BEAT 2

2nd International Workshop on Behavioural Types

23rd & 24th September 2013

University Complutense of Madrid

Affiliated to SEFM (Software Engineering and Formal Methods) 2013

Preface

BEAT 2 (full title: 2nd International Workshop on Behavioural Types), affiliated to SEFM, follows on from the BEAT 2013 workshop, which was affiliated to POPL 2013, and an invitational meeting which took place in Lisbon in April 2011.

Behavioural type systems go beyond data type systems in order to specify, characterize and reason about dynamic aspects of program execution. Behavioural types encompass: session types; contracts (for example in service-oriented systems); typestate; types for analysis of termination, deadlock-freedom, liveness, race-freedom and related properties; intersection types applied to behavioural properties; and other topics. Behavioural types can form a basis for both static analysis and dynamic monitoring. Recent years have seen a rapid increase in research on behavioural types, driven partly by the need to formalize and codify communication structures as computing moves from the data-processing era to the communication era, and partly by the realization that type-theoretic techniques can provide insight into the fine structure of computation.

The aim of BEAT 2 is to bring together researchers in all aspects of behavioural type theory and its applications, in order to share results, consolidate the community, and discover opportunities for new collaborations and future directions. The workshop was organised under the auspices of COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems (BETTY), and the Programme Committee for the workshop was formed by taking a representative from each country participating in BETTY.

Papers were submitted in two categories: original research papers, and presentations of papers already published elsewhere. There is also an invited lecture from Dr Achim Brucker of SAP, whose participation is funded by COST Action IC1201 and by the workshop registration fees. The workshop programme was completed by several talks offered by members of BETTY and by participants in the workshop, as follows:

- Typing Actors using Behavioural Types
  Adrian Francalanza and Joseph Masini
- Linear types in programming languages: progress and prospects
  Simon Gay
- Types for resources in psi-calculi
  Hans Hüttel
- Globally Governed Session Semantics
  Dimitrios Kouzapas and Nobuko Yoshida
- Behaviour inference for deadlock checking
  Violet Ka I Pun, Martin Steffen and Volker Stoltz
- Specification and Verification of Protocols for MPI Programs
  Eduardo R. B. Marques, Francisco Martins, Vasco T. Vasconcelos, Nicholas Ng, Nuno Dias Martins, César Santos and Nobuko Yoshida

Finally, I would like to thank the programme committee members for their hard work, and the SEFM workshop chair and local organizers for their help.

Simon Gay, PC Chair

Organisation

Programme Committee Chair

Simon Gay	University of Glasgow, UK

Programme Committee

Karthikeyan Bhargavan	INRIA Paris-Rocquencourt, France
Gabriel Ciobanu	Romanian Academy, ICS, Iaşi, Romania
Ricardo Colomo Palacios	Universidad Carlos III de Madrid, Spain
Ugo de'Liguoro	University of Torino, Italy
Adrian Francalanza	University of Malta, Malta
Tihana Galinac Grbac	University of Rijeka, Croatia
Vaidas Giedrimas	Šiauliai University, Lithuania
Thomas Hildebrandt	IT University of Copenhagen, Denmark
Einar Broch Johnsen	University of Oslo, Norway
Georgia Kapitsaki	University of Cyprus, Cyprus
Vasileios Koutavas	Trinity College Dublin, Ireland
Aleksandra Mileva	Goce Delčev University of štip, Macedonia
Samir Omanović	University of Sarajevo, Bosnia and Herzegovina
Jovanka Pantović	University of Novi Sad, Serbia
Nikolaos Sismanis	(Aristotle University of Thessaloniki, Greece
Peter Thiemann	University of Freiburg, Germany
Vasco Vasconcelos	University of Lisbon, Portugal
Björn Victor	Uppsala University, Sweden
PawełT. Wojciechowski	Poznań University of Technology, Poland
Peter Wong	SDL Fredhopper, The Netherlands

Contents

Service Compositions: Curse or Blessing for Security (Abstract)

Achim Brucker

SAP AG

Building large systems by composing reusable services is not a new idea, it is at least 25 years old. Still, only recently the scenario of dynamic interchangeable services that are consumed via public networks is becoming reality. Following the Software as a Service (SaaS) paradigm, an increasing number of complex applications is offered as a service that themselves can be used composed for building even larger and more complex applications. This will lead to situations in which users are likely to unknowingly consume services in a dynamic and ad hoc manner.

Leaving the rather static (and mostly on-premise) service composition scenarios of the past 25 years behind us, dynamic service compositions, have not only the potential to transform the software industry from a business perspective, they also requires new approaches for addressing the security, trustworthiness needs of users.

The EU FP7 project Aniketos develops new technology, methods, tools and security services that support the design-time creation and run-time dynamic behaviour of dynamic service compositions, addressing service developers, service providers and service end users.

In this talk, we will motivate several security and trustworthiness requirements that occur in dynamic service compositions and discuss the solutions developed within the project Aniketos. Based on our experiences, we will discuss open research challenges and potential opportunities for potential opportunities for applying type systems.

Behavioural Types Inspired by Cellular Thresholds

Bogdan Aman and Gabriel Ciobanu

Romanian Academy, Institute of Computer Science
Blvd. Carol I no.8, 700505 Iaşi, Romania
baman@iit.tuiasi.ro, gabriel@info.uaic.ro

Abstract

The sodium-potassium exchange pump is a transmembrane transport protein that establishes and maintains the appropriate internal concentrations of sodium and potassium ions in cells. This exchange is an important physiological process, and it is critical in maintaining the osmotic balance of the cell. Inspired by the functioning of this pump, we introduce and study a ratio-based type system using thresholds in a bio-inspired formalism. The intent is to avoid errors in the definition of the formal model used to model biologic processes. For this type system we prove a subject reduction theorem.

1 Introduction

Cell membranes are crucial to the life of the cell. Defining the boundary of the living cells, membranes have various functions and participate in many essential cell activities including barrier functions, transmembrane signalling and intercellular recognition. The sodium-potassium exchange pump [13] is a transmembrane transport protein in the plasma membrane that establishes and maintains the appropriate internal ratio of sodium (Na^+) and potassium ions (K^+) in cells. By using energy from the hydrolysis of ATP molecules, the pump extrudes three Na^+ ions, in exchange for two K^+. This exchange is an important physiological process, and it is critical in maintaining the osmotic balance of the cell, the resting membrane potential of most tissues and the excitability properties of muscle and nerve cells. Limitations on the values of the Na^+/K^+ ratio, together with their significance are described in [12]. If this ratio is unbalanced, it indicates physiological malfunctions within the cell: an unbalanced sodium/potassium ratio is associated with heart, kidney, liver, and immune deficiency diseases. The sodium/potassium ratio is also linked to adrenal gland function [9]. For example, the intracellular Na^+/K^+ ratios of the normal epithelial cells fall in a rather narrow range; their average is 0.11 [14].

Types articulate computation in a given paradigm. The traditional notion of types offers abstractions for data, objects and operations on them. The basic form of behavioural types articulates the ways in which interactions are performed. In this paper we introduce behavioural types inspired by cellular thresholds. We associate to each system a set of constraints that must be satisfied in order to assure that the application of the rules to a well-formed membrane system leads to a well-formed membrane system as well. We have a two-stage approach to the description of biological behaviours: the first describes reactions in an “untyped” setting, and the second rules out certain evolutions by imposing thresholds. This allows one to treat separately different aspects of modelling: which transitions are possible at all, and under which circumstances they can take place.

Membrane systems represent a formalism used to describe biological systems [15], thus also the sodium/potassium pump [3]. The model has compartments enclosed by membranes, floating objects, proteins associated with the internal and external surfaces of the membranes, and built-in proteins (the pump) that transport and process chemical substances. Evolution rules represent the formal counterpart of chemical reactions, and are given in the form of rewriting rules that operate on the objects, as well as on the compartmentalised structure (e.g., by dissolving, dividing, creating, or moving membranes).

2 A Multiset Model of Membranes

Given a finite set O of symbols, the set of all strings over O is denoted by O^*, and the set of all non-empty strings over O is denoted by $O^+ = O^* \backslash \lambda$, where λ is the empty string. A multiset over O [16] is a map $u : O \to \mathbb{N}$, where $u(a)$ denotes the multiplicity of the symbol $a \in O$ in the multiset u; $|u| = \sum_{a \in O} u(a)$ denotes the total number of objects appearing in a multiset u. A multiset u is included into a multiset v (denoted by $u \subseteq v$) if $u(a) \leq v(a)$, $\forall a \in O$. An object a is included into a multiset u (denoted by $a \in u$) if $u(a) > 0$. For $a \in O$, we write a instead of the multiset u if $u(a) = 1$ and $u(b) = 0$ for all $b \neq a$. The empty multiset is denoted by ϵ, and $\epsilon(a) = 0$, $\forall a \in O$. For two multisets u and v, we define the sum $u+v$ by $(u+v)(a) = u(a)+v(a)$, $\forall a \in O$, and the difference $u - v$ by $(u - v)(a) = max\{0, u(a) - v(a)\}$, $\forall a \in O$.

In what follows we work with terms ranged over by st, st_1, $\ldots$, that are built by means of a membrane constructor $[-]_{-}$, using a set O of objects. The syntax of terms $st \in ST$ is $st ::= u \mid [st]_v \mid st\ st$, where u denotes a (possibly empty) multiset of objects placed inside a membrane, v a multiset of objects within or on the surface of a membrane, and $st\ st$ is the parallel composition. Since we work with multisets of terms, we introduce a structural congruence relation following a standard approach from process algebra. The defined structural congruence is the least equivalence relation on terms satisfying also the rule: if $v_1 \equiv v_2$ and $st_1 \equiv st_2$ then $[st_1]_{v_1} \equiv [st_2]_{v_2}$.

A pattern is a term that may include variables. We denote with $\mathcal{P}$ the infinite set of patterns P of the form: $P ::= st \mid [\ P\ X\ ]_{v\ y} \mid P\ P$. We distinguish between "simple variables" (ranged over by x, y, z) that may occur only on the surface of membranes (i.e., they can be replaced only by multisets of objects) and "term variables" (ranged over by X, Y, Z) that may only occur inside regions (they can be replaced by arbitrary terms). Therefore, we assume two disjoint sets: V_{O^*} (set of simple variables) and V_{ST^*} (set of term variables). We denote by $V = V_{O^*} \cup V_{ST^*}$ the set of all variables, and with ρ any variable in V.

An instantiation is a partial function $\sigma : V \to ST^*$ that preserves the type of all variables: simple variables ($x \in V_{O^*}$) and term variables ($X \in V_{ST^*}$) are mapped into objects ($\sigma(x) \in O^*$) and terms ($\sigma(X) \in ST^*$), respectively. Given a pattern P, the term obtained by replacing all occurrences of each variable $\rho \in V$ with the term $\sigma(\rho)$ is denoted by $P\sigma$. The set of all possible instantiations is denoted by Σ, and the set of all variables appearing in P is denoted by $Var(P)$.

Formally, a rewriting rule r is a pair of patterns (P_1, P_2), denoted by $P_1 \to P_2$, where $P_1 \neq \epsilon$ (P_1 is a non-empty pattern) and $Var(P_1) \subseteq Var(P_2)$. A rewriting rule $P_1 \to P_2$ states that a term $P_1\sigma$ can be transformed into the term $P_2\sigma$, for some instantiation function σ.

The notion of context is used to complete the definition of a rewriting semantics for membrane systems with integral proteins. This is done by enriching the syntax with a new object $\square$ representing a hole. By definition, each context contains a single hole $\square$. The infinite set $\mathcal{C}$ of contexts (ranged over by C) is given by: $C ::= \square \mid C\ st \mid [\ C\ ]_v$.

Given $C_1, C_2 \in \mathcal{C}$, $C_1[\ st\ ]$ denotes the term obtained by replacing $\square$ with st in C_1, while $C_1[\ C_2\ ]$ denotes the context obtained by replacing $\square$ with C_2 in C_1.

Given a membrane system with integral proteins and a set of rewriting rules R, the reduction semantics of the system is the least transition relation $\to$ satisfying the following rule:

$$\frac{P_1 \to P_2 \in R \quad P_1\sigma \neq \epsilon \quad \sigma \in \Sigma \quad C \in \mathcal{C}}{C[P_1\sigma] \to C[P_2\sigma]}\ .$$

$\to^*$ denotes the reflexive and transitive closure of $\to$.

Unfortunately, the limitations of certain ratios (e.g., Na^+/K^+, ATP/ADP) as described in [12] cannot be modelled using the above class of membrane systems. From a physiological point of view, these ratios are important, as well as the lower and upper limits for the objects involved in ion channel transport. For this reason we define a type system for membrane systems with integral proteins that is able to impose constraints on the evolution with respect to the ratios between objects.

3 Ratio-based Type System Over Multisets

An important idea of type theory is to provide the possibility of distinguishing between different classes of objects (types). Types are fundamental both in logic and computer science, and have many applications. Recently it has been used in biological formalisms in order to transfer the complexity of biological properties from evolution rules to types. The syntax of types is simple, easy to understand and use, and these aspects make types ideal for expressing general constraints. The type system could be also used to decrease the number of rules in some models by defining a limited number of generic rules as in [1].

The behaviour of typed terms can be controlled by a type system in order to avoid unwanted evolutions. According to [12], the evolution of a healthy cell ensures that the ratio between objects (e.g., Na^+/K^+) of a cell is kept between certain values. We investigate the application of our type system to membrane systems with integral proteins.

Let T be a finite set of basic types ranged over by t. We classify each object in O with a unique element of T; we use Γ to denote this classification. In general, different objects a and b can have the same basic type t. When there is no ambiguity, we denote the type associated with an object a by t_a. We assume the existence of two functions, $min : T \times T \to (0, \infty) \cup \{\diamond\}$ and $max : T \times T \to (0, \infty) \cup \{\diamond\}$ for each ordered pair of basic types (t_1, t_2). These functions indicate the minimum and maximum ratio between the number of objects of basic types t_1 and t_2 that can be present inside a membrane.

For example, by considering the constraints $min(t_a, t_b) = 3$ and $max(t_a, t_b) = 5$, the number of objects of basic type t_a is larger than the number of objects of basic type t_b with a coefficient between three and five. $min(t_1, t_2) = \diamond$ and $max(t_1, t_2) = \diamond$ mean that these functions are undefined for the pair of types (t_1, t_2). Biologically speaking, the ratio between the types t_1 and t_2 is either unknown, or can be ignored.

We consider only local properties: the objects influence each other only if

- they are present inside the same membrane;
- they are integral on sibling membranes;
- one is present inside and the other is integral to the membrane;
- one is present outside and the other is integral to the membrane.

Definition 1 (Consistent Basic Types). *A system consisting of a set of basic types T and the functions min and max is consistent if:*

1. $\forall t_1, t_2 \in T$, $min(t_1, t_2) \neq \diamond$ *iff* $max(t_1, t_2) \neq \diamond$;
2. $\forall t_1, t_2 \in T$ *if* $min(t_1, t_2) \neq \diamond$ *then* $min(t_1, t_2) \leq max(t_1, t_2)$;
3. $\forall t_1, t_2 \in T$ *if* $min(t_1, t_2) \neq \diamond$ *and* $max(t_2, t_1) \neq \diamond$ *then* $min(t_1, t_2) \cdot max(t_2, t_1) = 1$.

The meaning of these constraints is explained below:

1. the minimum ratio between the number of objects of basic types t_1 and t_2 is defined iff the corresponding maximum ratio is defined;
2. the minimum ratio between the number of objects of basic types t_1 and t_2 must be lower than the maximum ratio between the number of objects of basic types t_1 and t_2;
3. the maximum ratio between the number of objects of basic types t_2 and t_1 must be equal to the inverse of the minimum ratio between the number of objects of types t_1 and t_2.

Definition 2 (Quantitative types). *Are triples (L, Pr, U) over the set T of basic types, where:*

- *L (lower) is the set of minimum ratios between basic types;*
- *Pr (present) is the multiset of basic types of present objects (the objects present at the top level of a pattern, i.e. in the outermost membrane);*
- *U (upper) is the set of maximum ratios between basic types.*

The number of objects of type t appearing in Pr is denoted by $Pr(t)$. In order to define well-formed types, the sets Pr_t (present), L_t (lower bounds) and U_t (upper bounds) are required:

- $Pr_t = \bigcup_{t' \in Pr} \left\{ t/t' : \dfrac{Pr(t)}{Pr(t')} \mid t \neq t', Pr(t') \neq 0 \right\}$;
- $L_t = \bigcup_{t' \in T} \{t/t' : min(t, t') \mid t \neq t', min(t, t') \neq \diamond\}$;
- $U_t = \bigcup_{t' \in T} \{t/t' : max(t, t') \mid t \neq t', max(t, t') \neq \diamond\}$.

These sets contain labelled values in order to be able to refer to them when needed: e.g., $t/t' : \dfrac{Pr(t)}{Pr(t')}$ denotes the fact that the ratio between the objects of types t and t' that are present in Pr has the label t/t' and the value $\dfrac{Pr(t)}{Pr(t')}$.

Definition 3 (Well-Formed Types). *A type (L, Pr, U) is well-formed if*

$$L = \bigcup_{t \in Pr} L_t,\ U = \bigcup_{t \in Pr} U_t \text{ and } L \leq \bigcup_{t \in Pr} Pr_t \leq U.$$

The constraints of this definition can be read as follows:

- $L = \bigcup_{t \in Pr} L_t$ - contains the minimum ratio constraints for the present objects;
- $U = \bigcup_{t \in Pr} U_t$ - contains the maximum ratio constraints for the present objects;
- $L \leq \bigcup_{t \in Pr} Pr_t$ - the ratio between present objects respects the minimum ratio from L: if for all $(t/t' : min(t, t')) \in L$ and $(t/t' : \dfrac{Pr(t)}{Pr(t')}) \in \bigcup_{t \in Pr} Pr_t$, then $min(t, t') \leq \dfrac{Pr(t)}{Pr(t')}$;
- $\bigcup_{t \in Pr} Pr_t \leq U$ - the ratio between present objects respects the maximum ratio from U: if for all $(t/t' : \dfrac{Pr(t)}{Pr(t')}) \in \bigcup_{t \in Pr} Pr_t$ and $(t/t' : max(t, t')) \in U$, then $\dfrac{Pr(t)}{Pr(t')} \leq max(t, t')$.

From now on we work only with well-formed types. For instance, the two well-formed types (L, Pr, U) and (L', Pr', U') of the following two definitions are constructed by using the ratio tables of *min* and *max*.

Definition 4 (Meet and Join)**.** *The meet of sets L and L' (denoted by $L \sqcap L'$) is:*

$$L \sqcap L' = \{t/t' : min(t,t') \mid (t/t' : min(t,t')) \in L \text{ or } (t/t' : min(t,t')) \in L'\}.$$

The join of sets U and U' (denoted by $U \sqcup U'$) is:

$$U \sqcup U' = \{t/t' : max(t,t') \mid (t/t' : max(t,t')) \in U \text{ or } (t/t' : max(t,t')) \in U'\}.$$

Definition 5 (Type Compatibility)**.** *Two well-formed types (L, Pr, U) and (L', Pr', U') are compatible, written $(L, Pr, U) \bowtie (L', Pr', U')$, if*

- $min(t,t') \leq \dfrac{(Pr + Pr')(t)}{(Pr + Pr')(t')}$ *holds for all* $(t/t' : min(t,t')) \in L \sqcap L'$ *and*

$$(t/t' : \frac{(Pr + Pr')(t)}{(Pr + Pr')(t')}) \in \bigcup_{t \in (Pr+Pr')} (Pr + Pr')_t;$$

- $\dfrac{(Pr + Pr')(t)}{(Pr + Pr')(t')} \leq max(t,t')$ *holds for all* $(t/t' : max(t,t')) \in U \sqcup U'$ *and*

$$(t/t' : \frac{(Pr + Pr')(t)}{(Pr + Pr')(t')}) \in \bigcup_{t \in (Pr+Pr')} (Pr + Pr')_t.$$

Definition 6. *A basis Δ assigning types to simple and term variables is defined by*

$$\Delta ::= \emptyset \mid \Delta, x : (L_t, t, U_t) \mid \Delta, X : (L, Pr, U).$$

A basis is well-formed if all types in the basis are well-formed.

A classification Γ maps each object in O to a unique element of the set T of basic types. The judgements are of the form$\Delta \vdash P : (L, Pr, U)$ indicating that a pattern P is well-typed having the type (L, Pr, U) relative to a typing environment Δ .

Types are assigned to patterns and terms according to the typing rules in Table 1. It is not difficult to verify that a derivation starting from well-formed bases produces only well-formed bases and well-formed types. The rules are rather trivial, except for the rules $(TPar)$ and $(TMem)$. The type of a parallel composition given by the $(TPar)$ rule is derived from the types of the two sub-patterns; if two patterns P and P' are compatible, then the type of the obtained pattern $P\ P'$ is derived from the types (L, Pr, U) and (L', Pr', U') of the connected patterns where $Pr + Pr'$ is the multiset sum of the present types pr and Pr', the minimum meet type $L \sqcap L'$ is and the maximum join type $U \sqcup U'$ are as in Definition 4. The type of rule $(TMem)$ is the type of the multiset of integral proteins v (because a membrane makes the objects inside it invisible to the outside). Since the objects on the membrane are influenced by the ones inside it, the type of the multiset placed on the membrane and the type of the pattern placed inside the membrane must be compatible in order to obtain the overall type of the membrane.

We define a typed semantics, since we are interested in applying reduction rules only to correct terms having well-formed types, and whose requirements are satisfied. More formally, a term st is correct if $\emptyset \vdash st : (L, Pr, U)$ for some well-formed type (L, Pr, U). An instantiation σ agrees with a basis Δ (denoted by $\sigma \in \Sigma_\Delta$) if $\rho : (L, Pr, U) \in \Delta$ implies $\emptyset \vdash \sigma(\rho) : (L, Pr, U)$.

In order to apply the rules in a safe way, we introduce a restriction on rules based on the context of application rather than on the type of patterns involved in the rule. In this direction, we characterise contexts by the types of terms that can fill their hole, and the rules by the types of terms produced by their application.

Definition 7 (Typed Holes)**.** *Given a context C and a type (L, Pr, U) that is well-formed, the type (L, Pr, U)* **fits the context** *C if for some well-formed type (L', Pr', U') it can be shown that $X : (L, Pr, U) \vdash C[X] : (L', Pr', U')$.*

Table 1: Typing Rules

$$\Delta \vdash \epsilon : (\emptyset, \emptyset, \emptyset) \quad (TEps) \qquad\qquad \frac{a : t \in \Delta}{\Delta \vdash a : (L_t, t, U_t)} \quad (TObj)$$

$$\Delta, \rho : (L, Pr, U) \vdash \rho : (L, Pr, U) \qquad (TVar)$$

$$\frac{\Delta \vdash v : (L, Pr, U) \quad \Delta \vdash P' : (L', Pr', U') \quad (L, Pr, U) \bowtie (L', Pr', U')}{\Delta \vdash [P']_v : (L, Pr, U)} \quad (TMem)$$

$$\frac{\Delta \vdash P : (L, Pr, U) \quad \Delta \vdash P' : (L', Pr', U') \quad (L, Pr, U) \bowtie (L', Pr', U')}{\Delta \vdash P\ P' : (L \sqcap L', Pr + Pr', U \sqcup U')} \quad (TPar)$$

The above notion guarantees that we obtain a correct term filling a context with a term whose type fits the context: note that there may be more than one type (L, Pr, U) such that (L, Pr, U) fits the context C.

We can classify reduction rules according to the types that can be derived for the right hand sides of the rules, since they influence the type of the obtained term.

Definition 8 (Δ-(L, Pr, U) safe rules)**.** *A rewriting rule $P_1 \to P_2$ is Δ safe if for some well-formed type (L, Pr, U) it can be shown that $\Delta \vdash P_2 : (L, Pr, U)$.*

To ensure correctness, each application of a rewriting rule must verify that the type of the right hand side of the rule fits the context. Using Definitions 7 and 8, if it is applied a rule whose right hand side has type (L, Pr, U) and this type fits the context, a correct term is obtained.

Typed Semantics. Given a finite set R of rewriting rules, the typed semantics of membrane systems with integral proteins is given by the least relation $\Rightarrow$ closed with respect to $\equiv$ and satisfying the following rule pattern:

$$\frac{\begin{array}{l} P_1 \to P_2 \in R \text{ is a}\Delta\text{-}(\ L, Pr, U) \text{ safe rule},\ P_1\sigma \neq \epsilon \\ \sigma \in \Sigma_\Delta \qquad C \in \mathcal{C} \qquad \text{and} \qquad (L, Pr, U) \text{ fits } C \end{array}}{C[P_1\sigma] \Rightarrow C[P_2\sigma]} \qquad \text{(RPat)}$$

4 Subject Reduction and Other Results

The type system of Table 1 satisfies weakening and other properties.

Proposition 1 (Weakening)**.** *If $\Delta \vdash P : (L, Pr, U)$ and $\Delta \subseteq \Delta'$, then $\Delta' \vdash P : (L, Pr, U)$.*

Proposition 2. *If $\Delta \vdash C[P] : (L, Pr, U)$ then*

1. *$\Delta \vdash P : (L', Pr', U')$ for some (L', Pr', U');*
2. *$\Delta, X : (L', Pr', U') \vdash C[X] : (L, Pr, U)$;*
3. *if P' is such that $\Delta \vdash P' : (L', Pr', U')$ (namely, P' has the same type as P), then $\Delta \vdash C[P'] : (L, Pr, U)$.*

The link between substitutions and well-formed bases guarantees type preservation, as expressed in the following proposition.

Proposition 3. *If $\sigma \in \Sigma_\Delta$, then* $\quad \emptyset \vdash P\sigma : (L, Pr, U)$ *iff* $\Delta \vdash P : (L, Pr, U)$.

Starting from a correct term, all the terms obtained via Δ-(L, Pr, U) safe rules are correct, and thus we can avoid conditions over P_1 because they do not influence the type of the obtained term. As expected, typed reduction preserves correctness. The main result of the paper is given by the following subject reduction theorem.

Theorem 1 (Subject Reduction)**.** *If $\emptyset \vdash st : (L, Pr, U)$ and $st \Rightarrow st'$, then $\emptyset \vdash st' : (L', Pr', U')$ for a well-formed type (L', Pr', U').*

Proof. The given typed semantics implies $st = C[P_1\sigma]$ and $st' = C[P_2\sigma]$, while Definition 8 implies$\Delta \vdash P_2 : (L, Pr, U)$. From Proposition 3 and $\sigma \in \Sigma_\Delta$ it follows that $\emptyset \vdash P_2\sigma : (L, Pr, U)$. Since (L, Pr, U) fits C (according to the given typed semantics), it means that $X : (L, Pr, U) \vdash C[X] : (L', Pr', U')$ for some well-formed type (L', Pr', U'). According to Proposition 3, it follows that $\emptyset \vdash C[P_2\sigma] : (L', Pr', U')$ for some well-formed type (L', Pr', U'). $\square$

Example 1. *Let us assume the consistent system formed from a set of basic types $T = \{t_{Na}, t_K, t_{ATP}, t_{ADP}, t_P, t_E\}$, a classification $\Gamma = \{Na : t_{Na};\ K : t_K;\ ATP : t_{ATP};\ ADP : t_{ADP};\ P, P_i : t_P;\ E_1, E_2, E_1^P, E_2^P : t_E\}$, the functions min and max given by*

$$min(t_1, t_2) = \begin{cases} 0.6 & \text{if } t_1 = t_{Na} \text{ and } t_2 = t_K \\ 0.25 & \text{if } t_1 = t_K \text{ and } t_2 = t_{Na} \\ \diamond & \text{otherwise} \end{cases}$$

$$max(t_1, t_2) = \begin{cases} 4 & \text{if } t_1 = t_{Na} \text{ and } t_2 = t_K \\ 5/3 & \text{if } t_1 = t_K \text{ and } t_2 = t_{Na} \\ \diamond & \text{otherwise} \end{cases}.$$

and the set of rules

$$\begin{array}{lll}
r_1 & : & [\ Na^3\ X\ ]_{E_1\ x} \rightarrow [\ X\ ]_{E_1\ Na^3\ x} \\
r_2 & : & [\ ATP\ X\ ]_{E_1\ Na^3\ x} \rightarrow [ADP\ X\ ]_{E_1^P\ Na^3\ x} \\
r_3 & : & [\ X\ ]_{E_1^P\ Na^3\ x} \rightarrow [\ X\ ]_{E_2^P\ x}\ Na^3 \\
r_4 & : & [\ X\ ]_{E_2^P\ x}\ K^2 \rightarrow [\ X\ ]_{E_2^P\ K^2\ x} \\
r_5 & : & [\ X\ ]_{E_2^P\ K^2\ x} \rightarrow [\ P_i\ X\ ]_{E_1\ K^2\ x} \\
r_6 & : & [\ X\ ]_{E_1\ K^2\ x} \rightarrow [\ K^2\ X\ ]_{E_1\ x}
\end{array}$$

Using all of the above rules once, the well-formed term

$$[ATP^3\ Na^8\ K^2]_{E_1} Na^9\ K^5$$

is rewritten in a well-formed term

$$[ATP^3\ Na^8\ K^2]_{E_1} Na^9\ K^5 \Rightarrow^* [ATP^2\ ADP\ P_i\ Na^5\ K^4]_{E_1} Na^{12}\ K^3.$$

It is worth to note that another rule cannot be applied because we would obtain terms that are not well-formed. For instance, consider rule $r_1 : [\ Na^3\ X\ ]_{E_1\ x} \rightarrow [\ X\ ]_{E_1\ Na^3\ x}$, the context $C = \square\ Na^{12}\ K^3$ and the instantiations $\sigma(X) = ATP^2\ ADP\ P_i\ Na^2\ K^4$ and $\sigma(x) = \epsilon$.

By applying rule r_1 we obtain a term that is **not** *well-formed:*

$$[ATP^2\ ADP\ P_i\ Na^5\ K^4]_{E_1} Na^{12}\ K^3 \Rightarrow [ATP\ ADP^2\ P_i\ Na^2\ K^4]_{E^1\ Na^3} Na^{12}\ K^3$$

This is because the type $(L_{E_1} \sqcap L_{Na}, t_{Na}^3 t_{E_1}, U_{E_1} \sqcup U_{Na})$ of P_2 does not fit the context C. It does not fit since the term $C[P_2]$ has the type $(L_{E_1} \sqcap L_{Na} \sqcap LK, t_{Na}^{15} t_{E_1} t_K^3, U_{E_1} \sqcup U_{Na} \sqcup U_K)$ that is not well-formed because the ratio between t_{Na} and t_K at the top level is $\dfrac{12+3}{3} = 5$ which is greater than 4.

5 Conclusion

In [8], the Na–K pump was described by using the π-calculus. In [7], the transfer mechanisms were described step by step, and software tools of verification were also applied. This means that it would be possible to verify properties of the described systems by using a computer program, and the use of the verification software as a substitute for expensive lab experiments. A similar development for membrane systems would be a useful achievement.

Recent years have seen a rapid increase in research on behavioural types, driven partly by the need to formalise and codify communication structures. Behavioural type systems go beyond data type systems in order to specify, characterise and reason about dynamic aspects of execution. Behavioural types encompass session types [10], multiparty session types [11] and other functional types for communications in distributed systems. Here we put forward a new view inspired by cellular biology by introducing behavioural types that control the behaviour of systems, namely quantitative types based on ratio thresholds. The inspiration comes from sodium/potassium pump, which extrudes sodium ions in exchange for potassium ions. These exchanges take place only if the ratios of these elements are between certain lower and upper bounds. To properly cope with such constraints, we introduce a ratio-based type system over multisets. We associate to each system a set of constraints, and relate them to the ratios between elements. If the constraints are satisfied, we prove that if a system is well-typed and an evolution rule is applied, then the obtained system is also well-typed.

The proposed typed semantics completely excludes the fact that sometimes biological constraints can be broken leading to a disease or even to the death of the biological system. However, the typed semantics can be modified in order to allow transitions that lead to terms that are not typable. In this case the type system should signal that some undesired event has been reached. In this way, it can be checked if a term breaks some biological property, or if the system has some unwanted behaviour.

We previously modelled the sodium/potassium pump using untyped membrane systems [3] and typed symport/antiport membrane systems [1]. The novelty of [1] is that it introduces types for the symport/antiport membrane systems, defining typing rules which control the passage of objects through the membranes. We proved that if a system is well-typed and an evolution rule is applied, then the obtained system is also well-typed.

A formalism that is somewhat related to the membrane systems considered in this paper is the calculus of looping sequences (CLS), a formalism based on term rewriting. An essential difference is that membrane systems with integral proteins use multisets to describe objects within or on membranes, while CLS terms use (looping) sequences. There are various type systems for CLS [2, 4, 5]. Our work is related to [4], where a type systems defined for the calculus of looping sequences is based on the number of elements (and not on the ratios between elements). The quantitative type system for CLS preserves some biological properties depending on the minimum and the maximum number of elements of some type; the author uses the number of elements in a system, a fact that is not relevant in biology, where concentration and ratios are typical.

Acknowledgements. Many thanks to the reviewers for their useful comments. The work was supported by a grant of the Romanian National Authority for Scientific Research, project number PN-II-ID-PCE-2011-3-0919.

References

[1] Aman, B. and Ciobanu, G. (2010) Typed Membrane Systems. *Lecture Notes in Computer Science*, **5957**, 169–181.

[2] Aman, B., Dezani-Ciancaglini, M. and Troina, A. (2009) Type Disciplines for Analysing Biologically Relevant Properties. *Electronic Notes in Theoretical Computer Science*, **227**, 97–111.

[3] Besozzi, D. and Ciobanu, G. (2005) A P System Description of the Sodium-Potassium Pump. *Lecture Notes in Computer Science*, **3365**, 210–223.

[4] Bioglio, L. (2011) Enumerated Type Semantics for the Calculus of Looping Sequences. *RAIRO - Theoretical Informatics and Applications*, **45**, 35–58.

[5] Bioglio, L., Dezani-Ciancaglini, M., Giannini, P. and Troina, A. (2012) Typed stochastic semantics for the calculus of looping sequences. *Theoretical Computer Science*, **431**, 165–180.

[6] Cavaliere, M. and Sedwards, S. (2006) Modelling Cellular Processes Using Membrane Systems With Peripheral and Integral Proteins. *Lecture Notes in Bio-Informatics*, **4210**, 108–126.

[7] Ciobanu, G. (2004) Software Verification of the Biomolecular Systems. In *Modelling in Molecular Biology*, Natural Computing Series, Springer, 40–59.

[8] Ciobanu, G., Ciubotariu, V. and Tanasă, B. (2002). A Pi-Calculus Model of the Na/K Pump. *Genome Informatics*, Universal Academy Press, 469–472.

[9] Guyton, A. and Hall, J. (2010) *Textbook of Medical Physiology.* 12th edition, Elsevier.

[10] Honda, K., Vasconcelos V.T. and M. Kubo M. (1998) Language Primitives and Type Disciplines for Structured Communication-Based Programming. *Lecture Notes in Computer Science*, **1381**, 122–138.

[11] Honda, K., Yoshida N. and Carbone M. (2008) Multiparty Asynchronous Session Types. *Proceedings POPL*, ACM Press, 273–284.

[12] Kennedy, B.G., Lunn, G. and Hoffman, J.F. (1986) Effects of Altering the ATP/ADP Ratio on Pump-Mediated Na/K and Na/Na Exchanges in Resealed Human Red Blood Cell Ghosts. *The Journal of General Physiology*, **87**, 47–72.

[13] Lingrel, J.B. and Kuntzweiler, T. (1994) Na^+,K^+-ATPase, *Journal of Biological Chemistry*, **269**, 19659–19662.

[14] Zs.-Nagy, I., Lustyik, G., Zs.-Nagy, V., Zarandi. B., and Bertoni-Freddari, C. (1981) Intracellular N+/K+ Ratios in Human Cancer Cells as Revealed by Energy Dispersive X-ray Microanalysis. *The Journal of Cell Biology*, **90**, 769–777.

[15] Păun, Gh., Rozenberg, G. and Salomaa, A. (Eds.) (2010) *Handbook of Membrane Computing.* Oxford University Press.

[16] Salomaa, A. (1973) *Formal Languages.* Academic Press.

Compliance and testing preorders differ[*]

Giovanni Bernardi and Matthew Hennessy

School of Computer Science, University of Dublin, Trinity College, Ireland

Abstract

Contracts play an essential role in the Service Oriented Computing, for which they need to be equiped with a *sub-contract relation*. We compare two possible formulations, one based on *compliance* and the other on the testing theory of De Nicola and Hennessy. We show that if the language of contracts is sufficiently expressive then the resulting *sub-contract relations* are incomparable.

However if we put natural restrictions on the contract language then the *sub-contract relations* coincide, at least when applied to servers. But when formulated for clients they remain incomparable, for many reasonable contract languages. Finally we give one example of a contract language for which the client-based *sub-contract relations* coincide.

1 Introduction

Contracts play a central role in the orchestration and development of web services, [CCLP06, LP07]. Existing services are advertised for use by third parties, which may combine these existing services to construct, and in turn advertise for further use, new services. The behavioural specification of advertised services is given via *contracts*, high-level descriptions of expected behaviour, which should come equipped with a *sub-contract* relation. Intuitively $\textsc{ct}_1 \sqsubseteq_{\textsc{crt}} \textsc{ct}_2$ means that a third party requiring a service to provide contract $\textsc{ct}_1$ may use one which already provides $\textsc{ct}_2$, so in this sense $\textsc{ct}_2$ is better than $\textsc{ct}_1$. The purpose of this short technical note is to compare and contrast two different approaches to defining this *sub-contract* relation.

The first method, [LP07, CGP09, Pad10], is based on a notion of *compliance* between two contracts, where one contract notionally formalises the behaviour offered by a server p, and the other one the behaviour offered by a client r. Contracts are interpreted as abstract processes, written in process algebras similar to CCS or CSP, [Mil89, Hoa85]. However, as pointed out by [Bd10, BH12] they can also be viewed as session types [THK94, GH05]. Intuitively p and r are in compliance, written $r \dashv p$, if when viewed as abstract processes they can continuously interact, and if this interaction ever stops then the client is in a *happy state*; the formal definition is co-inductive and is given in Definition 2.4. This leads to a natural comparison between server-oriented contracts : $p_1 \sqsubseteq^{\mathsf{cpl}}_{\textsc{svr}} p_2$ if every client which complies with p_1 also complies with p_2. As suggested in [Bd10], client-oriented contracts can also be compared, but in terms of the servers with which they comply, $r_1 \sqsubseteq^{\mathsf{cpl}}_{\textsc{clt}} r_2$.

It has been pointed out by various authors [LP07, CGP09, Pad09, Pad10] that the server contract preorder, $\sqsubseteq^{\mathsf{cpl}}_{\textsc{svr}}$, bears a striking resemblance to the well-known *must-testing* preorders from [DH84]. For example, the axioms for the strong sub-contract relation in [Pad09, Table 1], are essentially the same for the testing preorder in [Hen85, Figure 3.6]; and the behavioural characterisations of the sub-contract relation use *ready sets*, which were already in the behavioural characterisation of the *must-testing* preorder [DH84]. In this approach clients are viewed as *tests* for servers and servers are compared by their ability to guarantee that tests are satisfied. This is formalised as an *inductive* relation between tests and servers. Intuitively p $\textsc{must}$ r if

[*]Research supported by SFI project SFI 06 IN.1 1898.

whenever the two abstract processes p, r are executed in parallel the test r is guaranteed to reach a *happy state*. This in turn leads to a second pair of *sub-contract* relations, which we denote by $p_1 \sqsubseteq^{\mathsf{tst}}_{\text{SVR}} p_2$ and $r_1 \sqsubseteq^{\mathsf{tst}}_{\text{CLT}} r_2$ respectively.
In this paper we contrast these two different approaches to the notion of *sub-contract* by comparing the relations $\sqsubseteq^{\mathsf{cpl}}_{\star}$ and $\sqsubseteq^{\mathsf{tst}}_{\star}$, for both servers and clients. This study is of interest because the testing-based preorders have been thoroughly studied. In particular $\sqsubseteq^{\mathsf{tst}}_{\text{SVR}}$ has a behavioural characterisation, an axiomatisation (for finite terms) [DH84, Hen85], a logical characterisation [CH10], and an algorithm to decide it (on finite state LTSs) [CH93]; moreover the client preorder $\sqsubseteq^{\mathsf{tst}}_{\text{CLT}}$ has recently been investigated in [Ber13].

The outcome of the comparison depends on the expressive power of the language used to express contracts. We examine three different possibilities. The first is when there is no restriction on the contract language. We essentially allow any description of behaviour from the process calculus CCS; this includes infinite state and potentially divergent contracts. In this case the preorders are incomparable; see Section 3.1

In the second case we restrict the contract language to what we call $\mathsf{CCS}_{\mathsf{web}}$; this only allows finite-state contracts, which can never give rise to divergent behaviour; this language includes all the contract languages used in the standard literature, such as [LP07, Bd10, Pad09] and the concrete one of [CGP09]. In this setting the two server-contract preorders coincide:

$$p_1 \sqsubseteq^{\mathsf{cpl}}_{\text{SVR}} p_2 \text{ if and only if } p_1 \sqsubseteq^{\mathsf{tst}}_{\text{SVR}} p_2$$

However the client-contract preorders remain incomparable. This is discussed in Section 3.2.

It turns out that the difference in the formulation of the *compliance* relation between contracts and that based on *must-testing*, one co-inductive and the other inductive, has significant implications on the client-preorders, regardless of the expressivity of the contract language. This is explained via examples in Section 3.3. In particular it is difficult to think of a reasonable contract language in which they coincide. We provide one example, also in Section 3.3, which essentially coincides with the *finite session behaviours* of [Bd10]; one can think of these as *first-order* session types [THK94]. But, as we will see, introducing recursion into this contract language will once more enable us to differentiate between the two client-preorders.

The remainder of the paper is structured as follows. In the next section, Section 2, we provide formal definitions for the concepts introduced informally above, together with a description of the abstract language CCS, which is used as a general description language for contracts. Then the three different scenarios are discussed in turn in Section 3. Finally we discuss the related literature in Section 4.

2 LTS and behavioural preorders

A labelled transition system, LTS, consists of a triple $\langle P, \longrightarrow, \mathsf{Act}_{\tau\checkmark} \rangle$ where P is a set of processes and $\longrightarrow \subseteq P \times \mathsf{Act}_{\tau\checkmark} \times P$ is a transition relation between processes decorated with labels drawn from the set $\mathsf{Act}_{\tau\checkmark}$. We use the infix notation $p \xrightarrow{\mu} q$ in place of $(p, \mu, q) \in \longrightarrow$.
Let CCS be the set of terms defined by the grammar

$$p, q, r \quad ::= \quad \mathsf{1} \mid A \mid \mu.p \mid \textstyle\sum_{i \in I} p_i$$

where $\mu \in \mathsf{Act}_\tau$, I is a countable index sets, and $A, B, C, \ldots$ range over a set of definitional constants each of which has an associated definition $A \overset{\mathsf{def}}{=} p_A$. We use $\mathsf{0}$ to denote the empty external sum $\sum_{i \in \emptyset} p_i$ and $p_1 + p_2$ for the binary sum $\sum_{i \in \{1,2\}} p_i$. Note that we have omitted

2

$$\frac{}{\mathsf{1} \xrightarrow{\checkmark} \mathsf{0}}\ [\textsc{a-Ok}] \qquad \frac{}{\mu.p \xrightarrow{\mu} p}\ [\textsc{a-Pre}]$$

$$\frac{p \xrightarrow{\lambda} p'}{p + q \xrightarrow{\lambda} p'}\ [\textsc{r-Ext-l}] \qquad \frac{q \xrightarrow{\lambda} q'}{p + q \xrightarrow{\lambda} q'}\ [\textsc{r-Ext-r}]$$

$$\frac{p \xrightarrow{\lambda} p'}{A \xrightarrow{\lambda} p'}\ A \overset{\mathtt{def}}{=} p;\ [\textsc{r-Const}]$$

Figure 1: The operational semantics of CCS

$$\frac{q \xrightarrow{\lambda} q'}{q \mid\mid p \xrightarrow{\lambda} q' \mid\mid p}\ [\textsc{p-Left}] \qquad \frac{p \xrightarrow{\lambda} p'}{q \mid\mid p \xrightarrow{\lambda} q \mid\mid p'}\ [\textsc{p-Right}]$$

$$\frac{q \xrightarrow{\alpha} q' \quad p \xrightarrow{\overline{\alpha}} p'}{q \mid\mid p \xrightarrow{\tau} q' \mid\mid p'}\ [\textsc{p-Synch}]$$

Figure 2: The operational semantics of contract composition

the parallel operator $||$, as contracts, and their associated session types [Bd10, BH12], are normally expressed purely in terms of prefixing and choices.

The operational semantics of the language is given by the LTS generated by the relations $p \xrightarrow{\mu} q$ determined by the rules given in Figure 1. The *happy* or successful states mentioned in the Introduction are considered to be those CCS terms satisfying $p \xrightarrow{\checkmark}$.

We use standard notation for operations in LTSs. For example $\mathsf{Act}^{\star}_{\tau\checkmark}$, ranged over by t, denotes the set of *finite* sequences of actions from the set $\mathsf{Act}_{\tau\checkmark}$, and for any $t \in \mathsf{Act}^{\star}_{\tau\checkmark}$ we let $p \xrightarrow{t} q$ be the obvious generalisation of the single transition relations to sequences. For an infinite sequence $u \in \mathsf{Act}^{\infty}_{\tau\checkmark}$ of the form $\mu_0\mu_1\ldots$ we write $p \xrightarrow{u}$ to mean that there is an infinite sequence of actions $p \xrightarrow{\mu_0} p_o \xrightarrow{\mu_1} p_1 \ldots$. These action relations are lifted to the weak case in the standard manner, giving rise to $p \overset{s}{\Longrightarrow} q$ for $s \in \mathsf{Act}^{\star}_{\checkmark}$ and $p \overset{u}{\Longrightarrow}$ for $u \in Act^{\infty}$. Finally a process *diverges*, written $p \Uparrow$, if there is an infinite sequence of actions $p \xrightarrow{\tau} p_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} p_k \xrightarrow{\tau} \ldots$. Otherwise it is said to *converge*, written $p \Downarrow$.

To model the interactions that take place between the server and the client contracts, we introduce a binary composition of contracts, $r \mid\mid p$, whose operational semantics is in Figure (2).

Definition 2.1. [Compliance] Let $\mathcal{F}^{\dashv} : \mathcal{P}(\mathsf{CCS}^2) \longrightarrow \mathcal{P}\ (\mathsf{CCS}^2)$ be the rule functional defined so that $(r, p) \in \mathcal{F}^{\dashv}(\mathcal{R})$ whenever the following conditions are true:

(a) if $p \Uparrow$ then $r \xrightarrow{\checkmark}$

(b) if $r \mid\mid p \not\xrightarrow{\tau}$ then $r \xrightarrow{\checkmark}$

(c) if $r \mid\mid p \xrightarrow{\tau} r' \mid\mid p'$ then $r' \,\mathcal{R}\, p'$

If $X \subseteq \mathcal{F}^{\dashv}(X)$, then we say that X is a *co-inductive* compliance *relation.* The monotonicity of $\mathcal{F}^{\dashv}$ and the Knaster-Tarski theorem ensure that there exists the greatest solution of the equation $X = \mathcal{F}^{\dashv}(X)$; we call this solution the *compliance relation*, and we denote it $\dashv$. That is $\dashv \, = \, \nu X.\mathcal{F}^{\dashv}(X)$. If $r \dashv p$ we say that the client r *complies with* the server p. □

Thanks to its co-inductive nature, the compliance admits everlasting computations, even if the client side never reaches a happy state. This is a typical feature of the compliance relation.

Example 2.2. Let $C \overset{\text{def}}{=} \tau.\alpha.C$ and $S \overset{\text{def}}{=} \overline{\alpha}.S$. Even if C can not reach a happy state, it complies with S, for $\{ (C, S), (\alpha.C, S) \}$ is a co-inductive compliance. This set enjoys the properties required by Definition 2.1: Point (a) is trivially true for S converges, point (b) is true because $C \mid\mid S \xrightarrow{\tau}$ and $\alpha.C \mid\mid S \xrightarrow{\tau}$. A routine check shows that also point (c) is true. □

Another property of $\dashv$ is that it is preserved by the interactions of contracts.

Lemma 2.3. If $r \dashv p$ and $r \mid\mid p \xrightarrow{\tau}^{*} r' \mid\mid p'$ then $r' \dashv p'$.

Proof. It follows from induction on the number of reduction steps in $\xrightarrow{\tau}^{*}$, and point (c) of Definition 2.1. □

Definition 2.4. [Compliance preorders] In an arbitrary LTS we write

(1) $p_1 \sqsubseteq^{\mathsf{cpl}}_{\text{SVR}} p_2$ if for every r, $r \dashv p_1$ implies $r \dashv p_2$

(2) $r_1 \sqsubseteq^{\mathsf{cpl}}_{\text{CLT}} r_2$ if for every p, $r_1 \dashv p$ implies $r_2 \dashv p$ □

Note that our compliance relation is slightly different than that of [LP07]; we require that a client that complies with a divergent server report success immediately, whereas in [LP07] the client may report success in the future, and cannot engage in any interaction. This does not affect the resulting *sub-contract* relations on the language of contracts discussed in [CGP09, Pad10].

We also briefly recall the notion of *must-testing* from [DH84] A *computation* consists of series of τ actions of the form

$$r \mid\mid p = r_0 \mid\mid p_0 \xrightarrow{\tau} r_1 \mid\mid p_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} r_k \mid\mid p_k \xrightarrow{\tau} \ldots \tag{1}$$

It is *maximal* if it is infinite, or whenever $r_n \mid\mid p_n$ is the last state then $r_n \mid\mid p_n \overset{\tau}{\not\longrightarrow}$. A computation may be viewed as two processes p, r, one a server and the other a client, co-operating to achieve individual goals. We say that (1) is *client-successful* if there exists some $k \geq 0$ such that $r_k \xrightarrow{\checkmark}$.

Definition 2.5. [Testing preorders] In an arbitrary LTS we write p MUST r if every maximal computation of $r \mid\mid p$ is *client-successful.* Then

(1) $p_1 \sqsubsim^{\mathsf{tst}}_{\text{SVR}} p_2$ if for every r, p_1 MUST r implies p_2 MUST r

(2) $r_1 \sqsubsim^{\mathsf{tst}}_{\text{CLT}} r_2$ if for every p, p MUST r_1 implies p MUST r_2 □

Before comparing the testing and the compliance preorders, we highlight the differences between $\dashv$ and MUST. We use standard examples [LP07, Ber13]. The discussion on the preorders will mirror the differences shown in these examples.

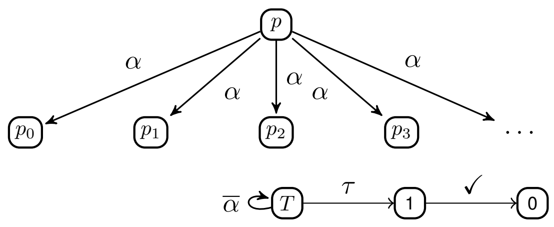


Figure 3: While $p \sqsubseteq^{\mathsf{cpl}}_{\text{SVR}} q$, the test T witnesses that $p \not\sqsubseteq^{\mathsf{tst}}_{\text{SVR}} q$ (see Example 3.1)

Example 2.6. [Meaning of livelocks] In this example we prove that $r \dashv p$ does not imply p MUST r. Recall the contracts C and S from Example 2.2. In that example we have seen that since $\{(C, S), (\alpha.C, S)\}$ is a co-inductive compliance, $C \dashv S$.

The fact that S ~~M~~UST C is true because C does not perform $\checkmark$, and so no computation of $C \mid\mid S$ is client-successful. □

The previous example shows that while the compliance admits livelocks where clients do not report success, the must testing does not. The testing relation requires clients to reach a successful state in every (maximal) computation.

Example 2.7. [Meaning $\checkmark$] In this example we prove that p MUST r does not imply $r \dashv p$. Let $r = \mathsf{1} + \tau.\mathsf{0}$. For every p, p MUST r because $r \xrightarrow{\checkmark}$, so all the computations of $r \mid\mid p$ are client-successful. For every p, the proof that $r \not\dashv p$ relies on the following computation,

$$r \mid\mid p \xrightarrow{\tau} \mathsf{0} \mid\mid p \xrightarrow{\tau} \dots$$

Since $\mathsf{0} \not\dashv p$, Lemma 2.3 implies that $r \not\dashv p$. □

In the must testing, the behaviour of a client that has reported success is completely disregarded; that is p MUST r and $r \mid\mid p \xrightarrow{\tau} r' \mid\mid p'$ does not imply p' MUST r'. For the compliance it is the contrary, as we have seen in Lemma 2.3.

3 Examples

We have three sub-sections, each examining one of the scenarios for contracts alluded to in the Introduction.

3.1 General contracts

Here we assume that contracts may be any term in the language CCS defined above. First we show that the server-contract preorders are incomparable.

Example 3.1. [Infinite traces and servers] Here we prove that $p \sqsubseteq^{\mathsf{cpl}}_{\text{SVR}} q$ but $p \not\sqsubseteq^{\mathsf{tst}}_{\text{SVR}} q$ where these terms are depicted in Figure 3.

The symbol p_k denotes a process which performs a sequence of k α actions and then becomes $\mathsf{0}$; so the process p performs every finite sequence of αs. In contrast, the process q performs also an infinite sequence of αs.

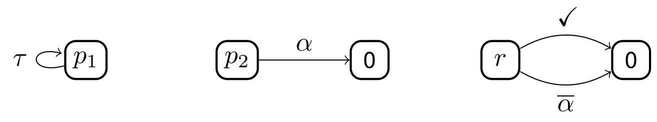


Figure 4: While $p_1 \sqsubseteq^{\mathsf{tst}}_{\mathrm{SVR}} p_2$, the client r lets us prove that $p_1 \not\sqsubseteq^{\mathsf{cpl}}_{\mathrm{SVR}} p_2$ (see Example 3.2)

To prove that $p \sqsubseteq^{\mathsf{cpl}}_{\mathrm{SVR}} q$, we have to show that $r \dashv p$ then $r \dashv q$. It suffices to prove that the following relation is a co-inductive compliance,

$$\mathcal{R} = \{\, (r',\, q) \mid r \dashv p,\, r \overset{\overline{\alpha}^k}{\Longrightarrow} r',\ \text{for some } k \in \mathbb{N} \text{ and } r \in \mathsf{CCS}\,\}$$

We have to show that if $r' \;\mathcal{R}\; q$ then the pair $(r',\, q)$ satisfies the conditions given in Definition 2.1. Pick a pair $(r',\, q)$ in the relation $\mathcal{R}$. By construction of $\mathcal{R}$ and of q, we know that $r \overset{\overline{\alpha}^k}{\Longrightarrow} r'$ for some $k \in \mathbb{N}$ and some r such that $r \dashv p$.

Condition (a) is trivially true, because q converges. We discuss condition (b) and (c). Suppose that $r' \mid\mid q \overset{\tau}{\not\longrightarrow}$; this implies that $r' \overset{\tau}{\not\longrightarrow}$. By construction $p \overset{\alpha^k}{\Longrightarrow} \mathsf{0}$, so we infer $r \mid\mid p \Longrightarrow r' \mid\mid \mathsf{0} \overset{\tau}{\not\longrightarrow}$. Now $r \dashv p$ and Lemma 2.3 imply that $r' \dashv \mathsf{0}$; Definition 2.1 ensures that $r' \overset{\checkmark}{\longrightarrow}$.

Suppose that $r' \mid\mid q \overset{\tau}{\longrightarrow} r'' \mid\mid q'$; we prove that $r'' \;\mathcal{R}\; q'$. The argument is a case analysis on the rule used to infer the reduction. In every case $q = q'$. If rule [p-Left] was applied then $r' \overset{\tau}{\longrightarrow} r''$; as $r \overset{\overline{\alpha}^k}{\Longrightarrow} r''$ the definition of $\mathcal{R}$ implies that $r'' \;\mathcal{R}\; q'$. Rule [p-Right] cannot have been applied, for $q \overset{\tau}{\not\longrightarrow}$. If rule [p-Synch] was applied, then the reduction is due to an interaction. As q engages only in α, it follows $r \overset{\overline{\alpha}^{k+1}}{\Longrightarrow} r''$. The definition of $\mathcal{R}$ implies that $r'' \;\mathcal{R}\; q'$.

We have proven that the relation $\mathcal{R}$ is a co-inductive compliance, so $p \sqsubseteq^{\mathsf{cpl}}_{\mathrm{SVR}} q$.

Now we prove that $p \not\sqsubseteq^{\mathsf{tst}}_{\mathrm{SVR}} q$; we define a test that is passed by p and not by q. Let $T \overset{\mathsf{def}}{=} \tau.\mathsf{1} + \overline{\alpha}.T$. The LTS of T is depicted in Figure (3). Every computation of $T \mid\mid p$ is finite and successful, so p MUST T. However when q is run as a server interacting with T, there is the possibility of an indefinite synchronisation on α, which is not a successful computation; q ~~M~~UST T. □

Example 3.2. [Convergence of servers] In this example we prove that $p_1 \sqsubseteq^{\mathsf{tst}}_{\mathrm{SVR}} p_2$ but $p_1 \not\sqsubseteq^{\mathsf{cpl}}_{\mathrm{SVR}} p_2$, where $p_1 = \tau^\infty$ and $p_2 = \alpha.\mathsf{0}$. The LTS of these processes is in Figure (4).

We prove that $p_1 \sqsubseteq^{\mathsf{tst}}_{\mathrm{SVR}} p_2$. First note that $p_1 \Uparrow$, so if p_1 MUST r, then $r \overset{\checkmark}{\longrightarrow}$; this is because of the infinite computation due only to the divergence of p_1. It follows that if p_1 MUST r then p_2 MUST r.

Now we define a client that lets us prove $p_1 \not\sqsubseteq^{\mathsf{cpl}}_{\mathrm{SVR}} p_2$. Let $r = \mathsf{1} + \overline{\alpha}.\mathsf{0}$. To prove that $r \dashv p_1$ Definition 2.1 requires us to show a co-inductive compliance that contains the pair $(r,\, p_1)$. The following relation $\{(r,\, p_1)\}$ will do, because the only state ever reached by $r \mid\mid p_1$ is itself. We have to prove that $r \not\dashv p_2$. Consider the computation $r \mid\mid p_2 \Longrightarrow \mathsf{0} \mid\mid \mathsf{0} \overset{\tau}{\not\longrightarrow}$. Since $\mathsf{0} \overset{\checkmark}{\not\longrightarrow}$, Definition 2.1 ensures that $\mathsf{0} \not\dashv \mathsf{0}$. An application of Lemma 2.3 leads to $r \not\dashv p_2$. □

6

Let us now consider the client preorders in this setting of general contracts. The fact that $\sqsubseteq^{\mathsf{tst}}_{\text{CLT}} \not\subseteq \sqsubseteq^{\mathsf{cpl}}_{\text{CLT}}$ will follow from Example 3.5. One final example is needed to show the converse.

Example 3.3. [Infinite traces and clients] Here we prove that $\sqsubseteq^{\mathsf{cpl}}_{\text{CLT}} \not\subseteq \sqsubseteq^{\mathsf{tst}}_{\text{CLT}}$. Let us define r as the the process p of Example 3.1, but with a $\checkmark$ transition after each finite sequence of αs. Recall also the process T of Example 3.1. To see why $r \sqsubseteq^{\mathsf{cpl}}_{\text{CLT}} T$, it is enough to check that the relation

$$\begin{aligned}\mathcal{R} = \;& \{\, (T,\, p') \mid r \dashv p,\, p \overset{\overline{\alpha}^k}{\Longrightarrow} p' \text{ for some } k \in \mathbb{N} \text{ and } p \in \mathsf{CCS} \,\} \\ & \cup \{\, (1,\, p) \mid p \in \mathsf{CCS} \,\}\end{aligned}$$

is a co-inductive compliance. To prove this, an argument similar to the one of Example 3.1 will do.

Now we show that $r \not\sqsubseteq^{\mathsf{tst}}_{\text{CLT}} T$; to see why, consider the server $S \overset{\mathsf{def}}{=} \overline{\alpha}.S$. All the maximal computations of $r \mid\mid S$ are client-successful, so S MUST r; while $T \mid\mid S$ performs an infinite computation with no client-successful states. □

The two essential differences in how servers are treated by the compliance relation and the testing relation are crystallised Example 3.1 and Example 3.2. In the former we see that a server may fail a test because of the presence of an infinite sequence of actions, although this does not impede the test, or client, from complying with the server. In the latter we see that divergent computations affect the preorders differently. The relation $\sqsubseteq^{\mathsf{tst}}_{\text{SVR}}$ is sensitive to the divergence of servers: any server that diverges is a least element of $\sqsubseteq^{\mathsf{tst}}_{\text{SVR}}$. So if $p_1 \sqsubseteq^{\mathsf{tst}}_{\text{SVR}} p_2$ and p_1 diverges, the traces that p_2 performs need not be matched by the traces of p_1. This is not the case if $p_1 \sqsubseteq^{\mathsf{cpl}}_{\text{SVR}} p_2$; the traces of p_2 have to be matched suitably by the traces of p_1, regardless of the divergence of p_1.

3.2 Contracts for web-services

There are natural constraints on the contract language which avoid the phenomena described above. We say that a process p *converges strongly* if for every $s \in Act^\star$, $p \overset{s}{\Longrightarrow} p'$ implies $p' \Downarrow$. Then let $\mathsf{CCS_{web}}$ denote the subset of processes in CCS which both strongly converge and are finite-state. Note that Konigs Lemma ensures that for every $p \in \mathsf{CCS_{web}}$, p can perform an infinite sequence of actions u whenever it can perform all finite subsequences of u. Thus neither Example 3.1 nor Example 3.2 can be formulated in $\mathsf{CCS_{web}}$. Nevertheless it is still a very expressive contract language. It encompasses (via an interpretation) first-order session types [Bd10, BH12], and, up to syntactic differences, the LTSs of contracts for web-services used in [LP07, CGP09, Pad10] are contained in the LTS $\langle\, \mathsf{CCS_{web}},\, \longrightarrow,\, \mathsf{Act}_{\tau\checkmark} \,\rangle$.

Theorem 3.4. In $\mathsf{CCS_{web}}$, $p_1 \sqsubseteq^{\mathsf{tst}}_{\text{SVR}} p_2$ if and only if $p_1 \sqsubseteq^{\mathsf{cpl}}_{\text{SVR}} p_2$.

Proof. See Proposition 5.1.21 of [Ber13]. The proof relies on the behavioural characterisation of the two preorders, which is the same relation $\precsim_{\text{SVR}}$. Roughly speaking, $p_1 \precsim_{\text{SVR}} p_2$ if and only if for every trace $s \in Act^\star$, the potential deadlocks of p_2 after s are matched the potential deadlocks[1] of p_1 after s. These properties characterise both $\sqsubseteq^{\mathsf{tst}}_{\text{SVR}}$ and $\sqsubseteq^{\mathsf{cpl}}_{\text{SVR}}$, that is $\sqsubseteq^{\mathsf{tst}}_{\text{SVR}} = \precsim_{\text{SVR}}$ and $\sqsubseteq^{\mathsf{cpl}}_{\text{SVR}} = \precsim_{\text{SVR}}$. The theorem follows from these equalities. □

However even in $\mathsf{CCS_{web}}$ the client sub-contract preorders remain different. In Example 3.5 and Example 3.6 below we prove that the client preorders are not comparable; Theorem 3.4 is false for the client preorders. and Also the converse (negative) inequality is true; we prove it in Example 3.6 below.

[1]More precisely, the acceptance sets.

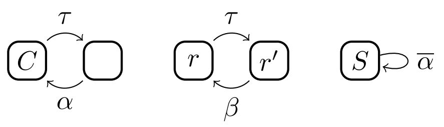


Figure 5: In any LTS that contains C and r, and where $\overline{\alpha} \neq \overline{\beta}$, S witnesses that $C \not\sqsubseteq^{\mathsf{cpl}}_{\text{CLT}} r_2$. However $C \underset{\sim}{\sqsubset}_{\text{CLT}} r$ (see Example 3.5)

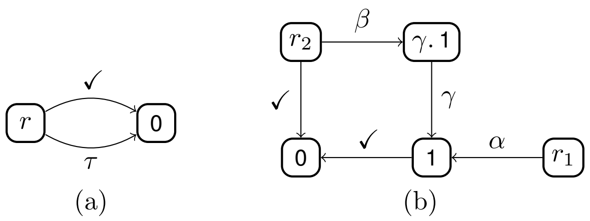


Figure 6: Clients that let us prove that the client preorders are not comparable even in the finite fragment of $\mathsf{CCS}_{\mathsf{web}}$ (see Example 3.6 and Example 3.7)

Example 3.5. [Client preorders and livelocks] In this example we prove that in $\mathsf{CCS}_{\mathsf{web}}$, $\underset{\sim}{\sqsubset}^{\mathsf{tst}}_{\text{CLT}} \not\subseteq \sqsubseteq^{\mathsf{cpl}}_{\text{CLT}}$. Suppose that for two actions α,β we have $\overline{\alpha} \neq \overline{\beta}$, recall the processes C, S of Example 2.6. Their LTS are depicted in Figure 5 along with the LTS of a process r.

We prove that $C \underset{\sim}{\sqsubset}^{\mathsf{tst}}_{\text{CLT}} r$ and that $C \not\sqsubseteq^{\mathsf{cpl}}_{\text{CLT}} r$. The inequality $C \underset{\sim}{\sqsubset}^{\mathsf{tst}}_{\text{CLT}} r$ is trivially true, because C does not perform $\checkmark$, so p $\not\text{M}$UST C for every C.

To show that $C \not\sqsubseteq^{\mathsf{cpl}}_{\text{CLT}} r$ we have to exhibit a server with which C complies, while r does not. This server is S. In Example 2.2 we have already proven that $C \dashv S$. On the other hand, since $\overline{\alpha}$ cannot interact with β, we have $r' \mid\mid S \overset{\tau}{\not\longrightarrow}$. As $r' \overset{\checkmark}{\not\longrightarrow}$, Definition 2.1 and Lemma 2.3 ensure that $r \not\dashv S$. □

3.3 Finite session behaviours

Underlying Example 3.5 is the treatment of *livelocks*. These are catastrophic for the testing based preorder, but can be accommodated by the compliance based one. However, there is another completely independent reason for which the two client preorders are different. Both are sensitive to the presence of the $\checkmark$ action, but in different ways.

In the examples below, Example 3.6 and Example 3.7, we prove that because of this difference, even for finite clients, with no recursion, the client preorders are incomparable. These examples show that any test which immediately performs $\checkmark$ is a top element in the testing based preorder, even if it subsequently evolves to a state in which $\checkmark$ is no longer possible. On the other hand for the compliance relation the action $\checkmark$ matters only in the stuck states of the client; its presence in all other states is immaterial.

Example 3.6. [$\mathsf{1}$ and internal moves] Here we prove that $\sqsubseteq^{\mathsf{cpl}}_{\text{CLT}} \not\subseteq \underset{\sim}{\sqsubset}^{\mathsf{tst}}_{\text{CLT}}$ even for finite clients (without recursion). Recall the client r of Example 2.7; its LTS is depicted in column (a) of

8

Figure 6.

On the one hand, $r \sqsubseteq^{\mathsf{cpl}}_{\mathrm{CLT}} \mathsf{0}$. This is true because for every p, $r \mid\mid p \xrightarrow{\tau} \mathsf{0} \mid\mid p$, thus $r \dashv p$ and Lemma 2.3 imply that $\mathsf{0} \dashv p$. On the other hand $r \not\underset{\sim}{\sqsubset}^{\mathsf{tst}}_{\mathrm{CLT}} \mathsf{0}$, because $\mathsf{0}$ MUST r (as $r \xrightarrow{\checkmark}$), however $\mathsf{0}$ ~~M~~UST $\mathsf{0}$. □

Example 3.7. [$\mathsf{1}$ and interactions] Here we show that $\underset{\sim}{\sqsubset}^{\mathsf{tst}}_{\mathrm{CLT}} \not\subseteq \sqsubseteq^{\mathsf{cpl}}_{\mathrm{CLT}}$. Let $r_1 = \alpha.\mathsf{1}$ and $r_2 = \mathsf{1} + \beta.\gamma.\mathsf{1}$; their LTS is in column (b) of Figure 6.

Regardless of the server p we have p MUST r_2 because $r_2 \xrightarrow{\checkmark}$. It follows trivially that $r_1 \underset{\sim}{\sqsubset}^{\mathsf{tst}}_{\mathrm{CLT}} r_2$. However, $r_1 \not\sqsubseteq^{\mathsf{cpl}}_{\mathrm{CLT}} r_2$; a typical server which distinguishes the two clients is $p = \overline{\alpha}.\mathsf{0} + \overline{\beta}.\mathsf{0}$. The proof that $r_1 \dashv p$ amounts in checking that the relation $\{ (r_1, p), (\mathsf{1}, \mathsf{0}) \}$ is a co-inductive compliance. The fact that $r_2 \not\dashv p$ is due to Lemma 2.3 and the computation $r_2 \mid\mid p \xrightarrow{\tau} \gamma.\mathsf{1} \mid\mid \mathsf{0} \stackrel{\tau}{\not\longrightarrow}$. □

A further restriction of $\mathsf{CCS_{web}}$ provides a language in which this difference in the treatment of $\checkmark$ does not materialise. Let $\mathsf{SB^f}$ be the language given by the following grammar,

$$p, q, r \quad ::= \quad \mathsf{1} \mid \textstyle\sum_{i \in I} \alpha_i.p_i \mid \sum_{i \in I} \tau.\overline{\alpha_i}.p_i$$

where $\alpha \in \mathit{Act}$, I is a finite non-empty set, and the actions α_is are pairwise distinct. This language gives rise to the LTS $\langle\ \mathsf{SB^f}, \longrightarrow, \mathsf{Act}_{\tau\checkmark}\ \rangle$ in the usual manner. The language $\mathsf{SB^f}$ is essentially the finite part of the session behaviours of [Bd10], which we can think of as the *first-order* part of the session types used in [GH05]. Here the language is finite so as to avoid duplicating Example 3.5.

Theorem 3.8. In $\mathsf{SB^f}$,

(1) $p_1 \underset{\sim}{\sqsubset}^{\mathsf{tst}}_{\mathrm{SVR}} p_2$ if and only if $p_1 \sqsubseteq^{\mathsf{cpl}}_{\mathrm{SVR}} p_2$

(2) $r_1 \underset{\sim}{\sqsubset}^{\mathsf{tst}}_{\mathrm{CLT}} r_2$ if and only if $r_1 \sqsubseteq^{\mathsf{cpl}}_{\mathrm{CLT}} r_2$

Outline. Part (1) follows from Corollary 6.4.8 of [Ber13]. Part (2) follows from Corollary 6.5.15, Proposition 6.2.12, and the fact that for every $r \in \mathsf{SB^f}$ there exists a p such that p MUST r. The proof of the latter fact is by structural induction on r. □

4 Conclusion

In this paper we have shown the differences between the sub-contract preorders [CGP09, Pad10] and the testing preorders [DH84, Ber13]. Another study of sub-contract relations is [BMPR10]. There different compliances are used; two similar to $\dashv$, and a fair one.

The sub-contract relation was first proposed in [CCLP06], and further developed in [LP07, CGP09, Pad10]. For instance, the latter papers show how to adapt the behaviour of contracts by applying filters, or orchestrators; thereby defining weak sub-contracts, whose elements can be forced (by filtering) into the sub-contract. [CGP09] also shows an encoding of WS-BPEL activities into the language of contracts. A result similar to Theorem 3.4 was already established in [LP07], and it has been referenced by [Pad09],[Pad10, Proposition 2.7], and [CGP09, pag. 13]. The sub-contract for clients was proposed first in [Bd10], and it is instrumental in modelling the subtyping for first-order session types [GH05]. The preorder that models the subtyping coincides with a combination of the client sub-contract and a server one. This model was proven sound in [Bd10] and fully-abstract in [BH12].

References

[Bd10] Franco Barbanera and Ugo de'Liguoro. Two notions of sub-behaviour for session-based client/server systems. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *PPDP*, pages 155–164. ACM, 2010.

[Ber13] Giovanni Bernardi. *Behavioural Equivalences for Web Services.* PhD thesis, Trinity College Dublin, June 2013. Preliminary version available at https://www.scss.tcd.ie/∼bernargi.

[BH12] Giovanni Bernardi and Matthew Hennessy. Modelling session types using contracts. In Sascha Ossowski and Paola Lecca, editors, *SAC*, pages 1941–1946. ACM, 2012.

[BMPR10] Michele Bugliesi, Damiano Macedonio, Luca Pino, and Sabina Rossi. Compliance preorders for web services. In Cosimo Laneve and Jianwen Su, editors, *Web Services and Formal Methods*, volume 6194 of *Lecture Notes in Computer Science*, pages 76–91. Springer Berlin / Heidelberg, 2010.

[BPZ09] Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, editors. *Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Bertinoro, Italy, June 1-6, 2009, Advanced Lectures*, volume 5569 of *Lecture Notes in Computer Science.* Springer, 2009.

[CCLP06] Samuele Carpineti, Giuseppe Castagna, Cosimo Laneve, and Luca Padovani. A formal account of contracts for web services. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2006.

[CGP09] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Trans. Program. Lang. Syst.*, 31(5):1–61, 2009. Supersedes the article in POPL '08.

[CH93] Rance Cleaveland and Matthew Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Asp. Comput.*, 5(1):1–20, 1993.

[CH10] Andrea Cerone and Matthew Hennessy. Process behaviour: Formulae vs. tests (extended abstract). In Sibylle B. Fröschle and Frank D. Valencia, editors, *EXPRESS'10*, volume 41 of *EPTCS*, pages 31–45, 2010.

[DH84] Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[GH05] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.

[Hen85] Matthew Hennessy. *Algebraic theory of processes.* MIT Press, Cambridge, MA, USA, 1985.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[LP07] Cosimo Laneve and Luca Padovani. The must preorder revisited. In *Proceedings of the 18th international conference on Concurrency Theory*, pages 212–225, Berlin, Heidelberg, 2007. Springer-Verlag.

[Mil89] Robin Milner. *Communication and concurrency.* PHI Series in computer science. Prentice Hall, 1989.

[Pad09] Luca Padovani. Contract-based discovery and adaptation of web services. In Bernardo et al. [BPZ09], pages 213–260.

[Pad10] Luca Padovani. Contract-based discovery of web services modulo simple orchestrators. *Theor. Comput. Sci.*, 411(37):3328–3347, 2010.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994.

Towards Global and Local Types for Adaptation[*]

Mario Bravetti[1], Marco Carbone[2], Thomas Hildebrandt[2], Ivan Lanese[1], Jacopo Mauro[1], Jorge A. Perez[3] and Gianluigi Zavattaro[1]

[1] University of Bologna, Lab. Focus INRIA
[2] IT University of Copenhagen
[3] CITI and DI, FCT Universidade Nova de Lisboa

Abstract

Choreographies allow designers to specify the protocols followed by participants of a distributed interaction. Adapting the behavior of a participant, either because of external requests or as a self-update to better suit a changing environment, requires to update in a coordinated way (possibly) all the participants interacting with him. We propose a language able to describe a choreography together with its adaptation strategies, and we discuss the main issues that have to be solved to enable adaptation on a participant code dealing with many interleaved protocols.

1 Introduction

Modern complex distributed software systems face the great challenge of adapting to varying contextual conditions, user requirements or execution environments. Service-oriented Computing (SOC), and service-oriented architectures in general, have been designed to support a specific form of adaptation: services can be dynamically discovered and properly combined in order to achieve an overall service composition that satisfies some specific desiderata that could be known only at service composition time. Rather sophisticated theories have been defined for checking and guaranteeing the correctness of this service assemblies (see, e.g., the rich literature on choreography/orchestration languages [2, 13], behavioral contracts [7, 6], and session types [4, 11, 5]). In this paper, we consider a more fine-grained form of adaptation that can occur when the services have been already combined, but have not yet completed their task. This form of adaptation can occur, for instance, when the desiderata dynamically change or when some unexpected external event occurs. In these cases, it could be necessary to dynamically modify the choreography and behavior of the combined services, and this modification must occur in a consistent and coordinated manner in order to avoid breaking the correctness of the overall service composition.

More precisely, we initiate the investigation of new models and theories for service composition that properly take into account this form of adaptation. First of all, we extend a previous language for the description of service choreographies [2] with two operators: one allows for the specification of *adaptable scopes* that can be dynamically modified, while the second may dynamically update code in one of such scopes. This language is designed for the description, from a global perspective, of dynamically adaptable multi-party interaction protocols. As a second step in the development of our theory, we define a service behavioral contract language for the description, from a local perspective, of the input-output communications. In order to support adaptation, also in this case we enhance a previous service contract language [2] with two new operators for adaptable scope declaration and code update, respectively. The most complex

[*]This work was partially supported by COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems (BETTY).

aspect to be taken into account is the fact that, at the local level, peers should synchronize their local adaptations in order to guarantee a consistent adaptation of their behavior.

As mentioned above, these two languages are expected to be used to describe multi-party protocols from a global and a local perspective, respectively. The relationship between the two languages is formalized in terms of a *projection* function from a global specification to a corresponding local one. The complete theory that we plan to develop will also consider a concrete language for programming services; such a language will include update mechanisms like those provided by, for instance, the Jorba service orchestration language [12]. The ultimate aim of our research is to define appropriate behavioral typing techniques able to check whether the concretely programmed services correctly implement the specified multi-party adaptable protocols. This will be achieved by considering the global specification of the protocol, by projecting such specification on the considered peer, and then by checking whether the actual service correctly implements the projected behavior. In order to clarify our objective, we discuss an example inspired by a health-care scenario [15]. Two adaptable protocols are described by using the proposed choreography languages: the first protocol describes the interaction between the *doctor* and the *laboratory* agents, while the second involves a *doctor*, a *nurse*, and a *patient*. In case of emergency, the doctor may speed up the used protocols by interrupting running tests and avoiding the possibility that the nurse refuses to use a medicine she does not trust —this possibility is normally allowed by the protocol. Then, using a π-calculus-like language, we present the actual behavior of some of the involved peers and discuss the kinds of problems that we will have to address in order to define appropriate behavioral type checking techniques.

Disclaimer. This paper presents the main ideas of an ongoing thread of research: many details are still preliminary.

2 Choreography and Endpoint Languages

2.1 Choreography Language

2.1.1 Syntax

We describe here the syntax of our choreography language.

$$\begin{array}{rlllll}
C ::= & a_{r_1 \to r_2} & (\mathit{action}) & \mid & C \,;\, C & (\mathit{sequence}) \\
\mid & C \mid C & (\mathit{parallel}) & \mid & C + C & (\mathit{choice}) \\
\mid & C^* & (\mathit{star}) & \mid & \mathbf{1} & (\mathit{one}) \\
\mid & \mathbf{0} & (\mathit{nil}) & & & \\
\mid & X : T[C] & (\mathit{scope}) & \mid & X_r\{C\} & (\mathit{update})
\end{array}$$

The basic element of a choreography C is an interaction $a_{r_1 \to r_2}$, with the intended meaning that participant r_1 sends a message to participant r_2 over channel a. Two choreographies C_1 and C_2 can be composed in sequence ($C_1 \,;\, C_2$), in parallel ($C_1 \mid C_2$), and using nondeterministic choice ($C_1 + C_2$). Also, a choreography may be iterated zero or more times using the Kleene star *. The empty choreography, which just successfully terminates, is denoted by $\mathbf{1}$. The deadlocked choreography $\mathbf{0}$ is needed for the definition of the semantics: we assume it is never used when writing a choreography.

The two last operators deal with adaptation. Adaptation is specified by defining a *scope* that delimits a choreography that at runtime may be replaced by a new choreography, coming from either inside or outside the system. Adaptations coming from outside may be decided by the user through some adaptation interface, by some manager module, or by the environment. In contrast, adaptations coming from inside represent self-updates, decided by a part of the system towards itself or towards another part of the system, usually as a result of some interaction producing unexpected values. Adaptations from outside and from inside are indeed quite similar, e.g. an update decided by a manager module may be from inside if the manager behavior is part of the choreography, from outside if it is not. Construct $X : T[C]$ defines a scope named X currently executing choreography C — the name is needed to designate it as a target for a particular adaptation. Type T is the set of roles (possibly) occurring in the scope. This is needed since a given update can be applied to a scope only if it specifies how all the involved roles are adapted. Operator $X_r\{C\}$ defines *internal updates*, i.e., updates offered by a participant of the choreography. Here r is the name of the participant offering the update, X the name of the target scope, and C the new choreography.

Not all choreography terms generated by the syntax above are actually choreographies. We call choreographies only the choreography terms satisfying the conditions below. They rely on the set of roles $roles(C)$ inside a choreography C. This includes roles of actions, i.e., r_1, r_2 in $a_{r_1 \to r_2}$, types of scopes, i.e., T in $X : T[C']$, and roles originating update prefixes, i.e., r in $X_r\{C''\}$, but not roles in C''. Roles in C'' are not considered since the update can be an external update towards another choreography.

Definition 1. *C is a choreography if:*

1. *names of scopes are unique: we call $type(X)$ the type T included in the unique occurrence $X : T[C']$ of scope X;*
2. *C is well-typed, i.e., for every scope $X : T[C']$ occurring in C we have $roles(C') \subseteq T$ and every update prefix $X_r\{C''\}$ occurring in C is such that $roles(C'') \subseteq T$.*

2.1.2 Semantics

As in the syntax, the most interesting part of the semantics concerns update constructs.

Definition 2. *The semantics of choreographies is the smallest labeled transition system closed under the rules in Table 1 where T is a set of roles and $C[C'/X]$ replaces all scopes with name $X : T$ occurring in C not inside update prefixes with $X : T[C']$.*

Rules in the first four rows of the table are standard (cf. [2]). Rule (Comm) executes an interaction, making it visible in the label. While rule (Seq) allows the first component of a sequential composition to compute, rule (SeqTick) allows it to terminate, starting the execution of the second component. Rule (Par) allows parallel components to interleave their executions. Rule (ParTick) allows parallel components to synchronize their termination. Rule (Cho) selects a branch in a nondeterministic choice. Rule (Star) unfolds the Kleene star. Note that the unfolding may break uniqueness of scopes with a given name. We will come back to this point later on. Rules (StarTick) and (One) allow termination of a Kleene star and of the empty choreography, respectively.

The other rules in the table deal with adaptation. Rule (CommUpd) makes an internal adaptation available, moving the information to the label. Adaptations propagate through sequence, parallel composition, and Kleene star using rules (SeqUpd), (ParUpd), and (StarUpd), respectively. Note that, while propagating, the update is applied to the continuation of the sequential composition, to parallel terms and to the body of Kleene star. Notably, the update is applied to both enabled and non enabled occurrences of the desired scope. Rule (ScopeUpd) allows a

$$(\textsc{One})\ \frac{}{\mathbf{1} \xrightarrow{\checkmark} \mathbf{0}} \qquad (\textsc{Comm})\ \frac{}{a_{r_1 \to r_2} \xrightarrow{a_{r_1 \to r_2}} \mathbf{1}} \qquad (\textsc{Seq})\ \frac{C_1 \xrightarrow{a_{r_1 \to r_2}} C_1'}{C_1;\ C_2 \xrightarrow{a_{r_1 \to r_2}} C_1';C_2}$$

$$(\textsc{SeqTick})\ \frac{C_1 \xrightarrow{\checkmark} C_1' \qquad C_2 \xrightarrow{\alpha} C_2'}{C_1;\ C_2 \xrightarrow{\alpha} C_2'} \qquad (\textsc{Par})\ \frac{C_1 \xrightarrow{a_{r_1 \to r_2}} C_1'}{C_1 \mid C_2 \xrightarrow{a_{r_1 \to r_2}} C_1' \mid C_2}$$

$$(\textsc{ParTick})\ \frac{C_1 \xrightarrow{\checkmark} C_1' \qquad C_2 \xrightarrow{\checkmark} C_2'}{C_1 \mid C_2 \xrightarrow{\checkmark} C_1' \mid C_2'} \qquad (\textsc{Cho})\ \frac{C_1 \xrightarrow{\alpha} C_1'}{C_1 + C_2 \xrightarrow{\alpha} C_1'}$$

$$(\textsc{Star})\ \frac{C \xrightarrow{a_{r_1 \to r_2}} C'}{C^* \xrightarrow{a_{r_1 \to r_2}} C';\ C^*} \qquad (\textsc{StarTick})\ \frac{}{C^* \xrightarrow{\checkmark} \mathbf{0}}$$

$$(\textsc{CommUpd})\ \frac{}{X_r\{C\} \xrightarrow{X_r\{C\}} \mathbf{1}} \qquad (\textsc{SeqUpd})\ \frac{C_1 \xrightarrow{X_r\{C\}} C_1'}{C_1;\ C_2 \xrightarrow{X_r\{C\}} C_1';(C_2[C/X])}$$

$$(\textsc{ParUpd})\ \frac{C_1 \xrightarrow{X_r\{C\}} C_1'}{C_1 \mid C_2 \xrightarrow{X_r\{C\}} C_1' \mid (C_2[C/X])} \qquad (\textsc{StarUpd})\ \frac{C_1 \xrightarrow{X_r\{C\}} C_1'}{C_1^* \xrightarrow{X_r\{C\}} C_1';\ (C_1[C/X])^*}$$

$$(\textsc{ScopeUpd})\ \frac{C_1 \xrightarrow{X_r\{C\}} C_1'}{X{:}T[C_1] \xrightarrow{X_r\{C\}} X{:}T[C]} \qquad (\textsc{Scope})\ \frac{C_1 \xrightarrow{\alpha} C_1' \qquad \alpha \neq X_r\{C\} \text{ for any } r,C}{X{:}T[C_1] \xrightarrow{\alpha} X{:}T[C_1']}$$

Table 1: Semantics of choreographies

scope to update itself (provided that the names coincide), while propagating the update to the rest of the choreography. Rule (Scope) allows a scope to compute.

We can now define traces. We consider both internal transitions, and open transitions corresponding to updates from a parallel choreography.

Definition 3. *Traces of a choreography C are (possibly infinite) sequences of states and transitions arising from the semantics of C. Open traces of a choreography C are traces in the set $OTr(C)$ defined as follows. Let Tr be the (union of) the set of traces of $C'|C$ for any choreography C' such that $roles(C)$ and $roles(C')$ are disjoint and $C'|C$ is a choreography. $OTr(C) = \{w(tr) \mid tr \in Tr\}$, where $w(tr)$ is obtained from tr by: (i) removing all actions concerning roles not in $roles(C)$ and all $X_r\{C''\}$ update transitions, for any C'', arising from roles r not in $roles(C)$ and such that the scope named X does not occur inside C; and (ii) relabeling $X_r\{C''\}$ update transitions, for any C'', arising from roles r not in $roles(C)$ and such that the scope named X occurs inside C, into $X\{C''\}$ (i.e. labels representing updates performed by the environment).*

As we have said, in a choreography we assume scope names to be unique. However, uniqueness is not preserved by transitions. Nevertheless a slightly weaker property is indeed preserved, and it simplifies the implementation of the adaptation mechanisms at the level of endpoints.

Proposition 1. *Let C be a choreography and let C' be a choreography term reachable from C via zero or more transitions (possibly open). For every X there exists at most an occurrence of a scope named X which is enabled (i.e., which can compute).*

2.1.3 An Example

Below we give an example of an adaptable choreography to illustrate the features introduced above. The example is based on a health-care workflow inspired by field study [15] carried out in previous work. The field study was also considered as inspiration for recent work on session types for health-care processes [10] and adaptable declarative case management processes [16], but the combination of session types and adaptability has not been treated previously.

In the considered scenario, Doctors, Nurses and Patients employ a distributed, electronic health-care record system where each actor (including the patient) uses a tablet pc/smartphone to coordinate the treatment. Below, iteration C^+ stands for $C; C^*$.

$$X : \{D, N\}\big[\big((prescribe_{D\to N})^+; (sign_{D\to N} + X_D\{sign_{D\to N}\}; up_{D\to N}); trust_{N\to D}\big)^+\big]; \\ medicine_{N\to P}$$

The doctor first records one or more prescriptions, which are sent to the nurse's tablet $(prescribe_{D\to N})^+$. When receiving a signature, $sign_{D\to N}$, the nurse informs the doctor if the prescription is trusted. If not trusted then the doctor must prescribe a new medicine. If trusted, the nurse proceeds and gives the medicine to the patient, which is recorded at the patient's smartphone, $medicine_{N\to P}$. However, instead of signing and waiting for the nurse to trust the medicine, in emergency cases the doctor may update the protocol so that the possibility of not trusting the prescription is removed: the nurse would have to give the medicine to the patient right after receiving the signature. In the example, this is done by a self-update $(X_D\{sign_{D\to N}\})$ of the running scope. In other scenarios, this could have been done by an entity not represented in the choreography, such as the hospital director, thus resulting in an external update. The doctor notifies the protocol update to the nurse using the $up_{D\to N}$ interaction.

Now consider the further complication that the doctor may run a test protocol with a laboratory, after prescribing a medicine and before signing:

$$X'\{D, L\} : [orderTest_{D\to L}\ ;\ (results_{L\to D} + \tilde{X}'_D\{\mathbf{1}\})]$$

We allow the test protocol also to be adaptable, since the doctor may decide that there is an emergency while waiting for the results, and thus also having to interrupt the test protocol. If the two protocols are performed in interleaving by the same code, then the updates of the two protocols should be coordinated. We illustrate this in § 3 below.

2.2 Endpoint Language

Since choreographies are at the very high level of abstraction, defining a description of the same system nearer to an actual implementation is of interest. In particular, for each participant in a choreography (also called endpoint) we want to describe the actions it has to take in order to follow the choreography. The syntax of endpoint processes is as follows:

$$\begin{array}{llllll}
P ::= & \overline{a}_r & (output) & \mid & a_r & (input) \\
\mid & P\ ;\ P & (sequence) & \mid & P \mid P & (parallel) \\
\mid & P + P & (choice) & \mid & P^* & (star) \\
\mid & \mathbf{1} & (one) & \mid & \mathbf{0} & (zero) \\
\mid & X[P]^F & (scope) & \mid & X_{(r_1,\ldots,r_n)}\{P_1,\ldots,P_n\} & (update)
\end{array}$$

where F is either A, denoting an active (running) scope, or ε, denoting a scope still to be started (ε is omitted in the following).

As for choreographies, endpoint processes contain some standard operators and some operators dealing with adaptation. Communication is performed by output $\overline{a}_r$, denoting an output on channel a towards participant r, and the corresponding input a_r, waiting for a message from participant r on channel a. Two endpoint processes P_1 and P_2 can be composed in sequence ($P_1 \,;\, P_2$), in parallel ($P_1 \mid P_2$), and using nondeterministic choice ($P_1 + P_2$). Endpoint processes can be iterated using a Kleene star *. The empty endpoint process is denoted by $\mathbf{1}$ and the deadlocked endpoint process (needed only for the definition of the semantics) is denoted by $\mathbf{0}$.

Adaptation is applied to scopes. $X[P]^F$ denotes a scope named X executing process P. F is a flag distinguishing scopes whose execution has already begun (A) from scopes which have not started yet (ε). The update operator $X_{(r_1,\dots,r_n)}\{P_1,\dots,P_n\}$ provides an update for scope named X, involving roles $r_1,\dots,r_n$. The new process for role r_i is P_i.

Endpoints are of the form $[\![P]\!]_r$, where r is the name of the endpoint and P its process. Systems are composed by parallel endpoints:

$$S ::= \quad [\![P]\!]_r \qquad (\textit{endpoint}) \qquad | \qquad S \| S \qquad (\textit{parallel system})$$

As for choreographies, not all systems are endpoint specifications. Given a function $\textit{type}(X)$ associating a set of roles to each scope name X, endpoint specifications are defined as follows.

Definition 4. *S is an endpoint specification if: (i) no active scopes are present, (ii) endpoint names are unique, (iii) all roles r occurring in terms of the form $\overline{a}_r$, a_r, or such that $r \in \textit{type}(X)$ for some scope X are endpoints of S (iv) a scope with name X con occur (outside updates) only in endpoints $r \in \textit{type}(X)$, (v) every update has the form $X_{\textit{type}(X)}\{P_1,\dots,P_n\}$ (vi) outputs $\overline{a}_r$ and inputs a_r included in $X_{\textit{type}(X)}\{P_1,\dots,P_n\}$ are such that $r \in \textit{type}(X)$.*

For space reasons, we do not define a formal semantics for endpoints: we just point out that it should include all the labels of the semantics of choreographies, plus some additional labels corresponding to partial activities, such as an input. We also highlight the fact that, for all scopes with the same name in a system, the scope start transition (transforming a scope from inactive to active) and the scope end transition (removing it) are synchronized: this is needed to ensure that scopes which correspond to the same choreography scope evolve together.

2.3 Projection

Since choreographies provide system descriptions at the high level of abstraction and endpoint specifications provide more low level descriptions, a main issue is to derive from a given choreography an endpoint specification executing it. This is done using the notion of *projection*.

Definition 5 (Projection)**.** *The projection of a choreography C on the role r of an endpoint specification, denoted by $C \upharpoonright_r$, is defined by the clauses below*

- $a_{r_1 \to r_2} \upharpoonright_r = \overline{a}_{r_2}$, *if* $r = r_1$;
  $a_{r_1 \to r_2} \upharpoonright_r = a_{r_1}$ *if* $r = r_2$;
  $a_{r_1 \to r_2} \upharpoonright_r = \mathbf{1}$, *otherwise.*
- $X_{r'}\{C\} \upharpoonright_r = X_{(r_1,\dots,r_n)}\{C \upharpoonright_{r_1},\dots,C \upharpoonright_{r_n}\}$, *where* $\{r_1,\dots,r_n\} = \textit{type}(X)$, *if* $r = r'$;
  $X_{r'}\{C\} \upharpoonright_r = \mathbf{1}$, *otherwise.*
- $X : T[C] \upharpoonright_r = X[C \upharpoonright_r]$ *if* $r \in \textit{type}(X)$;
  $X : T[C] \upharpoonright_r = \mathbf{1}$, *otherwise.*

and is an homomorphism on the other operators. The endpoint specification resulting from a choreography C is obtained by composing in parallel roles $[\![C \upharpoonright_r]\!]_r$, where $r \in \textit{roles}(C)$.

One can see that the term S obtained by projecting a choreography is an endpoint specification. Ideally, traces of the projected system should be included in the traces of the original choreography. Actually, this occurs only for choreographies satisfying suitable connectedness conditions. The conditions needed for our choreographies, and that we do not present here, extend the ones in [13]. This is not an actual restriction, since choreographies that do not respect the conditions can be transformed into choreographies that respect them [14].

Theorem 1. *Traces of projection of connected choreographies are included into traces of the original choreography.*

To be precise, the definition of trace inclusion in the theorem maps labels $X_r\{C\}$ of transitions of the choreography into labels $[X_{(r_1,\ldots,r_n)}\{P_1,\ldots,P_n\}]_r$ of the transitions of the endpoint specification obtained by projection, where $type(X) = \{r_1,\ldots,r_n\}$ and $P_1 = C\restriction_{r_1},\ldots,P_n = C\restriction_{r_n}$ are obtained, themselves, by projection from C. The proof exploits uniqueness of scope names.

As an example, the endpoint projection obtained from the prescribe choreography introduced in §2.1 is $[P_N]_N \| [P_D]_D \| [P_P]_P$ where

$$\begin{aligned}&P_N = X[((prescribe_D)^+ \ ; \ (sign_D + up_D \ ); \ \overline{trust}_D)^+] \ ; \ \overline{medicine}_P\\ &P_D = X[((\overline{prescribe}_N)^+ \ ; \ (\overline{sign}_N + X_{D,N}\{\overline{sign}_N, sign_D\} \ ; \ \overline{up}_N); trust_N)^+]\\ &P_P = medicine_N\end{aligned}$$

For presentation purposes, we dropped **1** processes generated by the projection when not needed.

3 Typing a Concrete Language

As demonstrated by our examples, choreography and endpoint terms provide a useful language for expressing protocols with adaptation. In this section, we investigate the idea of using such protocols as specifications for a programming language with adaptation. We plan to follow the approach taken in multiparty session types [11], where choreographies (and endpoints) are interpreted as behavioral types for typing sessions in a language modeled as a variant of the π-calculus. In the sequel, we investigate the core points of such a language by giving an implementation that uses the protocols specified in the examples of the previous sections. In particular, we discuss what are the relevant aspects for developing a type system for such a language, whose types are the choreographies introduced in § 2.1.

In both protocols (the prescribe protocol and the test protocol), the doctor plays a key role since (s)he initiates the workflow with prescriptions, decides when tests have to be requested, and decides when the protocols have to be interrupted due to an emergency. A possible implementation of the doctor could be given by the following program:

```
$P_D$ = $\overline{\mathsf{pr}}(k)$; $X[$ repeat {repeat {
                $k : \overline{prescribe}_N\langle e_{pr}\rangle$; $\overline{\mathsf{test}}(k')$;
                $X'[k' : \overline{orderTest}_L\langle e_o\rangle$;
                  $(k' : results_L(x) + X'_{(D,L)}\{X_{(D,N)}\{k : \overline{sign}_N\langle e_s\rangle, k : sign_D(z)\}, \mathbf{1}\})]$
              } until ok$(x)$;
          $(k : \overline{sign}_N\langle e_s\rangle + X_{(D,N)}\{k : \overline{sign}_N\langle e_s\rangle, k : sign_D(z)\}$ ; $k : \overline{up}_N\langle\rangle)$;
          $k : trust_N(t)$} until trusted$(t)$ $]$
```

In the code there are two kinds of communication operations, namely protocol initiation operations, where a new protocol (or session) is initiated, and in-session operations where protocol internal operations are implemented. The communication $\overline{\mathsf{pr}}(k)$ is for initiating a protocol called pr and its semantics is to create a fresh protocol identifier k that corresponds to a particular instance of protocol pr. In-session communications are standard.

The novelty in the process above is in the scope $X[\ldots]$ and update $X_{\ldots}\{\ldots\}$, which state respectively that the program can be adapted at any time in that particular point, and that an adaptation is available. Interestingly enough, the way the program P_D uses the protocols needs care. If the doctor wants to adapt to emergency while waiting for tests, both the test protocol and the prescription protocol need to be adapted as shown in line 4. If the doctor adapts to emergency after having received tests that are ok, then only the prescription protocol needs to be adapted. One can see that session *pr* can be typed using the prescribe endpoint specification and session *test* using the test endpoint specification. The update of X in line 4 does not appear in the protocol test since it acts as an external update for a different protocol.

4 Conclusion

Adaptation is a pressing issue in the design of service-oriented systems, which are typically open and execute in highly dynamic environments. There is a rather delicate tension between adaptation and the correctness requirements defined at service composition time: we would like to adapt the system's behavior whenever necessary/possible, but we would also like adaptation actions to preserve overall correctness. Here we have reported ongoing work on adaptation mechanisms for service-oriented systems specified in terms of choreographies. By enhancing an existing language for choreographies with constructs defining adaptation scopes and dynamic code update, we obtained a simple, global model for distributed, adaptable systems. We also defined an endpoint language for local descriptions, and a projection mechanism for obtaining (low-level) endpoint specifications from (high-level) choreographies.

We now briefly comment on related works. The work in [8] is closely related, for it develops a framework for rule-based adaptation in a choreographic setting. Both choreographies and endpoints are defined; their relation is formally defined via projection. The main difference w.r.t. the work described here is our choice of expressing adaptation in terms of scopes and code update constructs, rather than using rules. Also, we consider choreographies as types and we allow multiple protocols to interleave inside code. These problems are not considered in [8]. Our approach bears some similarities with works on multiparty sessions [11, 3]. Our focus so far has been on formally relating global and local descriptions of choreographies via projection and trace inclusion; investigating correctness properties (e.g., communication safety) via typing in our setting is part of ongoing work. We also note that exceptions and runtime adaptation are similar but conceptually different phenomena: while the former are typically related to foreseen unexpected behaviors in (low-level) programs, adaptation appears as a more general issue, for it should account for (unforeseen) interactions between the system and its (varying) environment. We have borrowed inspiration from [16], in which adaptive case management is investigated by relying on Dynamic Condition Response (DCR) Graphs, a declarative process model. Finally, the adaptation constructs we have considered for choreographies and endpoints draw inspiration from the *adaptable processes* defined in [1]. The application of adaptable processes in models of structured communications (focusing on the case of binary sessions) has been studied in [9].

An immediate topic for future work is the full formalization of the concrete language and its typing disciplines. Other avenues for future research include the investigation of refinement theories with a testing-like approach, enabled by having both systems and adaptation strategies modeled in the same language, and the development of prototype implementations.

References

[1] M. Bravetti, C. Di Giusto, J. A. Pérez, and G. Zavattaro. Adaptable processes. *Logical Methods in Computer Science*, 8(4), 2012.

[2] M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *SC*, volume 4829 of *LNCS*, pages 34–50. Springer, 2007.

[3] S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty sessions. In *FSTTCS*, volume 8 of *LIPIcs*, pages 338–351. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[4] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.

[5] M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013.

[6] S. Carpineti and C. Laneve. A basic contract language for web services. In *ESOP*, volume 3924 of *LNCS*, pages 197–213. Springer, 2006.

[7] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *POPL*, pages 261–272. ACM, 2008.

[8] M. Dalla Preda, I. Lanese, J. Mauro, and M. Gabbrielli. Adaptive choreographies, 2013. Unpublished. Available at `http://www.cs.unibo.it/~lanese/publications/adaptchor.pdf.gz`.

[9] C. Di Giusto and J. A. Pérez. Disciplined structured communications with consistent runtime adaptation. In *SAC*, pages 1913–1918. ACM, 2013.

[10] A. S. Henriksen, L. Nielsen, T. Hildebrandt, N. Yoshida, and F. Henglein. Trustworthy pervasive healthcare services via multiparty session types. In *FHIES*, LNCS, pages 124–141, 2013.

[11] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

[12] I. Lanese, A. Bucchiarone, and F. Montesi. A framework for rule-based dynamic adaptation. In *TGC*, volume 6084 of *LNCS*, pages 284–300. Springer, 2010.

[13] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *SEFM*, pages 323–332. IEEE Computer Society, 2008.

[14] I. Lanese, F. Montesi, and G. Zavattaro. Amending choreographies. In *WWV*, volume 123 of *EPTCS*, pages 34–48. Open Publishing Association, 2013.

[15] K. M. Lyng, T. Hildebrandt, and R. R. Mukkamala. From paper based clinical practice guidelines to declarative workflow management. In *ProHealth*, pages 36–43. BPM 2008 Workshops, 2008.

[16] R. R. Mukkamala, T. Hildebrandt, and T. Slaats. Towards trustworthy adaptive case management with dynamic condition response graphs. In *EDOC*, 2013.

A concurrent programming language with refined session types

Juliana Franco and Vasco Thudichum Vasconcelos

LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

Abstract

We present SePi, a concurrent programming language based on the monadic pi-calculus, where interaction is governed by linearly refined session types. On top of the core calculus and type system, and in order to facilitate programming, we introduce a number of abbreviations and derived constructs. This paper provides a brief introduction to the language.

1 Introduction

Session types [9] are by now a well-established methodology for typed, message-passing concurrent computations. By assigning session types to communication channels, and by checking programs against session type systems, a number of program properties can be established, including the absence of races in channel manipulation operations, and the guarantee that channels are used as prescribed by their types. As a simple example, a type of the form `!string .!integer.end` describes a channel end on which processes may first output a string, then output an integer value, after which the channel provides no further interaction. The process holding the other end of the channel must first input a string, then an integer, as described by the complementary (or dual) type, `?string.?integer.end`. If the string denotes a credit card number and the integer value the amount to be charged to the credit card, then we may further *refine* the type by requiring that the capability to charge the credit card has been offered, as in `?ccard:string.?amount:{x:integer|charge(ccard,x)}.end`. The most common approach to handle refinement types is classical first-order logic which is certainly sufficient for many purposes but cannot treat formulae as resources. In particular it cannot guarantee that the credit card is charged with the given amount once only.

SePi is an exercise in the design and implementation of a concurrent programming language solely based on the message passing mechanism of the pi calculus [13], where process interaction is governed by (linearly refined) session types. SePi allows to explore the practical applicability of recent work on session-based type systems [1, 22], as well as to provide a tool where new program idioms and type developments may be tested and eventually incorporated. In this respect, SePi shares its goal with Pict [16] and TyCO [19].

The SePi core language is the monadic synchronous pi-calculus [13] with replication rather than recursion [12], labelled choice [9], and with assume/assert primitives [1]. On top of this core we provide a few derived constructs aiming at facilitating code development. The current version of the language includes support for mutually recursive process definitions and type declarations, and for polyadic message passing. The type system of SePi is that of linearly refined session types [1], the algorithmic rules for the refinement-free type language are adapted from [22], and those for refinements are described in this paper.

SePi is currently implemented as an Eclipse plug-in, allowing code development (and interpretation) with the usual advantages of an IDE, such as syntax highlighting, syntactic and semantic validation, code completion and refactoring. There is also a command line alternative, in the form of a `jar` file. Installation details and examples can be found at `http://gloss.di.fc.ul.pt/sepi`. The interpreter is based on Turner's abstract machine [18].

The rest of this paper is structured as follows. The next section reviews related work. Section 3 briefly introduces SePi, based on a running example. Section 4 presents a few technical aspects of the language. Section 5 concludes the paper, pointing possible future language extensions.

2 Related work

This section briefly reviews programming language implementations either based on the pi-calculus or that incorporate session types.

There are a few programming languages based on the pi-calculus, but neither incorporates session types. Pict [16] is a language in the ML-tradition, featuring labelled records, higher-order polymorphism, recursive types and subtyping. Similarly to the SePi approach, Pict builds a on a tiny core (a variant of the asynchronous pi-calculus [3, 8]) by adding a few derived constructs. TyCO [20] is another language based on a variant of the asynchronous pi-calculus, featuring labelled messages (atomic select/output) and labelled receptors (atomic branch/input) [19], predicative polymorphism and full type inference. In turn, SePi is based on the monadic synchronous pi-calculus with labelled choice [9], explicitly typed and equipped with refined session types [1]. Polymorphism and subtyping are absent from the current version of SePi.

On the other hand, there are some programming languages that feature session types or variants of these, but are based on paradigms other than the pi-calculus. For functional languages, we find those that take advantage of the rich system of Haskell, monads in particular, and those based on ML. Neubauer and Thiemann implemented session types on Haskell using explicit continuation passing [14]. Sackman and Eisenbach improve this work, augmenting the expressive power of the language [17]. Given that session types are encoded, the Haskell code for session-based programs can be daunting. SePi works directly with session types, thus hopefully leading to readable programs. Bhargavan et al. [2] present a ML-like language for specifying multiparty sessions [10] for cryptographic protocols, with integrity and secrecy support.

For object-oriented languages, Fähndrich et al. developed Sing# [6], a variant of C# that supports message-based communication via shared-memory where session types are used to describe communication patterns. Hu et al. introduced SJ [11], an extension of Java with specific syntax for session types and structured communication operations. Based on a work by Gay et al. [7], Bica [4] is an extension of the Java 5 compiler that checks conventional Java source code against session type specifications for classes. Type specifications, included in Java annotations, describe the order by which methods in classes should be called, as well as the tests clients must perform on results from method calls. Following a similar approach, but using session types with lin/un annotations [22], Mool [5] is a minimal object based language.

Finally, for imperative languages, Ng et al. developed Session C [15], a multiparty session-based programming environment for the C programming language and its runtime libraries [15]. Neither of the languages discussed above feature any form of refinement types, linear or classical.

3 A gentle introduction to the SePi language

This section introduces the SePi language, its syntax, type system and operational semantics. The presentation is intentionally informal. Technical details can be found on the theoretical work the language builds upon, namely [22] for the base language and [1] for refinements.

Our running example is based on the online petition service [21] and on the online store [1]. An Online Donation Server manages donation campaigns. Clients seeking to start a donation campaign for a given cause begin by setting up a session with the server. The session is conducted on a channel on which the campaign related data is provided. The same channel may then be disseminated and used by different benefactors for the purpose of collecting the actual donations. Parties donating for some cause do so by providing a credit card number and the amount to be charged to the card. The type system makes sure that the exact amount specified by the donor is charged, and that the card is charged exactly once.

SePi is about message passing on bi-directional synchronous channels. Each channel is described by two end points. At each point in a program, processes may write on one end or else read from the other end. Channels are governed by types that describe the sequence of messages a channel may carry. We start with *input/output types*. A type of the form **!integer.end** describes a channel end where processes may write an integer value, after which the channel offers no further interaction. Similarly, a type **?integer.end** describes a channel end from which processes may read an integer value, after which the channel offers no further interaction.

To *create a channel* of the above type one writes

```
new w r: !integer.end
```

Such a declaration introduces two identifiers: w of type **!integer.end**, and r of type **?integer.end**. A semantically equivalent declaration is **new** r w: **?integer.end**. To *write* the integer value 2013 on the newly created channel, one writes w!2013. To *read* from the channel and store the value on program variable x one writes r?x. For the purpose of printing integer values on the console, SePi provides a primitive channel printInteger, and similarly for the remaining base types: **boolean** and **string**. Code such as channel writing or reading can be composed by *prefixing* via the dot notation. To read an integer value and then to print it, one writes r?x.printInteger!x. To run two processes in *parallel* one uses the vertical bar notation. Putting everything together one obtains our first complete program, composed of a channel declaration and two processes running in parallel while sharing the channel.

```
new w r: !integer.end
w!2013 | r?x.printInteger!x
```

Running such a program would produce 2013 on the console, after which the program terminates.

We now move on to *choice types*. The donation server allows clients to setup donation campaigns piece-wise. The required information (title, description, due date, etc.) may be introduced in any order, possibly more than once each. Once satisfied, the client presses the "commit" button. A channel end that allows a writer to *select* either the setDate option or the commit option is written as

```
+{setDate:end, commit:end}
```

Conversely, a channel end that provides a menu composed of the two same choices can be written as &{setDate:**end**, commit:**end**}. To select the setDate option on a + channel end we write w **select** setDate. Conversely to branch on a & channel end one may write **case** r **of** setDate → ... commit → ... . Putting everything together one obtains

```
new w r: +{setDate:end, commit:end}
w select setDate |
case r of setDate → printString!"Got setDate"
          commit → printString!"Got commit"
```

Types are composed by prefixing, using the dot notation: **!integer.end** means write an integer and then go on as **end**. We can compose the output and the select type we have seen above, so that the output of an integer is required after the setDate choice is taken. We leave to the reader composing the two programs above so that it interacts correctly on a channel whose client end is of type +{setDate:!integer.end, commit:end}.

The problem with this type is that it does not reflect the idea of "uploading the campaign information until satisfied, and then press the commit button". All a client can do is *either* set the date *or else* commit. What we would like to say is that after the setDate choice is taken the whole menu is again available. For this we require a *recursive* type of the form:

```
rec a.+{setDate:!integer.a, commit:end}
```

A client w may now upload the date two times before committing:

```
w select setDate. w!2012. w select setDate. w!2013. w select commit
```

The donation server, governed by type **rec** a.&{setDate:?**integer**.a, commit:**end**}, needs to continuously offer the setDate and commit options. Such behaviour cannot be achieved with a finite composition of the primitives we have seen so far. We need some form unbounded behaviour, which SePi provides in the form a **def** process. The Setup process below is the part of the donation server responsible for downloading the campaign information. To simplify the example, only the due date is considered and even this information, x, is immediately discarded. We will see that Setup is a form of input channel that survives interaction, thus justifying its invocation with the exact same syntax as message sending: Setup!r.

```
def Setup r: rec a.&{setDate:?integer.a, commit:end} =
    case r of setDate → r?x. Setup!r
              commit  → ...
```

Process definition, **def**, is the second form of declaration in SePi (the first is **new**). There is yet a third kind of declaration (rather, an abbreviation): **type**. Introducing the name Donation for the above recursive type, one may write

```
type Donation = +{setDate: !integer.Donation, commit: end}
```

A sequence of declarations followed by a process is a SePi process. Declarations are mutually recursive. One may write

```
def X z:T = z!true. Y!z
new r w: T
type U = ?boolean.T
def Y z:U = z?b. printBoolean!b. X!z
type T = !boolean.U
X!r | Y!w
```

where the system of equations {T = !**boolean**.U, U =?**boolean**.T} is solved in order to obtain two recursive types, one for T, the other for U.

There is a further handy abbreviation. Session types tend to be quite long; if a channel's end point is of type **rec** a.+{setDate:!**integer**.a, commit:**end**}, the other end is of type **rec** a.&{setDate:?**integer**.a, commit:**end**}. In this case we say that one type is *dual* of the other, a notion central to session types. Given that we abbreviated the first type to Donation, the second can be abbreviated to **dualof** Donation. Putting every together we obtain the following process.

```
type Donation = +{setDate: !integer.Donation, commit: ...}
def Setup r: dualof Donation =
    case r of setDate → r?x. Setup!r
              commit  → ...
new w r: Donation    // the donation channel
```

4

```
w select setDate. w!2012. w select setDate. w!2013. w select commit // a client
Setup!r // a server
```

Continuing with the example. After setup comes the promotion phase. Here the donation channel is used to collect donations from benefactors. Benefactors donate to a cause by providing a credit card number and the amount to be charged to the card. So we rewrite the donation type to:

```
type Donation = +{setDate: !integer.Donation, commit: Promotion}
type CreditCard = string
```

How does type Promotion look like? If we make it !CreditCard.!**integer**.**end**, then the server accepts a single donation. Clearly undesirable. If we choose **rec** a.!CreditCard.!**integer**.a, then we accept an infinite number of donations. And this is undesirable for two reasons: regrettably, no campaign will ever receive an infinite number of donations, and all these donations would have to be issued from the same thread (a process without parallel composition), one after the other. The first problem can be easily circumvented with a rec-choice combination, as in type Donation. The root of the second problem lies in the fact that types are *linear* by default, meaning that each channel end can be known, at any given point in the program, by exactly one thread. And this goes against the idea of disseminating the channel in such a way that any party may individually donate, just by knowing the channel. What we need is to able to classify the channel as *shared* or *unrestricted*. We do so by adding to the type a **un** qualifier.[1]

The type system keeps track of how many threads know a channel end: if **lin** then exactly one, if **un** then zero or more. Linear channels are exempt from races: we do not want two threads competing to set up a donation campaign. Shared channels are prone to races: we do want many (as many as possible) simultaneous benefactors carrying out their donations. Care must however be exerted when using shared channels. Imagine that type Promotion looks like **rec** a.**un**!CreditCard.**un**!**integer**.a, and that we have two donors trying to interact with the server,

```
w!"2345".w!500 | w!"1324".w!2000 | r?x.r?y...
```

Further imagine that the first donor wins the race, and exchanges message "2345". We are left with a process of the form w!500 | w!"1324".w!2000 | r?y..., where the value transmitted on the next message exchange can be an integer value (500) or a string ("1324"), a situation clearly undesirable. To circumvent this situation we pass the two values in a single message, by making w of type **rec** a.**un**!(CreditCard,**integer**).a. This pattern, **rec** a.**un**!T.a, is so common that we provide an abbreviation for it: $*$!T, and similarly for input. So here is the new type for Promotion.

```
type Promotion = *!(CreditCard,integer)
```

Now a client can donate twice (in parallel); it may also pass the channel to all its acquaintances so that they may donate and/or further disseminate the channel. Notice the parallel composition operator enclosed in braces when used within a process.

```
w select setDate. w!2014. w select commit. {
    w!("2345", 500) | w!("1324", 2000) | acquaintance!w
}
```

The ability to define types that "start" as linear (e.g. Donation) and end up as unrestricted (Promotion) was introduced in [22].

So far our example is composed of one server and one client. What if we require more than one client (the plausible scenario for an online system) or more than one server (perhaps for

[1]The **lin** qualifier is optional. For example, !**integer**.**end** abbreviates **lin** !**integer**.**end**.

load balancing)? If we add a second client, in parallel with the above code for the server and the client, the program does not compile anymore: there is a race between the two clients for the linear channel end w. On the one hand we have seen that the donation channel must be linear; on the other hand we want a donation server reading on a well-known, public, **un**, channel. We start by installing the server on a channel end of type ∗?Donation, and disseminate the client end of the channel (of type **dualof** ∗!Donation, that is ∗?Donation). Our main program with two clients looks as follows.

```
new c s: *?Donation // create a Online Donation channel
DonationServer!s |  // send one end to the Donation Server
Client1!c | Client2!c   // let the whole world know the other
```

To obtain a (linear) Donation channel from a (shared) ∗!Donation channel we use a technique called *session initiation*: the server creates a new channel, keeps the **dualof** Donation end to itself and writes the Donation end on the known-to-the-client ∗!Donation channel end.

```
def DonationServer donationServer: *!Donation =
    def Setup r: dualof Donation = <as above>
    new r w: Donation   // create a channel for a new donation campaign
    donationServer!r.   // send one end
    Setup!w.    // keep the other
    DonationServer!donationServer   // serve another client
def Client1 donationServer: *?Donation =
    donationServer?w.   // get a session channel from the donation server
    w select setDate. w!2012... // interact on it
```

We now concentrate on how the donation server charges credit cards. In general, merchants cannot directly charge credit cards. As such our donation server forwards the transaction details (the credit card number and amount to be charged) to the credit card issuer (a bank, for example). Assume the following definition for a bank: **def** Bank (ccard: CreditCard, amount: **integer**). Well behaved servers receive the data and forward it to the bank:

```
r?(ccard, amount). Bank!(ccard, amount)
```

Not so honest servers may try to charge a different amount (perhaps an hidden tax),

```
r?(ccard, amount). Bank!(ccard, amount+10)
```

or to charge the right amount, only that twice.

```
r?(ccard, amount). {Bank!(ccard, amount) | Bank!(ccard, amount)}
```

While types cannot constitute a general panacea for fraudulent merchants, the situation can be improved. The idea is that the bank is not interested in arbitrary (ccard,amount) pairs but else on pairs for which a charge (ccard,amount) capability has been granted. We then *refine* the type of the amount in the Bank's signature. We are now interested on amounts x of type integer for which the predicate charge (ccard,x) holds, that is, parameter amount is of type

```
{x: integer | charge(ccard,x)}
```

The capability of charging a given amount on a specific credit card is usually granted by the benefactor, by *assuming* an instance of the charge predicate, as in:

```
assume charge("2345", 500) | w!("2345", 500)
```

The Bank, in turn, makes sure the transaction details were granted by the client, by *asserting* the same predicate:

```
def Bank (ccard: CreditCard, amount: {x: integer | charge(ccard,x)}) =
    assert charge(ccard, amount)...
```

Assumptions and assertions have no operational significance on well-typed programs. At the type system level, assumptions and assertions are treated *linearly*: for each **assert** there must be exactly one **assume**, and conversely. In this way formulae are treated as resources: they are introduced in the type system via **assume** processes, passed around in refinement types, and consumed via **assert** processes. As such, the code for servers that try to charge twice the right amount (see above) does not compile, for the Bank's "second" **assert** is not matched by any assumption. The code for servers that try to charge a different amount (see above) does not compile either. In this case the benefactor's assumption charge("2345", 500) would never be asserted, whereas the bank's assertion charge("2345", 510) would not have a corresponding assumption. Linearity also means that code for banks that forget to **assert** charge(ccard, amount) does not compile. We leave as an exercise writing a typeful server code that charges an amount different from that stated (and assumed) by the benefactor, and that charges twice the right amount, by careful manipulation of **assume**/**assert** *in the server code.*

Benefactors that wish to be charged twice, may issue two separate assumptions or join them on a single formulae, as in the code below.

```
assume charge("2345",500)*charge("2345",500) | w!("2345",500) | w!("2345",500)
```

Likewise, multiple assertions can be conjoined in one, via the tensor ($*$) formula constructor.

4 Technical aspects of the language

SePi is based on the synchronous monadic pi calculus (as in [22]) extended with **assert** and **assume** primitives (inspired in [1]). Formulae in the current version of the language are built from uninterpreted predicates (over values only), tensor and **unit**. On top of this core calculus we added a few derived constructs, namely support for mutually recursive process definitions and type declarations, and for polyadic message passing. The types of SePi are linearly refined session types [1]. We have also added the **dualof** type operator, the $*$?T and the $*$!T abbreviations[2] (see Section 3), and made the **lin** annotation optional.

The **dualof** type operator produces a new type where input ? is replaced by output !, branching & is replaced by selection +, and conversely in both cases. Furthermore, **dualof end** is **end**; **dualof rec** a.T is **rec** a. **dualof** T; and **dualof** a is a. The operator is defined on these type constructors only.

Mutually recursive declarations (cf. the X!r | Y!w example in Section 3) are elaborated in three phases: first by solving the system of type equations, then by checking channel creation, and lastly by analysing the process definitions. Systems of type equations are guaranteed to have a solution due to the presence of recursive types in the syntax of the language and to the fact that types are required to be contractive.[3]

Each declaration of the form **def** X y:T = P introduces a new channel as in **new** X X1: $*$?(**dualof** T), where X1 is a fresh identifier. In the scope of all new channels (those explicitly introduced with **new** and those introduced by virtue of **def**), we a add *replicated* input processes of the form $*$X1?y.P. Replicated processes survive message reception. For example:

```
new w r: *!integer
w!2013 | *r?x.printInteger!x | w!2014
```

[2]For completeness, the following abbreviations are also available: $*$+{m1,...,mn}, and $*$&{m1,...,mn}.

[3]A type is contractive if it contains no sub-expression of the form **rec** a1. ... **rec** an.a1.

prints two integers, while

```
w!2013 | r?x.printInteger!x | w!2014
```

will print one only. Putting everything together, the X!r | Y!w example in Section 3 is semantically equivalent to

```
new r w: rec a. lin!boolean. lin?boolean.a
new X X1: rec b.un!(rec a. lin!boolean. lin?boolean.a).b
new Y Y1: rec b.un!(rec a. lin?boolean. lin!boolean.a).b
*X1?z. z!true. Y!z |
*Y1?z. z?b. printBoolean!b. X!z |
X!r | Y!w
```

Process declaration obviates in most cases the direct usage of replicated input processes. More important, it hides one of the channel ends (X1) thus simplifying code development, and they are amenable to an optimisation in code interpretation [18].

In order to simulate interference-free polyadic message passing on shared (**un**) channels, we use a standard encoding for the send and receive operations (cf. [22]). For example, the pair-type (ccard: CreditCard, amount: {x: **integer**|charge(ccard,x)}) in the signature of the Bank definition (Section 3) is equivalent to the refined linear session type

```
lin?ccard:CreditCard. lin?amount:{x:integer|charge(ccard,x)}. end
```

where each interaction point in the type is labelled with an optional identifier (e.g., ccard) that may be referred to in the continuation type (e.g., charge(ccard,x)). On the process side, the output process b!("2345", 500).P abbreviates

```
new r w: lin?ccard:CreditCard. lin?amount:{x:integer|charge(ccard,x)}. end
b!r. w!"2345". w!500. P
```

and the input process b?(x,y).P abbreviates

```
b?z. z?x. z?y. P
```

The type system of the SePi language is decidable. The algorithmic rules are those in [22], with minor adaptations in the rules for replicated input and **case** processes. Algorithmic typing systems crucially rely on the decidability of type equivalence. Type equivalence for the non-refined language is decidable [22]. Type equivalence for SePi is also decidable thanks to the extremely simple syntax of formulae. In essence, we keep separated a typing context and a multiset of predicates. An invariant of the type system says that context entries do not contain refinement types at the top level. The type equivalence procedure (basically, equality of regular infinite trees) may use (hence, remove) predicates from the multiset, if required.

The rules for **assume** and **assert** in [1] are not algorithmic. Nevertheless, algorithmic rules are easy to obtain. Processes of the form **assume** A add A to the multiset after breaking the tensors and eliminating occurrences of **unit**; processes of the form **assert** A try to remove the predicates in A from the multiset. Input processes of the form x?y.P eliminate the top-level refinements in the type for y; the resulting type is added to the typing environment, the predicates are added to the multiset (this operation in unnecessary for **new** processes given that channels types cannot be directly refined). The remaining rules remain as in [22], except that they now work with the new procedure for type equivalence.

5 Conclusion and future work

We presented SePi, a concurrent programming language based on the monadic pi-calculus where communication between processes is governed by session types and where linearly refinement types may be used to specify properties about the values exchanged. In order to facilitate programming we added to SePi a few derived constructs, such as output and input of multiple values, mutually recursive process definitions and type declaration, as well as the **dualof** type operator.

Our early experience with the language unveiled a few further constructs that may speed up code development, including: a simple `import` clause allowing the inclusion of code in a different source file, thus providing for limited support for API development; an abbreviation for *session initiation* where a process creates a new channel, keeps one end and passes the other (a form of bound output). In order to keep the language simple, the current version of SePi uses predicates over values only, thus preventing formulae containing expressions, such as `p(x+1)`. We plan to add expressions to predicates, together with the appropriate theories (e.g., arithmetic), combining the current type checking algorithm with an SMT solver. Finally, we acknowledge that the current language of formulae is quite limited (essentially a multiset of uninterpreted formulae). We are working on a system that provides for the persistent availability of resources in a form of replicated (or exponential) resources. Polymorphism and subtyping may be incorporated in future versions of the language. We are also interested in extending the type system so that it may guarantee some form of progress for well typed processes.

Acknowledgements. This work was partially supported by project PTDC/EIA–CCO/1175 13/2010. We are grateful to Dimitris Mostrous, Hugo Vieira, and to anonymous referees for their feedback.

References

[1] Pedro Baltazar, Dimitris Mostrous, and Vasco T. Vasconcelos. Linearly refined session types. *EPTCS*, 101:38–49, 2012.

[2] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *Computer Security Foundations Symposium*, pages 124–140. IEEE, 2009.

[3] Gérard Boudol. Asynchrony and the pi-calculus (note). Rapport de Recherche 1702, INRIA, Sophia-Antipolis, 1992.

[4] Alexandre Caldeira and Vasco T. Vasconcelos. Bica. `http://gloss.di.fc.ul.pt/bica`.

[5] Joana Campos and Vasco T. Vasconcelos. Mool. `http://gloss.di.fc.ul.pt/mool`.

[6] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity OS. *Operating Systems Review*, 40(4):177–190, 2006.

[7] Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Principles of Programming Languages*, pages 299–312. ACM, 2010.

[8] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *European Conference on Object-Oriented Programming*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.

[9] 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 of *LNCS*, pages 22–138. Springer, 1998.

[10] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Principles of Programming Languages*, pages 273–284. ACM, 2008.

[11] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *European Conference on Object-Oriented Programming*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

[12] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, 1992.

[14] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *Practical Aspects of Declarative Languages*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.

[15] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: safe parallel programming with message optimisation. In *Conference on Objects, Models, Components, Patterns*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.

[16] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[17] Matthew Sackman and Susan Eisenbach. Session types in Haskell: Updating message passing for the 21st century. Technical report, Imperial College, Department of Computing, 2008.

[18] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1995.

[19] Vasco T. Vasconcelos. Typed concurrent objects. In *European Conference on Object-Oriented Programming*, volume 821 of *LNCS*, pages 100–117. Springer, 1994.

[20] Vasco T. Vasconcelos. TyCO gently. DI/FCUL TR 01–4, Department of Informatics, Faculty of Sciences, University of Lisbon, 2001.

[21] Vasco T. Vasconcelos. Sessions, from types to programming languages. *Bulletin of the European Association for Theoretical Computer Science*, 103:53–73, 2011.

[22] Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.

Ensuring Faultless Communication Behaviour in an E-Commerce Cloud Application

Ross Horne, Rustem A. Kamun and Timur Umarov

Faculty of Information Technology, Kazakh-British Technical University, Almaty, Kazakhstan
ross.horne@gmail.com r.kamun@gmail.com t.umarov@kbtu.kz

Abstract. The goal of this work is to ensure that processes that integrate several services in a Cloud correctly interact according to a specification of their communication behaviour. To accomplish this goal, we employ session types to analyse the global and local communication patterns. A session type represents a "formal blueprint" of how users and services should interact with the Cloud at an appropriate level of abstraction for specifying message flows.
This work confirms the feasibility of applying session types to business protocols used by an e-commerce Cloud provider. The protocols are developed in SessionJ, an extension of Java implementing session-based programming. Furthermore, we highlight how our approach can be used to intergrate services across multiple Cloud providers, each of whom must correctly cooperate.

1 Introduction

Cloud providers typically offer a portfolio of services, where access and billing for all services are integrated in a single distributed system. Services are then made available on demand to anyone with a credit card, eliminating the up front commitment of users [1]. Furthermore, there is a drive for services to be integrated, not only within a Cloud, but also between multiple Cloud providers [14].

Protocols that integrate heterogeneous services with a single point of access and billing strategy can become complex. Thus we require an appropriate level of abstraction to specify and implement such protocols. Further to the complexity, the protocols are a critical component of the business strategy of a Cloud provider. Failure of the protocols could result in divergent behaviour that jeopardises services, leading to loss of customers or even legal disputes. These risks can be limited by using techniques that statically prove that protocols are correct and dynamically check that protocols are not violated at runtime.

It is challenging to manage service interactions that go beyond simple sequences of requests and responses or involve large numbers of participants. One technique for managing protocols between multiple services is to specify the protocol using a choreography. A choreography specifies a global view of the interactions between participating services. However, by itsself, a choreography does not determine how the global view can be executed.

The challenge of controlling interactions of participants motivated the design of Web Services Choreography Description Language (WS-CDL) [8]. The WS-CDL working group identified critical issues [3] including:

1. the need for tools to validate conformance to choreography specifications to ensure correct cooperation between Web Services;
2. design time verification of choreographies to guarantee correctness of properties such as deadlock and livelock, as well as the conformance of the behaviour of participants.

The aforementioned challenges can be tackled by adopting a solid foundational model, such as session types [8,7]. Successful approaches related to session types include: SessionJ [12], Scribble [11] and Session C [13] due to Honda and Yoshida; Sing# [4] that extends Spec# with choreographies; and UBF(B) [2] for Erlang.

In this paper, we present a case study where the interaction of process that integrate services in a commercial Cloud provider[1] are controlled using session types. Session types ensure communication safety by verifying that session implementations of each participant (the users, the services and the Cloud), conform to the specified protocols. In our case study, we use SessionJ, an extension of Java supporting sessions, to specify protocols used by the Cloud provider that involve iteration and higher order communication.

In Section 2 we provide an overview of SessionJ. In Section 3, we explain and refine a protocol used by a Cloud provider and implemented using SessionJ. Finally, in Section 4, we suggest that session types can be used in the design of reliable inter-Cloud protocols, following the techniques employed in this work.

2 Methodology for Verifying Protocols in SessionJ

We chose SessionJ for the core of our application, since Java was already used for several services. Furthermore, the language has a concise syntax and an active community.

We briefly outline how SessionJ is employed to correctly implement protocols. Firstly, the global protocol is specified using a global calculus similar to sequence diagrams. Secondly, the global calculus is projected to sessions types, which specify the protocol for each participant. Thirdly, the session is implemented using operations on session sockets. The correctness of the global protocol can be verified by proving that the implementation of each session conforms to the corresponding session type.

Protocol Specification. The body of a protocol defines a *session type*, according to the grammar in Figure 1. The session type specifies the actions that the participant in a session should perform. In SessionJ, the behaviour of an implementation of a session is monitored by the associated protocol, as enforced by the SessionJ compiler and runtime. The constructs in Figure 1 can describe a diverse range of complex interactions, including message passing, branching and iteration. Each session type construct has its dual construct, because a typical requirement is that two parties implement compatible protocols such that the specification of one party is dual to another party.

Higher Order Communication. SessionJ allows message types to themselves be session types. This is called higher-order communication and is supported by using subtyping [15]. Consider the dual constructs $!\langle ?(\text{int})\rangle$ and $?(?(\text{int}))$. These types specify

[1] V3na Cloud Platform. AlmaCloud Ltd., Kazakhstan. http://v3na.com

L_1, L_2	label
p	protocol name
$M ::= \mathit{Datatype} \mid T$	message
$S ::= p\,\{T\}$	protocol

$T ::= T\,.\,T$	Sequencing
$\mid$ `begin`	Session initiation
$\mid\ !\langle M \rangle$	Message send
$\mid\ ?(M)$	Message receive
$\mid \{L_1 : T_1, \ldots, L_n : T_n\}$	Session branching
$\mid [T]*$	Session iteration
$\mid$ `rec` $L\,[T]$	Session recursion scope
$\mid \#L$	Recursive jump
$\mid @p$	Protocol reference

Fig. 1: SessionJ protocol specification using session types (T).

sessions that expect to respectively send and receive a session of type ?(int). Higher order communication is often referred to a session delegation. Figure 2 shows a basic delegation scenario.

In Figure 2, the left diagram represents the session configuration before the delegation is performed: the user is engaged in a session s of type $!\langle\text{int}\rangle$ with the Cloud, while the Cloud is also involved in a session s' with a service of type $!\langle?(\text{int})\rangle$. So, instead of accepting the integer from the user, Cloud delegates his role in s to the service. The diagram on the right of Figure 2 represents the session configuration after the delegation has been performed: the user now directly interacts with the service for the session s. The delegation action corresponds to a higher-order send type for the session s' between the Cloud and the service.

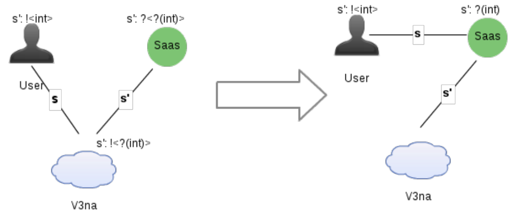


Fig. 2: Session delegation

Protocol Implementation using SessionJ. Session sockets represent the participants of a session connection. Each socket implements the session code according to the specified session type. In SessionJ session sockets extend the abstract *SJSocket* class. The session is implemented within a session-try scope using a vocabulary of session operations.

3 Case Study: Protocols for a Cloud

Our case study is an e-commerce Cloud application, V3na, that provides integrated Software as a Service solutions for business. V3na provides a central access point to several services, including document storage, document flow, and customer relations management. The central component of the e-commerce Cloud is responsible for seamless integration and maintenance of the services that a user subscribes to, while managing user accounts and billing.

A typical scenario is when a user requires the document storage service. The user will first subscribe for the service either by registering to be billed or by entering a trial period. When the user has subscribed for the document service and they attempt to access their documents, requests to the API of the document store are delegated, by V3na, to the relevant document server for a lease period. After delegation, the client interacts directly with the API of the document store until the lease expires.

A major challenge was to automate the process of service integration as a reliable service. In particular, V3na implements the following problems that can be addressed using sessions types:

- A customer can connect to a service for a trial period;
- V3na provides one entry point to all services a user subscribes to;
- A subscription may be extended or frozen;
- Billing and payment for use of a service can be managed.

In this section, we illustrate two refinements of the first scenario above.

3.1 Scenario 1: Forwarding and Branching

To begin, we specify a simple business protocol for connecting to a service. The protocol is informally specified as follows:

Protocol 1.1: User

```
protocol p_uv {
  begin.
  !<JSONMsg>.
  ?{
    OK: ?(JSONMsg),
    FAIL:
  }
}
```

Protocol 1.2: Cloud

```
protocol p_vu {
  begin.?(JSONMsg).!{
    OK: !<JSONMsg>,
    FAIL:
  }
}
protocol p_vs {
  begin.!<JSONMsg>.
        ?(JSONMsg)
}
```

Protocol 1.3: Service

```
protocol p_sv {
  begin.
  ?(JSONMsg).!<JSONMsg>
}
```

Fig. 3: Protocol specifications for Scenario 1

1. The user begins a request session with Cloud service (V3na) and sends the request "connect to service" as JSON-encoded message.
2. V3na selects either:
   (a) FAIL, if the user has no active session (not signed in).

(b) OK, if the user has logged in and the request data has passed validation steps.

3. If OK is selected, then, instead of responding immediately to the user, the Cloud initiates a new session with the service. In the new session the Cloud forwards the JSON message from the client to the service and receives a response from the server. The session between the Cloud and the service terminates. Finally, the original session resumes and the Cloud forwards the response from the service to the Client. From the perspective of the user it appears that the Cloud responded directly.

In Figure 3 we provide the protocol specifications for each participant (User, Cloud or Service). The protocols between the user and the Cloud and between the Cloud and the service are dual, i.e. the specification of interaction from one perspective is opposite to the other perspective. SessionJ employs the `outbranch` and `inbranch` operations to implement the branching behaviour.

3.2 Scenario 2: Session Delegation and Iteration

We present a refined example that demonstrates iteration and session delegation. To avoid becoming a bottleneck, the core processes of the Cloud should delegate the session to a service as soon as the user is authenticated for the service.

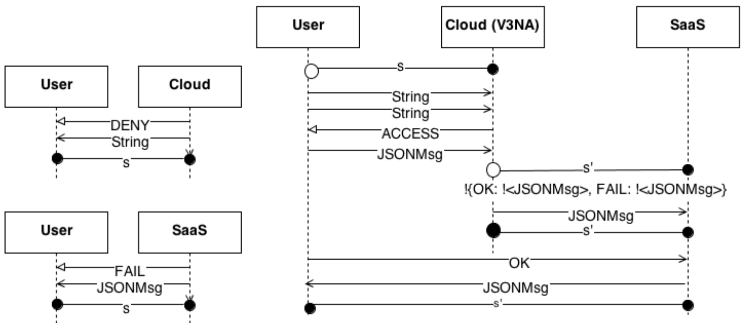


Fig. 4: Sequence diagram of interactions for Scenario 2

Figure 4 depicts two related sessions s and s'. Session s begins with interactions between the user and the Cloud; but, after authentication, s' delegates the rest of session s from the Cloud to the service. Session s is completed by an exchange between the user and the service directly. We informally describe the global protocol in more detail:

1. The user begins a request session (session s in Fig. 4) with the Cloud (V3na).
2. The user logs in by providing the Cloud with a user name and password.
3. V3na receives the user credentials and verifies them: If the user is not authenticated and still has tries go back to step 2, otherwise continue.
4. If the user is not allowed to access the Cloud, the interaction between the user and the Cloud continues on the DENY-branch, otherwise on the ACCESS-branch.

5. In the case of the ACCESS branch, the user sends his connection request in a JSON message to the Cloud. The Cloud creates a new session with the service (session s' in Fig. 4). The new session delegates the remaining session with the user to the service, and also forwards relevant user request details to the service. Session s' is then terminated.
6. The service continues session s, but now interactions are between the user and the service. The service either responds to the user with OK or FAIL. In either case, the user receives the reason and status of his request directly from the service in a JSON message. Finally, session s is terminated.

Protocol 2.1: User

```
protocol p_uv {
  begin.?[!<String>.!<String> ]*.
  ?{
   ACCESS: !<JSONMsg>.
       ?{
         OK: ?(JSONMsg),
         FAIL: ?(JSONMsg)
       },
   DENY: ?(String)
  }
}
```

Protocol 2.2: Cloud

```
protocol p_vu {
  begin.
   ![ ?(String).?(String) ]*. // login
   !{
     ACCESS: ?(JSONMsg).
        !{
          OK: !<JSONMsg>,
          FAIL: !<JSONMsg>
        },
     DENY: !<String>
   }
}
```

Fig. 5: User-Cloud interaction protocol specifications for Scenario 2

In Figure 5, the protocol between the user and the cloud provider appear to interact perfectly. The iterative login step works as expected, so either the ACCESS or DENY branch will be selected. If the ACCESS branch is selected, then, as expected, a JSON message is sent from the user to the Cloud. However, instead of the Cloud choosing OK or FAIL directly, the session (s') in Figure 6 is triggered.

Protocol 2.3: Cloud

```
protocol p_vs {
  begin.
  !<!{
     OK: !<JSONMsg>,
     FAIL: !<JSONMsg>
  }>.
  !<JSONMsg>
}
```

Protocol 2.4: Service

```
protocol p_sv {
  begin.
  ?(!{
     OK: !<JSONMsg>,
     FAIL: !<JSONMsg>
  }).
  ?(JSONMsg)
}
```

Fig. 6: Cloud-Service interaction protocol specifications for Scenario 2

The session in Figure 6 delegates the part of the session where either OK or FAIL is selected to the service. The delegation is enabled by a higher order session type, where a socket of session type $!\{\texttt{OK}{:}\ !\langle \mathit{JSONMessage}\rangle, \texttt{FAIL}{:}\ !\langle \mathit{JSONMessage}\rangle\}$ is sent from the Cloud in protocol *p_vs* and received by the service in protocol *p_sv*. Following, the delegation, a JSON message is sent from the Cloud to the service containing the relevant user request details.

Once the delegation has taken place, the service is able to complete the session that was begun by the Cloud. The service can negotiate directly with the client and either choose the OK branch or the FAIL branch, followed by sending the appropriate JSON message. The simple choice between an OK and a FAIL message could be replaced by a more complex iterated session between the user and the service.

4 Future Work: Inter-Cloud Protocols and Session Types

For customers of Cloud providers, there are considerable benefits when service can be hosted on more than one Cloud provider [6,5,1]. If data is replicated across multiple Cloud providers, customers can avoid becoming locked in to one provider. Thus customers are less exposed to risks such as fluctuations in prices and quality of service at a single provider. If a Cloud provider goes out of business, then customers entirely dependent on one Cloud provider are critically exposed.

Several visions have been proposed for inter-Cloud protocols [10,9]. In [14], they identify the main components of general inter-Cloud architecture: a *Cloud coordinator*, for exposing Cloud services; a *Cloud broker*, for mediating between service consumers and Cloud coordinators; and a *Cloud exchange*, for collecting consumers' demands and locating Cloud providers. Based on our experience in this work, we suggest that multi-party session types [13] are appropriate for specifying and correctly implementing protocols between Cloud providers and Cloud integrators.

5 Conclusion

This case study demonstrates the ability of session types to control interaction patterns between communicating processes in a Cloud. Participants are statically type-checked at compile time and dynamically monitored at run-time to ensure that components critical to a Cloud provider communicate correctly. The high level of abstraction of session types, implemented in the SessionJ language, enabled effortless translation of business scenarios into protocols. We were able to refine our protocol from Scenario 1 to Scenario 2, due to support of higher-order message passing (session delegation). Further to scenarios presented here, our case study covered payment and wallet recharging transactions, where we discovered the benefits of combining session delegation and threading provided by SessionJ. Our experience shows that the session-programming approach is suited to correctly implementing inter-Cloud protocols, which is our future objective.

Acknowlegements. We thank the anonymous reviewers for their clear and constructive comments. We are grateful to Ramesh Kini for his support for this project.

References

1. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
2. Joe Armstrong. Getting Erlang to talk to the outside world. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 64–72. ACM, 2002.
3. Alistair Barros, Marlon Dumas, and Phillipa Oaks. A critical overview of the web services choreography description language. *BPTrends Newsletter*, 3:1–24, 2005.
4. Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. *ACM SIGPLAN Notices*, 47(1):191–202, 2012.
5. David Bernstein, Erik Ludvigson, Krishna Sankar, Steve Diamond, and Monique Morrow. Blueprint for the intercloud-protocols and formats for cloud computing interoperability. In *ICIW'09.*, pages 328–336. IEEE, 2009.
6. Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.
7. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for Web services. In *Programming Languages and Systems*, pages 2–17. Springer, 2007.
8. Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. A theoretical basis of communication-centred concurrent programming. WS-CDL working report, W3C, 2006.
9. E. Cavalcante, F. Lopes, T. Batista, N. Cacho, F.C. Delicato, and P.F. Pires. Cloud integrator: Building value-added services on the Cloud. In *Network Cloud Computing and Applications (NCCA'11)*, pages 135–142, 2011.
10. Vij D. Bernstein. Using xmpp as a transport in intercloud protocols. In *In Proceedings of CloudComp 2010, the 2nd International Conference on Cloud Computing*, pages 973–982. CiteSeer.
11. Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In *Distributed Computing and Internet Technology*, pages 55–75. Springer, 2011.
12. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP 2008*, pages 516–541. Springer, 2008.
13. Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty Session C: Safe parallel programming with message optimisation. In *Objects, Models, Components, Patterns*, pages 202–218. Springer, 2012.
14. Rajiv Ranjan Rajkumar Buyya and Rodrigo N. Calheiros. InterCloud: Utility-oriented federation of Cloud Computing environments for scaling of application services. In *Algorithms and architectures for parallel processing*, pages 13–31. Springer, 2010.
15. Malcolm Hole Simon Gay. Subtyping for session types in the pi calculus. *Journal Acta Informatica*, 42(2-3):191–225, 2005.

A Typing System for Privacy

Dimitrios Kouzapas[1] and Anna Philippou[2]

[1] Department of Computing, Imperial College London
dk208@doc.ic.ac.uk
[2] Department of Computer Science, University of Cyprus
annap@cs.ucy.ac.cy

Abstract

In this paper we report on work-in-progress towards defining a formal framework for studying privacy. Our framework is based on the π-calculus with groups [1] accompanied by a type system for capturing privacy-related notions. The typing system we propose combines a number of concepts from the literature: it includes the use of *groups* to enable reasoning about information collection, it builds on *read/write capabilities* to control information processing, and it employs *type linearity* to restrict information dissemination. We illustrate the use of our typing system via simple examples.

1 Introduction

The notion of privacy does not have a single solid definition. It is generally viewed as a collection of related rights as opposed to a single concept and attempts towards its formalization have been intertwined with philosophy, legal systems, and society in general. The ongoing advances of network and information technology introduce new concerns on the matter of privacy. The formation of large databases that aggregate sensitive information of citizens, the exchange of information through e-commerce as well as the rise of social networks, impose new challenges for protecting individuals from violation of their right to privacy as well as for providing solid foundations for understanding privacy a term.

A study of the diverse types of privacy, their interplay with technology, and the need for formal methodologies for understanding and protecting privacy is discussed in [7], where the authors base their arguments on the taxonomy of privacy rights by Solove [6]. According to [6], the possible privacy violations within a system can be categorized into four groups: *invasions*, *information collection*, *information processing*, and *information dissemination*. These violations are typically expressed within a model consisting of three entities: the *data subject* about whom a *data holder* has information and the *environment*, the data holder being responsible to protect the information of the data subject against unauthorized adversaries in the environment.

The motivation for this work stems from the need to provide a formal framework (or a set of different formal frameworks) for reasoning about privacy-related concepts, as discussed above. Such a framework would provide solid foundations for understanding the notion privacy and it would allow to rigorously model and study privacy-related situations. Our interest for formal privacy is primarily focused on the processes of *information collection*, *information processing*, and *information dissemination* and how these can be controlled in order to guarantee the preservation of privacy within a system.

1.1 Privacy and the π-Calculus

The approach we follow in this paper attempts to give a correspondence between the requirements of the last paragraph and the theory and meta-theory of the π-calculus [4]. The π-calculus is a formal model of concurrent computation that uses message-passing communication as the primitive computational function. A rich theory of operational, behavioural and type system semantics of the π-calculus is used as a tool for the specification and the study of concurrent systems. Our aim is to use the π-calculus machinery to describe notions of privacy. Specifically, we are interested in the development of a meta-theory, via a typing system, for the π-calculus that can enforce properties of privacy, as discussed above.

The semantics for the $G\pi$-calculus, a π-calculus that disallows the leakage of information (secrets) is presented in [1]. That work proposes the *group type* along with a simple typing system that is used to restrict the scope of a name's existence, i.e., a name cannot exist outside its group scope. We find the semantics of the $G\pi$-calculus convenient to achieve the privacy properties regarding the *information collection* category. A data holder can use the group type to disallow unauthorized adversaries from collecting information about a data subject.

Consider for example the processes:

$$\begin{array}{rcl} \mathsf{DBadmin} & = & \overline{a}\langle c\rangle.\mathbf{0} \\ \mathsf{Nurse} & = & a(x).\overline{b}\langle x\rangle.\mathbf{0} \\ \mathsf{Doctor} & = & b(x).x(y).\overline{x}\langle \mathsf{data}\rangle.\mathbf{0} \end{array}$$

The database administrator process DBadmin sends a reference c to a patient's data to a doctor process Doctor using a nurse process Nurse as a delegate. Channel c is sent to the nurse via channel a and is then forwarded to the doctor via channel b by the nurse. The doctor then uses c to read and write data on the patient's records. The composition of the above processes under the fresh hospital group Hosp, and an appropriate typing of c, enforces that no external adversary will be able to collect the information exchanged in the above scenario, namely c: name c, belonging to group G, is not possible to be leaked outside the defined context because (1) groups are not values and cannot be communicated and (2) the group G is only known by the three processes (see [1] for the details).

$$(\nu\ \mathsf{Hosp})(((\nu c : \mathsf{Hosp}[])\mathsf{DBadmin}) \mid \mathsf{Nurse} \mid \mathsf{Doctor})$$

Let us now move on to the concept of *information processing* and re-consider the example above with the additional requirement that the nurse should not be able to read or write on the patient's record in contrast to the doctor who is allowed both of these capabilities. To address this issue we turn to the input/output typing system for the π-calculus of Pierce and Sangiorgi, [5]. Therein, the input/output subtyping is used to control the input and output capabilities on names and it is a prime candidate for achieving privacy with respect to the requirement in question: A type system that controls read and write capabilities[1] can be used by a data holder to control how the information about a data subject can be processed. Thus, in the case of our example, the requirements may be fulfilled by extending the specification with a read/write typing system as follows:

$$\begin{array}{rcl} T_{\mathsf{data}} & = & \mathsf{Hosp}[\mathsf{MedicalData}]^{-} \\ T_c & = & \mathsf{Hosp}[T_{\mathsf{data}}]^{\mathtt{rw}} \\ T_a & = & \mathsf{Hosp}[\mathsf{Hosp}[T_{\mathsf{data}}]^{-}]^{\mathtt{rw}} \\ T_b & = & \mathsf{Hosp}[\mathsf{Hosp}[T_{\mathsf{data}}]^{\mathtt{rw}}]^{\mathtt{rw}} \end{array}$$

where names a, b and c are of types T_a, T_b and T_c, respectively. The medical data are a basic type with no capability of read and write. Channel c can be used for reading and writing medical data. Channel a is used to pass information to the nurse without giving permission to the nurse to process the received information, while channel b provides read and write capabilities to the doctor. Nonetheless, the above system suffers from the following problem. Although the nurse acquires restricted capabilities for channel c via channel a, it is still possible for a nurse process to exercise its read capability on b and, thus, acquire read and write capability on the c channel. To avoid this problem, the system may be

[1]The terminology for read and write capabilities is equivalent with input and output terminology.

redefined as follows:

$$\begin{array}{rcl}\mathsf{DBadmin} & = & (\nu b : T_b)\, \overline{tonurse}\langle b\rangle.\overline{todoc}\langle b\rangle.\bar{a}\langle c\rangle.\mathbf{0} \\ \mathsf{Nurse} & = & tonurse(z).a(x).\bar{z}\langle x\rangle.\mathbf{0} \\ \mathsf{Doctor} & = & todoc(z).z(x).x(y).\bar{x}\langle \mathsf{data}\rangle.\mathbf{0}\end{array}$$

where channel *todoc* has type $\mathsf{Hosp}[T_b]^{rw}$ but channel *tonurse* has type $\mathsf{Hosp}[T'_b]^{rw}$, where $T'_b = \mathsf{Hosp}[\mathsf{Hosp}[T_{\mathsf{data}}]^{\mathtt{rw}}]^{\mathtt{w}}$. In other words, the nurse is not assigned read capabilities on channel b.

Note that the above typing is not completely sound: for instance the nurse process is expected to pass on to the doctor process more capabilities than those it acquires via channel a. Nevertheless in our theory we use a more complex type structure able to solve this problem.

Regarding the *information dissemination* category of privacy violations, we propose to handle information as a *linear resource*. Linear resources are resources that can be used for some specific number of times. A typing system for linearity was originally proposed in [3]. A linear typing system can be used by the data holder to control the number of times an information can be disseminated. In our example, we require from the nurse the capability of sending the reference of the patient only once, while we require from the doctor not to share the information with anyone else:

$$\begin{array}{rcl} T_{\mathsf{data}} & = & \mathsf{Hosp}[\mathsf{MedicalData}]^{-*} \\ T_c & = & \mathsf{Hosp}[T_{\mathsf{data}}]^{\mathtt{rw}*} \\ T_a & = & \mathsf{Hosp}[\mathsf{Hosp}[T_{\mathsf{data}}]^{-1}]^{\mathtt{rw}0} \\ T_b & = & \mathsf{Hosp}[\mathsf{Hosp}[T_{\mathsf{data}}]^{\mathtt{rw}0}]^{\mathtt{rw}0} \\ T'_b & = & \mathsf{Hosp}[\mathsf{Hosp}[T_{\mathsf{data}}]^{\mathtt{rw}0}]^{\mathtt{w}0} \end{array}$$

The $*$ annotation on the types above defines a shared (or unlimited) resource. Such resources are the patient's data and the reference to the patient's data. Channels a and b communicate values that can be disseminated one and zero times respectively. (Again there is a soundness problem solved by a more complex typing structure.) Furthermore channels a and b cannot be sent to other entities.

A central aspect of our theory is the distinction between the basic entities. The operational semantics of the π-calculus focuses on the communication between processes that are composed in parallel. Although a process can be thought of as a computational entity, it is difficult to distinguish at the operational level which processes constitute a logical entity. In our approach, we do not require any operational distinction between entities, since this would compromise the above basic intuition for the π-calculus, but we do require the logical distinction between the different entities that compose a system.

Finally, we note that our typing system employs a combination of i/o types and linear types, which are low-level π-calculus types, to express restrictions on system behavior. We point out that the expressivity of such ordinary π-calculus types has been studied in the literature and, for instance, in [2] the authors in fact prove that linear and variant types can be used to encode session types.

2 The Calculus

Our study of privacy is based on the π-calculus with groups proposed by Cardelli et al. [1]. In this section we briefly overview the syntax and reduction semantics of the calculus.

Beginning with the syntax, this is standard π-calculus syntax with the addition of the group restriction construct, $(\nu\ G)P$, and the requirement for typing on bound names (the definition of types is in Section 3).

$$P \quad ::= \quad x(y{:}T).P \ \mid\ \bar{x}\langle z\rangle.P \ \mid\ (\nu\ G)P \ \mid\ (\nu\ a{:}T)P \ \mid\ P_1 \mid P_2 \ \mid\ !P \ \mid\ \mathbf{0}$$

Free names $\mathtt{fn}(P)$, bound names $\mathtt{bn}(P)$, free variables $\mathtt{fv}(P)$, and bound variables $\mathtt{bv}(P)$ are defined in the standard way for π-calculus processes. We extend this notion to the sets of free groups in a process P and a type T which we denote as $\mathtt{fg}(P)$ and $\mathtt{fg}(T)$, respectively.

We now turn to defining the reduction semantics of the calculus. This employs the notion of *structural congruence* which allows the structural rearrangement of a process so that the reduction rules can be performed. Structural congruence is the least congruence relation, written $\equiv$, that satisfies the rules:

$$\begin{array}{ll}
P \mid \mathbf{0} \equiv P & (\nu\, a{:}T)P_1 \mid P_2 \equiv (\nu\, a : T)(P_1 \mid P_2) \text{ if } a \notin \mathtt{fn}(P_2) \\
P_1 \mid P_2 \equiv P_2 \mid P_1 & (\nu\, a{:}T_1)(\nu\, b{:}T_2)P \equiv (\nu\, b{:}T_2)(\nu\, a{:}T_1)P \\
(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) & (\nu\, G)P_1 \mid P_2 \equiv (\nu\, G)(P_1 \mid P_2) \text{ if } G \notin \mathtt{fg}(P_2) \\
!P \equiv P \mid !P & (\nu\, G_1)(\nu\, G_2)P \equiv (\nu\, G_2)(\nu\, G_1)P
\end{array}$$

$$(\nu\, G_1)(\nu\, a{:}T)P \equiv (\nu\, a{:}T)(\nu\, G_1)P \text{ if } G \notin \mathtt{fg}(T)$$

We may now present the reduction relation $P \longrightarrow Q$ which consists of the standard π-calculus reduction relation extended with a new rule for group creation.

$$\begin{array}{rcl}
 & \overline{a}\langle b\rangle.P_1 \mid a(x : T).P_2 \longrightarrow P_1 \mid P_2\{b/x\} & \\
P_1 \longrightarrow P_2 & \text{implies} & P_1 \mid P_3 \longrightarrow P_2 \mid P_3 \\
P_1 \longrightarrow P_2 & \text{implies} & (\nu\, G)P_1 \longrightarrow (\nu\, G)P_2 \\
P_1 \longrightarrow P_2 & \text{implies} & (\nu\, a : T)P_1 \longrightarrow (\nu\, a : T)P_2 \\
P_1 \equiv P_1', P_1' \longrightarrow P_2', P_2' \equiv P_2 & \text{implies} & P_1 \longrightarrow P_2
\end{array}$$

3 Types and Typing System

In this section we define a typing system for the calculus which builds upon the typing of [1]. The typing system includes: (i) the notion of groups of [1], (ii) the read/write capabilities of [5] extended with the empty capability, and (iii) a notion of linearity on the dissemination of names. The type structure is used for static control over the permissions and the disseminations on names in a process.

For each channel, its type specifies (1) the group it belongs to, (2) the type of values that can be exchanged on the channel, (3) the ways in which the channel may be used in input/output positions (permissions p below) and (4) the number of times it may be disseminated (linearity λ below):

$$\begin{array}{lcll}
T & ::= & G[\,]^{p\lambda} \mid G[T]^{p\lambda} & \\
p & ::= & - \mid \mathtt{r} \mid \mathtt{w} \mid \mathtt{rw} & \\
\lambda & ::= & * \mid i & \text{where } i \geq 0
\end{array}$$

For example, a channel of type $T = G[\,]^{\mathtt{r}2}$ is a channel belonging to group G that does not communicate any names, can be used in input position and twice in object position. Similarly, a name of type $G'[T]^{\mathtt{rw}*}$ is a channel of group G' that can be used in input and output position for exchanging names of type T and can be sent as the object of a communication for an arbitrary number of times.

Subtyping. Our typing system makes use of a subtyping relation which, in turn is, based on two pre-orders, one for permissions p, denoted as $\sqsubseteq_p$, and one for linearities λ, denoted as $\sqsubseteq_\lambda$:

$\sqsubseteq_p$:	$\mathtt{rw} \sqsubseteq_p \mathtt{w}$	$\mathtt{rw} \sqsubseteq_p \mathtt{r}$	$\mathtt{rw}, \mathtt{r}, \mathtt{w} \sqsubseteq_p -$
$\sqsubseteq_\lambda$:	$* \sqsubseteq_\lambda i$ for all i	$i \sqsubseteq_\lambda j$ if $i \geq j$	

The preorder for permissions is as expected with the empty capability being the greatest element. For linearities, *fewer* permissions are included in *larger* permissions and $*$ is the least element.

Let Type be the set of all types T. The subtyping relation, written $\leq$ as an infix notation, may be defined coinductively as the largest fixed point ($\mathscr{F}^\omega(\mathsf{Type} \times \mathsf{Type})$) of the monotone function:

$$\mathscr{F} : (\mathsf{Type} \times \mathsf{Type}) \longrightarrow (\mathsf{Type} \times \mathsf{Type})$$

where

$$\begin{array}{rcl}
\mathscr{F}(\mathscr{R}) & = & \{(G[]^{-0}, G[]^{-0})\} \\
& \cup & \{(G[T_1]^{p\lambda_1}, G[T_2]^{-\lambda_2})) \mid (T_1, T_2) \in \mathscr{R}, (T_2, T_1) \in \mathscr{R}, \lambda_1 \sqsubseteq_\lambda \lambda_2\} \\
& \cup & \{(G[T_1]^{p\lambda_1}, G[T_2]^{\mathtt{r}\lambda_2}) \mid (T_1, T_2) \in \mathscr{R}, p \sqsubseteq_p \mathtt{r}, \lambda_1 \sqsubseteq_\lambda \lambda_2\} \\
& \cup & \{(G[T_1]^{p\lambda_1}, G[T_2]^{\mathtt{w}\lambda_2}) \mid (T_2, T_1) \in \mathscr{R}, p \sqsubseteq_p \mathtt{w}, \lambda_1 \sqsubseteq_\lambda \lambda_2\} \\
& \cup & \{(G[T_1]^{\mathtt{rw}\lambda_1}, G[T_2]^{\mathtt{rw}\lambda_2}) \mid (T_1, T_2), (T_2, T_1) \in \mathscr{R}, \lambda_1 \sqsubseteq_\lambda \lambda_2\}
\end{array}$$

The first pair in the construction of $\mathscr{F}$ says that the least base type is reflexive. The next four cases define subtyping based on the preorders defined for permissions and linearities. According to the second case, the empty permission is associated with an invariant subtyping relation because the empty permission disallows for a name to be used for reading and/or writing. The read permission follows covariant subtyping, the write permission follows contravariant subtyping, while the read/write permission follows invariant subtyping. Note that linearities are required to respect the relation $\lambda_1 \sqsubseteq \lambda_2$ for subtyping in all cases. For example, according to the subtyping relation, the following hold: $G_1[G_2[]^{\mathtt{rw}5}]^{\mathtt{rw}*} \leq G_1[G_2[]^{\mathtt{w}3}]^{\mathtt{r}3}$, $G_1[G_2[]^{-3}]^{\mathtt{rw}*} \leq G_1[G_2[]^{\mathtt{w}3}]^{\mathtt{w}0}$, and $G_1[G_2[]^{\mathtt{w}5}]^{\mathtt{rw}*} \leq G_1[G_2[]^{\mathtt{w}5}]^{-1}$.

Typing Judgements. We now turn to the typing system of our calculus. This assigns an extended notion of a type on names which is constructed as follows:

$$\mathsf{T} = (T_1, T_2)$$

In a pair T we record the current capabilities of a name, captured by T_1, and its future capabilities after its dissemination, captured by T_2.

Based on these extended types, the environment on which type checking is carried out in our calculus consists of the components Π and Γ. These declare the names (free and bound) and groups in scope during type checking. We define Γ-environments by $\Gamma ::= \emptyset \mid \Gamma \cdot x : \mathsf{T} \mid \Gamma \cdot G$. The domain of an environment Γ, $\mathtt{dom}(\Gamma)$, is considered to contain all names and groups recorded in Γ. We assume that any name and group in $\mathtt{dom}(\Gamma)$ occurs exactly once in Γ. Then, a Π-environment is defined by $\Pi ::= \emptyset \mid \Pi \cdot x : \mathsf{T}$, where $\mathtt{dom}(\Pi)$ contains all variables in Π, each of which must exist in Π exactly once.

We define three typing judgements: $\Gamma \vdash x \triangleright T$, $\Pi \vdash x \triangleright T$, and $\Pi, \Gamma \vdash P$. The first two typing judgement say that under the typing environment Γ, respectively Π, variable x has type T. The third typing judgement stipulates that process P is well typed under the environments Π, Γ, where Γ records the groups and the types of the free names of P and Π the types of all bound names x that are created via a (νx) construct within P. We require that these bound names are uniquely named within P and, if needed, we employ α conversion to achieve this. In essence, this restriction requires for all freshly-created names to be recorded a-priori within the typing environment. If an unrecorded name is encountered, then the typing system will lead to failure as is implemented by the typing system. It turns out that recording this information on bound names of a process is necessary in order to control the internal processing of names that carry sensitive data.

Typing System. We now move on to the rules of our typing system. First, we present two auxiliary functions. To begin with we define the linearity addition operator $\oplus$ where $\lambda_1 \oplus \lambda_2 = *$, if $\lambda_1 = *$ or $\lambda_2 = *$, and $\lambda_1 \oplus \lambda_2 = \lambda_1 + \lambda_2$, otherwise. We may now lift this notion to the level of typing environments via operator $\odot$ which composes its arguments by concatenating their declarations with the exception of the common domain where linearities are added up via $\oplus$:

$$\begin{array}{rcl} \Gamma_1 \odot \Gamma_2 & = & \Gamma_1 \backslash \Gamma_2 \cdot \Gamma_2 \backslash \Gamma_1 \\ & \cdot & \{x : G[T]^{p\lambda_1 \oplus \lambda_2} \mid x : G[T]^{p\lambda_1} \in \Gamma_1, x : G[T]^{p\lambda_2} \in \Gamma_2\} \end{array}$$

At this point we make the implicit assumption that Γ_1 and Γ_2 are compatible in the sense that the declared types of common names may differ only in the linearity component.

We are ready now define the typing system:

$$(\mathsf{Name}) \quad \frac{x \notin \mathtt{dom}(\Gamma \cdot \Gamma') \qquad \mathtt{fg}(\mathsf{T}) \subseteq \mathtt{dom}(\Gamma \cdot \Gamma')}{\Gamma \cdot x : \mathsf{T} \cdot \Gamma' \vdash x \triangleright \mathsf{T}} \qquad (\mathsf{SubN}) \quad \frac{\Gamma \vdash x \triangleright (T_1', T_2'), T_1' \leq T_1, T_2' \leq T_2}{\Gamma \vdash x \triangleright (T_1, T_2)}$$

$$(\mathsf{In}) \quad \frac{\Pi, \Gamma \cdot y : (T_1, T_2) \vdash P \qquad \Gamma \vdash x \triangleright (G[T_1]^{\mathtt{r}0}, G[T_2]^{\mathtt{r}0})}{\Pi, \Gamma \vdash x(y : T_1).P} \qquad (\mathsf{Out}) \quad \frac{\Pi, \Gamma \cdot y : (G_y[T_1]^{-\lambda}, T_2) \vdash P \qquad \Gamma \vdash x \triangleright (G_x[T_2]^{\mathtt{w}0}, G_x[T_2]^{\mathtt{w}0})}{\Pi, \Gamma \cdot y : (G_y[T_1]^{-(\lambda \oplus 1)}, T_2) \vdash \overline{x}\langle y \rangle.P}$$

$$(\mathsf{ResG}) \quad \frac{\Pi, \Gamma \cdot G \vdash P}{\Pi, \Gamma \vdash (\nu\ G)P} \qquad (\mathsf{ResN}) \quad \frac{\Pi, \Gamma \cdot x : (T, T') \vdash P}{\Pi \cdot x : (T, T'), \Gamma \vdash (\nu\ x : T)P}$$

$$(\mathsf{Par}) \quad \frac{\Pi_1, \Gamma_1 \vdash P_1 \quad \Pi_2, \Gamma_2 \vdash P_2}{\Pi_1 \odot \Pi_2, \Gamma_1 \odot \Gamma_2 \vdash P_1 \mid P_2} \qquad (\mathsf{Rep}) \quad \frac{\Pi, \Gamma \vdash P \qquad \forall x \in \mathtt{fn}(P) \text{ if } \Gamma \vdash x \triangleright (G[T_1]^{p\lambda_1}, G[T_2]^{p\lambda_2}) \text{ then } \lambda_1 \in \{0, *\}}{\Pi, \Gamma \vdash !P}$$

$$(\mathsf{Nil}) \quad \Pi, \Gamma \vdash \mathbf{0} \qquad (\mathsf{SubP}) \quad \frac{\Pi, \Gamma \cdot x : (T_1', T_2') \vdash P \quad T_1' \leq T_1, T_2' \leq T_2}{\Pi, \Gamma \cdot x : (T_1, T_2) \vdash P}$$

Rule (Name) is used to type names. Note that in name typing we require that all group names of the type are present in the typing environment. Rule (SubN) defines a subsumption based on subtyping for channels. Rule (In) types the input prefixed process. We first require that the input subject has at least permission for reading. Then, the type y is included in the type environment Γ with a type that matches the type of the input channel x. This is to ensure that the input object will be used as specified. The rule for the output prefix (Out) checks that the output subject has write permissions. Furthermore, x should be a channel that can communicate names up-to type T_2, the maximum type by which y can be disseminated. Then, the continuation of the process P, should be typed according to the original type of y and with its linearity reduced by one. Finally, the output object should have at least the empty permission.

In rule (ResG) we record a newly-created name in Γ. For name restriction (ResN) specifies that a process type checks only if the restricted name is recorded in environment Π. In this way is is possible to control the internal behavior of a process, in order to avoid possible privacy violations. Parallel composition uses the $\odot$ operator to compose typing environments, since we want to add up the linearity usage of each name. For the replication operator, axiom (Rep) we require that free names of P have either linearity zero (i.e. they are not sent by P) or infinite linearity (i.e. they can be sent as many times as needed). The inactive process can be typed under any typing environment (axiom (Nil)). Finally we have a subsumption rule, (SubP) that uses subtyping to control the permissions on processes.

Type Soundness. We prove that the typing system is sound through a subject reduction theorem. Before we proceed with the subject reduction theorem we state the basic auxiliary lemmas.

Lemma 1 (Weakening)**.**

1. If $\Gamma \vdash x \triangleright \mathsf{T}$ and $y \notin \mathtt{dom}(\Gamma)$ then $\Gamma \cdot y : \mathsf{T}' \vdash x \triangleright \mathsf{T}$.
2. If $\Pi, \Gamma \vdash P$ and $y \notin \mathtt{dom}(\Gamma)$ then $\Pi, \Gamma \cdot y : \mathsf{T} \vdash P$.

Lemma 2 (Strengthening)**.**

1. If $\Gamma \cdot y : \mathsf{T}' \vdash x \triangleright \mathsf{T}$, $y \neq x$, then $\Gamma \vdash x \triangleright \mathsf{T}$.
2. If $\Pi, \Gamma \cdot y : \mathsf{T} \vdash P$ and $y \notin \mathtt{fn}(P)$ then $\Pi, \Gamma \vdash P$.

Lemma 3 (Substitution)**.** If $\Pi, \Gamma \cdot x : \mathsf{T} \vdash P$ and $\Gamma \vdash y \triangleright \mathsf{T}$ then $\Pi, \Gamma \vdash P\{y/x\}$

Lemma 4 (Subject Congruence)**.** If $\Pi, \Gamma \vdash P_1$ and $P_1 \equiv P_2$ then $\Pi, \Gamma \vdash P_2$.

We are now ready to state the Subject Reduction theorem.

Theorem 1 (Subject Reduction)**.** Let $\Pi, \Gamma \vdash P$ and $P \longrightarrow P'$ then $\Pi, \Gamma \vdash P'$.

Proof. The proof is by induction on the reduction structure of P.
Basic Step:
$P = \overline{a}\langle b \rangle.P_1 \mid a(x).P_2 \longrightarrow P_1 \mid P_2\{b/x\}$ and $\Pi, \Gamma \vdash P$. From the typing system we get that

$$\Gamma = \Gamma_1 \odot \Gamma_2 \tag{1}$$
$$\Pi_1, \Gamma_1 \vdash \overline{a}\langle b \rangle.P_1 \tag{2}$$
$$\Pi_2, \Gamma_2 \vdash a(x).P_2 \tag{3}$$

From the typing system we get that $\Pi_1, \Gamma_1 \vdash P_1$ for (2) and $\Pi_2, \Gamma_2 \cdot x : T \vdash P_2$ for (3). We apply the substitution lemma (Lemma 3) to get that $\Pi_2, \Gamma_2 \cdot b : T \vdash P_2\{b/x\}$. We can now conclude that $\Pi, \Gamma \vdash P_1 \mid P_2\{b/x\}$.
Induction Step:
Case: Parallel Composition. Let $P_1 \mid P_2 \longrightarrow P_1' \mid P_2$ with $\Pi, \Gamma \vdash P_1 \mid P_2$. From the induction hypothesis we know that $\Pi_1, \Gamma_1 \vdash P_1$ and $\Pi_1, \Gamma_1 \vdash P_1'$. From these two results and the parallel composition typing we can conclude that $\Pi_1 \odot \Pi_2, \Gamma_1 \odot \Gamma_2 \vdash P_1 \mid P_2$ and $\Pi_1 \odot \Pi_2, \Gamma_1 \odot \Gamma_2 \vdash P_1' \mid P_2$ as required.
Case: Group Restriction. Let $(\nu\ G)P \longrightarrow (\nu\ G)P'$ with $\Pi, \Gamma \vdash (\nu\ G)P$. From the induction hypothesis we know that $\Pi, \Gamma \cdot G \vdash P$ and $\Pi, \Gamma \cdot G \vdash P'$. If we apply the name restriction rule on the last result we get $\Pi, \Gamma \vdash (\nu\ G)P'$.
Case: Name Restriction. Let $(\nu\ a : T)P \longrightarrow (\nu\ a : T)P'$ with $\Pi \cdot a : T, \Gamma \vdash P$. From the induction hypothesis we know that $\Pi, \Gamma \cdot a : T \vdash P$ and $\Pi, \Gamma \cdot a : T \vdash P'$. If we apply the name restriction rule on the last result we get $\Pi \cdot a : T, \Gamma \vdash (\nu\ a : T)P'$.
Case: Structural Congruence closure. We use the subject congruence lemma (Lemma 4).

Let $P = P_1, P_1 \longrightarrow P_2, P_2 \cong P'$ with $\Pi, \Gamma \vdash P$. We apply subject congruence on P to get $\Pi, \Gamma \vdash P_1$. Then we apply the induction hypothesis and subject congruence once more to get the required result. □

4 Examples

In this section we show simple use cases that apply the theory developed. We also show how we tackle different problems that might arise.

4.1 Patient Privacy

Our first example revisits our example from the introduction and completes the associated type system. Recall the scenario where a database administrator (process $\mathsf{DBadmin}$) sends a reference to the medical data of a patient to a doctor (process Doctor) using a nurse (process Nurse) as a delegate.

$$\begin{array}{rcl}
\mathsf{DBadmin} & = & (\nu b : T_b)\, \overline{\mathsf{tonurse}}\langle b\rangle.\overline{\mathsf{todoc}}\langle b\rangle.\overline{a}\langle c\rangle.\mathbf{0} \\
\mathsf{Nurse} & = & \mathsf{tonurse}(z).a(x).\overline{z}\langle x\rangle.\mathbf{0} \\
\mathsf{Doctor} & = & \mathsf{todoc}(z).z(x).x(y).\overline{x}\langle \mathsf{data}\rangle.\mathbf{0}
\end{array}$$

The processes are composed together inside the hospital (Hosp) group.

$$\mathsf{Hospital} = (\nu\ \mathsf{Hosp})(((\nu c : T_c)\mathsf{DBadmin}) \mid \mathsf{Nurse} \mid \mathsf{Doctor})$$

Our prime interest is to avoid leakage of the data during their dissemination to the doctor. This means that the nurse should not have access to the patients' data. On the other hand the doctor should be able to read and update medical data, but not be able to send the data to anyone else. We can control the above permissions using the following typing.

We define the types

$$\begin{array}{rcl}
\mathsf{T}_{\mathsf{data}} & = & \mathsf{Hosp}[]^{-*} \\
\mathsf{T}_c & = & \mathsf{Hosp}[\mathsf{T}_{\mathsf{data}}]^{\mathtt{rw}*} \\
\mathsf{T}_a & = & \mathsf{Hosp}[\mathsf{Hosp}[\mathsf{T}_{\mathsf{data}}]^{-1}]^{\mathtt{rw}0} \\
\mathsf{T}'_a & = & \mathsf{Hosp}[\mathsf{T}_c]^{\mathtt{rw}0} \\
\mathsf{T}_b & = & \mathsf{Hosp}[\mathsf{Hosp}[\mathsf{T}_{\mathsf{data}}]^{\mathtt{rw}0}]^{\mathtt{rw}2} \\
\mathsf{T}^n_b & = & \mathsf{Hosp}[\mathsf{Hosp}[\mathsf{T}_{\mathsf{data}}]^{\mathtt{rw}0}]^{\mathtt{w}0} \\
\mathsf{T}^d_b & = & \mathsf{Hosp}[\mathsf{Hosp}[\mathsf{T}_{\mathsf{data}}]^{\mathtt{rw}0}]^{\mathtt{r}0} \\
\mathsf{T}_{td} & = & \mathsf{Hosp}[\mathsf{T}^n_b]^{\mathtt{rw}0} \\
\mathsf{T}_{tn} & = & \mathsf{Hosp}[\mathsf{T}^d_b]^{\mathtt{rw}0}
\end{array}$$

to construct:

$$\begin{array}{rcl}
D & = & (\mathsf{T}_{\mathsf{data}}, \mathsf{T}_{\mathsf{data}}) \\
C & = & (\mathsf{T}_c, \mathsf{T}_c) \\
A & = & (\mathsf{T}_a, \mathsf{T}'_a) \\
B & = & (\mathsf{T}_b, \mathsf{T}_b) \\
TD & = & (\mathsf{T}_{td}, \mathsf{T}_{td}) \\
TN & = & (\mathsf{T}_{tn}, \mathsf{T}_{tn}).
\end{array}$$

We can show that:

$$b : B \cdot c : C, \mathsf{tonurse} : TN \cdot \mathsf{todoc} : TD \cdot a : A \cdot \mathsf{data} : D \vdash \mathsf{Hospital}$$

Now, let us consider the case where the nurse sends channel c on a private channel in an attempt to gain access on the patient's medical data:

$$\mathsf{Nurse}_2 \;=\; \mathsf{tonurse}(z).a(x).(\nu\ e : T_b)(\overline{e}\langle x\rangle.\mathbf{0} \mid e(y).y(w).\mathbf{0})$$

In this case, in order for the resulting system to type-check, the type of name e would be recorded in the environment Π, as in

$$e : B \cdot b : B \cdot c : C, \mathsf{tonurse} : TN \cdot \mathsf{todoc} : TD \cdot a : A \cdot \mathsf{data} : D$$
$$\vdash (\nu\ \mathsf{Hosp})(((\nu c : T_c)\mathsf{DBadmin}) \mid \mathsf{Nurse}_2 \mid \mathsf{Doctor})$$

This implies that, if we allow the creation of e, there is possibility of violation in a well-typed process. To avoid this, the administrator of the system should observe all names created and included in Π and, in this specific case, disallow the creation of e. In future work we intend to address this point by providing typing policies that capture this type of problems and to refine our type system to disallow such privacy violations, possibly by controlling the process of name creation.

4.2 Social Network Privacy

Social networks allow users to share information within social groups. In the example that follows we define a type system to control the privacy requirements of participating users. In particular, we consider the problem where a user can make a piece of information public (e.g. a picture), but require that only specific people (his friends) can see it (and do nothing else with it).

The example considers a user who makes public the address, paddr, of a private object, pic, and wishes only the friend Friend to be able to read pic through the public address paddr. To achieve this the user makes available through the typing of name public only the object capability for paddr. However, by separately providing the friend with name a, it is possible to extend the capabilities of paddr to the read capability. In this way, channel a acts as a key for Friend to unlock this private information. Assuming that notAFriend does not gain access to a name of type T_a, as in the process below, he will never be able to obtain read capability on channel paddr.

$$\begin{array}{rcl}
\mathsf{User} & = & (\nu a : T_a)(\overline{\mathsf{tofriend}}\langle a\rangle.(\nu \mathsf{paddr} : T_{paddr})(!\overline{\mathsf{public}}\langle \mathsf{paddr}\rangle.\mathbf{0} \mid !\overline{\mathsf{paddr}}\langle \mathsf{pic}\rangle.\mathbf{0})) \\
\mathsf{notAFriend} & = & \mathsf{public}(z).\mathbf{0} \\
\mathsf{Friend} & = & \mathsf{tofriend}(x).\mathsf{public}(y).(\overline{x}\langle y\rangle.\mathbf{0} \mid x(z).z(w).\mathbf{0})
\end{array}$$

The processes are composed together inside the SN group.

$$\mathsf{SocialNetwork} = (\nu\ \mathsf{SN})(\mathsf{User} \mid \mathsf{notAFriend} \mid \mathsf{Friend})$$

To achieve this, we define the types

$$\begin{array}{rcl}
\mathsf{T}_{\mathsf{pic}} & = & \mathsf{SN}[]^{-*} \\
\mathsf{T}_{\mathsf{paddr}} & = & \mathsf{SN}[\mathsf{T}_{\mathsf{pic}}]^{\mathtt{rw}*} \\
\mathsf{T}'_{\mathsf{paddr}} & = & \mathsf{SN}[\mathsf{T}_{\mathsf{pic}}]^{-1} \\
\mathsf{T}_{a} & = & \mathsf{SN}[\mathsf{T}_{\mathsf{paddr}}]^{\mathtt{rw}*} \\
\mathsf{T}_{\mathsf{tofriend}} & = & \mathsf{SN}[\mathsf{T}_{a}]^{\mathtt{rw}0} \\
\mathsf{T}_{\mathsf{public}} & = & \mathsf{SN}[\mathsf{T}_{\mathsf{paddr}}]^{\mathtt{rw}0} \\
\mathsf{T}'_{\mathsf{public}} & = & \mathsf{SN}[\mathsf{T}'_{\mathsf{paddr}}]^{\mathtt{rw}0}
\end{array}$$

which are combined into the following tuples

$$\begin{aligned} PIC &= (\mathsf{T}_{\mathsf{pic}}, \mathsf{T}_{\mathsf{pic}}) \\ A &= (\mathsf{T}_a, \mathsf{T}_a) \\ PA &= (\mathsf{T}_{\mathsf{paddr}}, \mathsf{T}_{\mathsf{paddr}}) \\ TF &= (\mathsf{T}'_{\mathsf{tofriend}}, \mathsf{T}_{\mathsf{tofriend}}) \\ PB &= (\mathsf{T}'_{\mathsf{public}}, \mathsf{T}_{\mathsf{public}}) \end{aligned}$$

We can show that:

$$a : A \cdot \mathsf{paddr} : PA, \mathsf{tofriend} : TF \cdot \mathsf{public} : PB \cdot \mathsf{pic} : PIC \vdash \mathsf{SocialNetwork}$$

whereas for $\mathsf{notAFriend}' = \mathsf{public}(z).z(w).\mathbf{0}$ and

$$\mathsf{SocialNetwork}' = (\nu\ \mathsf{SN})(\mathsf{User} \mid \mathsf{notAFriend}' \mid \mathsf{Friend})$$

the following judgment fails.

$$a : A \cdot \mathsf{paddr} : PA, \mathsf{tofriend} : TF \cdot \mathsf{public} : PB \cdot \mathsf{pic} : PIC \vdash \mathsf{SocialNetwork}'$$

5 Conclusions

In this paper we have presented a formal framework based on the π-calculus with groups for studying privacy. Our framework is accompanied by a type system for capturing privacy-related notions: it includes the use of *groups* to enable reasoning about information collection, it builds on *read/write capabilities* to control information processing, and it employs *type linearity* to restrict information dissemination. We illustrate the use of our typing system via simple examples.

In future work we would like to develop a policy language for defining privacy policies associated to our framework and subsequently to refine our type system so that it can check the satisfaction/violation of these policies. Furthermore, we would like to study the relation of our type system to other typing systems in the literature.

References

[1] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.

[2] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *Proceedings of PPDP'12*, pages 139–150. ACM, 2012.

[3] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[4] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[5] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

[6] D. J. Solove. A Taxonomy of Privacy. *University of Pennsylvania Law Review*, 154(3):477–560, 2006.

[7] M. C. Tschantz and J. M. Wing. Formal methods for privacy. In *Proceedings of FM'09*, LNCS 5850, pages 1–15. Springer, 2009.

Scalable session programming for heterogeneous high-performance systems

Nicholas Ng, Nobuko Yoshida and Wayne Luk

Imperial College London

Abstract

This paper introduces a programming framework based on the theory of session types for safe and scalable parallel designs. Session-based languages can offer a clear and tractable framework to describe communications between parallel components and guarantee communication-safety and deadlock-freedom by compile-time type checking and parallel MPI code generation. Many representative communication topologies such as ring or scatter-gather can be programmed and verified in session-based programming languages. We use a case study involving N-body simulation to illustrate the session-based programming style. Finally, we outline a proposal to integrate session programming with heterogeneous systems for efficient and communication-safe parallel applications by a combination of code generation and type checking.

1 Introduction

Software programs that utilises parallelism to increase performance is no longer an exclusive feature of high performance applications. Modern day hardware, from multicore processor in smartphones to multicore multi-graphics card gaming systems, all take advantage of parallelism to improve performance. Message-passing is a scalable programming model for parallel programming, where the user has to make communication between components explicit using the basic primitives of message *send* and *receive*.

However, writing correct parallel programs is far from straightforward – blindly parallelising components with data dependencies might leave the overall program in an inconsistent state; arbitrary interleaving of parallel executions combined with complex flow control can easily lead to unexpected behaviour, such as blocked access to resources in a circular chain (i.e. deadlock) or mismatched send-receive pairs. These unsafe communications are a source of non-termination or incorrect execution of a program. Thus tracking and avoiding communication errors of parallel programs is as important as ensuring their functional correctness.

This work focuses on a programming framework which can automatically ensure deadlock-freedom and communication-safety i.e. matching communication pairs, for message-passing parallel programs based on the theory of *session types* [3, 4]. Towards the end of this paper, we discuss how this session-based programming framework can fit in heterogeneous computing environments with reconfigurable acceleration hardware such as Field Programmable Gate Arrays (FPGAs).

To illustrate how session types can track communication mismatches, consider the parallel program in Figure 1 that exchanges two values between two processes.

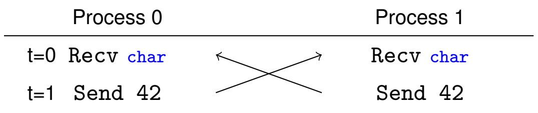


Figure 1: Mismatched communication.

In this notation, the arrow points from the sender of the message to the intended receiver. Both `Process0` and `Process1` start by waiting to receive a value from the other processes, hence we have a typical deadlock situation.

A simple solution is to swap the order of the receive and send commands for one of the processes, for example, `Process0`, shown in Figure 2.

	Process 0		Process 1
t=0	`Send 42`	$\longrightarrow$	`Recv char`
t=1	`Recv char`	$\longleftarrow$	`Send 42`

Figure 2: Communication order swapped.

However, the above program still has mismatched communication pairs and causing type error. Parallel programming usually involves debugging and resolving these communication problems, which is often a tedious task.

Using the session programming methodology, we can not only statically check that the above programs are incorrect, but can also encourage programmers to write safe designs from the beginning, guided by the information of types. Session types [3, 4] have been actively studied as a high-level abstraction of structured communication-based programming, which are able to accurately and intelligibly represent and capture complex interaction patterns between communicating parties.

The two examples above have session types shown in Figure 3 and Figure 4 respectively.

`Process 0`:	Recv `char`; Send `int`
`Process 1`:	Recv `char`; Send `int`

Figure 3: Session types for original example.

`Process 0`:	Send `int`; Recv `char`
`Process 1`:	Recv `char`; Send `int`

Figure 4: Session types for swapped example.

In the session types above, Send `int` stands for output with type `int` and Recv `int` stands for input with type `int`. The session types are used to check that the communications between `Process 0` and `Process 1` are *incompatible* (i.e. incorrect) because one process must have a *dual type* of the other.

On the other hand, the following program is correct, having neither deadlock nor type errors, since it has a *mutually dual* session types shown on the right hand side:

	Process 0		Process 1
t=0	`Send 'a'`	$\longrightarrow$	`Recv char`
t=1	`Recv int`	$\longleftarrow$	`Send 42`

`Process 0`:	Send `char`; Recv `int`
`Process 1`:	Recv `char`; Send `int`

In the session types theory, Recv `type` is dual to Send `type`, hence the type of `Process 0` is dual of the type of `Process 1`.

The above compatibility checking is simple and straightforward in the case of two parties. We can extend this idea to multiparty processes (i.e. more than two processes) based on multiparty session type theory [4]. Type-checking for parallel programs with multiparty processes is done statically and is efficient, with a polynomial-time bound with respect to the size of the program.

Below we list the contributions of this paper.

- Novel programming languages for communications in parallel designs and two session-based approaches to guarantee communication-safety and deadlock freedom (§ 2);
- Implementations of advanced communication topologies for parallel computer clusters by session types (§ 3)
- A case study with N-body simulation to illustrate session programming for clusters (§ 4)

2 Session-based language design

2.1 Overview

As a language independent framework for communication-based programming, session types can be applied to different programming languages and environments. Previous work on Session Java (SJ) [5, 8] integrated sessions into the object-oriented programming paradigm as an extension of the Java language, and was applied to parallel programming [8]. Session types have also been implemented in different languages such as OCaml, Haskell, F#, Scala and Python. This section explains session types and their applications, focussing on an implementation of sessions in the C language (Session C) as a parallel programming framework. Amongst all these different incarnations of session types, the key idea remains unchanged. A session-based system provides (1) a set of predefined primitives or interfaces for session communication and (2) a session typing system which can verify, at compile time, that each program conforms to its session type. Once the programs are type checked, they run correctly without deadlock nor communication errors.

2.2 Multiparty session programming

Session C [9,17] implements a generalised session type theory, *multiparty session types* (MPST) [4]. The MPST theory extends the original binary session types [3] by describing communications across multiple participants in the form of *global protocols*. Our development uses a Java-like protocol description language Scribble [1, 12] for describing the multiparty session types. Figure 5 explains two design flows of Session C programming. In the type checking approach, the programmer writes a global protocol starting from the keyword `protocol` and the protocol name. In the first box of Figure 5, the protocol named as `P` contains one communication with a value typed by `int` from participant `A` to participant `B`. For Session C implementation, the programmer uses the *endpoint protocol* generated by the projection algorithm in Scribble. For example, the above global protocol is projected to `A` to obtain `int to B` (as in the second box) and to `B` to obtain `int from A`. Each endpoint protocol gives a template for developing safe code for each participant and as a basis for static verification. Since we started from a correct global protocol, if endpoint programs (in the third box) conform to the induced endpoint protocols, it automatically ensures deadlock-free, well-matched interactions. This endpoint projection approach is particularly useful when many participants are communicating under complex communication topologies. Due to space limitation, this paper omits the full definition of global protocols, and will explain our framework and examples using only endpoint protocols introduced in the next subsection.

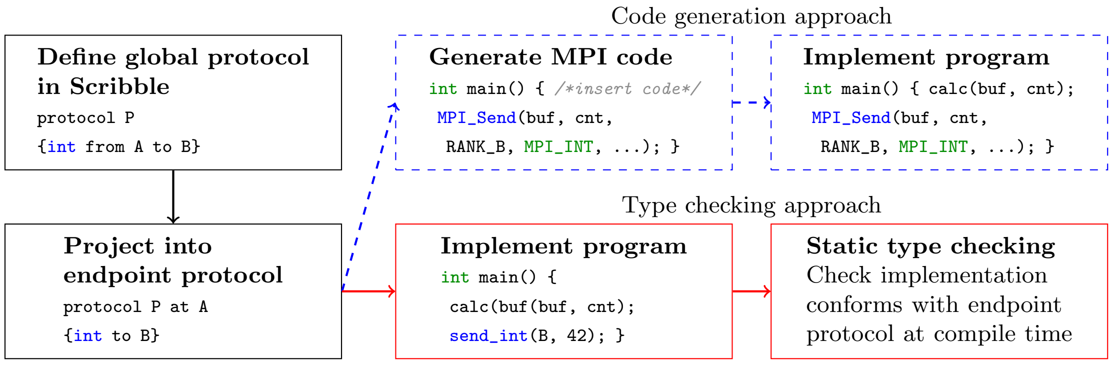


Figure 5: Session C design flows.

2.3 Protocols for session communications

The endpoint protocols include types for basic message-passing and for capturing control flow patterns. We use the endpoint protocol description derived from Scribble to algorithmically specify high-level communication of distributed parallel programs as a library of network communications. A protocol abstracts away the contents but keeps the high level structures of communications as a series of type primitives.

The syntax of Scribble is described in details in [1,12], and can be categorised to three types of operations: *message-passing*, *choice* and *iteration*.

Message passing. It represents that messages (or data) being communicated from one process to another; in the language it is denoted by the statements `datatype to P1` or `datatype from P0` which stands for sending/receiving data of `datatype` to the participant identified by `P0`/`P1` respectively. Notice that the protocol does not specify the value being sent/received, but instead designate the datatype (which could be primitive types such as `int` or composite types), indicating its nature as a high-level abstraction of communication.

Choice. It allows a communication to exhibit different behavioural flows in a program. We denote a choice by a pair of primitives, `choice from` and `choice to`, meaning a distributed choice receiver and choice maker, respectively. A choice maker first decides a branch to take, identified by its `label`, and executes its associated block of statements. The chosen label is sent to the choice receiver, which looks up the label in its choices and execute the its associated block of statements. This ensures the two processes are synchronised in terms of the choice taken.

Iteration. It can represent repetitive communication patterns. We represent recursion by the `rec` primitive (short for recursion), followed by the block of statements to be repeated, enclosed by braces. The operation does not require communication as it is a local recursion. However two communicating processes have to ensure both of their endpoint protocols contains recursion, otherwise their protocols will not be compatible.

2.4 Session C

We present two approaches to session programming in C, using the Session C framework. The first approach (§ 2.4.1) is by type checking of user written code, using a simple session programming API we provided. The second approach (§ 2.4.2) is by MPI code generation from protocols.

2.4.1 Type checking approach

In the type checking approach, a user implements a parallel program using the simple API provided by the library, following communication protocols stipulated in Scribble. Once a program is complete, the type checker verifies that the program code matches that of the endpoint protocol description in Scribble to ensure that the program is safe. The core runtime API corresponds the endpoint protocol as described below.

Message passing primitives in Session C are written as `send_datatype(participant, data)` for message send, which is `datatype to participant` in the protocol, and `recv_datatype(participant, &data)` for message receive (`datatype from participant` in the protocol).

Choice in Session C is a combination of ordinary C control-flow syntax and session primitives. For a choice maker, each if-then or if-else block in a session-typed choice starts with `outbranch(participant, branchLabel)` to mark the beginning of a choice. `inbranch(participant, &branchLabel)` is a choice receiver, used as the argument of a switch-case statement, and each case-block is distinguished by the `branchLabel` corresponding to a choice in the `choice from` block in the protocol.

Iteration in Session C corresponds to `while` loops in C. As no communication is required, the implementation simply repeats a block of code consisting of above session primitives in a `rec` recursion block.

2.4.2 Code generation approach

In the code generation approach, given a Scribble protocol, we generate an MPI parallel program skeleton. The program skeleton contains all the MPI code needed, the user inserts code that performs computation on the input data (e.g. for scientific calculation) between the MPI primitives, completing the program.

This approach is part of a larger extension of the Scribble language to support parameterised session types [18]. The extension, *Parameterised Scribble*, or Pabble, uses indices to parameterise participants. Participants can be defined and accessed in an array-like notation, in order to denote logical groupings of related participants. For example, a parallel algorithm that uses many parallel workers, can define a group of participants using `role participant[1..N]`, and a pipeline of message passing is written in Pabble as `datatype from participant[i:1..N-1] to participant[i+1]`. Pabble protocols can be written once, and a protocol with different number of participants can be instantiated by changing the value of `N`. MPI code generated from Pabble protocols can also take advantage of this feature and will be scalable over different number of processes.

These two approaches to session programming complement each other and cover different use cases: critical applications can use the type checking approach to ensure that the written program is communication and type safe; whereas scalable and parametric applications can use the MPI code generation capability to create communication safe and type safe parallel programs.

3 Advanced communication topologies for clusters

This section shows how session endpoint protocols introduced in § 2.3 can be used to specify advanced, complex communications for clusters. Consider a heterogeneous cluster with multiple kinds of acceleration hardware, such as GPUs or FPGAs, as Processing Elements (PEs). To allow a safe and high performance collaborative computation on the cluster, we can describe communications between PEs by our communication primitives. The PEs can be abstracted as small computation functions with a basic interface for data input and result output, hence we can easily describe high-level understanding of the program by the session types.

We list some widely used structured communication patterns that form the backbones of implementations of parallel algorithms. These patterns were chosen because they exemplify representative communication patterns used in clusters. Computation can interleave between statements if no conflict in the data dependencies exists. The implementation follows the theory of the optimisation for session types developed in [7], maximising overlapped messaging.

Node$_{0\le i\le n-1}$:

```
rec LOOP { // Repeat shifting ring
  datatype to Node[i+1]; // Next
  datatype from Node[i-1]; // Prev
LOOP }
```

Node$_n$:

```
rec LOOP { // Repeat shifting ring
  datatype from Node[N-1]; // Prev
  datatype to Node[0]; // Initial
LOOP }
```

Figure 6: Endpoint protocols of Ring.

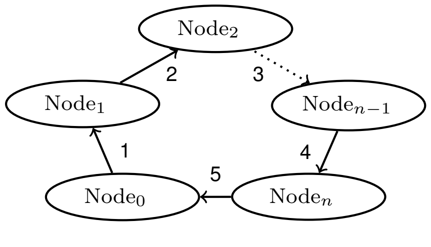


Figure 7: n-node ring pipeline.

Ring topology. In a ring topology, as depicted in Figure 7, processes or PEs are arranged in a pipeline, where the end of the node of the pipeline is connected to the initial node. Each of the connections of the nodes is represented by an individual endpoint session. We use N-body simulation as an example for ring topology. Note that the communication patterns between the middle $n-1$ Nodes are identical. The endpoint protocol is shown in Listing 6.

Map-reduce pattern. Map-reduce is a common scatter-gather pattern used to parallelise tasks that can be easily partitioned with few dependencies between the partitioned computations. The topology is shown in Figure 9. It combines the map pattern which partitions and distributes data to parallel workers by a Master coordination node, and the reduce pattern which collects and combines completed results from all parallel workers. At the end of a map-reduce, the Master coordination node will have a copy of the final results combined into a single datum. All Workers in a map-reduce topology share a simple communication pattern, where they only interact with the Master coordination node. The Master node will have a communication pattern containing all known Workers.

The MPI operation `MPI_Alltoall` is a communication-only instance of the map-reduce pattern for all of the nodes, and only applies memory concatenation to the collected set of data. Our endpoint types can represent this topology with more fine-grained primitives so that we can obtain performance gain by communication-computation overlap. Although collective operations are more efficient in cases where the implementations take advantage of the underlying architectures, fine-grained primitives can more readily allow partial data-structures to be distributed, without the need to create new copies of data or calculating offsets (as in `MPI_Alltoallv`) for transmission.

Master :

```
rec LOOP {
  // Map phase
  datatype to Worker[0], Worker[1];
  // Reduce phase
  datatype from Worker[0], Worker[1];
LOOP }
```

$\text{Worker}_{0\leq i\leq n}$:

```
rec LOOP {
  datatype from Master; // Map phase
  datatype to Master; // Reduce phase
LOOP }
```

Figure 8: Endpoint protocols of Map-reduce.

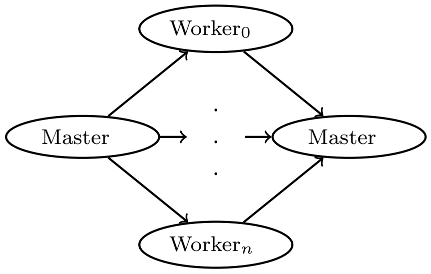


Figure 9: Map-reduce pattern.

4 Case study: N-body simulation

We implemented a 2-dimension N-body simulation using a ring topology. Each `Worker` is initially assigned to a partition of the input data. In every round of the ring propagation, each `Worker` receives a set of partitioned input from a neighbour, and pipelines the input data received from the previous round to the other neighbour. This propagation continues until the set of particles have been received by all `Workers` once. The algorithm will them perform one step of global update to calculate the new positions of the particles after one time step of the simulation.

```
global protocol Nbody(role Worker[0..N] {
  rec RING {
    // Workers 0 to N: Worker[i] -> Worker[i+1]
    int from Worker[i:1..N-1] to Worker[i+1];

    // Data from Worker[N] -> Worker[0]
    int from Worker[N] to Worker[0];

    continue RING;
  }
}
```

Listing 1: Protocol of N-body simulation.

```
protocol Nbody at Worker[0..N] {
  rec RING {
    // Workers 0 to N: Worker[i] -> Worker[i+1]
    if Worker[i:1..N] int from Worker[i-1];
    if Worker[i:0..N-1] int to Worker[i+1];
    // Data from Worker[N] -> Worker[0]
    if Worker[0] int from Worker[N];
    if Worker[N] int to Worker[0];
    continue RING;
  }
}
```

Listing 2: Worker endpoint of N-body protocol.

Listing 1 is the protocol specification of the `Worker` participant of our N-body simulation implementation, and Listing 2 is the automatically generated endpoint version, both written in the syntax of parameterised Scribble.

The block `rec RING { }` means recursion, and represents the repeating ring propagation in the algorithm. The line `if Worker[i:1..N] int from Worker[i-1]` stands for receiving a message from my previous neighbour `Worker[i-1]` with a message of type `int`, given that the current participant is one of `Worker[1]`, ..., or `Worker[N]`. The above protocol generates MPI code equivalent to below:

```
while (i++<N) {
  if (1<=rank && rank<=N) MPI_Recv(rbuf, count, rank-1, MPI_INT, ..);
// (Sub-compute) Send received data to FPGA to process ..
  if (0<=rank && rank<=N-1) MPI_Send(sbuf, count, rank+1, MPI_INT, ..);
  if (rank==0) MPI_Recv(rbuf, count, N, MPI_INT, ..);
// (Sub-compute) Send received data to FPGA to process ..
  if (rank==N) MPI_Send(sbuf, count, 0, MPI_INT, ..); }
// Perform global update after round
```

In MPI, all processes share the same source code and compiled program file, and they are only distinguished at runtime by their assigned process id. The process id is stored in the `rank` variable, and is available throughout the program to calculate participants addresses. In the above MPI code, `MPI_Send` and `MPI_Recv` are the primitives in the MPI library to send and receive data, and all the lines are guarded by a rank check. The variables `sbuf` and `rbuf` stand for send buffer and receive buffer respectively, `count` is the number of elements to send/receive (i.e. array size); `MPI_INT` is an MPI defined macro to indicate the data being sent/received is of type `int`.

The ring topology above is a simple yet powerful topology to distribute data between multiple participants in small chunks. This allows more sub-computation and will potentially allow more overlapping between communication and computation.

A Scribble protocol contains the interaction patterns (i.e. the session typing) for a set of participants. It contains sufficient information to generate the MPI code shown above.

5 Related work and conclusion

ISP [15] and the distributed DAMPI [16] are formal dynamic verifiers which apply model-checking techniques to standard MPI C source code to detect deadlocks using a test harness. The tool exploits independence between thread actions to reduce the state space of possible thread interleavings of an execution, and checks for potentially violating situations. TASS [13] is another suite of tools for formal verification of MPI-based parallel programs by model-checking. It constructs an abstract model of a given MPI program and uses symbolic execution to evaluate the model, which is checked for a number of safety properties including potential deadlocks and functional equivalences.

Compared to the test-based and model-checking approaches which may not be able to cover all possible states of the model, the session type-based approach does not depend on external testing or extraction of models from program code for safety. It encourages designing communication-correct programs from the start, especially given the high level communication structure which session types captures.

Recent works [2,6] used annotated MPI code and a software verifier to check the annotated MPI code for compliance against session types. Their bottom-up approach focusses on accurately representing MPI primitives and datatypes, whereas Session C treats them as high level abstractions, often ignoring details such as send/receive data payload size.

There are a lot of challenges of verifying real-world MPI source code. MPI is a standardised and platform independent message-passing API, the ubiquitous nature in supercomputing makes it a convenient abstraction layer between software and underlying hardware. In cases such as [11], it was used as a programming model for FPGAs. Hence its specification is intentionally vague, in order to allow different implementations to take advantage of any platform-specific optimisations. For example, there are a number of message transport modes such as the more commonly used `MPI_Send/Recv` (standard mode) or `MPI_Isend/Irecv` (immediate/non-blocking). The modes do not correspond directly to standard synchronous or asynchronous communication modes as one would expect. The different communication modes in MPI have subtle differences in their semantics. Care must be taken when making assumptions and correspondences with high-level Scribble protocols. In addition to standard point-to-point communication primitives, MPI also includes a huge number of primitives such as collective operators, topology construction and process management. A complete session type checking framework will be able to consider these additional information to extract the session types from the source code. Combining the flexibility of the host language (C) and the large number of MPI primitives makes our approach more challenging compared to model checking based approaches. This is because MPI model checkers work by observing the behaviours of the programs, which the same behaviour can be implemented in many different ways; whereas our type based approach requires us to understand the consequences of each primitive because we construct a type model without executing the program.

This paper is an extension of our previous works on Session C [9,10]. In both of the works, parametric protocols and MPI code generation were not explored, this work is a short insight into the benefits of using parametric protocols and potentials of integrating with specialised accelerators, as the framework was evaluated on [14], a heterogeneous cluster with FPGAs.

6 Future work

At BEAT workshop, we plan to mainly present our progress on the first topic.

Integration with heterogeneous workflow. Immediate future works includes refining our MPI code generation tool to better integrate with APIs of specialised hardware. This includes streamlining the data received/sent from MPI directly into input/output buffers of acceleration hardware.Tighter integration between MPI and acceleration hardware will achieve better overall performance of the heterogeneous system.

Type-directed optimisations. Extending our type checker to support inferring parameterised MPST from MPI code is a prerequisite for type-directed optimisations. Once parameterised MPST can be extract from MPI code, Session C framework can then extend the support of asynchronous message optimisation [7] described in Session C framework [9] to expressive parameterised protocols. The theoretical and engineering challenges of this future work will be keeping type checking process decidable and representing most, if not all, of the common MPI primitives in Scribble.

Assertion and error recovery. We propose the use of runtime assertions for session-based programming in the Session C framework. Assertions are properties that are expected to hold during runtime, and they can complement static type checking. Error recovery is also a topic of interest, as large scale high performance parallel applications often need to gracefully handle unexpected errors such as hardware failures. Type-based approach to error handling and recovery will be explored as part of ongoing research on Scribble.

Acknowledgements. The research leading to these results has received funding from EPSRC EP/F003757/01, EP/G015635/01 and the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521. The support by UK EPSRC, the HiPEAC NoE, the Maxeler University Program, and Xilinx is gratefully acknowledged.

References

[1] K. Honda et al. Scribbling interactions with a formal foundation. In *ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.

[2] K. Honda, E. Marques, F. Martins, N. Ng, V. Vasconcelos, and N. Yoshida. Verification of MPI programs using session types. In *Proc. EuroMPI 2012*, volume 7940 of *LNCS*, pages 291–293. Springer, 2012.

[3] K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.

[4] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, volume 5201, page 273, 2008.

[5] R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP*, volume 5142 of *LNCS*, pages 516–541, 2008.

[6] E. Marques, F. Martins, V. Vasconcelos, N. Ng, and N. Martins. Towards deductive verification of MPI programs against session types. In *Proc. PLACES 2013*, 2013. To appear.

[7] D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP*, volume 5502 of *LNCS*, pages 316–332, 2009.

[8] N. Ng et al. Safe Parallel Programming with Session Java. In *COORDINATION*, volume 6721 of *LNCS*, pages 110–126, 2011.

[9] N. Ng, N. Yoshida, and K. Honda. Multiparty Session C: Safe parallel programming with message optimisation. In *TOOLS*, pages 203–219, 2012.

[10] N. Ng, N. Yoshida, X. Y. Niu, K. H. Tsoi, and W. Luk. Session types: towards safe and fast reconfigurable programming. *SIGARCH Comput. Archit. News*, 40(5):22–27, Mar. 2012.

[11] M. Saldaña et al. MPI as a Programming Model for High-Performance Reconfigurable Computers. *TRETS*, 3(4):1–29, Nov. 2010.

[12] Scribble homepage. `http://www.jboss.org/scribble`.

[13] S. F. Siegel and T. K. Zirkel. Automatic formal verification of MPI-based parallel programs. In *PPoPP'11*, page 309. ACM Press, Feb. 2011.

[14] K. H. Tsoi and W. Luk. Axel: A Heterogeneous Cluster with FPGAs and GPUs. In *FPGA'10*, pages 115–124. ACM, 2010.

[15] A. Vo et al. Formal verification of practical MPI programs. In *PPoPP'09*, pages 261–270, 2009.

[16] A. Vo et al. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *SC'10*, pages 1–10. IEEE, 2010.

[17] Session C homepage. `http://www.doc.ic.ac.uk/~cn06/sessionc/`.

[18] N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised Multiparty Session Types. In *FOSSACS*, volume 6014 of *LNCS*, pages 128–145, 2010.

Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types

Pierre-Malo Deniélou Nobuko Yoshida

Abstract Multiparty session types are a type system that can ensure the safety and liveness of distributed peers via the global specification of their interactions. To construct a global specification from a set of distributed uncontrolled behaviours, this paper explores the problem of fully characterising multiparty session types in terms of communicating automata. We equip global and local session types with labelled transition systems (LTSs) that faithfully represent asynchronous communications through unbounded buffered channels. Using the equivalence between the two LTSs, we identify a class of communicating automata that exactly correspond to the projected local types. We exhibit an algorithm to synthesise a global type from a collection of communicating automata. The key property of our findings is the notion of *multiparty compatibility* which non-trivially extends the duality condition for binary session types.

1 Introduction

Over the last decade, *session types* [12, 18] have been studied as data types or functional types for communications and distributed systems. A recent discovery by [4, 20], which establishes a Curry-Howard isomorphism between binary session types and linear logics, confirms that session types and the notion of duality between type constructs have canonical meanings. Multiparty session types [13, 2] were proposed as a major generalisation of binary session types. They can enforce communication safety and deadlock-freedom for more than two peers thanks to a choreographic specification (called *global type*) of the interaction. Global types are projected to end-point types (*local types*), against which processes can be statically type-checked and verified to behave correctly.

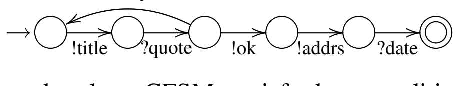
The motivation of this paper comes from our practical experiences that, in many situations, even where we start from the end-point projections of a choreography, we need to reconstruct a global type from distributed specifications. End-point specifications are usually available, either through inference from the control flow, or through existing service interfaces, and always in forms akin to individual communicating finite state machines. If one knows the precise conditions under which a global type can be constructed (i.e. the conditions of *synthesis*), not only the global safety property which multiparty session types ensure is guaranteed, but also the generated global type can be used as a refinement and be integrated within the distributed system development life-cycle (see [17]). This paper attempts to give the synthesis condition as a sound and complete characterisation of multiparty session types with respect to Communicating Finite State Machines (CFSMs) [3]. CFSMs have been a well-studied formalism for analysing distributed safety properties and are widely present in industry tools. They can be seen as generalised end-point specifications, therefore an excellent target for a common comparison ground and for synthesis. As explained below, to identify a complete set of CFSMs for synthesis, we first need to answer a question – *what is the canonical duality notion in multiparty session types?*

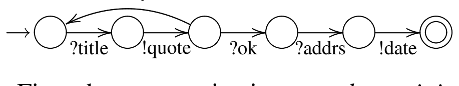
Characterisation of binary session types as communicating automata The subclass which fully characterises *binary session types* was actually proposed by Gouda, Manning and Yu in 1984 [11] in a pure communicating automata context. Consider a simple business protocol between a Buyer and a Seller from the Buyer's viewpoint: Buyer sends the title of a book, Seller answers with a quote. If Buyer is satisfied by the quote, then he sends his address and Seller sends back the delivery date; otherwise it retries the same conversation. This can be described by the following session type:

$$\mu\mathsf{t}.!\,\mathsf{title};\ ?\mathsf{quote};\ !\{\ \mathsf{ok}:!\,\mathsf{addrs};?\mathsf{date};\mathsf{end},\quad \mathsf{retry}:\mathsf{t}\ \} \tag{1.1}$$

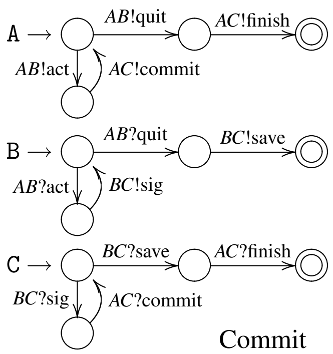
where the operator $!\,\mathsf{title}$ denotes an output of the title, whereas $?\mathsf{quote}$ denotes an input of a quote. The output choice features the two options ok and retry and ; denotes sequencing. end represents the termination of the session, and $\mu\mathsf{t}$ is recursion.

The simplicity and tractability of binary sessions come from the notion of *duality* in interactions [10]. The interaction pattern of the Seller is fully given as the dual of the type in (1.1) (exchanging input ! and output ? in the original type). When composing two parties, we only have to check they have mutually dual types, and the resulting communication is guaranteed to be deadlock-free. Essentially the same characterisation is given in communicating automata. Buyer and Seller's session types are represented by the following two machines.





We can observe that these CFSMs satisfy three conditions. First, the communications are *deterministic*: messages that are part of the same choice, ok and retry here, are distinct. Secondly, there is no mixed state (each state has either only sending actions or only receiving actions). Third, these two machines have *compatible* traces (i.e. dual): the Seller machine can be defined by exchanging sending to receiving actions and vice versa. Breaking one of these conditions allows deadlock situations and breaking one of the first two conditions makes the compatibility checking undecidable [11, 19].



Commit

Multiparty compatibility This notion of duality is no longer effective in multiparty communications, where the whole conversation cannot be reconstructed from only a single behaviour. To bypass the gap between binary and multiparty, we take the *synthesis* approach, that is to find conditions which allow a global choreography to be built from the local machine behaviour. Instead of directly trying to decide whether the communications of a system will satisfy safety (which is undecidable in the general case), inferring a global type guarantees the safety as a direct consequence.
We give a simple example above to illustrate the problem. The Commit protocol involves three machines: Alice A, Bob B and Carol C. A orders B to act or quit. If act is sent, B sends a signal to C, and A sends a commitment to C and continues. Otherwise B informs C to save the data and A gives the final notification to C to terminate the protocol.

This paper presents a decidable notion of *multiparty compatibility* as a generalisation of duality of binary sessions, which in turns characterises a synthesis condition. The idea is to check the duality between each automaton and the rest, up to the internal communications (1-bounded executions in the terminology of CFSMs, see § 2) that the other machines will independently perform. For example, in the Commit example, to check the compatibility of trace AB!quit AC!finish in A, we observe the dual trace AB?quit $\cdot$ AC?finish from B and C executing the internal communications between B and C such that BC!save $\cdot$ BC?save. If this extended duality is valid for all the machines from any 1-bounded reachable state, then they satisfy multiparty compatibility and can build a well-formed global choreography.

Contributions and Outline Section 3 defines new labelled transition systems for global and local types that represent the abstract observable behaviour of typed processes. We prove that a global type behaves exactly as its projected local types, and the same result between a single local type and its CFSMs interpretation. These correspondences are the key to prove the main theorems. Section 4 defines *multiparty compatibility*, studies its safety and liveness properties, gives an algorithm for the synthesis of global types from CFSMs, and proves the soundness and completeness results between global types and CFSMs. Section 5 discusses related work and concludes. The full proofs and applications of this work can be found in [17].

2 Communicating Finite State Machines

This section starts from some preliminary notations (following [6]). ε is the empty word. $\mathbb{A}$ is a finite alphabet and $\mathbb{A}^*$ is the set of all finite words over $\mathbb{A}$. $|x|$ is the length of a word x and $x.y$ or xy the concatenation of two words x and y. Let $\mathcal{P}$ be a set of *participants* fixed throughout the paper: $\mathcal{P} \subseteq \{\mathtt{A},\mathtt{B},\mathtt{C},\dots,\mathtt{p},\mathtt{q},\dots\}$.

Definition 1 (CFSM)**.** A communicating finite state machine is a finite transition system given by a 5-tuple $M = (Q, C, q_0, \mathbb{A}, \delta)$ where (1) Q is a finite set of *states*; (2) $C = \{\mathtt{pq} \in \mathcal{P}^2 \mid \mathtt{p} \neq \mathtt{q}\}$ is a set of channels; (3) $q_0 \in Q$ is an initial state; (4) $\mathbb{A}$ is a finite *alphabet* of messages, and (5) $\delta \subseteq Q \times (C \times \{!,?\} \times \mathbb{A}) \times Q$ is a finite set of *transitions*.

In transitions, $\mathtt{pq}!a$ denotes the *sending* action of a from participant p to participant q, and $\mathtt{pq}?a$ denotes the *receiving* action of a from p by q. ℓ,ℓ' range over actions and we define the *subject* of an action ℓ as the principal in charge of it: $subj(\mathtt{pq}!a) = subj(\mathtt{qp}?a) = \mathtt{p}$.

A state $q \in Q$ whose outgoing transitions are all labelled with sending (resp. receiving) actions is called a *sending* (resp. *receiving*) state. A state $q \in Q$ which does not have any outgoing transition is called *final*. If q has both sending and receiving outgoing transitions, q is called *mixed*. We say q is *directed* if it contains only sending (resp. receiving) actions to (resp. from) the same (identical) participant. A *path* in M is a finite sequence of $q_0,\dots,q_n$ ($n \geq 1$) such that $(q_i,\ell,q_{i+1}) \in \delta$ ($0 \leq i \leq n-1$), and we write $q \xrightarrow{\ell} q'$ if $(q,\ell,q') \in \delta$. M is *connected* if for every state $q \neq q_0$, there is a path from q_0 to q. Hereafter we assume each CFSM is connected.

A CFSM $M = (Q, C, q_0, \mathbb{A}, \delta)$ is *deterministic* if for all states $q \in Q$ and all actions ℓ, $(q,\ell,q'),(q,\ell,q'') \in \delta$

imply $q' = q''$.[1]

Definition 2 (CS)**.** A (communicating) system S is a tuple $S = (M_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}}$ of CFSMs such that $M_{\mathtt{p}} = (Q_{\mathtt{p}}, C, q_{0\mathtt{p}}, \mathbb{A}, \delta_{\mathtt{p}})$.

For $M_{\mathtt{p}} = (Q_{\mathtt{p}}, C, q_{0\mathtt{p}}, \mathbb{A}, \delta_{\mathtt{p}})$, we define a *configuration* of $S = (M_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}}$ to be a tuple $s = (\vec{q};\vec{w})$ where $\vec{q} = (q_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}}$ with $q_{\mathtt{p}} \in Q_{\mathtt{p}}$ and where $\vec{w} = (w_{\mathtt{pq}})_{\mathtt{p}\neq\mathtt{q}\in\mathcal{P}}$ with $w_{\mathtt{pq}} \in \mathbb{A}^*$. The element $\vec{q}$ is called a *control state* and $q \in Q_{\mathtt{p}}$ is the *local state* of machine $M_{\mathtt{p}}$.

Definition 3 (reachable state)**.** Let S be a communicating system. 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 $s \to s'$ or $s \xrightarrow{t} s'$, if there exists $a \in \mathbb{A}$ such that either: (1) $t = (q_{\mathtt{p}}, \mathtt{pq}!a, q'_{\mathtt{p}}) \in \delta_{\mathtt{p}}$ and (a) $q'_{\mathtt{p}'} = q_{\mathtt{p}'}$ for all $\mathtt{p}' \neq \mathtt{p}$; and (b) $w'_{\mathtt{pq}} = w_{\mathtt{pq}}.a$ and $w'_{\mathtt{p}'\mathtt{q}'} = w_{\mathtt{p}'\mathtt{q}'}$ for all $\mathtt{p}'\mathtt{q}' \neq \mathtt{pq}$; or (2) $t = (q_{\mathtt{q}}, \mathtt{pq}?a, q'_{\mathtt{q}}) \in \delta_{\mathtt{q}}$ and (a) $q'_{\mathtt{p}'} = q_{\mathtt{p}'}$ for all $\mathtt{p}' \neq \mathtt{q}$; and (b) $w_{\mathtt{pq}} = a.w'_{\mathtt{pq}}$ and $w'_{\mathtt{p}'\mathtt{q}'} = w_{\mathtt{p}'\mathtt{q}'}$ for all $\mathtt{p}'\mathtt{q}' \neq \mathtt{pq}$.

The condition (1-b) puts the content a to a channel pq, while (2-b) gets the content a from a channel pq. The reflexive and transitive closure of $\to$ is $\to^*$. For a transition $t = (s, \ell, s')$, we refer to ℓ by $act(t)$. We write $s_1 \xrightarrow{t_1\cdots t_m} s_{m+1}$ for $s_1 \xrightarrow{t_1} s_2 \cdots \xrightarrow{t_m} s_{m+1}$ and use φ to denote $t_1 \cdots t_m$. We extend act to these sequences: $act(t_1 \cdots t_n) = act(t_1) \cdots act(t_n)$.

The *initial configuration* of a system is $s_0 = (\vec{q}_0;\vec{\varepsilon})$ with $\vec{q}_0 = (q_{0\mathtt{p}})_{\mathtt{p}\in\mathcal{P}}$. A *final configuration* of the system is $s_f = (\vec{q};\vec{\varepsilon})$ with all $q_{\mathtt{p}} \in \vec{q}$ final. A configuration s is *reachable* if $s_0 \to^* s$ and we define the *reachable set* of S as $RS(S) = \{s \mid s_0 \to^* s\}$. We define the traces of a system S to be $Tr(S) = \{act(\varphi) \mid \exists s \in RS(S), s_0 \xrightarrow{\varphi} s\}$.

We now define several properties about communicating systems and their configurations. These properties will be used in § 4 to characterise the systems that correspond to multiparty session types. Let S be a communicating system, t one of its transitions and $s = (\vec{q};\vec{w})$ one of its configurations. The following definitions of configuration properties follow [6, Definition 12].

1. s is *stable* if all its buffers are empty, i.e., $\vec{w} = \vec{\varepsilon}$.

2. s is a *deadlock configuration* if s is not final, and $\vec{w} = \vec{\varepsilon}$ and each $q_{\mathtt{p}}$ is a receiving state, i.e. all machines are blocked, waiting for messages.

3. s is an *orphan message configuration* if all $q_{\mathtt{p}} \in \vec{q}$ are final but $\vec{w} \neq \emptyset$, i.e. there is at least an orphan message in a buffer.

4. s is an *unspecified reception configuration* if there exists $\mathtt{q} \in \mathcal{P}$ such that $q_{\mathtt{q}}$ is a receiving state and $(q_{\mathtt{q}}, \mathtt{pq}?a, q'_{\mathtt{q}}) \in \delta$ implies that $|w_{\mathtt{pq}}| > 0$ and $w_{\mathtt{pq}} \notin a\mathbb{A}^*$, i.e $q_{\mathtt{q}}$ is prevented from receiving any message from buffer pq.

A sequence of transitions is said to be *k-bounded* if no channel of any intermediate configuration s_i contains more than k messages. We define the k-reachability set of S to be the largest subset $RS_k(S)$ of $RS(S)$ within which each configuration s can be reached by a k-bounded execution from s_0. Note that, given a communicating system S, for every integer k, the set $RS_k(S)$ is finite and computable. We say that a trace φ is n-bound, written $bound(\varphi) = n$, if the number of send actions in φ never exceeds the number of receive actions by n. We then define the equivalences: (1) $S \approx S'$ is $\forall\varphi$, $\varphi \in Tr(S) \Leftrightarrow \varphi \in Tr(S')$; and (2) $S \approx_n S'$ is $\forall\varphi$, $bound(\varphi) \leq n \Rightarrow (\varphi \in Tr(S) \Leftrightarrow \varphi \in Tr(S'))$.

The following key properties will be examined throughout the paper as properties that multiparty session type can enforce. They are undecidable in general CFSMs.

Definition 4 (safety and liveness)**.** (1) A communicating system S is *deadlock-free* (resp. *orphan message-free*, *reception error-free*) if for all $s \in RS(S)$, s is not a deadlock (resp. orphan message, unspecified reception) configuration. (2) S satisfies the *liveness property* if for all $s \in RS(S)$, there exists $s \longrightarrow^* s'$ such that s' is final.

3 Global and local types: the LTSs and translations

This section presents multiparty session types, our main object of study. For the syntax of types, we follow [2] which is the most widely used syntax in the literature. We introduce two labelled transition systems, for local types and for global types, and show the equivalence between local types and communicating automata.

Syntax A *global type*, written $G, G', ..$, describes the whole conversation scenario of a multiparty session as a

[1] "Deterministic" often means the same channel should carry a unique value, i.e. if $(q, c!a, q') \in \delta$ and $(q, c!a', q'') \in \delta$ then $a = a'$ and $q' = q''$. Here we follow a different definition [6] in order to represent branching type constructs.

type signature, and a *local type*, written by $T, T', ..$, type-abstract sessions from each end-point's view. $\mathtt{p}, \mathtt{q}, \dots \in \mathcal{P}$ denote participants (see § 2 for conventions). The syntax of types is given as:

$$\begin{array}{lll} G & ::= & \mathtt{p} \to \mathtt{p}' : \{a_j.G_j\}_{j\in J} \mid \mu\mathtt{t}.G \mid \mathtt{t} \mid \mathsf{end} \\ T & ::= & \mathtt{p}?\{a_i.T_i\}_{i\in I} \mid \mathtt{p}!\{a_i.T_i\}_{i\in I} \mid \mu\mathtt{t}.T \mid \mathtt{t} \mid \mathsf{end} \end{array}$$

$a_j \in \mathbb{A}$ corresponds to the usual message label in session type theory. We omit the mention of the carried types from the syntax in this paper, as we are not directly concerned by typing processes. Global branching type $\mathtt{p} \to \mathtt{p}' : \{a_j.G_j\}_{j\in J}$ states that participant $\mathtt{p}$ can send a message with one of the a_i labels to participant $\mathtt{p}'$ and that interactions described in G_j follow. We require $\mathtt{p} \neq \mathtt{p}'$ to prevent self-sent messages and $a_i \neq a_k$ for all $i \neq k \in J$. Recursive type $\mu\mathtt{t}.G$ is for recursive protocols, assuming that type variables $(\mathtt{t}, \mathtt{t}', \dots)$ are guarded in the standard way, i.e. they only occur under branchings. Type end represents session termination (often omitted). $\mathtt{p} \in G$ means that $\mathtt{p}$ appears in G.

Concerning local types, the *branching type* $\mathtt{p}?\{a_i.T_i\}_{i\in I}$ specifies the reception of a message from $\mathtt{p}$ with a label among the a_i. The *selection type* $\mathtt{p}!\{a_i.T_i\}_{i\in I}$ is its dual. The remaining type constructors are the same as global types. When branching is a singleton, we write $\mathtt{p} \to \mathtt{p}' : a.G'$ for global, and $\mathtt{p}!a.T$ or $\mathtt{p}?a.T$ for local.

Projection The relation between global and local types is formalised by projection. Instead of the restricted original projection [2], we use the extension with the merging operator $\bowtie$ from [7]: it allows each branch of the global type to actually contain different interaction patterns. The *projection of G onto* $\mathtt{p}$ (written $G \upharpoonright \mathtt{p}$) is defined as:

$$\mathtt{p} \to \mathtt{p}' : \{a_j.G_j\}_{j\in J} \upharpoonright \mathtt{q} = \begin{cases} \mathtt{p}!\{a_j.G_j \upharpoonright \mathtt{q}\}_{j\in J} & \mathtt{q} = \mathtt{p} \\ \mathtt{p}?\{a_j.G_j \upharpoonright \mathtt{q}\}_{j\in J} & \mathtt{q} = \mathtt{p}' \\ \sqcup_{j\in J} G_j \upharpoonright \mathtt{q} & \text{otherwise} \end{cases} \qquad (\mu\mathtt{t}.G) \upharpoonright \mathtt{p} = \begin{cases} \mu\mathtt{t}.G \upharpoonright \mathtt{p} & G \upharpoonright \mathtt{p} \neq \mathtt{t} \\ \mathsf{end} & \text{otherwise} \end{cases}$$

$$\mathtt{t} \upharpoonright \mathtt{p} = \mathtt{t} \qquad\qquad \mathsf{end} \upharpoonright \mathtt{p} = \mathsf{end}$$

The mergeability relation $\bowtie$ is the smallest congruence relation over local types such that:

$$\frac{\forall i \in (K \cap J).T_i \bowtie T'_i \quad \forall k \in (K \setminus J), \forall j \in (J \setminus K).a_k \neq a_j}{\mathtt{p}?\{a_k.T_k\}_{k\in K} \bowtie \mathtt{p}?\{a_j.T'_j\}_{j\in J}}$$

When $T_1 \bowtie T_2$ holds, we define the operation $\sqcup$ as a partial commutative operator over two types such that $T \sqcup T = T$ for all types and that:

$$\mathtt{p}?\{a_k.T_k\}_{k\in K} \sqcup \mathtt{p}?\{a_j.T'_j\}_{j\in J} \;=\; \mathtt{p}?(\{a_k.(T_k \sqcup T'_k)\}_{k\in K\cap J} \cup \{a_k.T_k\}_{k\in K\setminus J} \cup \{a_j.T'_j\}_{j\in J\setminus K})$$

and homomorphic for other types (i.e. $\mathcal{C}[T_1] \sqcup \mathcal{C}[T_2] = \mathcal{C}[T_1 \sqcup T_2]$ where $\mathcal{C}$ is a context for local types). We say that G is *well-formed* if for all $\mathtt{p} \in \mathcal{P}$, $G \upharpoonright \mathtt{p}$ is defined.

Example 1 (Commit)**.** The global type for the commit protocol in § 1 is:

$\mu\mathtt{t}.\mathtt{A} \to \mathtt{B} : \{act.\mathtt{B} \to \mathtt{C} : \{sig.\,\mathtt{A} \to \mathtt{C} : commit.\mathtt{t}\},\ quit.\mathtt{B} \to \mathtt{C} : \{save.\mathtt{A} \to \mathtt{C} : finish.\mathsf{end}\}\}$

Then C's local type is: $\mu\mathtt{t}.\mathtt{B}?\{sig.\mathtt{A}?\{commit.\mathtt{t}\},\ save.\mathtt{A}?\{finish.\mathsf{end}\}\}$.

We now present labelled transition relations (LTS) for global and local types and their sound and complete correspondence.

LTS over global types We first designate the observables $(\ell, \ell', ...)$. We choose here to follow the definition of actions for CFSMs where a label ℓ denotes the sending or the reception of a message of label a from $\mathtt{p}$ to $\mathtt{p}'$: $\ell ::= \mathtt{p}\mathtt{p}'!a \mid \mathtt{p}\mathtt{p}'?a$

In order to define an LTS for global types, we need to represent intermediate states in the execution. For this reason, we introduce in the grammar of G the construct $\mathtt{p} \rightsquigarrow \mathtt{p}' : j\ \{a_i.G_i\}_{i\in I}$ to represent the fact that a_j has been sent but not yet received.

Definition 5 (LTS over global types)**.** The relation $G \xrightarrow{\ell} G'$ is defined as ($subj(\ell)$ is defined in § 2):

$$[\text{GR1}] \quad \mathtt{p} \to \mathtt{p}' : \{a_i.G_i\}_{i\in I} \xrightarrow{\mathtt{p}\mathtt{p}'!a_j} \mathtt{p} \rightsquigarrow \mathtt{p}' : j\ \{a_i.G_i\}_{i\in I} \quad (j \in I)$$

$$[\text{GR2}] \quad \mathtt{p} \rightsquigarrow \mathtt{p}' : j\ \{a_i.G_i\}_{i\in I} \xrightarrow{\mathtt{p}\mathtt{p}'?a_j} G_j \qquad [\text{GR3}]\ \frac{G[\mu\mathtt{t}.G/\mathtt{t}] \xrightarrow{\ell} G'}{\mu\mathtt{t}.G \xrightarrow{\ell} G'}$$

$$[\text{GR4}]\ \frac{\forall j \in I \quad G_j \xrightarrow{\ell} G'_j \quad \mathtt{p}, \mathtt{q} \notin subj(\ell)}{\mathtt{p} \to \mathtt{q} : \{a_i.G_i\}_{i\in I} \xrightarrow{\ell} \mathtt{p} \to \mathtt{q} : \{a_i.G'_i\}_{i\in I}} \ [\text{GR5}]\ \frac{G_j \xrightarrow{\ell} G'_j \quad \mathtt{q} \notin subj(\ell) \quad \forall i \in I \setminus j, G'_i = G_i}{\mathtt{p} \rightsquigarrow \mathtt{q} : j\ \{a_i.G_i\}_{i\in I} \xrightarrow{\ell} \mathtt{p} \rightsquigarrow \mathtt{q} : j\ \{a_i.G'_i\}_{i\in I}}$$

[GR1] represents the emission of a message while [GR2] describes the reception of a message. [GR3] governs recursive types. [GR4,5] define the asynchronous semantics of global types, where the syntactic order of messages is enforced only for the participants that are involved. For example, when the participants of two consecutive communications are disjoint, as in: $G_1 = \mathtt{A} \to \mathtt{B} : a.\mathtt{C} \to \mathtt{D} : b.\mathsf{end}$, we can observe the emission (and possibly the reception) of b before the interactions of a (by [GR4]).

A more interesting example is: $G_2 = \mathtt{A} \to \mathtt{B} : a.\mathtt{A} \to \mathtt{C} : b.\mathsf{end}$. We write $\ell_1 = AB!a$, $\ell_2 = AB?a$, $\ell_3 = AC!b$ and $\ell_4 = AC?b$. The LTS allows the following three sequences:

$$\begin{array}{llllllll}
G_2 & \xrightarrow{\ell_1} & \mathtt{A} \rightsquigarrow \mathtt{B} : a.\mathtt{A} \to \mathtt{C} : b.\mathsf{end} & \xrightarrow{\ell_2} & \mathtt{A} \to \mathtt{C} : b.\mathsf{end} & \xrightarrow{\ell_3} & \mathtt{A} \rightsquigarrow \mathtt{C} : b.\mathsf{end} & \xrightarrow{\ell_4} \mathsf{end} \\
G_2 & \xrightarrow{\ell_1} & \mathtt{A} \rightsquigarrow \mathtt{B} : a.\mathtt{A} \to \mathtt{C} : b.\mathsf{end} & \xrightarrow{\ell_3} & \mathtt{A} \rightsquigarrow \mathtt{B} : a.\mathtt{A} \rightsquigarrow \mathtt{C} : b.\mathsf{end} & \xrightarrow{\ell_2} & \mathtt{A} \rightsquigarrow \mathtt{C} : b.\mathsf{end} & \xrightarrow{\ell_4} \mathsf{end} \\
G_2 & \xrightarrow{\ell_1} & \mathtt{A} \rightsquigarrow \mathtt{B} : a.\mathtt{A} \to \mathtt{C} : b.\mathsf{end} & \xrightarrow{\ell_3} & \mathtt{A} \rightsquigarrow \mathtt{B} : a.\mathtt{A} \rightsquigarrow \mathtt{C} : b.\mathsf{end} & \xrightarrow{\ell_4} & \mathtt{A} \rightsquigarrow \mathtt{B} : a.\mathsf{end} & \xrightarrow{\ell_2} \mathsf{end}
\end{array}$$

The last sequence is the most interesting: the sender A has to follow the syntactic order but the receiver C can get the message b before B receives a. The respect of these constraints is enforced by the conditions $\mathtt{p}, \mathtt{q} \notin \mathit{subj}(\ell)$ and $\mathtt{q} \notin \mathit{subj}(\ell)$ in rules [GR4,5].

LTS over local types We define the LTS over local types. This is done in two steps, following the model of CFSMs, where the semantics is given first for individual automata and then extended to communicating systems. We use the same labels (ℓ,ℓ',...) as the ones for CFSMs.

Definition 6 (LTS over local types)**.** The relation $T \xrightarrow{\ell} T'$, for the local type of role p, is defined as:

$$[\mathrm{LR1}]\ \mathtt{q}!\{a_i.T_i\}_{i\in I} \xrightarrow{\mathtt{pq}!a_i} T_i \quad [\mathrm{LR2}]\ \mathtt{q}?\{a_i.T_i\}_{i\in I} \xrightarrow{\mathtt{qp}?a_j} T_j \quad [\mathrm{LR3}]\ \frac{T[\mu\mathtt{t}.T/\mathtt{t}] \xrightarrow{\ell} T'}{\mu\mathtt{t}.T \xrightarrow{\ell} T'}$$

The semantics of a local type follows the intuition that every action of the local type should obey the syntactic order. We define the LTS for collections of local types.

Definition 7 (LTS over collections of local types)**.** A configuration $s = (\vec{T}; \vec{w})$ of a system of local types $\{T_\mathtt{p}\}_{\mathtt{p}\in\mathcal{P}}$ is a pair with $\vec{T} = (T_\mathtt{p})_{\mathtt{p}\in\mathcal{P}}$ and $\vec{w} = (w_{\mathtt{pq}})_{\mathtt{p}\neq\mathtt{q}\in\mathcal{P}}$ with $w_{\mathtt{pq}} \in \mathbb{A}^*$. We then define the transition system for configurations. For a configuration $s_T = (\vec{T}; \vec{w})$, the visible transitions of $s_T \xrightarrow{\ell} s'_T = (\vec{T}'; \vec{w}')$ are defined as: (1) $T_\mathtt{p} \xrightarrow{\mathtt{pq}!a} T'_\mathtt{p}$ and (a) $T'_{\mathtt{p}'} = T_{\mathtt{p}'}$ for all $\mathtt{p}' \neq \mathtt{p}$; and (b) $w'_{\mathtt{pq}} = w_{\mathtt{pq}} \cdot a$ and $w'_{\mathtt{p}'\mathtt{q}'} = w_{\mathtt{p}'\mathtt{q}'}$ for all $\mathtt{p}'\mathtt{q}' \neq \mathtt{pq}$; or (2) $T_\mathtt{q} \xrightarrow{\mathtt{pq}?a} T'_\mathtt{q}$ and (a) $T'_{\mathtt{p}'} = T_{\mathtt{p}'}$ for all $\mathtt{p}' \neq \mathtt{q}$; and (b) $w_{\mathtt{pq}} = a \cdot w'_{\mathtt{pq}}$ and $w'_{\mathtt{p}'\mathtt{q}'} = w_{\mathtt{p}'\mathtt{q}'}$ for all $\mathtt{p}'\mathtt{q}' \neq \mathtt{pq}$.

The semantics of local types is therefore defined over configurations, following the definition of the semantics of CFSMs. $w_{\mathtt{pq}}$ represents the FIFO queue at channel pq. We write $\mathit{Tr}(G)$ to denote the set of the visible traces that can be obtained by reducing G. Similarly for $\mathit{Tr}(T)$ and $\mathit{Tr}(S)$. We extend the trace equivalences $\approx$ and $\approx_n$ in § 2 to global types and configurations of local types.

We now state the soundness and completeness of projection w.r.t. the LTSs.

Theorem 1 (soundness and completeness)**.** [2] *Let G be a global type with participants $\mathcal{P}$ and let $\vec{T} = \{G \upharpoonright \mathtt{p}\}_{\mathtt{p}\in\mathcal{P}}$ be the local types projected from G. Then $G \approx (\vec{T}; \vec{\varepsilon})$.*

Local types and CFSMs Next we show how to algorithmically go from local types to CFSMs and back while preserving the trace semantics. We start by translating local types into CFSMs.

Definition 8 (translation from local types to CFSMs)**.** Write $T' \in T$ if T' occurs in T. Let T_0 be the local type of participant p projected from G. The automaton corresponding to T_0 is $\mathcal{A}(T_0) = (Q, C, q_0, \mathbb{A}, \delta)$ where: (1) $Q = \{T' \mid T' \in T_0,\ T' \neq \mathtt{t}, T' \neq \mu\mathtt{t}.T\}$; (2) $q_0 = T'_0$ with $T_0 = \mu\vec{\mathtt{t}}.T'_0$ and $T'_0 \in Q$; (3) $C = \{\mathtt{pq} \mid \mathtt{p}, \mathtt{q} \in G\}$; (4) $\mathbb{A}$ is the set of $\{a \in G\}$; and (5) δ is defined as:

$$\text{If } T = \mathtt{p}'!\{a_j.T_j\}_{j\in J} \in Q, \text{ then } \begin{cases} (T, (\mathtt{pp}'!a_j), T_j) \in \delta & T_j \neq \mathtt{t} \\ (T, (\mathtt{pp}'!a_j), T') \in \delta & T_j = \mathtt{t},\ \mu\vec{\mathtt{t}}\mathtt{t}.T' \in T_0, T' \in Q \end{cases}$$

$$\text{If } T = \mathtt{p}'?\{a_j.T_j\}_{j\in J} \in Q, \text{ then } \begin{cases} (T, (\mathtt{p}'\mathtt{p}?a_j), T_j) \in \delta & T_j \neq \mathtt{t} \\ (T, (\mathtt{p}'\mathtt{p}?a_j), T') \in \delta & T_j = \mathtt{t},\ \mu\vec{\mathtt{t}}\mathtt{t}.T' \in T_0, T' \in Q \end{cases}$$

[2]The local type abstracts the behaviour of multiparty typed processes as proved in the subject reduction theorem in [13]. Hence this theorem implies that processes typed by global type G by the typing system in [13, 2] follow the LTS of G.

The definition says that the set of states Q are the suboccurrences of branching or selection or end in the local type; the initial state q_0 is the occurrence of (the recursion body of) T_0; the channels and alphabets correspond to those in T_0; and the transition is defined from the state T to its body T_j with the action $\mathtt{pp}'!a_j$ for the output and $\mathtt{pp}'?a_j$ for the input. If T_j is a recursive type variable $\mathtt{t}$, it points the state of the body of the corresponding recursive type. As an example, see C's local type in Example 1 and its corresponding automaton in § 1.

Proposition 1 (local types to CFSMs)**.** *Assume $T_\mathtt{p}$ is a local type. Then $\mathcal{A}(T_\mathtt{p})$ is deterministic, directed and has no mixed states.*

We say that a CFSM is *basic* if it is deterministic, directed and has no mixed states. Any basic CFSM can be translated into a local type.

Definition 9 (translation from a basic CFSM to a local type)**.** From a basic $M_\mathtt{p} = (Q, C, q_0, \mathbb{A}, \delta)$, we define the translation $\mathcal{T}(M_\mathtt{p})$ such that $\mathcal{T}(M_\mathtt{p}) = \mathcal{T}_\varepsilon(q_0)$ where $\mathcal{T}_{\tilde{q}}(q)$ is defined as:

(1) $\mathcal{T}_{\tilde{q}}(q) = \mu \mathtt{t}_q.\mathtt{p}'!\{a_j.\mathcal{T}^{\circ}_{\tilde{q}\cdot q}(q_j)\}_{j\in J}$ if $(q, \mathtt{pp}'!a_j, q_j) \in \delta$;

(2) $\mathcal{T}_{\tilde{q}}(q) = \mu \mathtt{t}_q.\mathtt{p}'?\{a_j.\mathcal{T}^{\circ}_{\tilde{q}\cdot q}(q_j)\}_{j\in J}$ if $(q, \mathtt{p}'\mathtt{p}?a_j, q_j) \in \delta$;

(3) $\mathcal{T}^{\circ}_{\tilde{q}}(q) = \mathcal{T}_\varepsilon(q) = \mathsf{end}$ if q is final; (4) $\mathcal{T}^{\circ}_{\tilde{q}}(q) = \mathtt{t}_{q_k}$ if $(q, \ell, q_k) \in \delta$ and $q_k \in \tilde{q}$; and

(5) $\mathcal{T}^{\circ}_{\tilde{q}}(q) = \mathcal{T}_{\tilde{q}}(q)$ otherwise.

Finally, we replace $\mu \mathtt{t}.T$ by T if $\mathtt{t}$ is not in T.

In $\mathcal{T}_{\tilde{q}}$, $\tilde{q}$ records visited states; (1,2) translate the receiving and sending states to branching and selection types, respectively; (3) translates the final state to end; and (4) is the case of a recursion: since q_k was visited, ℓ is dropped and replaced by the type variable.

The following proposition states that these translations preserve the semantics.

Proposition 2 (translations between CFSMs and local types)**.** *If a CFSM M is basic, then $M \approx \mathcal{T}(M)$. If T is a local type, then $T \approx \mathcal{A}(T)$.*

4 Completeness and synthesis

This section studies the synthesis and sound and complete characterisation of multiparty session types as communicating automata. A first idea would be to restrict basic CFSMs to the natural generalisation of half-duplex systems [6, § 4.1.1], in which each pair of machines linked by two channels, one in each direction, communicates in a half-duplex way. In this class, the safety properties of Definition 4 are however undecidable [6, Theorem 36]. We therefore need a stronger (and decidable) property to force basic CFSMs to behave as if they were the result of a projection from global types.

Multiparty compatibility In the two machines case, there exists a sound and complete condition called *compatible* [11]. Let us define the isomorphism $\Phi : (C \times \{!,?\} \times \mathbb{A})^* \longrightarrow (C \times \{!,?\} \times \mathbb{A})^*$ such that $\Phi(j?a) = j!a$, $\Phi(j!a) = j?a$, $\Phi(\varepsilon) = \varepsilon$, $\Phi(t_1 \cdots t_n) = \Phi(t_1) \cdots \Phi(t_n)$. Φ exchanges a sending action with the corresponding receiving one and vice versa. The compatibility of two machines can be immediately defined as $Tr(M_1) = \Phi(Tr(M_2))$ (i.e. the traces of M_1 are exactly the set of dual traces of M_2). The idea of the extension to the multiparty case comes from the observation that from the viewpoint of the participant $\mathtt{p}$, the rest of all the machines $(M_\mathtt{q})_{\mathtt{q}\in\mathcal{P}\setminus\mathtt{p}}$ should behave as if they were one CFSM which offers compatible traces $\Phi(Tr(M_\mathtt{p}))$, up to internal synchronisations (i.e. 1-bounded executions). Below we define a way to group CFSMs.

Definition 10 (Definition 37, [6])**.** Let $M_i = (Q_i, C_i, q_{0i}, \mathbb{A}_i, \delta_i)$. The *associated CFSM* of a system $S = (M_1, .., M_n)$ is $M = (Q, C, q_0, \mathbb{A}, \delta)$ such that: $Q = Q_1 \times Q_2 \times \cdots \times Q_n$, $q_0 = (q_{01}, \ldots, q_{0n})$; $C = \cup_i C_i$; $\mathbb{A} = \cup_i \mathbb{A}_i$; and δ is the smallest relation for which: if $(q_i, \ell, q_i') \in \delta_i$ $(1 \leq i \leq n)$, then $((q_1, ..., q_i, ..., q_n), \ell, (q_1, ..., q_i', ..., q_n)) \in \delta$.

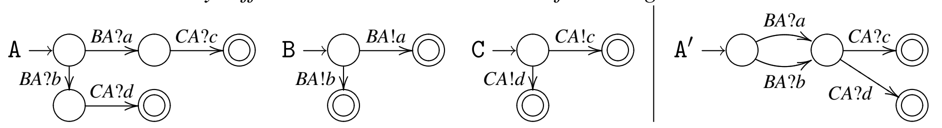
We now define a notion of compatibility extended to more than two CFSMs. We say that φ is an *alternation* if φ is an alternation of sending and corresponding receive actions (i.e. the action $\mathtt{pq}!a$ is immediately followed by $\mathtt{pq}?a$).

Definition 11 (multiparty compatible system)**.** A system $S = (M_1,..,M_n)$ $(n \geq 2)$ is *multiparty compatible* if for any 1-bounded reachable stable state $s \in RS_1(S)$, for any sending action ℓ and for at least one receiving action ℓ from s in M_i, there exists a sequence of transitions $\varphi \cdot t$ from s in a CFSM corresponding to $S^{-i} = (M_1,\ldots,M_{i-1},M_{i+1},\ldots,M_n)$ where φ is either empty or an alternation and $\ell = \Phi(act(t))$ and $i \notin act(\varphi)$ (i.e. φ does not contain actions to or from channel i).

The above definition states that for each M_i, the rest of machines S^{-i} can produce the compatible (dual) actions by executing alternations in S^{-i}. From M_i, these intermediate alternations can be seen as non-observable internal actions.

Example 2 (multiparty compatibility)**.** As an example, we can test the multiparty compatibility property on the commit example in § 1.We only detail here how to check the compatibility from the point of view of `A`. To check the compatibility for the actions $act(t_1 \cdot t_2) = AB!\text{quit} \cdot AC!\text{finish}$, the only possible action is $\Phi(act(t_1)) = AB?\text{quit}$ from `B`, then a 1-bounded execution is $BC!\text{save} \cdot BC?\text{save}$, and $\Phi(act(t_2)) = AC?\text{finish}$ from `C`. To check the compatibility for the actions $act(t_3 \cdot t_4) = AB!\text{act} \cdot AC!\text{commit}$, $\Phi(act(t_3)) = AB?\text{act}$ from `B`, the 1-bound execution is $BC!\text{sig} \cdot BC?\text{sig}$, and $\Phi(act(t_4)) = AC?\text{commit}$ from `C`.

Remark 1. *In Definition 11, we check the compatibility from any 1-bounded reachable stable state in the case one branch is selected by different senders. Consider the following machines:*



In `A`, `B` *and* `C`*, each action in each machine has its dual but they do not satisfy multiparty compatibility. For example, if $BA!a \cdot BA?a$ is executed, $CA!d$ does not have a dual action (hence they do not satisfy the safety properties). On the other hand, the machines* `A′`, `B` *and* `C` *satisfy the multiparty compatibility.*

Theorem 2. *Assume $S = (M_\mathtt{p})_{\mathtt{p}\in\mathcal{P}}$ is basic and multiparty compatible. Then S satisfies the three safety properties in Definition 4. Further, if there exists at least one $M_\mathtt{q}$ which includes a final state, then S satisfies the liveness property.*

Proposition 3. *If all the CFSMs $M_\mathtt{p}$ ($\mathtt{p} \in \mathcal{P}$) are basic, there is an algorithm to check whether $(M_\mathtt{p})_{\mathtt{p}\in\mathcal{P}}$ is multiparty compatible.*

The proof of Theorem 2 is non-trivial, using a detailed analysis of causal relations. The proof of Proposition 3 comes from the finiteness of $RS_1(S)$. See [17] for details.

Synthesis Below we state the lemma which will be crucial for the proof of synthesis and completeness. The lemma comes from the intuition that the transitions of multiparty compatible systems are always permutations of one-bounded executions as it is the case in multiparty session types. See [17] for the proof.

Lemma 1 (1-buffer equivalence)**.** *Suppose S_1 and S_2 are two basic and multiparty compatible communicating systems such that $S_1 \approx_1 S_2$, then $S_1 \approx S_2$.*

Theorem 3 (synthesis)**.** *Suppose S is a basic system and multiparty compatible. Then there is an algorithm which successfully builds well-formed G such that $S \approx G$ if such G exists, and otherwise terminates.*

Proof. We assume $S = (M_\mathtt{p})_{\mathtt{p}\in\mathcal{P}}$. The algorithm starts from the initial states of all machines $(q^{\mathtt{p}_1}{}_0,\ldots,q^{\mathtt{p}_n}{}_0)$. We take a pair of the initial states which is a sending state $q_0^\mathtt{p}$ and a receiving state $q_0^\mathtt{q}$ from p to q. We note that by directness, if there are more than two pairs, the participants in two pairs are disjoint, and by [G4] in Definition 5, the order does not matter. We apply the algorithm with the invariant that all buffers are empty and that we repeatedly pick up one pair such that $q_\mathtt{p}$ (sending state) and $q_\mathtt{q}$ (receiving state). We define $G(q_1,\ldots,q_n)$ where $(q_\mathtt{p},q_\mathtt{q} \in \{q_1,\ldots,q_n\})$ as follows:

- if $(q_1,\ldots,q_n)$ has already been examined and if all participants have been involved since then (or the ones that have not are in their final state), we set $G(q_1,\ldots,q_n)$ to be $\mathtt{t}_{q_1,\ldots,q_n}$. Otherwise, we select a pair sender/receiver from two participants that have not been involved (and are not final) and go to the next step;
- otherwise, in $q_\mathtt{p}$, from machine p, we know that all the transitions are sending actions towards p′ (by directedness), i.e. of the form $(q_\mathtt{p}, \mathtt{pq}!a_i, q_i) \in \delta_\mathtt{p}$ for $i \in I$.

    – we check that machine $\mathtt{q}$ is in a receiving state $q_\mathtt{q}$ such that $(q_\mathtt{q}, \mathtt{pq}?a_j, q'_j) \in \delta_{\mathtt{p}'}$ with $j \in J$ and $I \subseteq J$.
    – we set $\mu \mathtt{t}_{q_1,...,q_n}.\mathtt{p} \to \mathtt{q}\colon \{a_i.G(q_1,...,q_\mathtt{p} \leftarrow q_i,...,q_\mathtt{q} \leftarrow q'_i,...,q_n)\}_{i\in I}$ (we replace $q_\mathtt{p}$ and $q_\mathtt{q}$ by q_i and q'_i, respectively) and continue by recursive calls.
    – if all sending states in $q_1,...,q_n$ become final, then we set $G(q_1,...,q_n) = \mathsf{end}$.

- we erase unnecessary $\mu \mathtt{t}$ if $\mathtt{t} \notin G$.

Since the algorithm only explores 1-bounded executions, the reconstructed G satisfies $G \approx_1 S$. By Theorem 1, we know that $G \approx (\{G \upharpoonright \mathtt{p}\}_{\mathtt{p}\in\mathcal{P}}; \vec{\varepsilon})$. Hence, by Proposition 2, we have $G \approx S'$ where S' is the communicating system translated from the projected local types $\{G \upharpoonright \mathtt{p}\}_{\mathtt{p}\in\mathcal{P}}$ of G. By Lemma 1, $S \approx S'$ and therefore $S \approx G$. □ □

The algorithm can generate the global type in Example 1 from CFSMs in § 1and the global type $\mathtt{B} \to \mathtt{A}\{a : \mathtt{C} \to \mathtt{A} : \{c : \mathsf{end}, d : \mathsf{end}\}, b : \mathtt{C} \to \mathtt{A} : \{c : \mathsf{end}, d : \mathsf{end}\}\}$ from $\mathtt{A}'$, $\mathtt{B}$ and $\mathtt{C}$ in Remark 1. Note that $\mathtt{B} \to \mathtt{A}\{a : \mathtt{C} \to \mathtt{A} : \{c : \mathsf{end}\}, b : \mathtt{C} \to \mathtt{A} : \{d : \mathsf{end}\}\}$ generated by $\mathtt{A}$, $\mathtt{B}$ and $\mathtt{C}$ in Remark 1 is not projectable, hence not well-formed.

With Theorems 1 and 2, and Proposition 2, we can now conclude:

Theorem 4 (soundness and completeness)**.** *Suppose S is basic and multiparty compatible. Then there exists G such that $S \approx G$. Conversely, if G is well-formed, then there exists a basic and multiparty compatible system S which satisfies the three safety properties in Definition 4 such that $S \approx G$.*

5 Conclusion and related work

This paper investigated the sound and complete characterisation of multiparty session types into CFSMs and developed a decidable synthesis algorithm from basic CFSMs. The main tool we used is a new extension to multiparty interactions of the duality condition for binary session types, called *multiparty compatibility*. The basic condition (coming from binary session types) and the multiparty compatibility property are a *necessary and sufficient condition* to obtain safe global types. Our aim is to offer a duality notion which would be applicable to extend other theoretical foundations such as the Curry-Howard correspondence with linear logics [4, 20] to multiparty communications. Basic multiparty compatible CFSMs also define one of the few non-trivial decidable subclass of CFSMs which satisfy deadlock-freedom. The methods proposed here are palatable to a wide range of applications based on choreography protocol models and more widely, finite state machines. Multiparty compatibility is applicable for extending the synthesis algorithm to build more expressive graph-based global types (*general global types* [8]) which feature fork and join primitives [9].

Our previous work [8] presented the first translation from global and local types into CFSMs. It only analysed the properties of the automata resulting from such a translation. The complete characterisation of global types independently from the projected local types was left open, as was synthesis. This present paper closes this open problem. There are a large number of paper that can be found in the literature about the synthesis of CFSMs. See [16] for a summary of recent results. The main distinction with CFSM synthesis is, apart from the formal setting (i.e. types), about the kind of the target specifications to be generated (global types in our case). Not only our synthesis is concerned about trace properties (languages) like the standard synthesis of CFSMs (the problem of the closed synthesis of CFSMs is usually defined as the construction from a regular language L of a machine satisfying certain conditions related to buffer boundedness, deadlock-freedom and words swapping), but we also generate concrete syntax or choreography descriptions as *types* of programs or software. Hence they are directly applicable to programming languages and can be straightforwardly integrated into the existing frameworks that are based on session types.

Within the context of multiparty session types, [15] first studied the reconstruction of a global type from its projected local types up to asynchronous subtyping and [14] recently offers a typing system to synthesise global types from local types. Our synthesis based on CFSMs is more general since CFSMs do not depend on the syntax. For example, [15, 14] cannot treat the synthesis for $\mathtt{A}'$, $\mathtt{B}$ and $\mathtt{C}$ in Remark 1. These works also do not study the completeness (i.e. they build a global type from a set of projected local types (up to subtyping), and do not investigate necessary and sufficient conditions to build a well-formed global type). A difficulty of the completeness result is that it is generally unknown if the global type constructed by the synthesis can simulate executions with arbitrary buffer bounds since the synthesis only directly looks at 1-bounded executions. In this paper, we proved Lemma 1 and bridged this gap towards the complete characterisation. Recent work by [5, 1] focus on proving the semantic correspondence between global and local descriptions (see [8] for more detailed comparison), but no synthesis algorithm is studied.

References

[1] S. Basu, T. Bultan, and M. Ouederni. Deciding choreography realizability. In *POPL'12*, pages 191–202. ACM, 2012.

[2] L. Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008.

[3] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30:323–342, April 1983.

[4] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[5] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multi-party session. *LMCS*, 8(1), 2012.

[6] G. Cécé and A. Finkel. Verification of programs with half-duplex communication. *Inf. Comput.*, 202(2):166–190, 2005.

[7] P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446. ACM, 2011. Full version, Prototype at `http://www.doc.ic.ac.uk/~pmalo/dynamic`.

[8] P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.

[9] http://arxiv.org/abs/1304.1902.

[10] J.-Y. Girard. Linear logic. *TCS*, 50, 1987.

[11] M. Gouda, E. Manning, and Y. Yu. On the progress of communication between two finite state machines. *Information and Control.*, 63:200–216, 1984.

[12] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[13] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

[14] J. Lange and E. Tuosto. Synthesising choreographies from local session types. In *CONCUR*, volume 7454 of *LNCS*, pages 225–239. Springer, 2012.

[15] D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP'09*, volume 5502 of *LNCS*, pages 316–332. Springer, 2009.

[16] A. Muscholl. Analysis of communicating automata. In *LATA*, volume 6031 of *LNCS*, pages 50–57. Springer, 2010.

[17] DoC Technical Report, Imperial College London, Computing, DTR13-5, 2013.

[18] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[19] J. Villard. *Heaps and Hops*. PhD thesis, ENS Cachan, 2011.

[20] P. Wadler. Proposition as Sessions. In *ICFP'12*, pages 273–286, 2012.

Towards static deadlock resolution in the π-calculus

Marco Giunti and António Ravara

CITI and DI-FCT, Universidade Nova de Lisboa, Portugal

Background. Session types allow a concise description of protocols by detailing the sequence of messages involved in each particular run of the protocol [10, 11, 14]. Static analysis techniques based on session types discern concurrent programs that are *type-safe*, i. e., well-typed programs cannot go wrong in the sense that they do not reach neither the usual data errors nor *communication* errors, as those generated by two parallel programs waiting for an in input on the same session channel, or sending in output on the same session channel.[1]

A drawback of most of these systems is accepting processes that exhibit various forms of deadlocks — although they guarantee type safety, they do not guarantee deadlock-freedom. For that aim, several proposals appeared, guaranteeing progress by inspecting causality dependencies in the processes [2, 3, 15]. Not surprisingly, these systems reduce the set of typed processes, namely rejecting (as usual in static analysis, which is not complete) deadlock-free processes.

Aim of this work. Distributed programming is known to be very hard and one makes mistakes by not taking into consideration all possible executions of the code. Therefore, to assist in the software developing process, instead of simply rejecting a process that may contain a *resource self-holding deadlock* (input and output on the same channel occur in sequence in a given thread, an instance of *Wait For* deadlocks [4, 12]), we devise an algorithm that produces a "fix" for this kind of deadlocked processes.

This situation is easy to spot in a simple process, but it is not so obvious when the two co-actions (input and output) occur far away from each other in the code, or end up in the same thread due to reduction. Assisting the programmer in finding and solving these errors may lead to spare time when debugging.

Contribution. Herein, we propose a first step towards a compromise solution to the identified drawback: rather than require stronger conditions for the analysis and type *less* processes, we devise a procedure that detects synchronisation errors leading to self-holding deadlocked processes, while automatically generating type-safe, deadlock-free, code that faithfully represents the intended protocol of the original process, as described by the session types. The mechanism crucially relies on the help of types to infer this kind of errors. Following the behaviour of a process as specified by a session type environment, the algorithm uses a process typing and transformation function that puts in parallel threaded sequences of input/output in the same channels, releasing deadlocks.

[1] The interested reader may have look at a recent overview [5].

The framework is the standard π-calculus (supporting session delegation as the π-calculus communication) equipped with the Giunti and Vasconcelos session type system [8, 9]; other type-disciplines based on session [6, 11] and linear [13] types can be embedded in the framework.

Further information. This procedure may help the software development process: the typing algorithm detects a deadlock, but instead of rejecting the code, fixes it by looking into the session types and producing new safe code that obeys the protocols and is deadlock-free. The synthetised code can be submitted to the programmer that decides if the "fix makes sense".

We implemented the algorithm (in *Standard ML*), and analysed several examples [1]. This work was presented at the Symposium on Trustworthy Global Computing (TGC'13), being part of the post-proceedings [7].

References

1. LOCKRES: a deadlock resolver for the pi calculus, standard ML of New Jersey, http://tinyurl.com/mg-source
2. Bettini, L., et al.: Global progress in dynamically interleaved multiparty sessions. In: CONCUR. LNCS, vol. 5201, pp. 418–433 (2008)
3. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR. LNCS, vol. 6269, pp. 222–236. Springer (2010)
4. Coffman, E.G., Elphick, M., Shoshani, A.: System deadlocks. ACM Computing Surveys 3(2), 67–78 (1971)
5. Dezani-Ciancaglini, M., de'Liguoro, U.: Sessions and session types: An overview. In: WS-FM. LNCS, vol. 6194, pp. 1–28. Springer (2009)
6. Gay, S.J., Hole, M.J.: Subtyping for session types in the pi calculus. Acta Informatica 42(2/3), 191–225 (2005)
7. Giunti, M., Ravara, A.: Towards static deadlock resolution in the pi calculus. In: TGC. LNCS, Springer, to appear
8. Giunti, M., Vasconcelos, V.T.: A linear account of session types in the pi calculus. In: CONCUR. LNCS, vol. 6269, pp. 432–446. Springer (2010)
9. Giunti, M., Vasconcelos, V.T.: Linearity, session types and the pi calculus. Mathematical Structures in Computer Science (2013), in press
10. Honda, K.: Types for dyadic interaction. In: CONCUR. LNCS, vol. 715, pp. 509–523. Springer (1993)
11. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. LNCS, vol. 1381, pp. 122–138. Springer (1998)
12. Knapp, E.: Deadlock detection in distributed databases. ACM Computing Surveys 19(4), 303–328 (1987)
13. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Transactions on Programming Languages and Systems 21(5), 914–947 (1999)
14. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE. LNCS, vol. 817, pp. 398–413. Springer (1994)
15. Wadler, P.: Propositions as sessions. In: ICFP. pp. 273–286 (2012)

Distributed Governance with Scribble

Raymond Hu, Rumyana Neykova and Nobuko Yoshida

Imperial College London

Abstract

Scribble is a multiparty session-based specification language for modelling application-level protocols. Scribble protocols enjoy safety and liveness properties such as deadlock-freedom and progress between distributed endpoints. To use our language for describing choreographic communications in a large cyberinfrastucture for the oceanography, we have been working with an NSF project, Ocean Observatories Initiative over four years. Our aim is to identify a major class of communication patterns via their use cases, and specify them in Scribble. We demonstrate our experiences that Scribble enables to easily specify industrial-scale protocols, ensuring essential properties for distributed governance in distributed systems.

Note: The part of this paper is based on Kohei Honda's last paper, *Structuring Communication with Session Types*, which will appear in the proceeding of *Concurrent Objects and Beyond*. In the Beat workshop, we plan to give a tutorial of the updated Scribble and demonstrations how we have integrated our tool to the OOI governance architectures.

1 Writing Protocols in Scribble

Session types describe a way, or a pattern, in which interactions can take place in sessions. Session types have been called *protocols* for many years in network and other engineering disciplines which need to treat such patterns. For this reason, and because session types are sufficiently different in nature from data types, we know in sequential computing (although the former share the key principle from data types as we shall discuss later), hereafter we often use the term "protocols" instead of "session types" when discussing their use for programming.

One of the key ingredients of session-based programming is the use of protocols as an essential element of design and programming, because a clear understanding of an interaction scenario is an essential ingredient of communications programming. For this reason, one of the key features of programming with sessions is a *protocol description language*, the language with which engineers read and write their protocols. They are close to types in sequential programming: like data and function types, there is a tight linkage to language primitive. Like data and function types, protocols may be inferred from programs or declared by programmers so that programs may be checked against them. A difference is that a protocol describes interactions for a session, and that, for this reason, each session involves a sequence of interactions (which may not necessarily be contiguous, since interactions in other sessions or internal computation may interleave).

A Simple Protocol. To illustrate how we can specify a protocol, we take a simple scenario, and show how the corresponding protocol can be specified using an experimental protocol description language we are developing, called Scribble [7, 11, 10] (the name comes from our desire to create an effective tool for architects, designers and developers alike to quickly and accurately write down protocols).

A key feature of Scribble is that all of its constructs are fully founded on the formal theory of multiparty session types, starting from the core language features for message passing, choice and recursion [8, 1], to more advanced features, such as parallel [5], interrupts [2], sub-sessions [4] and run-time monitoring [3], and studies relating session types to alternatives such as communicating automata [5]. The development of Scribble is a collaboration between researchers and industry partners [10, 9]. Most of the examples presented in this section are supported

```
1  type <ysd> "ListingFormat"
       from "ListingFormat.ysd" as lf;
2
3  protocol ListResources(role client as cl,
       role resource_registry as rr) {
4     request(resource:String) from cl to rr;
5     rec loop {
6        choice at rr {
7           response(element:lf) from rr
                to cl;
8           continue loop;
9        } or {
10          completed() from rr to cl;
11 }}}
```

Figure 1: List Resources protocol

```
protocol ListResources
  <type ListingFormat as lf>
(role client as cl,
 role resource_registry as rr) {

   request(resource:String) from cl to rr;
   rec loop {
      choice at rr {
         response(element:lf) from rr
              to cl;
         continue loop;
      } or {
         completed() from rr to cl;
}}}
```

Figure 2: A refined List Resources protocol (1)

by the current working version of Scribble [11], with a few exceptions that we note as being planned for future release.

The initial scenario we treat is called "List Resources", where a Client obtains a list of resources of some kind from a Resource Registry. This is a basic use case applicable to many environments where a user may be provided with a variety of resources by the infrastructure, e.g. remotely operable instruments or systems resources such as bandwidth. The scenario consists of two steps:

Step 1: Client asks Registry to send her a resource list, specifying the kind of resources it is interested in.

Step 2: Registry responds by sending the list of the resources of the kind specified, until the list is exhausted.

It is a simple elaboration of a remote procedural call. Note, however, that Step 2 involves a repetition of sending actions. This use case may be further elaborated in various ways, but this simple version is sufficient for our first exercise.

Writing down a protocol goes through a natural flow, practised for decades in the networking community. We first list the *message formats*, followed by the participating *actors* (and other parameters). Then we scribble away the structure of the *conversation* between the actors. The result for our mini use case is given in Figure 1.

Line 1 starts from importing an message type `ListingFormat`, specified in YAML (`ysd`), from the external source (file) `ListingFormat.ysd`. This message type can then be referred to in this Scribble protocol specification by the given alias `lf`. Message type imports allow Scribble to be used in conjunction and orthogonally with externally defined message formats: here we are using a YAML schema, but any data format given in a well-defined schema/type language may be used as far as the protocol validator is notified. Data format is of course fundamental in protocols to ensure interacting parties understand what the other is saying.

In Line 3, we give the name to the protocol, `ListResources`, followed by its parameters. The parameters consist of the names of the two actors roles which participants can play, `client` and `resource_registry`, aliased as `cl` and `rr` (short names are often good for scribbling away protocols). This completes the header of the protocol.

The remaining lines (Lines 4–11) constitute the *protocol body*, which describes the structured flow of the conversation in a session. Line 4 is reminiscent of a method/function declaration found in APIs and modules of high-level sequential programming languages: an interaction signature is a symmetric, peer-to-peer version of the familiar notion of "interface" of functions and objects. As such, Line 4 does *not* specify constraints on *concrete values* a message may carry, but specifies only the *type* of an interaction. For this reason, we call the description in

Line 4 as a whole, an *interaction signature*. Line 5 declares the *recursion label* `loop` that names the *recursion body* starting from Line 6 and reaching Line 11. The recursion body consists of a single *choice* statement.

Lines 7–8 and Line 10 are respectively two distinct *branch*es of the choice, separated by `or` on Line 9. In the first branch, Line 7 says that Registry sends a `response` message to Client, with message content annotated as (list) `element` and typed as `Format`. Again we specify only a sender, a receiver and a message signature. This is followed by Line 8, a *recurrence* denoted by the `continue` keyword, which says that the protocol flow at this point returns to the start of the recursion body labelled by `loop`, i.e. to Line 5.

The other branch consists of a single interaction, Line 10, where a `completed` message with an empty payload is sent from Registry to Client, indicating the end of the list, i.e. the end of the recursion – since there is no recurrence, the loop terminates if this branch is chosen. As described in Step 2 above, at the level of the application logic, the repetition should terminate only when all the resource data for the specified kind has been sent by Registry: our protocol description again abstracts from exactly how this may be determined in the program logic (although the protocol assertions we discuss later can constrain this behaviour in some way or another). After this action, the flow exits the choice and the recursion, and (since no further interactions are specified) the session terminates.

Elaborating Protocols. For protocols to assist computer software development, be it a newly built system or an upgrade of an existing system, they had better be *reusable*, i.e. once you author a protocol, it should be able to be used for many concrete applications. From this viewpoint, the `ListResources` protocol in Figure 1 may not be fully satisfactory. In particular, it works only for the message type defined in the specification by the concrete `ListingFormat` YAML schema. Even if only one listing format is known now, new formats may arise later. Why should we write different protocols for all different formats, given the structure of interactions is identical? We use a basic technique from programming theory, *parametrisation*, to solve the problem.

There are several different, and natural, ways we may employ parametericity in the protocol of this example. The approach, supported in the current version of Scribble, is given in Figure 2. Here, we directly abstract the message type as a parameter to the protocol. In Line 1, the protocol has gotten an additional parameter, `<type ListingFormat>`, as well as dispensing with the "import" statement. This additional parameter means, with the keyword `<type>`, that `ListingFormat` (again aliased as `Format`) is now a type name to be instantiated each time this protocol is instantiated as a whole into a run-time session. Later, in the `response` interaction in Line 9, Registry is obliged to send the list elements according to the concrete type known at run-time, while the Client should be ready to receive them. The protocol again gives a contract among participants, while now flexibly catering for arbitrary data formats.

Nested protocols. Consider the protocol given in Figure 3. It has two actors, a Requester and an Authority. In Lines 3, Requester sends a `check` message to query on whether a subject is permitted to do an operation on a resource, carrying the identities of a subject and a resource, the name of an operation, and the certificate of Requester (for authentication, possibly validated via a separate protocol) in its payload. In Lines 4–10, Authority responds, saying the operation is allowed or not, or else by saying `other`, to deal with cases when the answer cannot be delivered for some reason, such as an unqualified Requester.

Now consider the following elaboration of our original "List Resources" use case:

Step 1: Client asks Resource Registry to send a resource list (as before).

Step 2: Registry checks if Client has sufficient privileges.

Step 3: If everything is fine, the Registry replies by a sequence of data for resources of the specified kind to Client.

This use case incorporates a privilege check as part of the protocol, as an extension to the original use case. Note this use case composes two previous use cases, by nesting a protocol

```
protocol CheckPrivileges(
    role requester as req,
    role authority as au) {
  check(subject:URI, resource:URI,
     operation:String, certificate:String)
         from req to au;
  choice at au {
    allowed() from au to req;
  } or {
    not_allowed(reason:String) from au
        to req;
  } or {
    other(reason:String) from au to req;
  }}
```

Figure 3: Check Privileges protocol

```
protocol RequestResponse(role Client as cl,
    role Server as sr) {
  choice at cl {
    GET() from cl to sr;
    choice at sr {
      sc200(s:String) from sr to cl;
    } or {
      sc500(reason:String) from sr to cl;
      ...
  } or {
    POST() from cl to sr;
    ...
  ...
}
```

Figure 4: A HTTP-like protocol extract

inside another protocol. Can we realise such composite use cases as a protocol?

In Figure 5, we show how such a composition is done in Scribble, by combining the previously specified `CheckPrivileges` and `ListResources` (the Figure 1 version). In Line 6, we use the `introduces` keyword to indicate that Registry will "introduce" a new actor, `authority`. After this preparation, the `CheckPrivileges` protocol is launched (`spawn`) by Registry (`at rr`) in Line 7. Note the arguments include Authority which has just been introduced, as well as Registry (who will play the `requester` role in the spawned session). We call the nested `CheckPrivileges` session spawned during the execution of the `ListResources` protocol a *child session*, or a *sub-session*, of the *parent* `ListResources` session. The lifetime of a child session is, in the standard run-time semantics [4], dependent on its parent (e.g. if a parent session aborts, its child session(s) should also abort). Where such causal dependency is not desired, these unrelated protocols may well be specified separately, to be instantiated into distinct sessions at run-time.

Returning to Figure 5, after the `CheckPrivileges` sub-session is carried out, Registry, now knowing the qualification of Client for this query, responds to Client with either an `ok` or an `error` message with the reason (a `String` payload). When `ok`, the remainder of the protocol is the same as in Figure 1 (and also Figures 2). Note that the result of running `CheckPrivileges` is likely to be related to whether `ok` or `error` is selected at the application logic but, at this type level, we do not specify such detailed constraints.

2 Writing Programs with Sessions

We next take a brief look at how we can use the proposed concept of protocols and sessions to implement clear and understandable communication programs, taking a Python implementation of the List Resources protocol from Figure 1 as an example. We cannot give a full implementation in its entirety here, but we hope the reader can get the flavour.

Preliminaries. Our session-oriented programs are constructed using "socket" abstractions that can be seen as standard TCP sockets generalised for multiparty messaging. Explicit structuring of conversation flows makes the description of multiple flows of interactions within an endpoint implementation clear with regards to the dependencies within each flow and between flows. Since interactions in a session are ensured to never violate the underlying protocol, either by static checking [8, 1] or through run-time monitoring (by protocol machines) [3], each endpoint knows what kinds of messages are coming from which other participants at each stage of a conversation.

To demonstrate the description of multiple conversation flows, our example implementation shall integrate the List Resources protocol with a separately specified HTTP-like request-

```
import Authentication.CheckPrivilege as
    CheckPrivilege;

protocol ListResources(role client as cl,
    role resource_registry as rr) {
  request(resource_kind:String,
       type ListingFormat) from cl
           to rr;
  rr introduces au;
  spawn CheckPrivileges(rr as requester,
      au as authority)
      at rr;
choice at rr {
    ok() from rr to cl;
    rec loop {
      choice at rr {
        response(list:ListingFormat) from rr
            to cl;
        continue loop;
      } or {
        completed() from rr to cl;}}
  } or {
    error(reason:String) from rr to cl;}}
```

Figure 5: A refined List Resources protocol (2)

response protocol (simply called Request-Response). We first give the relevant part of the Scribble for Request-Response in Figure 4 before proceeding to the code. In the figure, "sc" in e.g. `sc200` stands for the "status code" of a message.

Program. We now consider a Python program that uses the `ListResources` and `RequestResponse` protocols (the latter for transparently receiving user requests) in combination. The program is an implementation of a service proxy that obtains data from the Registry on behalf of the User. We call this endpoint program simply "Proxy" from now on. Proxy needs to carry out two kinds of conversations:

1. As a Request-Response server, it will engage in sessions with Users, accepting the User query and returning the results from the Registry.
2. As a List Resources client, it will engage in sessions with the Registry, passing on the User query and receiving the list of resources following Figure 1.

Proxy will return the results to User in HTML format, in a similar manner to a standard CGI application. The main Python code for Proxy related to implementing these sessions is given in Figure 6.

In Line 1, Proxy (receives and) accepts an invitation to interact in the Request-Response session with User. The `proxy_uri` object represents Proxy as a network *principal*, and may roughly be considered as a conversation programming counterpart to a TCP server socket. Proxy can then `accept` an invitation through this interface, with respect to the `RequestResponse` protocol, playing the role of `Server` to User. Specifying the protocol and role for this endpoint prescribes the local programming interface for `c1`, by which Proxy will interact with User.

In Line 2, through `c1`, Proxy receives from User (denoted by its role name `Client` in the protocol), a message `msg`. The basic attributes of a session message include `op`, the operation name for the message (i.e. the message label or header), and the `value` array, the message payload. In Line 3, we check if the operation of `msg` is `GET`. We assume that the kind of resources is specified by the message value, parsed by the `parse_query` function and the result stored in `resource_kind`.

This example demonstrates the interleaving of multiple sessions in a single application. Here we introduce a second session in which Proxy now acts as `client` according to `ListResources`. In Line 5, we initialises a new session, using the class named `Conversation`. When creating a session, we specify the protocol name `ListResources` (taken to be the simplest version presented earlier, in Figure 1). In Line 6, after initialisation, Proxy "joins" the session as the `client` role specified in the protocol.

In Line 7, Proxy *invites* the remote `registry_uri` principal to this newly created session (to play the role `resource_registry`). The method returns when an acknowledgement is returned by the principal to accept the invitation. Now that both roles have joined, in Line 8, Proxy sends to Registry (role name `resource_registry`), a message with the `request` operation and the

```
1    c1 = proxy_uri.accept("RequestResponse", "Server
        ")
2    msg = c1.receive("Client")
3     if msg.op == "GET":
4        resource_kind = parse_query(msg.value)
5        c2 = Conversation("ListResources")
6        c2.join("client")
7        registry_uri.invite(c2, "resource_registry")
8        c2.send("resource_registry", "request",
             resource_kind)
```

```
20   def loop():
21       msg = c2.receive("resource_registry")
22         if msg.operator == "response":
23           # set the response string
24           loop()
25         elif msg.operator == "completed":
26           return
27   loop()
28   c1.send("Client", "sc200")
29   c2.close()
30   c1.close()
```

Figure 6: Conversation endpoint program for a service proxy program in Python (extract)

kind of resources it is interested in. Note the message format precisely follows the protocol.

The next part of the code gives a tail recursive routine for repeated data delivery, whose flow exactly matches that in the `ListResources` protocol. Lines 20–26 define a function `loop`. In its body, first in Line 21, the client receives a `msg` from Registry. Then we have two cases, depending on the operation of the message:

- If the operation is `response` (Line 22), a HTML-formatted version of the original message (which was specified in the protocol to have a YAML format) is appended to the string (Line 23), and the recursion is enacted (Line 24).
- If the operation is `completed`, the recursion is terminated (Line 25).

Line 27 executes this recursive function, and Line 28 returns the HTML request to User. Finally, Line 29-Line 30 close the sessions.

We have illustrated above a simple use of sessions in communications programming. The use of sessions in programs makes it possible to build the application logic with a clear understanding on explicit conversation flows. These flows are clearly visible: by going through how conversation channels are mentioned in a given program (the **red** part in Figure 6), one can clearly capture these flows.

The resulting organisation of communication actions enable not only programs with a clear presentation of interaction structures, but also static validation of conformance to the underlying protocols through type checking; and its dynamic counterpart through finite state machine based protocols monitors. In the latter (dynamic) validation, it is assumed that we can identify the underlying session by inspecting a message, if that message belongs to a session. In this way the runtime messages also get organised, dividing numerous message exchanges in distributed computing environments into different chunks with a binding to underlying protocols. This is how sessions structure communication-centred computing.

3 Using Scribble for End-to-end Cyberinfrastructure

Ocean Observatories Initiative [9], often abbreviated as OOI, is a large-scale NSF-funded project to build a cyberinfrastructure for observing oceans in the United States and beyond, with usage span of 30 years. It integrates real-time data acquisition, processing and data storage for ocean research (e.g. sensor arrays, underwater gliders, high-resolution under-water cameras), providing access for a wide ranging user community under different administrative domains. It consists of multiple marine networks where we lay cables over a large area under the sea, which are integrated by a distributed cyberinfrastructure. This cyberinfrastructure, called OOI CI (CI for CyberInfrastructure), is itself a network, consisting of distributed infrastructural services whose main sites are two large clouds but whose distributed components in the shape of containers also reside all over its distributed sites residing in hundreds of universities and marine institutions.

A central features of the OOI CI is its end-to-end nature, in the sense that its design allows and encourages scientists to register data (which often takes the form of real-time streaming data from sensors over different time scales) and data products (which are derivatives from raw data by application of models). Thus, in this system, multiple heterogeneous organisations and individuals participate, run their software and we need to ensure a high-level quality of usage One of the architectural decisions of OOI CI is to regulate the behaviours of heterogeneous participants in the OOI CI by imposing high-level abstractions based on interaction patterns, which are in turn regulated by high-level policies through runtime monitors. The catalogue of interaction patterns will in turn assist developers to implement their distributed services with ease and clarity. Thus we need a descriptive means to write down these interaction patterns clearly and without ambiguity, use them for software development, and regulate communications behaviour of participating endpoints at runtime through induced protocol machines, augmented with regulation by policies on their basis. For the description of interaction patterns, the use of session types is considered, building a framework to regulate interaction behaviour based on policies on its basis. This policy-based regulation is called "governance" by the OOI CI architects, centring on the notion of commitments [6]. To use session types as a basis of regulating behaviour in this distributed computing platform, several technical challenges were identified, which include (restricted to those proper to session types):

- Can we accurately describe interaction patterns which are and will potentially be used in distributed applications in OOI CI?
- Can we ground them to programming? Can we help developers to build safe and robust systems with ease?
- Can we have a simple and efficient execution framework for these programs?
- Can we guarantee their communication safety at runtime? What would be a simplest mechanism?

The research team on session types are contributing to the OOI CI development through the following technical elements:

- A protocol description language, *Scribble*, and development/execution environments centring on this language.
- A tool chain for protocol validation, endpoint projection, FSM translations, APIs and runtimes.
- Part of the monitor architecture based on the protocol machines (FSM) translated from protocols.

The FSM translation is a direct application of the theory which links automata theory (communication automata) and session types, recently introduced in [5], where a session type can be directly translated into a communication automaton.

The development efforts are producing several interesting findings. For example, one of the methods for facilitating the use of session types for developers who are not accustomed to session types is to use the interface of the standard communication APIs such as RPC. These libraries were independently developed in the OOI CI to support application development based on traditional technologies: the idea is to replace them with distributed runtimes for session types. What we found is that this approach, where we implement libraries using session primitives, has rewarding practical merits in the tractability and transparency in engineering. For instance, each library is now a short scripting code by using the underlying session machinery, automatically monitored by the corresponding protocol. As one example, RPCs with diverse signatures are now based on a single parametrised protocol, and its interactions are checked by a generic monitor for general session types. This conversion is feasible because not even a single line of application code needs be changed: the resulting behaviour is the same, we can use the same interface file, with a formal foundation automatically assuring correctness of interactions. The layer for typed sessions is called Conversation Layer in OOI CI. As well as the extensive experiments on Conversation Layer itself, our development efforts are focusing on the governance functions to be realised on top of Conversation Layer.

Demonstrations

In the Beat Workshop, we plan to give a case-study-driven demonstration to outline how the toolchain is integrated and used in the OOI infrastructure. We shall present the implementation of one of the main governance protocols, a *negotiation protocol* between an agent and a management service for acquiring a resource. The request for permissions can be approved by the management service or rejected based on the agent usage proposal. If rejected, the negotiation (with a different proposal) continues until agreement is reached. First, we write the global protocol in Scribble, emphasising errors that can be reported by the validation tool. Then we show the projection facilities and how the local files are stored and integrated in the system. Finally, we start the OOI services for agent and management, deployed on a separate machine with the monitors in place. We show different fault implementations of the agent code and observe how monitors are created, monitors are receiving events from the system, how errors are detected and logged in the system.

References

[1] Lorenzo Bettini, Mario Coppo, Loris DÁntoni, Marco De Luca, and Mariangiola Dezani-Ciancaglini. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[2] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. In *FSTTCS*, volume 8 of *LIPIcs*, pages 338–351, 2010.

[3] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 7454 of *LNCS*, pages 25–45, 2011.

[4] Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *LNCS*, pages 272–286, 2012.

[5] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213, 2012.

[6] Nirmit Desai, Amit K. Chopra, Matthew Arrott, Bill Specht, and Munindar P. Singh. Engineering foreign exchange processes via commitment protocols. In *IEEE SCC'07*, pages 514–521, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[7] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, volume 6536 of *LNCS*, pages 55–75, 2011.

[8] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

[9] Ocean Observatories Initiative (OOI). `http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/`.

[10] Scribble development tool site. `http://www.jboss.org/scribble`.

[11] Scribble github project. `https://github.com/scribble`.