- Their APIs cannot statically prevent *common usage errors*. As a result, users are often confronted with runtime errors that are hard to debug. A common example is unsafe closure serialization [12].
- Typically, only high-level user-facing abstractions are statically typed. The absence of static types in lower layers of the system makes *maintenance* tasks, such as code refactorings, more difficult.
- The absence of certain kinds of static type information precludes systems-centric *optimizations*. Importantly, type-based static meta-programming enables fast serialization [11], but this is only possible if also lower layers (namely those dealing with object serialization) are statically typed. Several studies [2, 8, 14, 16] report on the high overhead of serialization in widely-used runtime environments such as the JVM. This overhead is so important in practice that popular systems, like

[1]See `http://hadoop.apache.org/`
[2]See `https://github.com/twitter/scalding/`
[3]See `http://nicta.github.io/scoobi/`

Spark [17] and Akka [15], leverage alternative serialization frameworks such as Protocol Buffers (Google), Apache Avro [1], or Kryo [13].

Contributions This paper makes the following contributions:

- A new asynchronous programming model, called SCP ("safe closure passing"), for functional processing of distributed data (see Sec. 2). We propose to address the above problems through a novel combination of: (a) *safe closures;* to prevent common usage errors. Closures that are not guaranteed to be serializable are rejected at compile time; (b) *a statically-typed implementation of a generic distributed, persistent data structure.* Preserving static types through more system layers improves maintainability and enables type-based optimizations.
- An implementation of the SCP model in Scala (see Sec. 3). In addition, we report on preliminary experimental experience using SCP to evaluate the end-to-end performance impact of a type-based optimization of serialization.

An important goal of our model is to better understand programming systems such as Spark, Scalding, and Scoobi, by incorporating several of their core principles; namely, immutable distributed data and distributed closure passing. By focussing on simplicity, expressiveness, and performance (and ignoring many of the more ad-hoc refinements of the mentioned programming models) our programming model–together with its prototype implementation–enables exploring implementation trade-offs, and capturing the semantics of the core constructs more precisely.

To ensure safe and efficient distribution of closures, our model leverages both syntactic and type-based restrictions. For instance, closures sent to remote nodes are required to conform to the restrictions imposed by the so-called "spore" abstraction that the authors presented in previous work [12]. Among others, the syntax and static semantics of spores can guarantee the absence of runtime serialization errors due to closure environments that are not serializable.

The following sections 2 (programming model) and 3 (implementation) present our main contributions. Section 4 relates to prior work. Section 5 summarizes future work and concludes.

2 Programming Model

The programming model has a few basic abstractions at its center: first, the so-called *silo*. A silo is a typed data container. It is stationary in the sense that it does not move between machines. A silo remains on the machine where it was created. Data stored in a silo is typically loaded from stable storage, such as a distributed file system. A program operating on data stored in a silo can only do so using a reference to the silo, a so-called *SiloRef*. Similar to a proxy object, a SiloRef represents, and allows interacting with, a silo possibly located on a remote node. Some programming patterns require combining data contained in silos located on different nodes (*e.g.,* joins). To support such patterns, our model includes a *pump* primitive for emitting data to silos on arbitrary nodes (explained further below).

A SiloRef has the following main operations:

```
trait SiloRef[T] {
  def apply(s: Spore[T, S]): SiloRef[S]
  def send(): Future[T]
}
```

Apply The `apply` method takes a spore, a kind of closure (see Appendix A for an overview), that is to be applied to the data in the silo of the receiver SiloRef. Rather than immediately sending the spore across the network, and waiting for the operation to finish, the apply method is *lazy*. It immediately returns a SiloRef that refers to the result silo.

To realize something like the `map` combinator of a (distributed) collection using `apply`, it is helpful to think of the spore argument to `apply` ("s") as the composition of a user-defined function passed to the `map` combinator with the actual implementation of `map`:

```
val ref: SiloRef[List[Int]] = ...
val userFun: Int => String = ...
val mapFun: (Int => String) => List[Int] => List[String] = ...
val ref2: SiloRef[List[String]] = ref.apply(mapFun(userFun))
```

In the above example, the higher-order `mapFun` function is expressed in curried style where the user's function argument is passed as the first argument. Applying `mapFun` to a function of type `Int => String` returns a function of type `List[Int] => List[String]`.

The result of invoking `apply` is another SiloRef, which has a reference to the spore and the SiloRef that it was derived from. Note that this is semantically the same as programming with normal functional data structures, where a new data structure is defined by a transformation of an original data structure.

Send The `send` method takes no argument, and returns a future. Unlike `apply`, `send` is *eager* (readers familiar with the concept of *views* might recognize a similarity to forcing a view). That is, it sends whatever operations are queued up (by invocations of `apply`) on a given SiloRef to the node that contains the corresponding silo, and kicks off the materialization of the result silo. Once the materialization is done, the future returned by `send` is completed. Example:

```
val ref2 = ref1.apply(s)  // lazy
val fut  = ref2.send()    // eager
```

The invocation of `send` kicks off the following sequence of actions:

1. A "send" control message is sent to the node where `ref2`'s silo is located.
2. Since `ref2` is derived from `ref1`, `ref1`'s silo is located on the same node. Thus, the runtime demands `ref1` to be materialized; once this is done, spore `s` is applied, populating `ref2`'s silo (on the same node).
3. Once `ref2`'s silo is materialized, its data is sent to the node executing the `send`, completing `fut`.

Note that since a `send` operation sends the data of a silo across the network, it should only be invoked on silos containing small bits of data.

pumpTo The SiloRef singleton object provides an additional method for combining silos storing collections:

```
def pumpTo[T <: Traversable[U], V, R](
      p: Place,
      silo1: SiloRef[T],
      silo2: SiloRef[T],
      fun: Spore[(U, Emitter[V]), Unit],
      bf: BuilderFactory[V, R]): SiloRef[R]
```

The pumpTo method requires the silos silo1 and silo2 to contain collections of element type U (Traversable[U]). Using pumpTo, the elements (of type U) of the two silos are passed one-by-one to the user-provided spore fun. This spore takes a pair as argument containing two components: first, a single element of one of the silo's collections, and second, an *emitter* to which the spore can output values of type V. By emitting such elements, a new silo at the destination (Place p) is filled, yielding a collection of type R. A BuilderFactory[V, R] provides the functionality for building a collection of type R based on elements of type V. Finally, pumpTo returns the SiloRef of the silo that was created at Place p.

Although conceptually related, the Emitter and the BuilderFactory address two separate issues: the Emitter provides a way to output elements from the source silo; the BuilderFactory provides a way to input data into a newly created silo at the destination.

An Emitter is a simple trait which allows the spore parameter fun of pumpTo to emit zero, one, or multiple values per element of a silo's collection, using an emit function:

```
def emit(v: T)(implicit p: Pickler[T]): Unit
```

Note that Emitter differs from the well-known *observable* abstraction [10] in one important way: the emit method requires an implicit type-specific *pickler*. In Scala, type-specialized picklers enable fast serialization through compile-time meta-programming [11]. Thus, Emitter is an abstraction specially designed for distributed programming. The main reason why the pickler is required already at this point is that we would like to enable picklers to be *specialized* to the type of the pickled values. However, this means a pickler has to be constructed at the point when the static type of the emitted value is still available (essentially, before the value loses its type when treated as generic data to be sent across the network).

2.1 Combining Multiple Silos

Operations on distributed collections such as *union*, *groupByKey*, or *join*, involve multiple data sets, possibly located on different nodes. In the following we explain how such operations can be expressed using the introduced primitives.

union The union of two unordered collections stored in two different silos can be expressed directly using the above pumpTo primitive.

join Suppose we are given two silos with the following types:

```
val silo1: SiloRef[List[A]]
val silo2: SiloRef[List[B]]
```

as well as two hash functions computing hashes for elements of type A and B, respectively:

```
val hashA: A => K = ...
val hashB: B => K = ...
```

The goal is to compute the hash-join of silo1 and silo2:

```
val hashJoin: SiloRef[List[(K, (A, B))]] = ???
```

To be able to use pumpTo, the types of the two silos first have to be made equal, through initial apply invocations:

```
val silo12: SiloRef[List[(K, Option[A], Option[B])] =
      silo1.apply { x => (hashA(x), Some(x), None) }
val silo22: SiloRef[List[(K, Option[A], Option[B])] =
      silo2.apply { x => (hashB(x), None, Some(x)) }
```

Then, we can use `pumpTo` to create a new silo (at some destination place), which contains the elements of both `silo12` and `silo22`:

```
val combined = SiloRef.pumpTo(destPlace, silo12, silo22,
                              (elem, emitter) => emitter.emit(elem),
                              listBuilderFactory[...])
```

The combined silo contains triples of type `(K, Option[A], Option[B])`. Using an additional `apply`, the collection can be sorted by key, and adjacent triples be combined, yielding finally a `SiloRef[List[(K, (A, B))]]` as required.

Partitioning and groupByKey A *groupByKey* operation on a group of silos containing collections needs to create multiple result silos, on each node, with ranges of keys supposed to be shipped to destination nodes. These destination nodes are determined using a partitioning function. Our goal, concretely:

```
val groupedSilos = groupByKey(silos)
```

Furthermore, we assume that `silos.size` $= N$ where N is the number of nodes, with nodes N_1, N_2, etc. We assume each silo contains an unordered collection of key-value pairs (a multi-map). Then, *groupByKey* can be implemented as follows:

- For each node N_i, the master node creates N SiloRefs.
- Each node N_i applies a *partitioning function* (example: `hash(key) mod N`) to the key-value pairs in its silo, yielding N (local) silos.
- Using `pumpTo`, each pair of silos containing keys of the same range can be combined and materialized on the right destination node.

3 Implementation and Preliminary Experimental Results

We have developed a prototype[4] of the SCP model in Scala, which builds on our earlier work on Scala Pickling [11] and Spores [12]. The implementation does not require extensions to the Scala language or compiler; it is developed using the current stable Scala release 2.11.

We have used our implementation to measure the impact of compile-time-generated serializers [11] on end-to-end application performance. We ran our experiments on a 2.3 GHz Intel Core i7 with 16 GB RAM under Mac OS X 10.9.5 using Java HotSpot Server 1.8.0-b132. In our benchmark application, a group of 4 silos is distributed across 4 different nodes/JVMs. The silos are first transformed using *map*, and then using *groupBy*. For an input size of 100’000 “person” records, the use of compile-time-generated serializers resulted in an overall speedup of about 48% with respect to the same system but without using compile-time-generated serializers.

[4]See `https://github.com/heathermiller/f-p`

4 Related Work

Cloud Haskell [4] leverages guaranteed-serializable, static closures for a message-passing communication model inspired by Erlang. In contrast, in our model spores are sent between passive, persistent silos. Closures and continuations in Termite Scheme [5] are always serializable; references to non-serializable objects (like open files) are automatically wrapped in processes that are serialized as their process ID. Similar to Cloud Haskell, Termite is inspired by Erlang. In contrast to Termite, SCP is statically typed, enabling advanced type-based optimizations. In non-process-oriented models, parallel closures [9] and RiverTrail [6] address important safety issues. SCP integrates a distributed, persistent data structure. Other prior work related to spores is discussed in [12].

5 Conclusion and Future Work

We have presented a new asynchronous distributed programming model. A novel combination of (a) closures with syntactic and semantic restrictions and (b) abstractions for distributed data “silos” prevents usage errors common in widely-used “big data” frameworks. We have implemented our model in Scala; preliminary experimental results evaluate the performance impact of a static type-based optimization. In future work, we intend to expand the practical experiments and explore the impact of implementation trade-offs. Furthermore, we would like to exploit the simplicity of the programming model for a formal treatment of its properties.

References

[1] Apache. Avro®. `http://avro.apache.org`. Accessed: 2013-08-11.

[2] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object serialization for marshalling data in a Java interface to MPI. In *Java Grande*, pages 66–71, 1999.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[4] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards Haskell in the cloud. In *Proc. Haskell Symposium*, pages 118–129. ACM, 2011.

[5] Guillaume Germain. Concurrency oriented programming in Termite Scheme. In *Erlang Workshop*, 2006.

[6] Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. River trail: a path to parallelism in JavaScript. In *OOPSLA*, pages 729–744, 2013.

[7] Ralf Lämmel. Google’s mapreduce programming model - revisited. *Sci. Comput. Program*, 70(1):1–30, 2008.

[8] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of Java’s remote method invocation. In *PPOPP*, pages 173–182, August 1999.

[9] Nicholas D. Matsakis. Parallel closures: a new twist on an old idea. In *HotPar*. USENIX, 2012.

[10] Erik Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, 2012.

[11] Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In *OOPSLA*. ACM, 2013.

[12] Heather Miller, Philipp Haller, and Martin Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *ECOOP*, pages 308–333. Springer, 2014.

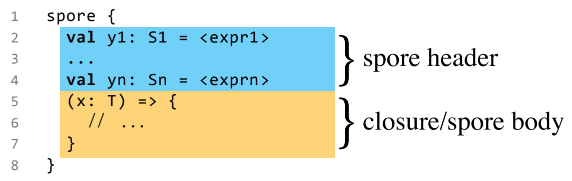
[13] Nathan Sweet et al. Kryo. `https://code.google.com/p/kryo/`. Accessed: 2013-08-11.

[14] Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency - Practice and Experience*, 12(7):495–518, 2000.

[15] Typesafe. Akka. `http://akka.io/`, 2009. Accessed: 2013-08-11.

[16] Matt Welsh and David E. Culler. Jaguar: enabling efficient communication and I/O in Java. *Concurrency - Practice and Experience*, 12(7), 2000.

[17] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX, 2012.



```
spore {
  val y1: S1 = <expr1>
  ...
  val yn: Sn = <exprn>
  (x: T) => {
    // ...
  }
}
```

Figure 1: The syntactic shape of a spore.

```
{
  val y1: S1 = <expr1>
  ...
  val yn: Sn = <exprn>
  (x: T) => {
    // ...
  }
}
```

(a) A closure block.

```
spore {
  val y1: S1 = <expr1>
  ...
  val yn: Sn = <exprn>
  (x: T) => {
    // ...
  }
}
```

(b) A spore.

Figure 2: The evaluation semantics of a spore is equivalent to that of a closure, obtained by simply leaving out the spore marker.

A Spores

Spores are a closure-like abstraction and type system which aims to give users a principled way of controlling the environment which a closure can capture. This is achieved by (a) enforcing a specific syntactic shape which dictates how the environment of a spore is declared, and (b) providing additional type-checking to ensure that types being captured have certain properties. A crucial insight of spores is that, by including type information of captured variables in the type of a spore, type-based constraints for captured variables can be composed and checked, making spores safer to use in a concurrent, distributed, or in arbitrary settings where closures must be controlled.

A.1 Spore Syntax

A spore is a closure with a specific shape that dictates how the environment of a spore is declared. The shape of a spore is shown in Figure 1. A spore consists of two parts:

- **the spore header**, composed of a list of value definitions.
- **the spore body** (sometimes referred to as the "spore closure"), a regular closure.

The characteristic property of a spore is that the *spore body* is only allowed to access its parameter, the values in the spore header, as well as top-level singleton objects (public, global state). In particular, the spore closure is not allowed to capture variables in the environment. Only an expression on the right-hand side of a value definition in the spore header is allowed to capture variables.

By enforcing this shape, the environment of a spore is always declared explicitly in the spore header, which avoids accidentally capturing problematic references. Moreover, importantly for object-oriented languages, it's no longer possible to accidentally capture the `this` reference.

```
trait Function1[-A, +B] {
  def apply(x: A): B
}
```

(a) Scala's arity-1 function type.

```
trait Spore[-A, +B]
extends Function1[A, B] {
  type Captured
  type Excluded
}
```

(b) The arity-1 `Spore` type.

Figure 3: The `Spore` type.

A.1.1 Evaluation Semantics

The evaluation semantics of a spore is equivalent to a closure obtained by leaving out the `spore` marker, as shown in Figure 2. In Scala, the block shown in Figure 2a first initializes all value definitions in order and then evaluates to a closure that captures the introduced local variables `y1, ..., yn`. The corresponding spore, shown in Figure 2b has the exact same evaluation semantics. Interestingly, this closure shape is already used in production systems such as Spark in an effort to avoid problems with accidentally captured references, such as `this`. However, in systems like Spark, the above shape is merely a convention that is not enforced.

A.2 The `Spore` Type

Figure 3 shows Scala's arity-1 function type and the arity-1 spore type.[5] Functions are contravariant in their argument type `A` (indicated using `-`) and covariant in their result type `B` (indicated using `+`). The `apply` method of `Function1` is abstract; a concrete implementation applies the body of the function that is being defined to the parameter `x`.

Individual spores have *refinement types* of the base `Spore` type, which, to be compatible with normal Scala functions, is itself a subtype of `Function1`. Like functions, spores are contravariant in their argument type `A`, and covariant in their result type `B`. Unlike a normal function, however, the `Spore` type additionally contains information about *captured* and *excluded* types. This information is represented as (potentially abstract) `Captured` and `Excluded` type members. In a concrete spore, the `Captured` type is defined to be a tuple with the types of all captured variables. [12] discusses the `Excluded` type member in detail.

A.3 Basic Usage

A.3.1 Definition

A spore can be defined as shown in Figure 4a, with its corresponding type shown in Figure 4b. As can be seen, the types of the environment listed in the spore header are represented by the `Captured` type member in the spore's type.

A.3.2 Using Spores in APIs

Consider the following method definition:

```
def sendOverWire(s: Spore[Int, Int]): Unit = ...
```

[5]For simplicity, we omit `Function1`'s definitions of the `andThen` and `compose` methods.

```
val s = spore {
  val y1: String = expr1;
  val y2: Int = expr2;
  (x: Int) => y1 + y2 + x
}
```

(a) A spore `s` which captures a `String` and an `Int` in its spore header.

```
Spore[Int, String] {
  type Captured = (String, Int)
}
```

(b) `s`'s corresponding type.

Figure 4: An example of the `Captured` type member.
Note: we omit the `Excluded` type member for simplicity; we discuss it in detail in [12].

In this example, the `Captured` (and `Excluded`) type member is not specified, meaning it is left abstract. In this case, so long as the spore's parameter and result types match, a spore type is always compatible, regardless of which types are captured.

Using spores in this way enables libraries to enforce the use of spores instead of plain closures, thereby reducing the risk for common programming errors, even in this very simple form.

A.3.3 Composition

Like normal functions, spores can be composed. By representing the environment of spores using refinement types, it is possible to preserve the captured type information (and later, constraints) of spores when they are composed.

For example, assume we are given two spores `s1` and `s2` with types:

```
s1: Spore[Int, String] { type Captured = (String, Int) }
s2: Spore[String, Int] { type Captured = Nothing }
```

The fact that the `Captured` type in `s2` is defined to be `Nothing` means that the spore does not capture anything (`Nothing` is Scala's bottom type). The composition of `s1` and `s2`, written `s1 compose s2`, would therefore have the following refinement type:

```
Spore[String, String] { type Captured = (String, Int) }
```

Note that the `Captured` type member of the result spore is equal to the `Captured` type of `s1`, since it is guaranteed that the result spore does not capture more than what `s1` already captures. Thus, not only are spores composable, but so are their (refinement) types.

Reversible Communicating Processes

Geoffrey Brown and Amr Sabry

School of Informatics and Computing
Indiana University
Bloomington, IN
geobrown@indiana.edu, sabry@indiana.edu

Abstract

Reversible distributed programs have the ability to abort unproductive computation paths and backtrack, while unwinding communication that occurred in the aborted paths. While it is natural to assume that reversibility implies full state recovery (as with traditional roll-back recovery protocols), an interesting alternative is to separate backtracking from local state recovery. For example, such a model could be used to create complex transactions out of nested compensable transactions where a programmer supplied compensation defines the work required to "unwind" a transaction.

Reversible distributed computing has received considerable theoretical attention, but little reduction to practice; the few published implementations of languages supporting reversibility depend upon a high degree of central control. The objective of this paper is to demonstrate that a practical reversible distributed programming language can be efficiently implemented in a fully distributed manner.

We discuss such a language, supporting CSP-style synchronous communication, embedded in Scala. While this language provided the motivation for the work described in this paper, our focus is upon the distributed implementation. In particular, we demonstrate that a "high-level" semantic model can be implemented using a simple point-to-point protocol.

1 Introduction

Speculative execution either by intent or through misfortune (in response to error conditions) is pervasive in system design and yet it remains difficult to handle at the program level [10]. Indeed, we find that despite the importance of speculative computation, there is very little programmatic support for it in distributed languages at the foundational level it deserves. We note that, from a programming language perspective, speculative execution requires a backtracking mechanism and that, even in the sequential case, backtracking in the presence of various computational effects (e.g. assignments, exceptions, etc.) has significant subtleties [12]. The introduction of concurrency additionally requires a "distributed backtracking" algorithm that must "undo" the effects of any communication events that occurred in the scope over which we wish to backtrack. While this has been successfully accomplished at the algorithmic level (e.g. in virtual time based simulation [15, 14]), in models of concurrent languages (e.g.[2, 3, 16, 17, 22]) and in some restricted parallel shared-memory environments (e.g. [6, 11, 18, 23, 20]), it does not appear that any concurrent languages based upon message passing have directly supported backtracking with no restrictions. The language constructs we introduce are inspired by the stabilizers of [24]; however, that work depends upon central control to manage backtracking. Our work was also inspired by the work of Hoare and others [19, 13]. Communicating message transactions [6, 18] is an interesting related approach that relies upon global shared data structures.

The work presented in this paper has a natural relationship to the rich history of rollback-recovery protocols [8]. Rollback-recovery protocols were developed to handle the (presumably rare) situation where a processor fails and it is necessary to restart a computation from a previously saved state. The fundamental requirement of these protocols is that the behavior is as if no error ever occurred. In contrast, we are interested in systems where backtracking might take the computation in a new direction based upon state information gleaned from an abandoned execution path; the (possibly frequent) decision to backtrack is entirely under program control. Because check-pointing in traditional rollback-recovery protocols involves saving a complete snapshot of a process's state, it is a relatively expensive operation. Much of the research in rollback-recovery protocols focuses upon minimizing these costs. The cost of check-pointing is much lower for our domain – saving control state is no more expensive than for a conventional exception handler; the amount of data state preserved is program dependent.

Implementing a reversible concurrent language is not a trivial undertaking and, as we found, there are many opportunities to introduce subtle errors. Ideally, such a language implementation should be accompanied by a suitable semantics that provides both a high-level view which a programmer can use to understand the expected behavior of a program text, and a low-level view which the language implementer can use to develop a correct implementation. In order to accommodate these two constituencies, we have developed two separate semantic models. We have developed a refinement mapping to demonstrate that the low-level model is a correct implementation of the high-level model, which is outlined in the paper.

The remainder of this paper is organized as follows. We begin with a small example that illustrates the main ideas using our Scala implementation. We then, in Sec. 3, present a formal "high-level" semantic model focusing on the semantics of forward communication and backtracking. In Sec. 4, we discuss a communication protocol for maintaining backtracking state across distributed communicating agents. Sec. 5 introduces (a fragment of) a "low-level" model that utilizes the channel protocol to implement the high-level model, and Sec. 6 outlines the proof of correctness of the low-level semantics with respect to the high-level semantics. We end with a brief discussion.

2 Example

We illustrate the main ideas of our language with an example written in our Scala realization. A programmer wishing to use our distributed reversible extensions imports our libraries for processes and channels and then defines extensions of the base class `CspProc` by overriding the method `uCode`. The user-defined code must use our channel implementation for communication and may additionally use the keywords `stable` and `backtrack` for managing backtracking over speculative executions. The following excerpt provides the code for two processes `p1` and `p2` that communicate over channel `c` – not shown is the code that creates these processes and the channel:

```
 7  class p1 (c: SndPort, name: String)
        extends CspProc(name) {
 9    override def uCode = {
        println("p1: start")
11      var count = 2
        stable {
13        println("p1: snd " + count)
          send(c,count)
15        stable {
            println("p1: snd " + count)
17          send(c,count)
            count = count - 1
19          if (count > 0) {
              println ("p1: backtrack")
21            backtrack }
          }}
23    }}
```

```
25  class p2 (c: RcvPort, name: String)
        extends CspProc(name) {
27    override def uCode = {
        println("p2: start")
29      stable {
          var x = receive(c)
31        println("p2: recv " + x)
          var y = receive(c)
33        println("p2: recv " + y)
        }
35    }}
```

Output:

```
p1: start
p2: start
p1: snd 2
p2: recv 2
p1: snd 2
p2: recv 2
p1: backtrack
p2: start
p1: snd 1
p2: recv 1
p1: snd 1
p2: recv 1
```

In the code, the `stable` regions denote the scope of saved contexts; executing `backtrack` within a stable region returns control to the beginning of the stable region – much like "throwing" an exception, but with the additional effect of unwinding any communication that may have occurred within the stable region. Process `p1` starts its execution by sending a first message to `p2`, entering a nested stable region, and sending another message to `p2`. Meanwhile process `p2` also starts a stable region in which it receives the two messages. At this point in the execution, process `p1` decides at line 15 to backtrack. As a result, process `p1` transfers its control to the inner stable region (line 9). This jump invalidates the communication on channel `c` at line 11. Process `p1` then blocks until process `p2` takes action. When process `p2` notices that the second communication event within the stable region (line 8) was invalidated, it backtracks to the start of its stable region. This jump invalidates the first communication action (line 6) which in turn invalidates the corresponding action in `p1` at line 8. In other words, process `p1` is forced to backtrack to its outer stable region to establish a causally consistent state. The trace in the output shows a possible interleaving of the execution – it is important to remember that all processes have an implicit stable region that includes their full code body; in this case `p2` is forced to backtrack to the beginning of its code. The crucial point is that after the backtracking, both communication events between `p1` and `p2` are re-executed.

3 High-Level Semantics of a Reversible Process Language

We will present two semantic models for our language. The first "high-level" semantics formalizes both the forward communication events that occur under "normal" program execution and the backwards communication events that occur when processes are backtracking to previously saved states as atomic steps. In the low-level semantics, these communication events are further subdivided into actions that communicating senders and receivers may take independently in a distributed environment and hence trades additional complexity for a specification that is close to a direct implementation. This low-level semantics is based upon a channel protocol that we have verified using the SAL infinite-state model checker [4, 5, 7]; the invariants validated using SAL were necessary to prove that the low-level semantics is a refinement of the high level semantics. Both of our semantic models are based upon virtual time – a commonly used technique for conventional rollback recovery protocols [9, 21]. Our approach differs in utilizing synchronous communication and also by providing a fully distributed rollback protocol.

3.1 User-Level Syntax

We begin with the syntax of a core calculus which is rich enough to express the main semantic notions of interest:

$$\begin{array}{llll}
(\textit{channel name}) & \ell & & \\
(\textit{constants}) & c & ::= & () \mid 0 \mid 1 \mid \ldots \mid + \mid - \mid \geq \mid \ldots \\
(\textit{expression}) & e & ::= & c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \textsf{send}\ \ell\ e \mid \textsf{recv}\ \ell(x).e \mid \textsf{stable}\ e \mid \textsf{backtrack}\ e \\
(\textit{process}) & p & ::= & p_1 \,\|\, p_2 \quad \mid \quad \langle e \rangle
\end{array}$$

A program is a collection of processes executing in parallel. Expressions extend the call-by-value λ-calculus with communication and backtracking primitives. The communication primitives are $\textsf{send}\ \ell\ e$ which commits to sending the value of e on the channel ℓ and $\textsf{recv}\ \ell(x).e$ which receives x on channel ℓ. Both our Scala implementation and full model support input "choice"; we have omitted the details in the interest of brevity. The backtracking primitives are $\textsf{stable}\ e$ which is used to delimit the scope of possible backtracking events within e. The expression $\textsf{backtrack}\ e$ typically has two effects: the control state in the process executing the instruction jumps back to the dynamically closest nested block with the value of e *and* all intervening communication events are invalidated. The latter action might force neighboring processes to also backtrack, possibly resulting in a cascade of backtracking for a poorly written program.

3.2 Internal Syntax

In order to formalize the semantics, we define a few auxiliary syntactic categories that are used to model run-time data structures and internal states used by the distributed reversible protocol. These additional categories include process names, time stamps, channel maps, evaluation contexts, and stacks and are collected below:

$$\begin{array}{llll}
(\textit{process name}) & n & & \\
(\textit{time stamp}) & t & & \\
(\textit{values}) & v & ::= & c \mid x \mid \lambda x.e \mid \textsf{stable}\ (\lambda x.e) \\
(\textit{expressions}) & e & ::= & \ldots \mid \underline{\textsf{stable}}\ e \\
(\textit{evaluation contexts}) & E & ::= & \square \mid E\ e \mid v\ E \mid \textsf{send}\ \ell\ E \mid \textsf{stable}\ E \mid \underline{\textsf{stable}}\ E \mid \textsf{backtrack}\ E \\
(\textit{channel map}) & \Xi & = & \ell \mapsto (n, t, n) \\
(\textit{stacks}) & \Gamma & = & \bullet \mid \Gamma, (E, t, \Xi) \\
(\textit{processes}) & p & ::= & p_1 \,\|\, p_2 \quad \mid \quad \langle n@t :\ \Gamma, e \rangle
\end{array}$$

Expressions are extended with $\underline{\textsf{stable}}\ e$ which indicates an *active* region. The syntax of processes $\langle n@t :\ \Gamma, e\rangle$ is extended to record additional information: a process id n, a virtual time t, a context stack Γ, and an expression e to evaluate. The processes communicate using channels ℓ whose state is maintained in a global map Ξ. Each entry in Ξ maps a channel to the sender and receiver processes (which are fixed throughout the lifetime of the channel) and the current virtual time of the channel. Contexts are pushed on the stack when a process enters a new stable region and popped when a process backtracks or exits a stable region. Each context includes a conventional continuation (modeled by an evaluation context), a time stamp, and a local channel map describing the state of the communication channels at the time of the checkpoint. An invariant maintained by the semantics is that a process executing in the forward direction will have the times of its channels in the global map greater than or equal to the times associated with the channels in the top stack frame. Similarly, the time associated with a process will always be at least as great as the times associated with its channels. We assume that in the initial system state, all channels have time 0 and every process is of the form $\langle n@0 : \bullet, \textsf{stable}\ (\lambda\_.e)\ ()\rangle$; i.e. process n is entering a stable region containing the expression e with an empty context stack at time 0.

We now informally describe the key semantic specifications that the above structures must satisfy and illustrate some of the key points using examples. Aiming for clarity, the presentation is simplified in several inessential aspects that are formalized in the remaining sections. Any computation step that does not involve communication, stable regions, or backtracking is considered a local computation step. None of the above structures need to be consulted or updated during such local computation steps. In particular, the virtual time stamp is not incremented and local computation steps proceed at "full native speed." In order to establish notation for the remaining examples, here is a simple computation step:

$$\{\ell_1 \mapsto (p_1, 2, p_2)\} \fatsemi \langle p_1@5 : \bullet, 1+2\rangle \to \{\ell_1 \mapsto (p_1, 2, p_2)\} \fatsemi \langle p_1@5 : \bullet, 3\rangle$$

In this example, a process p_1 with local virtual time 5 and making forward progress encounters the computation $1 + 2$. The process's context stack is currently empty ($\bullet$).[1] We also assume the existence of a channel ℓ_1 which is associated with time 2 and connects p_1 to p_2. Intuitively this means that the last communication by that process on that channel happened three virtual time units in the past. The reduction rule leaves all structures intact and simply performs the local calculation.

Communication between processes is synchronous and involves a handshake. We require that in addition to the usual exchange of information between sender and receiver, that the handshake additionally exchanges several virtual times to establish the following invariant. At the end of handshake, the virtual times of the sending process, the receiving process, and the used channel are all equal, and that this new virtual time is larger than any of the prior times for these structures. Here is a small example illustrating this communication handshake:

$$\begin{array}{rl} & \{\ell_1 \mapsto (p_1, 3, p_2)\} \fatsemi \langle p_1@5 : \bullet, \textsf{send}\ \ell_1\ 10\rangle \| \langle p_2@4 : \bullet, \textsf{recv}\ \ell_1(x).x+1\rangle \\ \to & \{\ell_1 \mapsto (p_1, 6, p_2)\} \fatsemi \langle p_1@6 : \bullet, ()\rangle \| \langle p_2@6 : \bullet, 10+1\rangle \end{array}$$

Initially, we have two processes willing to communicate on channel ℓ_1. Process p_1 is sending the value 10 and process p_2 is willing to receive an x on channel ℓ_1 and proceed with $x + 1$. After the reduction, the value 10 is exchanged and each process proceeds to the next step. The important invariant that has been established is that the virtual times of the two processes as well as the virtual time of the channel ℓ_1 have all been synchronized to time 6 which is greater than any of the previous times.

When a process executing in the forward direction encounters a new stable region, it increments its virtual time and pushes a new context containing a continuation (or an exception handler) and the current virtual times of all its current communication ports. If the execution of this stable region ends "normally," the context is popped and forward execution continues. As an example, consider:

$$\begin{array}{rl} & \{\ell_1 \mapsto (p_1, 2, p_2)\} \fatsemi \langle p_1@5 : \bullet, 7 + (\textsf{stable}\ f)\ v\rangle \\ \to & \{\ell_1 \mapsto (p_1, 2, p_2)\} \fatsemi \langle p_1@6 : \bullet, (7 + (\textsf{stable}\ f)\ \square, 5, \{\ell_1 \mapsto (p_1, 2, p_2)\}), 7 + \underline{\textsf{stable}}\ (f\ v)\rangle \end{array}$$

Process p_1 encounters the expression $7 + (\textsf{stable}\ f)\ v$ where f is some function and v is some value. The $\textsf{stable}$ construct indicates that execution might have to revert back to the current state if any backtracking actions are encountered during the execution of $f\ v$. The first step is to increase the virtual time to establish a new unique event. Then to be prepared for the eventuality of backtracking, process p_1 pushes $(7 + (\textsf{stable}\ f)\ \square, 5, \{\ell_1 \mapsto (p_1, 2, p_2)\})$ on its context stack. The pushed information consists of the continuation $7 + \textsf{stable}\ f\ \square$ which indicates the local control point to jump back to, the virtual time 5 which indicates the time to which to return, and the current channel map which captures the state of the communication channels to be restored. Execution continues with $7 + \underline{\textsf{stable}}\ (f\ v)$ where the underline indicates that the region is currently active. If the execution of $f\ v$ finishes normally, for example, by

[1]Actually, each process starts with an implicit stable region which is omitted for clarity.

performing communication on channel ℓ_1 and then returning the value 100, then the evaluation progresses as follows:

$$\begin{array}{ll} & \{\ell_1 \mapsto (p_1, 8, p_2)\} \fatsemi \langle p_1@8 :\ \bullet, (7 + (\mathsf{stable}\ f)\ \square, 5, \{\ell_1 \mapsto (p_1, 2, p_2)\}), 7 + \underline{\mathsf{stable}}\ 100\rangle \\ \to & \langle p_1@8 :\ \bullet, 7 + 100\rangle \end{array}$$

The context stack is popped and execution continues in the forward direction.

The more challenging situation occurs when the execution of $f\ v$ encounters a backtracking command (whether directly initiated by the process itself or indirectly initiated via a communicating partner). This case will be described in detail in the next section.

3.3 Forward Semantics

In the remainder of this section we define the high-level semantics through a set of transition rules on semantic configurations where a semantic configuration $\Xi \fatsemi p_1 \parallel p_2 \ldots$ consists of the global channel map Ξ and a number of processes.

Internal evaluation by a single process is captured with fairly conventional rules. We present the rule for application of λ-expressions; the rules for applying primitive operations are similar:

$$\Xi \fatsemi \langle n@t :\ \Gamma, E[(\lambda x.e)\ v]\rangle \xrightarrow{\epsilon} \Xi \fatsemi \langle n@t :\ \Gamma, E[e[v/x]]\rangle \qquad \text{(H1)}$$

In the rule, the runnable expression in the process is decomposed into an evaluation context E and a current "instruction" $(\lambda x.e)\ v$. This instruction is performed in one step that replaces the parameter x with the value v in the body of the procedure e. The notation for this substitution is $e[v/x]$.[2]

The rule for forward communication is:

$$\begin{array}{l} \Xi\{\ell \mapsto (n_1, t_c, n_2)\} \fatsemi \langle n_1@t_1 :\ \Gamma_1, E_1[\mathsf{send}\ \ell\ v]\rangle \parallel \langle n_2@t_2 :\ \Gamma_2, E_2[\mathsf{recv}\ \ell(x).e]\rangle \\ \xrightarrow{\ell@t[v]} \\ \Xi\{\ell \mapsto (n_1, t, n_2)\} \fatsemi \langle n_1@t :\ \Gamma_1, E_1[()]\rangle \parallel \langle n_2@t :\ \Gamma_2, E_2[e[v/x]]\rangle \end{array} \qquad \text{(H2)}$$

Where $t > \max(t_1, t_2)$. The notation $\Xi\{\ell \mapsto (n_1, t_c, n_2)\}$ says that, in channel map Ξ, channel ℓ connects sender n_1 and receiver n_2 and has time t_c. Two processes communicate on a shared channel when one is prepared to send and the other is prepared to receive. The act of communication causes data to be transferred from the sender to receiver and a new (later) virtual time to be assigned to each of the processes and the channel. An important model invariant is that $t_1 \geq t_c \wedge t_2 \geq t_c$. Notice further that this transition produces a visible event $\ell@t[v]$ signifying that value v was transferred on channel ℓ at time t.

Finally there are rules corresponding to entering and exiting stable regions. Entry into a stable region causes a new context to be pushed onto the stack. Notice that a value is passed into the stable region being executed. The syntax $\underline{\mathsf{stable}}\ e$ means that e is being evaluated within a stable region:

$$\begin{array}{l} \Xi \fatsemi \langle n@t :\ \Gamma, E[(\mathsf{stable}\ (\lambda x.e))\ v]\rangle \\ \xrightarrow{\epsilon} \\ \Xi \fatsemi \langle n@t' :\ (\Gamma, (E[(\mathsf{stable}\ (\lambda x.e))\ \square], t, \Xi_n)), E[\underline{\mathsf{stable}}\ e[v/x]]\rangle \end{array} \qquad \text{(H3)}$$

Where $t' > t$ and Ξ_n is the subset of Ξ referring to the channels of n. The saved context $E[(\mathsf{stable}\ (\lambda x.e))\ \square]$ on the stack will be reinstated in case of a backtracking action. As the rules in the next section will show, backtracking may occur either from within the current process or indirectly because another neighboring process has retracted a communication event. In the first case, the context will be resumed with a value of the programmer's choice; in the latter case, the context will be resumed, asynchronously, with the value $()$. A well-typed program should have the function argument to stable ($\lambda x.e$ in the rule above) be prepared to handle either situation.

When a stable region is completed, the stored context is popped:

$$\Xi \fatsemi \langle n@t :\ (\Gamma, (E', t', \Xi')), E[\underline{\mathsf{stable}}\ v]\rangle \xrightarrow{\epsilon} \Xi \fatsemi \langle n@t :\ \Gamma, E[v]\rangle \qquad \text{(H4)}$$

[2]For readability, the color version of the paper highlights the components of the configuration that are modified by each rule.

3.4 Backtracking Semantics

The novelty of our language is in its support of backtracking. Backtracking is initiated by a process that encounters a backtrack expression. A process executing a backtrack event can synchronize with other processes through "backwards communication events." Any process engaging in a backwards communication event is forced into the backtracking state:

$$\begin{array}{l} \Xi\{\ell \mapsto (c_1, t_c, c_2)\} \fatsemi \langle n_1@t_1 : \ \Gamma_1, e_1\rangle \parallel \langle n_2@t_2 : \ \Gamma_2, E_2[\mathsf{backtrack}\ v]\rangle \\ \xrightarrow{\overline{\ell}@t} \\ \Xi\{\ell \mapsto (c_1, t, c_2)\} \fatsemi \langle n_1@t_1 : \ \Gamma_1, \mathsf{backtrack}\ ()\rangle \parallel \langle n_2@t_2 : \ \Gamma_2, E_2[\mathsf{backtrack}\ v]\rangle \end{array} \tag{H5}$$

Where $0 \leq t < t_c$ and $\{c_1, c_2\} = \{n_1, n_2\}$. (Case where n_1 is already backtracking omitted).

A backwards communication event occurs between two processes sharing a channel where at least one of the processes is in the backtracking state. The label $\overline{\ell}@t$ means that all communication events on channel ℓ at times later than t are retracted. Notice that only the channel time is reduced – this preserves our invariant that the virtual time of every process is greater than or equal to that of its channels. The virtual time of a backtracking process is only updated when it returns to forward execution.

While our semantic rules impose as few constraints as possible, our Scala implementation demonstrates that it is possible to constrain the application of these rules to obtain an efficient implementation.

A backwards communication event may bring a process to a state in which the top context on its stack is "later" than required by one of its channels. In this case, repeated backtracking is required – in effect popping its context stack.

$$\begin{array}{l} \Xi\{\ell \mapsto (-, t, -)\} \fatsemi \langle n_1@t_1 : \ \Gamma_1, (E_1, t'_1, \Xi_1\{\ell \mapsto (-, t', -)\}), E[\mathsf{backtrack}\ v]\rangle \\ \rightarrow \\ \Xi \fatsemi \langle n_1@t_1 : \ \Gamma_1, E[\mathsf{backtrack}\ v]\rangle \end{array} \tag{H6}$$

Where $t < t'$. Because of our initial conditions, this rule is always possible if $0 < t'$ – in effect we guarantee there is a such a stored context.

Finally, a process that is backtracking can return to forward action if all its channels are in a "consistent" state – that is, when all channels in its stored channel map have time-stamps matching what is found in the global channel map.

$$\Xi \fatsemi \langle n_1@t_1 : \ \Gamma_1, (E_1, t'_1, \Xi_1), E[\mathsf{backtrack}\ v]\rangle \rightarrow \Xi \fatsemi \langle n_1@t'_1 : \ \Gamma_1, E_1[v]\rangle \tag{H7}$$

Where $\Xi_1\{\ell \mapsto (-, t, -)\} \Rightarrow \Xi\{\ell \mapsto (-, t, -)\}$. Again, our initial conditions guarantee that such a state is possible.

4 Channel Protocol

The principal difference between the low-level (Section 5) and high-level semantics is that the low-level communication is implemented using a multi-phase handshaking protocol where the sender requests a communication event that the receiver subsequently acknowledges. The state corresponding to a channel is divided into two parts – one maintained by the sender and the other maintained by the receiver. A process may only write the state associated with its channel end, but may read (a delayed version of) that maintained by its communicating partner. Necessarily, the channel implementation has two time-stamps – one maintained by the sender and one maintained by the receiver. It is convenient to think of the time-stamp maintained by the receiver as the "true" channel time. In addition to independent time-stamps, each end of the channel has a token bit and a "direction" flag. The token bits jointly determine which end of the channel may make the next "move," and the flag (loosely) determines the direction of communication, forward or backwards.

We developed a formal model for the channel protocol using the Symbolic Analysis Laboratory (SAL) tools [5, 4, 7] and present the protocol using the SAL syntax. The invariants verified with the SAL model are crucial to the refinement proofs that link our two semantic models. For example, the SAL invariants show that if the processes using a channel obey the protocol, the various timestamps will obey the ordering assertions our semantics depend upon.

In the SAL language the key protocol types are defined as:

```
TIME : TYPE = NATURAL;
DIR  : TYPE = { B, I, F }; % backwards, idle, forward
```

The sender state is defined by four state variables:

```
OUTPUT s_b  : BOOLEAN   % sender token
OUTPUT s_t  : TIME      % sender time
OUTPUT s_d  : DIR       % sender direction
OUTPUT v    : NATURAL   % sender data
```

Similarly for the receiver state:

```
OUTPUT r_b   : BOOLEAN  % receiver token
OUTPUT r_t   : TIME     % receiver time
OUTPUT r_d   : DIR      % receiver direction
```

The state of the channel consists of the union of the sender and receiver states. In general, the right to act alternates between the sender and the receiver. The sender is permitted to initiate a communication event (forwards or backwards) when the two token bits are equal. The receiver is permitted to complete a communication event when the two token bits are unequal. Thus the sender (receiver) "holds" the token when these bits are equal (unequal). This alternating behavior is a characteristic of handshake protocols.

An important wrinkle in our protocol is the ability of blocked sender or receiver to signal its partner that it wishes to switch from forward to backwards communication. The receiver state also includes an auxiliary Boolean variable sync used to support the proof. The receiver sets sync when it completes a communication event and resets it when it refuses a communication event. This variable is not visible to the sender; however, our automated proof demonstrates that the sender can infer its value from the visible state even in the presence of potential races.

Forward communication is initiated when the sender executes the guarded transition:

Trans 1.
```
(s_b = r_b) AND (r_d = F)  -->  s_b'  = NOT s_b;
                           s_t'  IN { x : TIME | x > r_t} ;
                           s_d'  = F;
                           v' IN { x : NATURAL | true}
```

In the SAL language, transition rules are simply predicates defining pre- and post-conditions; the next state of `s_b` is `s_b'`. Thus, the sender may initiate forward communication whenever it holds the "token" (`s_b = r_b`) and the receiver is accepting forward transactions (`r_d = F`). By executing the transition, the sender selects a new time (`s_t'`), relinquishes the token, indicates that it is executing a forward transaction (`s_d' = F`), and selects (arbitrary) data to transfer.

The receiver completes the handshake by executing the following transition in which it updates its clock (to a value at least that offered by the sender), and flips its token bit. This transition is only permitted when both the sender and the receiver wish to engage in forward communication.

Trans 2.
```
(s_b /= r_b) AND (s_d /= B) AND (r_d = F) -->  r_b' = s_b;
               r_t' IN { x : TIME | x >= s_t};
```

A receiver may also refuse a forward transaction by indicating that it desires to engage only in backwards communication.

Trans 3.
```
 (s_b /= r_b) AND (s_d = F) --> r_b' = s_b;
                           r_d' = B;
```

Our protocol also supports backwards communication events. The sender may initiate a backwards event whenever it holds the token.

Trans 4.
```
(s_b = r_b) --> s_b' = NOT s_b;
      s_t' IN { x : TIME | x < r_t} ;
      s_d' = B
```

In a manner analogous to forward communication, the receiver may complete the event by executing the following transition. One subtlety of this transition is that the receiver may also signal whether it is ready to resume forward communication (`r_d' = F`) or wishes to engage in subsequent backward events (`r_d'= B`). The latter occurs when the sender has offered a new time that is not sufficiently in the past to satisfy the needs of the receiver.

Trans 5.
```
(s_b /= r_b) AND (s_d = B) --> r_b' = s_b;
          r_d' IN { x : DIR | x = B or x = F };
          r_t' = s_t;
```

While the protocol presented supports both forward and backwards communication, the sender may be blocked waiting for a response from a receiver when it wishes to backtrack. The following transition allows the sender to *request* that a forward transaction be retracted.

Trans 6.
```
(s_b /= r_b) AND (s_d = F) --> s_d' = I
```

The receiver may either accept the original offer to communicate (Trans 2) or allow the retraction:

Trans 7.
```
(s_b /= r_b) AND (s_d = I) -->  r_b' = s_b;
          r_d' IN {x : DIR | x =  r_d or x = B};
```

Similarly a blocked receiver may signal the sender that it wishes to backtrack.

Trans 8.
```
(s_b = r_b) -->  r_d' = B
```

In a distributed environment, where channel state changes made by the sender or receiver take time to propagate, these two rules introduce potential race conditions. Our SAL model accounts for race conditions by verifying a model where communication is buffered. Thus, the invariants proved using the SAL tools are valid in a distributed environment. Our SAL model includes a "shadow variable," `sync`, that the receiver sets whenever it accepts a communication event and clears whenever it rejects one. Whenever the sender holds the token, `sync` is true exactly when `r_t >= s_t`. Thus, the sender can determine which of the racing events occurred.

Consider that the sender may attempt to retract a forward request while the receiver simultaneously acknowledges that request. Similarly, the receiver may decide, after it has acknowledged a request, that it wishes to backtrack. In either case the later decision "overwrites" state that may or may not have been seen by the partner. For our semantic model, it is crucial that the sender be able to determine whether the synchronization event occurred or was successfully retracted. Indeed we prove a key invariant:

```
(s_b = r_b) => ((s_t <= r_t)) = sync)
```

Thus, when the sender holds the token, it can determine whether its last communication request was completed.

5 Low-Level Processes

Our low-level model is derived from the high-level model by implementing those rules involving synchronization using finer-grained rules based upon the the protocol model. Necessarily, there are more transition rules associated with the low-level model. For example, the single high-level transition implementing forward communication requires three transitions (two internal and one external or visible) in the low-level model. High level transitions not involving communication (H1, H3, H4, and H6) are adopted in the low-level model with minimal changes to account for the differences in channel state. Due to space limitations, we consider only those transitions relating to forward communication. We use these transitions to illustrate how low-level events "map" to high-level events.

We define the state of a channel as a tuple:

$$(s : (n, t, b, d, v), r : (n, t, b, d))$$

where s is the state of the sender and r is the state of the receiver; $s.n$ is the sender id and $r.n$ is the receiver id. As mentioned, the sender and receiver both maintain (non-negative) time-stamps ($s.t$, $r.t$) and Boolean tokens ($s.b$, $r.b$). Each also maintains a direction flag ($s.d$, $r.d$) indicating "forward" or "backward" synchronization. The sender state includes a value $s.v$ to be transferred when communication occurs. These state elements correspond the those of the channel protocol.

Where the high-level semantics included rules H2 and H5 that require simultaneous changes in two processes, none of the low-level rules affects more than one process. Indeed, the only preconditions on any of our low-level rules are the state of a single process and the state of its channels. Furthermore, these rules modify only the process state and the portion of a channel state (send or receive) owned by the process.

Forward communication (H2 in the High-level model) is executed in three steps by the underlying channel protocol.

1. In the first step, which corresponds to **Trans 1** of the SAL model, the sender initiates the communication. The sender marks its state as "in progress" with the new expression $\underline{\mathsf{send}}\ \ell\ v$, which has no direct equivalent in the high-level model and which may only occur as a result of this transition rule.

$$\begin{aligned}&\Xi\{\ell \mapsto (s:(n,t_s,b,-,-), r:(n_r,t_r,b,F)\} \fatsemi \langle n@t:\ \Gamma, E[\mathsf{send}\ \ell\ v]\rangle\\ &\xrightarrow{\epsilon}\\ &\Xi\{\ell \mapsto (s:(n,t'_s,\bar{b},F,v), r:(n_r,t_r,b,F)\} \fatsemi \langle n@t:\ \Gamma, E[\underline{\mathsf{send}}\ \ell\ v]\rangle\end{aligned} \tag{L1}$$

   Where $t'_s > t_r$. Recall that the sender has the "token" when the two channel token bits are equal ($s.b = r.b$), and initiates communication by inverting its token bit ($s.b$).

2. In the second step, (**Trans 2**) the receiver "sees" that the sender has initiated communication, reads the data, updates its local virtual time, updates the channel's time, and flips its token bit to enable the sender to take the next and final step in the communication. (Note that the sender stays blocked until the receiver takes this step.) After taking this step the receiver can proceed with its execution.

$$\begin{aligned}&\Xi\{\ell \mapsto (s:(n_s,t_s,b,d_s,v), r:(n,t_r,\bar{b},F)\} \fatsemi \langle n@t:\ \Gamma, E[\mathsf{recv}\ \ell(x).e]\rangle\\ &\xrightarrow{\ell@t[v]}\\ &\Xi\{\ell \mapsto (s:(n_s,t_s,b,d_s,v), r:(n,t,b,F)\} \fatsemi \langle n@t:\ \Gamma, E[e[v/x]]\rangle\end{aligned} \tag{L2}$$

   Where $t > \max(t_s, t)$ and $d_s \in \{F, I\}$. From L1 we can show that $t_s > t_r$. A required invariant for our model is that when the conditions of this rule are satisfied, the sender n_s is executing $\underline{\mathsf{send}}\ \ell$.

3. In the final step, the sender notes that its active communication event has been acknowledged by the receiver. It updates its local time and unblocks.

$$\begin{aligned}&\Xi\{\ell \mapsto (s:(n,t_s,b,d_s,v), r:(n_r,t_r,b,d_r)\} \fatsemi \langle n@t:\ \Gamma, E[\underline{\mathsf{send}}\ \ell\ v]\rangle\\ &\xrightarrow{\epsilon}\\ &\Xi\{\ell \mapsto (s:(n,t_s,b,d_s,v), r:(n_r,t_r,b,d_r)\} \fatsemi \langle n@t_r:\ \Gamma, E[()]\rangle\end{aligned} \tag{L3}$$

   Where $d_s \neq B$ and $t_s \leq t_r$.

6 Refinement Mapping Proof Outline

We have (partially) presented two semantic models and and a channel protocol. We can use the high-level model to inform programmers about expected program behavior, the low-level model to guide implementers, and the channel protocol to help prove invariant properties about the low-level model. In this section we outline the refinement mapping that we used to prove that the low level processes (henceforth LP); i.e., a function that maps every state of LP to a state of HP and where every transition of LP maps to a transition of HP.

Following Abadi and Lamport [1], a specification S is defined by (Σ, F, N) where Σ is a state space, F is the set of initial states, and N is the next-state relation. To prove that $S_1 = (\Sigma_1, F_1, N_1)$ implements $S_2 = (\Sigma_1, F_2, N_2)$ we need to define a mapping $f : \Sigma_1 \to \Sigma_2$ such that:

R2. For all $f(F_1) \subseteq F_2$ (f takes initial states into initial states.)

R3. If $\langle s, t\rangle \in N_1$ then $\langle f(s), f(t)\rangle \in N_2$ or $f(s) = f(t)$. (A state transition allowed by N_1 is mapped into a [possibly stuttering] transition allowed by N_2.)

Note that we have omitted the portions of the Abadi and Lamport definition that handle supplemental properties along with rules R1 and R4 which deal with externally visible state and supplemental properties (respectively).

The states of HP are defined by parallel composition of a finite set of processes as defined by syntax of Section 3.2. The initial states of HP are those where every context stack is empty, every process is of the form $\langle n@0 :\ \bullet, \mathsf{stable}\ (\lambda\_.e)\ ()\rangle$, and every channel is at time 0. The initial states of LP are exactly the initial states of HP but where each channel state has two additional token bits, both false, two clocks, both 0, and two direction flags, both F.

We can quickly dispense with mapping rule R2 by stating that our mapping function converts channel maps of LP to those of HP by dropping the additional state information, and maps the syntax to LP to that of HP except for two process forms, neither of which is permitted in the initial state:

$$\langle n@t :\ \Gamma, \underline{\mathsf{send}}\ \ell\ e\rangle$$
$$\langle n@t :\ \Gamma, \underline{\mathsf{backtrack}}\ \ell\ e\rangle$$

$\underline{\mathsf{send}}\ \ell$ was discussed in Section 5 where it was used to capture the intermediate state of a forward communication transaction; $\underline{\mathsf{backtrack}}\ \ell$ serves as an intermediate state for the backtracking rules that we have omitted due to space limitations.

As an example of mapping transition rules, consider the following cases for mapping of $\underline{\mathsf{send}}\ \ell\ v$. The first case corresponds to the state after transition L1 and the second to the state after transition L2. (recall that ℓ is a channel, and $\ell.x.y$ correspond to fields of the channel state).

$$f(E[\underline{\mathsf{send}}\ \ell\ v]) = E[\mathsf{send}\ \ell\ v]\ \mathtt{if}\ (\ell.s.b \neq \ell.r.b) \vee \ell.s.t > \ell.r.t$$
$$f(E[\underline{\mathsf{send}}\ \ell\ v]) = E[()]\ \mathtt{if}\ (\ell.s.b = \ell.r.b) \wedge \ell.s.t \leq \ell.r.t$$

Thus in the mapping, L1 is a “silent” transition and L2, which is only executed in parallel with a process executing $\underline{\mathsf{send}}\ \ell\ v$, corresponds to high-level transition H2.

Validating the (complete) refinement mapping requires proving that every LP transition maps to a HP transition.

7 Discussion

We have introduced a CSP based language supporting reversible distributed computing along with two semantic models – a high-level model in which synchronous events are modeled by transitions that affect two processes simultaneously, and a low-level model in which transitions affect a single process. These two models are related by a verified communication protocol which is the basis for the finer grained transitions of the low-level model. We outlined a refinement mapping that we developed proving that the low-level model implements the high-level model. This proof required the invariants of the protocol that were verified with the SAL model checker. We have also proved that the high-level model obeys sensible causal ordering properties even in the face of backtracking.

While our Scala language implementation is somewhat richer than the simple models presented here (e.g., it supports communication choice, and dynamic process and channel creation); at its core it is implemented exactly as indicated by our low-level model. Channels are implemented via message passing where the messages carry the channel state of our protocol. Processes are implemented as Java threads. Processes learn that their peers wish to backtrack by examining the (local) state of their channels. Stable sections consist of: saving the channel timestamps on the context stack, executing the Stable code in a try/catch block, and popping the stack; backtracking is implemented by throwing an exception. The implementation required approximately 1200 lines of code. [3]

[3]Download: http://cs.indiana.edu/~geobrown/places-code.tar.gz.

This paper provides clear evidence that implementing reversible communicating processes in a distributed manner is both feasible and, from the perspective of communication overhead, relatively efficient. As we noted, our "high-level" model is unsatisfying because it exposes the programmer to the mechanics of backtracking. In our current model, even when a process has decided that it wishes to backtrack, its peers may continue forward execution for a period during which they may learn from their peers. If we were to restrict our attention to traditional roll-back recovery, where nothing is "learned" from unsuccessful forward execution, this could easily be abstracted. We continue to work towards a "compromise" between traditional rollback and the unrestricted model we have presented.

Acknowledgment This material is based upon work supported by the National Science Foundation under grant numbers 1116725 and 1217454 and while the first author was an NSF program officer. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, May 1991.

[2] V. Danos and J. Krivine. *Transactions in RCCS*, pages 398–412. Springer-Verlag, London, UK, 2005.

[3] V. Danos, J. Krivine, and F. Tarissan. Self-assembling trees. *Electron. Notes Theor. Comput. Sci.*, 175:19–32, May 2007.

[4] Leonardo de Moura. SAL: tutorial. Technical report, SRI International, 2004.

[5] Leonardo de Moura, Sam Owre, and N. Shankar. The SAL language manual. Technical report, SRI International, 2002.

[6] Edsko de Vries, Vasileios Koutavas, and Matthew Hennessy. Communicating transactions. In *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, pages 569–583. Springer-Verlag, 2010.

[7] Bruno Dutertre and Maria Sorea. Timed systems in SAL. Technical Report SRI-SDL-04-03, SRI International, 2004.

[8] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[9] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, pages 56–66, February 1988.

[10] Rachid Guerraoui. Foundations of speculative distributed computing. In Nancy Lynch and Alexander Shvartsman, editors, *Distributed Computing*, volume 6343 of *Lecture Notes in Computer Science*, pages 204–205. Springer-Verlag, 2010.

[11] M. Hermenegildo and K. Greene. The &-prolog system: Exploiting independent and-parallelism. *New Generation Computing*, 9:233–256, 1991.

[12] Ralf Hinze. Deriving backtracking monad transformers. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 186–197. ACM, 2000.

[13] Tony Hoare. Compensable transactions. In Peter Mller, editor, *Advanced Lectures on Software Engineering*, volume 6029 of *Lecture Notes in Computer Science*, pages 21–40. Springer Berlin Heidelberg, 2010.

[14] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, SOSP '87, pages 77–93. ACM, 1987.

[15] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7:404–425, 1985.

[16] Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Controlling reversibility in higher-order Pi. In *Proceedings of the 22nd international conference on Concurrency theory*, CONCUR'11, pages 297–311. Springer-Verlag, 2011.

[17] Ivan Lanese, ClaudioAntares Mezzina, and Jean-Bernard Stefani. Reversing higher-order Pi. In Paul Gastin and Franois Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 478–493. Springer Berlin Heidelberg, 2010.

[18] Mohsen Lesani and Jens Palsberg. Communicating memory transactions. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 157–168. ACM, 2011.

[19] Jing Li, Huibiao Zhu, and Jifeng He. Specifying and verifying web transactions. In Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, editors, *Formal Techniques for Networked and Distributed*

Systems FORTE 2008, volume 5048 of *Lecture Notes in Computer Science*, pages 149–168. Springer Berlin Heidelberg, 2008.

[20] Michael Lienhardt, Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. A reversible abstract machine and its space overhead. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems*, volume 7273 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2012.

[21] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1988.

[22] Iain Phillips and Irek Ulidowski. Reversibility and models for concurrency. *Electron. Notes Theor. Comput. Sci.*, 192:93–108, October 2007.

[23] John H. Reppy. *Concurrent programming in ML*. Cambridge University Press, New York, NY, USA, 1999.

[24] Lukasz Ziarek and Suresh Jagannathan. Lightweight checkpointing for concurrent ML. *J. Funct. Program.*, 20(2):137–173, March 2010.

Retractable Contracts[*]

Franco Barbanera[1], Mariangiola Dezani-Ciancaglini[2] [†]
Ivan Lanese[3] [‡]and Ugo de'Liguoro[2] [§]

[1] Dipartimento di Matematica e Informatica, University of Catania, barba@dmi.unict.it
[2] Dipartimento di Informatica, University of Torino, dezani@di.unito.it, deliguoro@di.unito.it
[3] Dipartimento di Informatica - Scienza e Ingegneria, University of Bologna/INRIA, ivan.lanese@gmail.com

Abstract

In calculi for modelling communication protocols, internal and external choices play dual roles. Two external choices can be viewed naturally as dual too, as they represent an agreement between the communicating parties. If the interaction fails, the past agreements are good candidates as points where to roll back, in order to take a different agreement. We propose a variant of contracts with synchronous rollbacks to agreement points in case of deadlock. The new calculus is equipped with a compliance relation which is shown to be decidable.

1 Introduction

In human as well as automatic negotiations, an interesting feature is the ability of rolling back to some previous point in case of failure, undoing previous choices and possibly trying a different path. *Rollbacks* are familiar to the users of web browsers, and so are also the troubles that these might cause during "undisciplined" interactions. Clicking the "back" button, or going to some previous point in the chronology when we are in the middle of a transaction, say the booking of a flight, can be as smart as dangerous. In any case, it is surely a behaviour that service programmers want to discipline. Also the converse has to be treated with care: a server discovering that an auxiliary service becomes available after having started a conversation could take advantage of it using some kind of rollback. However, such a server would be quite unfair if the rollback were completely hidden from the client.

Let us consider an example. A Buyer is looking for a bag ($\overline{\mathsf{bag}}$) or a belt ($\overline{\mathsf{belt}}$); she will decide how to pay, either by credit card ($\overline{\mathsf{card}}$) or by cash ($\overline{\mathsf{cash}}$), after knowing the price from a Seller. The Buyer behaviour can be described by the process:

$$\mathsf{Buyer} = \overline{\mathsf{bag}}.\mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) \oplus \overline{\mathsf{belt}}.\mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}})$$

where dot is sequential composition and $\oplus$ is internal choice. The Seller does not accept credit card payments for items of low price, like belts, but only for more expensive ones, like bags:

$$\mathsf{Seller} = \mathsf{belt}.\overline{\mathsf{price}}.\mathsf{cash} + \mathsf{bag}.\overline{\mathsf{price}}.(\mathsf{card} + \mathsf{cash})$$

where $+$ is external choice. According to contract theory [6], Buyer is not compliant with Seller, since she can choose to pay the belt by card. Also, there is no obvious way to represent the

[*]This work was partially supported by Italian MIUR PRIN Project CINA Prot. 2010LHT4KM and COST Action IC1201 BETTY.
[†]This author was partially supported by the Torino University/Compagnia San Paolo Project SALT.
[‡]This author was partially supported by the French ANR project REVER n. ANR 11 INSE 007.
[§]This author was partially supported by the Torino University/Compagnia San Paolo Project SALT.

buyer's will to be free in her decision about the payment and be compliant with a seller without asking the seller in advance. Nonetheless, when interacting with Seller, the buyer's decision is actually free at least in the case of purchase of a bag. For exploiting such a possibility the client (but also the server) should be able to tolerate a partial failure of her protocol, and to try a different path.

To this aim we add to (some) choices a possibility of rollback, in case the taken path fails to reach a success configuration. In this setting, choices among outputs are no more purely internal, since the environment may oblige to undo a wrong choice and choose a different alternative. For this reason, we denote choices between outputs which allow rollback as external, hence we use Buyer′ below instead of Buyer:

$$\mathsf{Buyer}' = \overline{\mathsf{bag}}.\mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) + \overline{\mathsf{belt}}.\mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}})$$

We thus explore a model of contract interaction in which synchronous rollback is triggered when client and server fail to reach an agreement.

In defining our model we build over some previous work reported in [2], where we have considered contracts with rollbacks. However, we depart from that model on three main aspects. First, in the present model rollback is used in a disciplined way to tolerate failures in the interaction, thus improving compatibility, while in [2] it is an internal decision of either client or server, which makes compatibility more difficult. Second, we embed checkpoints in the structure of contracts, avoiding explicit checkpoints. Third, we consider a stack of "pasts", called histories, instead of just one past for each participant, as in [2], thus allowing to undo many past choices looking for a successful alternative.

2 Contracts for retractable interactions

Our contracts can be obtained from the session behaviours of [1] or from the session contracts of [3] just adding external retractable choices between outputs.

Definition 2.1 (Retractable Contracts). *Let $\mathcal{N}$* (set of names) *be some countable set of symbols and $\overline{\mathcal{N}} = \{\overline{a} \mid a \in \mathcal{N}\}$* (set of conames)*, with $\mathcal{N} \cap \overline{\mathcal{N}} = \emptyset$. The set* RC *of* **retractable contracts** *is defined as the set of the* **closed** *expressions generated by the following grammar,*

$$\begin{array}{rcll}
\sigma, \rho & := & \mid\ \mathbf{1} & \textit{success} \\
 & & \mid\ \sum_{i \in I} a_i.\sigma_i & \textit{(retractable) input} \\
 & & \mid\ \sum_{i \in I} \overline{a}_i.\sigma_i & \textit{retractable output} \\
 & & \mid\ \bigoplus_{i \in I} \overline{a}_i.\sigma_i & \textit{unretractable output} \\
 & & \mid\ x & \textit{variable} \\
 & & \mid\ \mathsf{rec}\, x.\sigma & \textit{recursion}
\end{array}$$

where I is non-empty and finite, the names and the conames in choices are pairwise distinct and σ is not a variable in $\mathsf{rec}\, x.\sigma$.

Note that recursion in RC is guarded and hence contractive in the usual sense. We take an equi-recursive view of recursion by equating $\mathsf{rec}\, x.\sigma$ with $\sigma[\mathsf{rec}\, x.\sigma/x]$. We use α to range over $\mathcal{N} \cup \overline{\mathcal{N}}$, with the convention $\overline{\alpha} = \begin{cases} \overline{a} & \text{if } \alpha = a, \\ a & \text{if } \alpha = \overline{a}. \end{cases}$

We write $\alpha_1.\sigma_1 + \alpha_2.\sigma_2$ for binary input/retractable output and $\overline{a}_1.\sigma_1 \oplus \overline{a}_2.\sigma_2$ for binary unretractable output. They are both commutative by definition. Also, $\overline{a}.\sigma$ may denote both unary retractable choice and unary unretractable choice. This is not a source of confusion since they have the same semantics.

From now on we call just *contracts* the expressions in RC. They are written by omitting all trailing $\mathbf{1}$'s.

In order to deal with rollbacks we decorate contracts with histories, which memorise the alternatives in choices which have been discharged. We use '$\circ$' as a placeholder for *no-remaining-alternatives*.

Definition 2.2 (Contracts with histories)**.** *Let* $\mathsf{Histories}$ *be the expressions (referred to also as* stacks*) generated by the grammar:*

$$\boldsymbol{\gamma} ::= [\,] \mid \boldsymbol{\gamma}:\sigma$$

where $\sigma \in \mathsf{RC} \cup \{\circ\}$ *and* $\circ \notin \mathsf{RC}$. *Then the set of* contracts with histories *is defined by:*

$$\mathsf{RCH} = \{\boldsymbol{\gamma} \prec \sigma \mid \boldsymbol{\gamma} \in \mathsf{Histories}, \sigma \in \mathsf{RC} \cup \{\circ\}\,\}.$$

We write just $\sigma_1 : \cdots : \sigma_k$ for the stack $(\cdots([\,]:\sigma_1):\cdots):\sigma_k$. With a little abuse of notation we use '$:$' also to concatenate histories, and to add contracts in front of histories.

We can now discuss the operational semantics of our calculus (Definition 2.3). The reduction rule for the internal choice ($\oplus$) is standard, but for the presence of the $\boldsymbol{\gamma} \prec \cdot$. Whereas, when reducing retractable choices ($+$), the discharged branches are memorised. When a single action is executed, the history is modified by adding a '$\circ$', intuitively meaning that the only possible branch has been tried and no alternative is left. Rule (rb) recovers the contract on the top of the stack, replacing the current one with it.

Definition 2.3 (LTS of Contracts with Histories)**.**

$$(+) \quad \boldsymbol{\gamma} \prec \alpha.\sigma + \sigma' \xrightarrow{\alpha} \boldsymbol{\gamma}:\sigma' \prec \sigma \qquad\qquad (\oplus) \quad \boldsymbol{\gamma} \prec \overline{a}.\sigma \oplus \sigma' \xrightarrow{\tau} \boldsymbol{\gamma} \prec \overline{a}.\sigma$$

$$(\alpha) \quad \boldsymbol{\gamma} \prec \alpha.\sigma \xrightarrow{\alpha} \boldsymbol{\gamma}:\circ \prec \sigma \qquad\qquad (\mathsf{rb}) \quad \boldsymbol{\gamma}:\sigma' \prec \sigma \xrightarrow{\mathsf{rb}} \boldsymbol{\gamma} \prec \sigma'$$

The interaction of a client with a server is modelled by the reduction of their parallel composition, that can be either forward, consisting of CCS style synchronisations and single internal choices, or backward, only when there is no possible forward reduction, and the client is not satisfied, i.e. it is different from $\mathbf{1}$.

Definition 2.4 (TS of Client/Server Pairs)**.** *We define the relation* $\longrightarrow$ *over pairs of contracts with histories by the following rules:*

$$\frac{\boldsymbol{\delta} \prec \rho \xrightarrow{\alpha} \boldsymbol{\delta}' \prec \rho' \qquad \boldsymbol{\gamma} \prec \sigma \xrightarrow{\overline{\alpha}} \boldsymbol{\gamma}' \prec \sigma'}{\boldsymbol{\delta} \prec \rho \parallel \boldsymbol{\gamma} \prec \sigma \longrightarrow \boldsymbol{\delta}' \prec \rho' \parallel \boldsymbol{\gamma}' \prec \sigma'}\ (\mathsf{comm})$$

$$\frac{\boldsymbol{\delta} \prec \rho \xrightarrow{\tau} \boldsymbol{\delta} \prec \rho'}{\boldsymbol{\delta} \prec \rho \parallel \boldsymbol{\gamma} \prec \sigma \longrightarrow \boldsymbol{\delta} \prec \rho' \parallel \boldsymbol{\gamma} \prec \sigma}\ (\tau)$$

$$\frac{\boldsymbol{\gamma} \prec \rho \xrightarrow{\mathsf{rb}} \boldsymbol{\gamma}' \prec \rho' \qquad \boldsymbol{\delta} \prec \sigma \xrightarrow{\mathsf{rb}} \boldsymbol{\delta}' \prec \sigma' \qquad \rho \neq \mathbf{1}}{\boldsymbol{\gamma} \prec \rho \parallel \boldsymbol{\delta} \prec \sigma \longrightarrow \boldsymbol{\gamma}' \prec \rho' \parallel \boldsymbol{\delta}' \prec \sigma'}\ (\mathsf{rbk})$$

plus the rule symmetric to (τ) *w.r.t.* $\parallel$. *Moreover, rule* (rbk) *applies only if neither* (comm) *nor* (τ) *do.*

We will use $\xrightarrow{*}$ and $\not\longrightarrow$ with the standard meanings.
Notice that, since '$\circ$' cannot synchronise with anything, in case a partner rolls back to a '$\circ$', it is forced to recover an *older* past (if any).

The following examples show the different behaviours of retractable and unretractable outputs. We decorate arrows with the name of the used reduction rule. As a first example we consider a possible reduction of the process discussed in the Introduction.

Example 2.5. As in the Introduction, let $\mathsf{Buyer}' = \overline{\mathsf{bag}}.\mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) + \overline{\mathsf{belt}}.\mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}})$ be a client and $\mathsf{Seller} = \mathsf{belt}.\overline{\mathsf{price}}.\mathsf{cash} + \mathsf{bag}.\overline{\mathsf{price}}.(\mathsf{card} + \mathsf{cash})$ a server; then

$$\begin{array}{lrcl}
 & [\,] \prec \mathsf{Buyer}' & \| & [\,] \prec \mathsf{Seller} \\
\xrightarrow{\mathsf{comm}} & \overline{\mathsf{bag}}.\mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) \prec \mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) & \| & \mathsf{bag}.\overline{\mathsf{price}}.(\mathsf{card} + \mathsf{cash}) \prec \overline{\mathsf{price}}.\mathsf{cash} \\
\xrightarrow{\mathsf{comm}} & \overline{\mathsf{bag}}.\mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) : \circ \prec (\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) & \| & \mathsf{bag}.\overline{\mathsf{price}}.(\mathsf{card} + \mathsf{cash}) : \circ \prec \mathsf{cash} \\
\xrightarrow{\tau} & \overline{\mathsf{bag}}.\mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) : \circ \prec \overline{\mathsf{card}} & \| & \mathsf{bag}.\overline{\mathsf{price}}.(\mathsf{card} + \mathsf{cash}) : \circ \prec \mathsf{cash} \\
\xrightarrow{\mathsf{rbk}} & \overline{\mathsf{bag}}.\mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) \prec \circ & \| & \mathsf{bag}.\overline{\mathsf{price}}.(\mathsf{card} + \mathsf{cash}) \prec \circ \\
\xrightarrow{\mathsf{rbk}} & [\,] \prec \overline{\mathsf{bag}}.\mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) & \| & [\,] \prec \mathsf{bag}.\overline{\mathsf{price}}.(\mathsf{card} + \mathsf{cash}) \\
\xrightarrow{\mathsf{comm}} & \circ \prec \mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) & \| & \circ \prec \overline{\mathsf{price}}.(\mathsf{card} + \mathsf{cash}) \\
\xrightarrow{\mathsf{comm}} & \circ : \circ \prec (\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) & \| & \circ : \circ \prec (\mathsf{card} + \mathsf{cash}) \\
\xrightarrow{\tau} & \circ : \circ \prec \overline{\mathsf{card}} & \| & \circ : \circ \prec (\mathsf{card} + \mathsf{cash}) \\
\xrightarrow{\mathsf{comm}} & \circ : \circ : \circ \prec \mathbf{1} & \| & \circ : \circ : \mathsf{cash} \prec \mathbf{1} \\
\not\longrightarrow & & &
\end{array}$$

Example 2.6. Let $\rho = \mathsf{rec}\, x.(\overline{b}.x \oplus \overline{a}.c.x)$ and $\sigma = \mathsf{rec}\, x.(b.x + a.\overline{e}.x)$. The following reduction sequence leads the parallel composition of these contracts to a deadlock.

$$\begin{array}{rlccl}
\rho \parallel \sigma & \xrightarrow{\tau} & [\,] \prec \overline{a}.c.\rho & \| & [\,] \prec \mathsf{rec}\, x.(b.x + a.\overline{e}.x) \\
 & \xrightarrow{\mathsf{comm}} & \circ \prec c.\rho & \| & b.\sigma \prec \overline{e}.\sigma \\
 & \xrightarrow{\mathsf{rbk}} & [\,] \prec \circ & \| & [\,] \prec b.\sigma \\
 & \not\longrightarrow & & &
\end{array}$$

Example 2.7. Let us now modify the above example by using retractable outputs in the client, so making the two contracts in parallel always reducible. The following reduction shows that there can be an infinite number of rollbacks in a sequence, even if it is not possible to have an infinite reduction containing only rollbacks. Notice how the stack keeps growing indefinitely.

Let $\rho = \mathsf{rec}\, x.(\overline{b}.x + \overline{a}.c.x)$ and $\sigma = \mathsf{rec}\, x.(b.x + a.\overline{e}.x)$.

$$\begin{array}{rcrcl}
\rho \parallel \sigma & \xrightarrow{\mathsf{comm}} & \overline{b}.\rho \prec c.\rho & \parallel & b.\sigma \prec \overline{e}.\sigma \\
& \xrightarrow{\mathsf{rbk}} & [\,] \prec \overline{b}.\rho & \parallel & [\,] \prec b.\sigma \\
& \xrightarrow{\mathsf{comm}} & \circ \prec \rho & \parallel & \circ \prec \sigma \\
& \xrightarrow{\mathsf{comm}} & \circ : \overline{b}.\rho \prec c.\rho & \parallel & \circ : b.\sigma \prec \overline{e}.\sigma \\
& \xrightarrow{\mathsf{rbk}} & \circ \prec \overline{b}.\rho & \parallel & \circ \prec b.\sigma \\
& \xrightarrow{\mathsf{comm}} & \circ : \circ \prec \rho & \parallel & \circ : \circ \prec \sigma \\
& & & \vdots &
\end{array}$$

3 Compliance

The compliance relation for standard contracts consists in requiring that, whenever no reduction is possible, all client requests and offers have been satisfied, i.e. the client is in the success state $\mathbf{1}$. For retractable contracts we can adopt the same definition.

Definition 3.1 (Compliance Relation $\dashv\!\vdash$)**.**

i) The relation $\dashv\!\vdash$ *on contracts with histories is defined by:*
$\boldsymbol{\delta} \prec \rho \dashv\!\vdash \boldsymbol{\gamma} \prec \sigma$ *holds whenever* $\boldsymbol{\delta} \prec \rho \parallel \boldsymbol{\gamma} \prec \sigma \xrightarrow{*} \boldsymbol{\delta'} \prec \rho' \parallel \boldsymbol{\gamma'} \prec \sigma' \not\longrightarrow$ *implies* $\rho' = \mathbf{1}$ *for any* $\boldsymbol{\delta'}, \rho', \boldsymbol{\gamma'}, \sigma'$.

ii) The relation $\dashv\!\vdash$ *on contracts is defined by:*
$$\rho \dashv\!\vdash \sigma \quad \textit{iff} \quad [\,] \prec \rho \dashv\!\vdash [\,] \prec \sigma.$$

We now provide a formal system characterising compliance on retractable contracts. The judgments are of the shape $\Gamma \ \triangleright\ \rho \dashv \sigma$, where Γ is a set of expressions of the form $\rho' \dashv \sigma'$. We write $\triangleright\ \rho \dashv \sigma$ when Γ is empty. The only non standard rule is rule $(+,+)$, which assures compliance of two retractable choices when they contain respectively a name and the corresponding coname followed by compliant contracts. This contrasts with the rules $(\oplus,+)$ and $(+,\oplus)$, where all conames in unretractable choices between outputs must have corresponding names in the choices between inputs, followed by compliant contracts.

Definition 3.2 (Formal System for Compliance)**.**

$$\frac{}{\Gamma \ \triangleright\ \mathbf{1} \dashv \sigma}\,(\textsc{Ax}) \qquad \frac{}{\Gamma, \rho \dashv \sigma \ \triangleright\ \rho \dashv \sigma}\,(\textsc{Hyp}) \qquad \frac{\Gamma, \alpha.\rho + \rho' \dashv \overline{\alpha}.\sigma + \sigma' \ \triangleright\ \rho \dashv \sigma}{\Gamma \ \triangleright\ \alpha.\rho + \rho' \dashv \overline{\alpha}.\sigma + \sigma'}\,(+,+)$$

$$\frac{\forall i \in I.\ \Gamma, \bigoplus_{i\in I} \overline{a}_i.\rho_i \dashv \sum_{j\in I\cup J} a_j.\sigma_j \ \triangleright\ \rho_i \dashv \sigma_i}{\Gamma \ \triangleright\ \bigoplus_{i\in I} \overline{a}_i.\rho_i \dashv \sum_{j\in I\cup J} a_j.\sigma_j}\,(\oplus,+)$$

$$\frac{\forall i \in I.\ \Gamma, \sum_{j\in I\cup J} a_j.\sigma_j \dashv \bigoplus_{i\in I} \overline{a}_i.\rho_i \ \triangleright\ \rho_i \dashv \sigma_i}{\Gamma \ \triangleright\ \sum_{j\in I\cup J} a_j.\sigma_j \dashv \bigoplus_{i\in I} \overline{a}_i.\rho_i}\,(+,\oplus)$$

Notice that rule $(+,+)$ implicitly represents the fact that, in the decision procedure for two contracts made of retractable choices, the possible synchronising branches have to be tried, until either a successful one is found or all fail.

Example 3.3. Let us formally show that, for the Buyer' and Seller of the Introduction, we have $\mathsf{Buyer}' \dashv \mathsf{Seller}$.
For the sake of readability, let
$\Gamma' = \mathsf{Buyer}' \dashv \mathsf{Seller},\ \mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) \dashv \overline{\mathsf{price}}.(\mathsf{card}+\mathsf{cash})$ and $\Gamma'' = \Gamma',\ \overline{\mathsf{card}} \oplus \overline{\mathsf{cash}} \dashv \mathsf{card}+\mathsf{cash}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma'' \ \rhd\ \mathbf{1} \dashv \mathbf{1}}\,(\textsc{Ax}) \qquad \dfrac{}{\Gamma'' \ \rhd\ \mathbf{1} \dashv \mathbf{1}}\,(\textsc{Ax})}{\Gamma' \ \rhd\ \overline{\mathsf{card}} \oplus \overline{\mathsf{cash}} \dashv \mathsf{card} + \mathsf{cash}}\,(\oplus,+)}{\mathsf{Buyer}' \dashv \mathsf{Seller} \ \rhd\ \mathsf{price}.(\overline{\mathsf{card}} \oplus \overline{\mathsf{cash}}) \dashv \overline{\mathsf{price}}.(\mathsf{card} + \mathsf{cash})}\,(+,+)}{\rhd\ \mathsf{Buyer}' \dashv \mathsf{Seller}}\,(+,+)$$

Example 3.4. The contracts of Example 2.7 can be formally proved to be compliant by means of the following derivation in our formal system. Actually such a derivation can be looked at as the result of the decision procedure implicitly described by the formal system.

$$\dfrac{\dfrac{}{\overline{b}.\rho + \overline{a}.c.\rho \dashv b.\sigma + a.\overline{e}.\sigma \ \rhd\ \rho \dashv \sigma}\,(\textsc{Hyp})}{\rhd\ \overline{b}.\rho + \overline{a}.c.\rho \dashv b.\sigma + a.\overline{e}.\sigma}\,(+,+)$$

In applying the rules we exploit the fact that we consider contracts modulo recursion fold/unfold.

We can show that derivability in this formal system is decidable, since it is syntax directed and it does not admit infinite derivations.

We denote by $\mathcal{D}$ a derivation in the system of Definition 3.2. The procedure **Prove** in Figure 1 clearly implements the formal system, that is it is straightforward to check the following

Fact 3.5.

i) $\mathbf{Prove}(\Gamma \ \rhd\ \rho \dashv \sigma) \neq \mathbf{fail}$ *iff* $\Gamma \ \rhd\ \rho \dashv \sigma$.

ii) $\mathbf{Prove}(\Gamma \ \rhd\ \rho \dashv \sigma) = \mathcal{D} \neq \mathbf{fail}$ *implies* $\overset{\mathcal{D}}{\Gamma \ \rhd\ \rho \dashv \sigma}$

Theorem 3.6. *Derivability in the formal system is decidable.*

Proof. By Fact 3.5, we only need to show that the procedure **Prove** always terminates. Notice that in all recursive calls $\mathbf{Prove}(\Gamma, \rho \dashv \sigma \ \rhd\ \rho_k \dashv \sigma_k)$ inside $\mathbf{Prove}(\Gamma \ \rhd\ \rho \dashv \sigma)$ the expressions ρ_k and σ_k are subexpressions of ρ and σ respectively (because of unfolding of recursion they can also be ρ and σ). Since contract expressions generate regular trees, there are only finitely many such subexpressions. This implies that the number of different calls of procedure **Prove** is always finite. □

In the remaining of this section we will show the soundness and the completeness of the formal system using some auxiliary lemmas.

```
**Prove**$(\Gamma \ \rhd \ \rho \dashv \sigma)$
if    $\rho = \mathbf{1}$   then   $\dfrac{}{\Gamma \ \rhd \ \mathbf{1} \dashv \sigma}$ (Ax)
else  if  $\rho \dashv \sigma \in \Gamma$  then  $\dfrac{}{\Gamma, \rho \dashv \sigma \ \rhd \ \rho \dashv \sigma}$ (Hyp)
else  if  $\rho = \sum_{i \in I} \alpha_i.\rho_i$  and $\sigma = \sum_{j \in J} \overline{\alpha}_j.\sigma_j$
          and  exists $k \in I \cap J$ s.t. $\mathcal{D} = \mathbf{Prove}(\Gamma, \rho \dashv \sigma \ \rhd \ \rho_k \dashv \sigma_k) \neq \mathbf{fail}$
      then  $\dfrac{\mathcal{D}}{\Gamma \ \rhd \ \rho \dashv \sigma}$ $(+,+)$   else fail
else  if  $\rho = \bigoplus_{i \in I} \overline{a}_i.\rho_i$  and  $\sigma = \sum_{j \in J} a_j.\sigma_j$ and $I \subseteq J$
          and  for all $k \in I$  $\mathcal{D}_k = \mathbf{Prove}(\Gamma, \rho \dashv \sigma \ \rhd \ \rho_k \dashv \sigma_k) \neq \mathbf{fail}$
      then  $\dfrac{\forall k \in I \ \ \mathcal{D}_k}{\Gamma \ \rhd \ \rho \dashv \sigma}$ $(\oplus,+)$
else  if  $\rho = \sum_{i \in I} a_i.\rho_i$  and  $\sigma = \bigoplus_{j \in J} \overline{a}_j.\sigma_j$ and $I \supseteq J$
          and  for all $k \in J$  $\mathcal{D}_k = \mathbf{Prove}(\Gamma, \rho \dashv \sigma \ \rhd \ \rho_k \dashv \sigma_k) \neq \mathbf{fail}$
      then  $\dfrac{\forall k \in J \ \ \mathcal{D}_k}{\Gamma \ \rhd \ \rho \dashv \sigma}$ $(+,\oplus)$   else fail
else  fail
```

Figure 1: The procedure **Prove**.

Soundness It is useful to show that if a configuration is stuck, then both histories are empty. This is a consequence of the fact that the property "the histories of client and server have the same length" is preserved by reductions.

Lemma 3.7. *If $\boldsymbol{\delta} \prec \rho' \parallel \boldsymbol{\gamma} \prec \sigma' \not\longrightarrow$, then $\boldsymbol{\delta} = \boldsymbol{\gamma} = [\,]$.*

Proof. Clearly $\boldsymbol{\delta} \prec \rho' \parallel \boldsymbol{\gamma} \prec \sigma' \not\longrightarrow$ implies either $\boldsymbol{\delta} = [\,]$ or $\boldsymbol{\gamma} = [\,]$. Observe that:

- rule (comm) adds one element to both stacks;
- rule (τ) does not modify both stacks;
- rule (rbk) removes one element from both stacks.

Then starting from two stacks containing the same number of elements, the reduction always produces two stacks containing the same number of elements. So $\boldsymbol{\delta} = [\,]$ implies $\boldsymbol{\gamma} = [\,]$ and vice versa. □

Theorem 3.8 (Soundness)**.** *If $\rhd \ \rho \dashv \sigma$, then $\rho \dashv\!\!| \ \sigma$.*

Proof. The proof is by contradiction. Assume $\rho \not\dashv\!\!| \ \sigma$. Then there is a reduction

$$[\,] \prec \rho \parallel [\,] \prec \sigma \xrightarrow{*} [\,] \prec \rho' \parallel [\,] \prec \sigma' \not\longrightarrow$$

with $\rho' \neq \mathbf{1}$. Note that both the histories are empty thanks to Lemma 3.7. We proceed by induction on the number n of steps in the reduction.

Let us consider the base case ($n = 0$). In this case $\rho \neq \mathbf{1}$ and there is no possible synchronization. Rule Ax is not applicable since $\rho \neq \mathbf{1}$. Rule Hyp is not applicable since Γ is empty. The other rules are not applicable otherwise we would have a possible synchronization.

Let us consider the inductive case. We have a case analysis on the topmost operators in ρ and σ. Let us start with the case where both topmost operators are retractable sums, i.e., $\rho = a.\rho_k + \rho''$ and $\sigma = \overline{a}.\sigma_k + \sigma''$. Thus,

$$[\,] \prec a.\rho_k + \rho'' \parallel [\,] \prec \overline{a}.\sigma_k + \sigma'' \xrightarrow{*} [\,] \prec \rho'' \parallel [\,] \prec \sigma'' \xrightarrow{*} [\,] \prec \rho' \parallel [\,] \prec \sigma' \not\longrightarrow .$$

By definition $\rho'' \not\dashv\!\mid \sigma''$, and by inductive hypothesis $\not\vdash \rho'' \dashv \sigma''$. Also,

$$[\,] \prec \rho_k \parallel [\,] \prec \sigma_k \xrightarrow{*} [\,] \prec \rho'_k \parallel [\,] \prec \sigma'_k \not\longrightarrow .$$

By definition $\rho_k \not\dashv\!\mid \sigma_k$, and by inductive hypothesis $\not\vdash \rho_k \dashv \sigma_k$. Hence, rule $(+, +)$ is not applicable since no possible premise holds. Rule Hyp is not applicable since Γ is empty. The other rules are not applicable since the term does not have the correct shape.

Let us consider the case where $\rho = a.\rho_k + \rho''$ and $\sigma = \overline{a}.\sigma_k \oplus \sigma''$ (the other is symmetric). Thus we have a derivation

$$\begin{array}{l} [\,] \prec a.\rho_k + \rho'' \parallel [\,] \prec \overline{a}.\sigma_k \oplus \sigma'' \longrightarrow [\,] \prec a.\rho_k + \rho'' \parallel [\,] \prec \overline{a}.\sigma_k \longrightarrow \\ \rho'' \prec \rho_k \parallel \circ \prec \sigma_k \xrightarrow{*} \rho'' \prec \rho''' \parallel \circ \prec \sigma''' \longrightarrow [\,] \prec \rho'' \parallel [\,] \prec \circ \not\longrightarrow . \end{array}$$

Then we also have $[\,] \prec \rho_k \parallel [\,] \prec \sigma_k \xrightarrow{*} [\,] \prec \rho''' \parallel [\,] \prec \sigma''' \not\longrightarrow$. By definition $\rho_k \not\dashv\!\mid \sigma_k$, and by inductive hypothesis $\not\vdash \rho_k \dashv \sigma_k$. Then rule $(+, \oplus)$ cannot be applied and the thesis follows.

The case of two $\oplus$ follows trivially since there is no applicable rule. □

Completeness The following lemma proves that compliance is preserved by the concatenation of histories to the left of the current histories.

Lemma 3.9. *If $\boldsymbol{\delta} \prec \rho \dashv\!\mid \boldsymbol{\gamma} \prec \sigma$, then $\boldsymbol{\delta}' : \boldsymbol{\delta} \prec \rho \dashv\!\mid \boldsymbol{\gamma}' : \boldsymbol{\gamma} \prec \sigma$ for all $\boldsymbol{\delta}'$, $\boldsymbol{\gamma}'$.*

Proof. It suffices to show that

$$\boldsymbol{\delta} \prec \rho \dashv\!\mid \boldsymbol{\gamma} \prec \sigma \text{ implies } \rho' : \boldsymbol{\delta} \prec \rho \dashv\!\mid \boldsymbol{\gamma} \prec \sigma \text{ and } \boldsymbol{\delta} \prec \rho \dashv\!\mid \sigma' : \boldsymbol{\gamma} \prec \sigma$$

which we prove by contraposition.

Suppose that $\rho' : \boldsymbol{\delta} \prec \rho \not\dashv\!\mid \boldsymbol{\gamma} \prec \sigma$; then

$$\rho' : \boldsymbol{\delta} \prec \rho \parallel \boldsymbol{\gamma} \prec \sigma \xrightarrow{*} \boldsymbol{\delta}' \prec \rho'' \parallel \boldsymbol{\gamma}' \prec \sigma'' \not\longrightarrow \text{ and } \rho'' \neq \mathbf{1}$$

If ρ' is never used, then $\boldsymbol{\delta}' = \rho' : \boldsymbol{\delta}''$ and $\boldsymbol{\gamma}' = [\,]$, so that we get

$$\boldsymbol{\delta} \prec \rho \parallel \boldsymbol{\gamma} \prec \sigma \xrightarrow{*} \boldsymbol{\delta}'' \prec \rho'' \parallel [\,] \prec \sigma'' \not\longrightarrow$$

Otherwise we have that

$$\rho' : \boldsymbol{\delta} \prec \rho \parallel \boldsymbol{\gamma} \prec \sigma \xrightarrow{*} \rho' \prec \rho'' \parallel \boldsymbol{\gamma}' \prec \sigma'' \longrightarrow [\,] \prec \rho' \parallel \boldsymbol{\gamma}'' \prec \sigma'''$$

and we assume that $\xrightarrow{*}$ is the shortest such reduction. It follows that $\rho'' \neq \mathbf{1}$. By the minimality assumption about the length of $\xrightarrow{*}$ we know that ρ' neither has been restored by some previous application of rule (rbk), nor pushed back into the stack before. We get

$$\boldsymbol{\delta} \prec \rho \parallel \boldsymbol{\gamma} \prec \sigma \xrightarrow{*} [\,] \prec \rho'' \parallel \boldsymbol{\gamma}'' \prec \sigma'' \not\longrightarrow$$

In both cases we conclude that $\boldsymbol{\delta} \prec \rho \not\dashv\!\mid \boldsymbol{\gamma} \prec \sigma$ as desired.

Similarly we can show that $\boldsymbol{\delta} \prec \rho \not\dashv\!\mid \sigma' : \boldsymbol{\gamma} \prec \sigma$ implies $\boldsymbol{\delta} \prec \rho \not\dashv\!\mid \boldsymbol{\gamma} \prec \sigma$. □

The following lemma gives all possible shapes of compliant contracts. It is the key lemma for the proof of completeness.

Lemma 3.10. *If $\rho \dashv\!\vdash \sigma$, then one of the following conditions holds:*

1. $\rho = \mathbf{1}$;
2. $\rho = \sum_{i\in I} \alpha_i.\rho_i$, $\sigma = \sum_{j\in J} \overline{\alpha}_j.\sigma_j$ *and* $\exists k \in I \cap J.\ \rho_k \dashv\!\vdash \sigma_k$;
3. $\rho = \bigoplus_{i\in I} \overline{a}_i.\rho_i$, $\sigma = \sum_{j\in J} a_j.\sigma_j$, $I \subseteq J$ *and* $\forall k \in I.\ \rho_k \dashv\!\vdash \sigma_k$;
4. $\rho = \sum_{i\in I} a_i.\rho_i$, $\sigma = \bigoplus_{j\in J} \overline{a}_j.\sigma_j$, $I \supseteq J$ *and* $\forall k \in J.\ \rho_k \dashv\!\vdash \sigma_k$.

Proof. By contraposition and by cases of the possible shapes of ρ and σ.

Suppose $\rho = \sum_{i\in I} \alpha_i.\rho_i$, $\sigma = \sum_{j\in J} \overline{\alpha}_j.\sigma_j$, $I \cap J = \{k_1, \dots, k_n\}$ and $\rho_{k_i} \not\dashv\!\vdash \sigma_{k_i}$ for $1 \le i \le n$. Then we get

$$[\,] \prec \rho_{k_i} \parallel [\,] \prec \sigma_{k_i} \xrightarrow{*} \boldsymbol{\delta}_i \prec \rho'_i \parallel \boldsymbol{\gamma}_i \prec \sigma'_i \not\longrightarrow$$

for $1 \le i \le n$, where $\rho'_i \neq \mathbf{1}$ and $\boldsymbol{\delta}_i = \boldsymbol{\gamma}_i = [\,]$ by Lemma 3.7. This implies

$$\textstyle\sum_{i\in I\setminus\{k_1\}} \alpha_i.\rho_i \prec \rho_{k_1} \parallel \sum_{j\in J\setminus\{k_1\}} \overline{\alpha}_j.\sigma_j \prec \sigma_{k_1} \xrightarrow{*} \sum_{i\in I\setminus\{k_1\}} \alpha_i.\rho_i \prec \rho'_1 \parallel \sum_{j\in J\setminus\{k_1\}} \overline{\alpha}_j.\sigma_j \prec \sigma'_1$$

by Lemma 3.9. Let $I' = I \setminus J$ and $J' = J \setminus I$. We can reduce $[\,] \prec \rho \parallel [\,] \prec \sigma$ only as follows:

$$\begin{array}{rcll}
[\,] \prec \rho \parallel [\,] \prec \sigma & \longrightarrow & \sum_{i\in I\setminus\{k_1\}} \alpha_i.\rho_i \prec \rho_{k_1} \parallel \sum_{j\in J\setminus\{k_1\}} \overline{\alpha}_j.\sigma_j \prec \sigma_{k_1} & \text{by (comm)} \\
& \xrightarrow{*} & \sum_{i\in I\setminus\{k_1\}} \alpha_i.\rho_i \prec \rho'_1 \parallel \sum_{j\in J\setminus\{k_1\}} \overline{\alpha}_j.\sigma_j \prec \sigma'_1 & \\
& \longrightarrow & [\,] \prec \sum_{i\in I\setminus\{k_1\}} \alpha_i.\rho_i \parallel [\,] \prec \sum_{j\in J\setminus\{k_1\}} \overline{\alpha}_j.\sigma_j & \text{by (rbk)} \\
& \vdots & \qquad\qquad\vdots & \\
& \xrightarrow{*} & \sum_{i\in I'} \alpha_i.\rho_i \prec \rho'_n \parallel \sum_{j\in J'} \overline{\alpha}_j.\sigma_j \prec \sigma'_n & \\
& \longrightarrow & [\,] \prec \sum_{i\in I'} \alpha_i.\rho_i \parallel [\,] \prec \sum_{j\in J'} \overline{\alpha}_j.\sigma_j & \text{by (rbk)}
\end{array}$$

and $[\,] \prec \sum_{i\in I'} \alpha_i.\rho_i \parallel [\,] \prec \sum_{j\in J'} \overline{\alpha}_j.\sigma_j$ is stuck since $I' \cap J' = \emptyset$.

Suppose $\rho = \bigoplus_{i\in I} \overline{a}_i.\rho_i$ and $\sigma = \sum_{j\in J} a_j.\sigma_j$. If $I \not\subseteq J$ let $k \in I \setminus J$; then we get

$$\begin{array}{rcll}
[\,] \prec \rho \parallel [\,] \prec \sigma & \longrightarrow & [\,] \prec \overline{a}_k.\rho_k \parallel [\,] \prec \sigma & \text{by } (\tau) \\
& \not\longrightarrow & &
\end{array}$$

Otherwise $I \subseteq J$ and $\rho_k \not\dashv\!\vdash \sigma_k$ for some $k \in I$. By reasoning as above we have

$$[\,] \prec \rho_k \parallel [\,] \prec \sigma_k \xrightarrow{*} [\,] \prec \rho' \parallel [\,] \prec \sigma' \not\longrightarrow$$

and

$$\circ \prec \rho_k \parallel \textstyle\sum_{j\in J\setminus\{k\}} a_j.\sigma_j \prec \sigma_k \xrightarrow{*} \circ \prec \rho' \parallel \sum_{j\in J\setminus\{k\}} a_j.\sigma_j \prec \sigma'$$

which imply

$$\begin{array}{rcll}
[\,] \prec \rho \parallel [\,] \prec \sigma & \longrightarrow & [\,] \prec \overline{a}_k.\rho_k \parallel [\,] \prec \sigma & \text{by } (\tau) \\
& \longrightarrow & \circ \prec \rho_k \parallel \sum_{j\in J\setminus\{k\}} a_j.\sigma_j \prec \sigma_k & \text{by (comm)} \\
& \xrightarrow{*} & \circ \prec \rho' \parallel \sum_{j\in J\setminus\{k\}} a_j.\sigma_j \prec \sigma' & \\
& \longrightarrow & [\,] \prec \circ \parallel [\,] \prec \sum_{j\in J\setminus\{k\}} a_j.\sigma_j & \text{by (rbk)} \\
& \not\longrightarrow & &
\end{array}$$

In both cases we conclude that $\rho \not\dashv\!\vdash \sigma$.

The proof for the case $\rho = \sum_{i\in I} a_i.\rho_i$, $\sigma = \bigoplus_{j\in J} \overline{a}_j.\sigma_j$ is similar. □

Theorem 3.11 (Completeness)**.** *If* $\rho \dashv\!\!\mid \sigma$*, then* $\rhd\ \rho \dashv \sigma$.

Proof. By Theorem 3.6 each computation of **Prove**$(\rhd\ \rho \dashv \sigma)$ always terminates. By Lemma 3.10 and Fact 3.5, $\rho \dashv\!\!\mid \sigma$ implies that **Prove**$(\rhd\ \rho \dashv \sigma) \neq$ **fail**, and hence $\rhd\ \rho \dashv \sigma$. □

4 Related work and conclusions

Since the pioneering work by Danos and Krivine [7], reversible concurrent computations have been widely studied. A main point is that understanding which actions can be reversed is not trivial in a concurrent setting, since there is no unique "last" action. Since [7], the most common notion of reversibility in concurrency is *causal-consistent* reversibility: any action can be undone if no other action depending on it has been executed (and not yet undone). The name highlights the relation with causality, which makes the approach applicable even in settings where there is no unique notion of time, but makes it quite complex.

The first calculus for which a causal-consistent reversible extension has been defined is CCS in [7], using a stack of memories for each thread. Later, causal-consistent reversible extensions have been defined by Phillips and Ulidowski [14] for calculi definable by SOS rules in a general format (without mobility), using keys to bind synchronised actions together, and by Lanese et al. [11] for the higher-order π-calculus, using explicit memory processes to store history information and tags to track causality. A survey of causal-consistent reversibility can be found in [12].

In [10], Lanese et al. enrich the calculus of [11] with a fine-grained rollback primitive, showing the subtleties of defining a rollback operator in a concurrent setting. The first paper exploring reversibility in a context of sessions (see, e.g., [13] for a comparison between session types and contracts) is [15], by Tiezzi and Yoshida. This paper defines the semantics for reversible sessions by adapting the approach in [11], but does not consider compliance. Compliance has been first studied in [2]. We already discussed the differences between the present work and [2] in the Introduction.

A main point of our approach is that it exploits the fact that contracts describe sequential interactions (in a concurrent setting) to avoid the complexity of causal-consistent reversibility, allowing for a simpler semantics (compared, e.g., to the one of [10]).

Similarly to our approach, long running transactions with compensations, and in particular interacting transactions [8], allow to undo past agreements. In interacting transactions, however, a new possibility is tried when an exception is raised, not when an agreement cannot be found as in our case. Also, the possible options are sorted: first the normal execution, then the compensation. Finally, compliance of interacting transactions has never been studied. In the field of sessions, the most related works are probably the ones studying exceptions in binary sessions [5] and in multi-party sessions [4]. As for transactions, they aim at dealing with exceptions more than at avoiding to get stuck since an agreement cannot be found.

We plan to investigate whether our approach can be extended to multi-party sessions [9], the rationale being that parallelism is controlled by the global type, hence possibly part of the complexity due to concurrency can be avoided. The sub-behaviour relation induced by our notion of compliance is also worth being thoroughly studied.

Acknowledgments. We are grateful to the anonymous reviewers for their useful remarks.

References

[1] Franco Barbanera and Ugo de'Liguoro. Two notions of sub-behaviour for session-based client/server systems. In *PPDP*, pages 155–164. ACM Press, 2010.

[2] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Compliance for reversible client/server interactions. In *BEAT*, volume 162 of *EPTCS*, pages 35–42, 2014.

[3] Giovanni Bernardi and Matthew Hennessy. Modelling session types using contracts. *Math. Struct. in Comp. Science*, 2014. To appear.

[4] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. In *FSTTCS*, volume 8 of *LIPIcs*, pages 338–351, 2010.

[5] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

[6] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Trans. on Prog. Lang. and Sys.*, 31(5):19:1–19:61, 2009.

[7] Vincent Danos and Jean Krivine. Reversible communicating systems. In *CONCUR*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004.

[8] Edsko de Vries, Vasileios Koutavas, and Matthew Hennessy. Communicating transactions - (extended abstract). In *CONCUR*, volume 6269 of *LNCS*, pages 569–583. Springer, 2010.

[9] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM Press, 2008.

[10] Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Controlling reversibility in higher-order pi. In *CONCUR*, volume 6901 of *LNCS*, pages 297–311. Springer, 2011.

[11] Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversing higher-order pi. In *CONCUR*, volume 6269 of *LNCS*, pages 478–493. Springer, 2010.

[12] Ivan Lanese, Claudio Antares Mezzina, and Francesco Tiezzi. Causal-consistent reversibility. *Bulletin of the EATCS*, 114, 2014.

[13] Cosimo Laneve and Luca Padovani. The pairing of contracts and session types. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 681–700, 2008.

[14] Iain C. C. Phillips and Irek Ulidowski. Reversing algebraic process calculi. *J. of Logic and Alg. Progr.*, 73(1-2):70–96, 2007.

[15] Francesco Tiezzi and Nobuko Yoshida. Towards reversible sessions. In *PLACES*, volume 155 of *EPTCS*, pages 17–24, 2014.