Parameterized Concurrent Multi-Party Session Types

Minas Charalambides
charala1@illinois.edu

Peter Dinges
pdinges@acm.org

Gul Agha
agha@illinois.edu

Department of Computer Science
University of Illinois at Urbana–Champaign, USA
http://osl.cs.uiuc.edu/

Session types have been proposed as a means of statically verifying implementations of communication protocols. Although prior work has been successful in verifying some classes of protocols, it does not cope well with parameterized, multi-actor scenarios with inherent asynchrony. For example, the sliding window protocol is inexpressible in previously proposed session type systems. This paper describes System-A, a new typing language which overcomes many of the expressiveness limitations of prior work. System-A explicitly supports asynchrony and parallelism, as well as multiple forms of parameterization. We define System-A and show how it can be used for the static verification of a large class of asynchronous communication protocols.

1 Introduction

Session types [27] are a means of expressing the order of messages sent by actors [1] (or processes). In particular, session types can be used to statically check if a group of processes communicate according to a given specification. In these systems, a *global type* specifies the permissible sequences of messages that participants may exchange in a given *session*, as well as the types of these messages. The typing requires the programmer to provide the *global type*. A *projection* algorithm then generates the restrictions implied by the global type for each participant. Such restrictions are called *end-point types* or *local types* and describe the expected behavior of individual participants in the protocol. The actual program code implementing the participant behavior is checked for conformance against this localized behavior specification. We are interested in generalizing prior work on session types to typing coordination constraints on actors, which can then be enforced e.g. with *Synchronizers* [19, 20, 17] or other ways [30].

This requires addressing two limitations of previous work. First, session types do not (directly) support *asynchronous* events; asynchronous communication leads to delays which require considering arbitrary shuffles. Second, we wish to consider *parameterized* protocols. For example, consider two actors communicating through a sliding window protocol: the actors agree on the length of the window (i.e., the number of messages that may be buffered) and then proceed to an exchange of concurrent messages. Prior work on session types is not suitable for typing protocols like this: the reason for this deficiency is the fact that their respective type languages depend on other formalisms for type checking (such as typed λ-calculus [3] or System T [25]) and these formalisms do not support a parallel construct.

Contributions. We overcome many of these shortcomings by developing System-A, a new system for expressing types for multi-party interaction that does not depend on other formalisms for type checking. The main contributions are (a) *parameterized* constructs for expressing asynchrony, parallelism,

Natallia Kokash and António Ravara (Eds.): 11th International Workshop on
Foundations of Coordination Languages and Self Adaptation (FOCLASA'12)
© M. Charalambides, P. Dinges & G. Agha
EPTCS 91, 2012, pp. 16–30, doi:10.4204/EPTCS.91.2

sequence and choice (Section 4), (b) a projection mechanism to provide type constraints on individual actors (Section 5), (c) the conditions under which conformance of the latter with the global type is assured (Sections 7, 8) and finally (d) we show that structural equivalence of types is decidable in System-A, by proving strong normalization of our local types (Section 6). Proofs of our theorems are included in the long version of this paper [13].

Limitations. Using the strong normalization result, we can decide whether the local behavior of an actor follows the protocol. However, this result relies on a type inference mechanism for the actor's behavior (of the sort in Alur et al. [2]). We do not describe such a type inference mechanism in this article. Moreover, we omit support for session delegation. Finally, our realizability results rely on structural criteria and are hence conservative rather than precise [4].

2 Related Work

Session types [27, 39, 36, 26] originate from the context of π-calculi as statically derivable descriptions of process interaction behaviors. In two-party sessions, they allow us to statically verify that the participants have compatible behavior by requiring *dual* session types, that is, behaviors where each participant expects precisely the message sequence that the other participant sends and vice versa. Extensions to session types support asynchronous message passing [32] and introduce subtyping [21] for a looser notion of type compatibility. Session types have been integrated into functional [37, 35] and object-oriented [16, 29, 23] languages among others, with a wide range of applications including deadlock and livelock detection [24]. Other extensions deal with evolving system specifications using transformations [18]. Exception handling, which allows the participants of a protocol to escape the normal flow of control and coordinate on another, has been considered in [11, 10]. The present article combines three enhancements to session types that majorly extend their applicability: concurrent multi-party sessions, parameterized session types, and an enhanced syntax.

Asynchronous Multi-Party Sessions. Many real-world protocols involve more than two participants, which makes their description in terms of multiple two-party sessions unnatural. To overcome this limitation, Honda et al. [28] extend session types to support multiple participants: a *global type* specifies the interactions between all participants from a global perspective. A projection algorithm then mechanically derives the behavior specification of the individual participants, that is, the *local type*.

The notion of global type and the associated correctness requirements for projection were first studied by Carbone et al. [9]; Bonelli's work on multi-point session types [7] treats multi-party protocols from the local perspective only. Bettini et al. [6] allow multi-party sessions to interleave and derive a type system guaranteeing global progress. Gay et al. [22] consider subtyping in presence of asynchrony.

The present article builds on the foundation of a global protocol specification and its projection onto local behaviors [28]. However, we do not address the question of local type safety and inference of actual programs, which is a major part of Honda et al.'s work. Furthermore, unlike their approach (but following Castagna et al. [12]), we simplify the notation for global types by replacing recursion with the Kleene star and limiting each pair of participants to use a single channel. We introduce an explicit shuffle operator to preserve the commutativity of message arrivals that can be achieved using multiple channels. Explicit shuffles also reduce the need for a special subtyping relationship that allows the permutation of (Lamport-style) concurrent asynchronous events for optimization [31].

Following Castagna's global type syntax further, we support join operations. Joins cannot be expressed in Honda et al.'s global types because of the linearity requirement. However, as Deniélou remarks [15], join operations can only describe series–parallel graphs. Protocols such as the *alternating bit protocol* that require interleaved synchronization between two processes consequently cannot be expressed in our global type language. Our choice to not support generic graph structures as global types is founded on the desire to support parameterization and, at the same time, keep the language understandable; it remains unclear to us how to visualize parameterized graphs in an intuitive fashion.

Parameterized Session Types. Our major extension of global types over Honda et al. and Castagna et al.'s work is the introduction of parameters. The starting point for our parameterization of session types is the work of Yoshida et al. [38] and Bejleri [5]. Yoshida et al. augment the global types of Bettini et al. [6] with primitive recursive combinators to obtain dependent types that support the parameterization of the repetition count and the connection topology. This allows, for example, using a single global type for a highly participant-count dependent butterfly network. Static verifiability—without instantiating the type parameters—is maintained by projecting onto parameterized local types that allow syntactic comparisons. In [14], Deniélou et al. achieve parameterization by means of quantification over behavior specifications they call *roles*. Like Bettini et al. and unlike System-A, neither Deniélou et al. nor Yoshida et al. support arbitrary, concurrency-induced shuffles in their global and local types. While Bettini et al. regain parallel composition through the interleaving of global types, it is unclear how the results transfer to the other two approaches.

Modeling of Multi-Party Protocols. Formalisms for describing multi-party communication protocols have been studied in the context of designing distributed systems and cryptographic protocols. As modeling tools, the formalisms provide ways to check a protocol for desired properties [40], or to synthesize such protocols [34]. In contrast to session types, the formalisms lack ways to statically verify the compliance of an actual protocol implementation against the specification. Deniélou and Yoshida [15] discuss session types and their relation to work on distributed systems or cryptographic protocols in greater depth.

3 Motivation

Formalisms introduced in previous work are not expressive enough to define the types of some interesting protocols such as the sliding window protocol, a locking–unlocking protocol, and a case of limited resource sharing. In this section, we demonstrate how the behavior of these protocols can be described in System-A.

The Sliding Window Protocol. Assume an actor a sends messages of type m to an actor b, which acknowledges every received message with an *ack* message. The protocol determines that at most n messages can be unacknowledged at any given time, so that a ceases sending until it receives another *ack* message. In this example, the window size n is a parameter, which means we need a way to express the fact that n sending–acknowledging events can be in transit at any given instant in time. Following is the global type of the protocol.

$$\underbrace{\left(a \xrightarrow{m} b\,;\, b \xrightarrow{ack} a\right)^* \parallel \left(a \xrightarrow{m} b\,;\, b \xrightarrow{ack} a\right)^* \parallel \ldots \parallel \left(a \xrightarrow{m} b\,;\, b \xrightarrow{ack} a\right)^*}_{n \text{ times}}$$

$a \xrightarrow{m} b$ denotes that a sends a message of type m to b. Operator ; is used for sequencing interactions. $\|$ is used for composing its left and right arguments in parallel. The Kleene star has the usual semantics and takes precedence over $\|$.

The above type can be expressed using the notation of Castagna et al. [12], albeit with a fixed window size n. In System-A on the other hand, we can parameterize the type in n and statically verify that participants follow the protocol without knowing its value at runtime. Using $\prod_{i=1}^{n}$ to denote the parallel composition of n processes, we obtain the following type in our notation:

$$\prod_{i=1}^{n} \left(a \xrightarrow{m} b \,;\, b \xrightarrow{ack} a \right)^*$$

Locking / Unlocking. Consider a set of n processes, each of which needs to acquire exclusive access to a resource by sending it a *lock* message. The resource replies with *ack*, the process uses the resource and unlocks it by sending an *unlock* message, at which point the next process can do the same, and so on. The following type describes the locking–unlocking protocol for a fixed number of processes.

$$(c_1 \xrightarrow{lock} s \,;\, s \xrightarrow{ack} c_1 \,;\, c_1 \xrightarrow{unlock} s) \otimes \ldots \otimes (c_n \xrightarrow{lock} s \,;\, s \xrightarrow{ack} c_n \,;\, c_n \xrightarrow{unlock} s)$$

With $\otimes$ denoting shuffling, this formula expresses that any ordering of the $(c_i \xrightarrow{lock} s \,;\, s \xrightarrow{ack} c_i \,;\, c_i \xrightarrow{unlock} s)$ events is acceptable. To support a dynamic network topology, the number of participants should be a parameter. The following is the locking–unlocking example in System-A, where conformance to the protocol is statically verifiable without knowledge of the runtime value of n.

$$\bigotimes_{i=1}^{n} (c_i \xrightarrow{lock} s \,;\, s \xrightarrow{ack} c_i \,;\, c_i \xrightarrow{unlock} s)$$

Limited Resource Sharing. In this scenario, a server s grants two clients c_1 and c_2 exclusive access to a set of n resources. At any given point, a maximum of n resources can be locked, but the relevant lock–ack–unlock messages from both clients can be interleaved in any way. Following is the global type for this situation:

$$\prod_{i=1}^{n} \left(c_1 \xrightarrow{lock_i} s \,;\, s \xrightarrow{ack_i} c_1 \,;\, c_1 \xrightarrow{unlock_i} s \,\oplus\, c_2 \xrightarrow{lock_i} s \,;\, s \xrightarrow{ack_i} c_2 \,;\, c_2 \xrightarrow{unlock_i} s \right)^*$$

The parallel composition is parameterized in n, the number of resources. Each sequence of lock–ack–unlock messages is also parameterized in i, which ranges from 1 to n. This is necessary to ensure realizability of the protocol, as in the case of multiple outstanding requests, it allows the participants to disambiguate the responses they receive. Each parallel instance subsumed by the $\prod_{i=1}^{n}$ operator consists of a loop (Kleene Star) which entails a choice, indicated by $\oplus$. Either c_1 gets access to a resource, or c_2 and this happens repeatedly.

4 Type Syntax

4.1 Global Types

A global type describes a protocol which the whole system must adhere to. The examples in Section 3 are all global types, since they describe the behavior of all participants. Global types in System-A can be constructed according to the grammar in Table 1, with operator descriptions following.

Table 1: *The grammar of global types*

$\mathscr{G}$ ::=	$(\mathscr{G})$	(G-Paren)	|	ε	(G-Empty)
|	$\mathscr{G}\,;\mathscr{G}$	(G-Seq)	|	$\overset{n}{\underset{i=1}{\circledcirc}}\mathscr{G}_i$	(G-Seq-N)
|	$\mathscr{G} \oplus \mathscr{G}$	(G-Choice)	|	$\overset{n}{\underset{i=1}{\bigoplus}}\mathscr{G}_i$	(G-Choice-N)
|	$\mathscr{G} \parallel \mathscr{G}$	(G-Parallel)	|	$\overset{n}{\underset{i=1}{\Vert}}\mathscr{G}_i$	(G-Parallel-N)
|	$\mathscr{G} \otimes \mathscr{G}$	(G-Shuffle)	|	$\overset{n}{\underset{i=1}{\bigotimes}}\mathscr{G}_i$	(G-Shuffle-N)
|	$p_1 \xrightarrow{type} p_2$	(G-Interaction)	|	$\mathscr{G}^n$	(G-Exp)
			|	$\mathscr{G}^*$	(G-KleeneStar)

(G-Seq) is used for the sequential composition of events.

(G-Choice) denotes exclusive choice between the arguments. For instance, $\mathscr{G}_1 \oplus \mathscr{G}_2$ means that either $\mathscr{G}_1$ or $\mathscr{G}_2$ will be executed (but not both).

(G-Parallel) means that the arguments run in parallel; any interleaving of sequenced actions is possible. For instance, $(a \xrightarrow{t_1} b\,;\, a \xrightarrow{t_2} c) \parallel c \xrightarrow{t_3} b$ means that any of the interleavings ABC, ACB, CAB is possible, where A = $(a \xrightarrow{t_1} b)$, B = $(a \xrightarrow{t_2} c)$ and C = $(c \xrightarrow{t_3} b)$. Notice that B is not allowed to precede A, as the ordering of actions as determined by operator ; is not allowed to change.

(G-Shuffle) means that both arguments are executed atomically, in an unspecified order. Formally, $\mathscr{G}_1 \otimes \mathscr{G}_2 \equiv (\mathscr{G}_1\,;\mathscr{G}_2) \oplus (\mathscr{G}_2\,;\mathscr{G}_1)$.

(G-Interaction) denotes the sending and receiving of a message. For instance, $p_1 \xrightarrow{t} p_2$ means that process p_1 sends a message of type t to process p_2.

(G-KleeneStar) has the usual semantics, of zero or more repetitions of the argument.

The n-ary versions of the operators express behaviors where the value of n is unknown at compile time. (G-Seq-N), (G-Choice-N), (G-Parallel-N), (G-Shuffle-N) apply the respective binary operator $n-1$ times to n global types, parameterized in i. (G-Exp) denotes n-fold repetition of the argument (in sequence). Note that for known values of n, we do not need the right column of Table 1, as the desired behavior can be produced by suitable repeated applications of the binary operators.

All of the operators are commutative, with the exception of sequencing. All operators are furthermore associative, with the exception of shuffling. In particular,

$$\overset{n}{\underset{i=1}{\bigotimes}}\mathscr{G}_i \neq (\ldots(\mathscr{G}_1 \otimes \mathscr{G}_2) \otimes \mathscr{G}_3 \ldots) \otimes \ldots \otimes \mathscr{G}_n).$$

Instead, $\overset{n}{\underset{i=1}{\bigotimes}}\mathscr{G}_i$ means that all arguments $\mathscr{G}_i$ are executed atomically, but in an unspecified order.

The distinction between the Kleene star and exponentiation is fundamental. The use of $\mathscr{G}^n$ means that the protocol conformance checker will have to prove that the system is correct for any fixed value of the parameter n. $\mathscr{G}^*$ on the other hand means an arbitrary number of repetitions of $\mathscr{G}$. There is no parameter fixing this number and it may be different from instance to instance of the Kleene Star and/or among executions of the same program with the same run-time values for its parameters.

4.2 Local Types

A local type specifies the abstract behavior of a single protocol participant. The syntax of local types is given in Table 2, with descriptions following.

Table 2: *The grammar of local types*

$\mathscr{L} ::=$	$(\mathscr{L})$	(L-Paren)	$\mid$	ε	(L-Empty)
$\mid$	$a!t$	(L-Send)	$\mid$	$a?t$	(L-Recv)
$\mid$	$\mathscr{L}\,;\mathscr{L}$	(L-Seq)	$\mid$	$\overset{n}{\underset{i=1}{\circledcirc}}\mathscr{L}_i$	(L-Seq-N)
$\mid$	$\mathscr{L}\oplus\mathscr{L}$	(L-Choice)	$\mid$	$\overset{n}{\underset{i=1}{\bigoplus}}(\mathscr{L}_i)$	(L-Choice-N)
$\mid$	$\mathscr{L}\parallel\mathscr{L}$	(L-Parallel)	$\mid$	$\overset{n}{\underset{i=1}{\Vert}}\mathscr{L}_i$	(L-Parallel-N)
$\mid$	$\mathscr{L}\otimes\mathscr{L}$	(L-Shuffle)	$\mid$	$\overset{n}{\underset{i=1}{\bigotimes}}\mathscr{L}_i$	(L-Shuffle-N)
$\mid$	$\mathscr{L}^n$	(L-Exp)	$\mid$	$\mathscr{L}^*$	(L-KleeneStar)

(L-Seq), (L-Choice), (L-Parallel), (L-Shuffle), (L-Exp), (L-KleeneStar) are defined as in the case of global types (Section 4.1).

With (L-Parallel) being defined as in the global case, the local type $(a!t\,;a!u)\parallel a?v$ again allows three orderings of the events $T=a!t$, $U=a!u$, and $V=a?v$: TUV, TVU, and VTU. As above, the specification $a!t\,;a!u$ enforces that T happens before U.

(L-Send) denotes sending a message of type t to process a.

(L-Recv) denotes receiving a message of type t from process a.

In the sliding window example of Section 3, the behavior of the sender a is described by the local type $\overset{n}{\underset{i=1}{\Vert}}(b!m\,;b?\text{ack})^*$. Leaving out the initial $\overset{n}{\underset{i=1}{\Vert}}$ symbol for the time being, what remains is $(b!m\,;b?\text{ack})^*$. This means sending a message and then receiving an acknowledgment $(b!m\,;b?\text{ack})$, an arbitrary number of times. Assuming that the window size n is a parameter, any interleaving of n of these sequences is possible, with the obvious constraint of not receiving more acknowledgments than the number of messages sent. This is ensured by composing sequences of the form $(b!m\,;b?\text{ack})$, where ordering is forced by the ; operator.

5 Projection

The local type of the sliding window protocol in Section 4.2 is a restriction of the respective global type in Section 3 onto the individual processes. In this section, we investigate a way of automating this process. $\mathscr{G}\triangleright p$ is read "the projection of global type $\mathscr{G}$ onto process p" and the result is a local type as defined in Table 2. The projection function $\triangleright$ is formally defined in Table 3 and the result of applying it to all the processes in the system is an *environment* $\Delta=\{p_i:\mathscr{L}_i\}_{i\in I}$ which maps processes to local types.

Table 3: *The projection function*

$(a \xrightarrow{m} b) \triangleright p$	::=	$\begin{cases} b!m & \text{if } p = a \\ a?m & \text{if } p = b \\ \varepsilon & \text{otherwise} \end{cases}$	(P-Interaction)
$\mathscr{G}^n \triangleright p$	::=	$(\mathscr{G} \triangleright p)^n$	(P-Exp)
$(\mathscr{G}_1 \oplus \mathscr{G}_2) \triangleright p$	::=	$(\mathscr{G}_1 \triangleright p) \oplus (\mathscr{G}_2 \triangleright p)$	(P-Choice)
$(\mathscr{G}_1 \parallel \mathscr{G}_2) \triangleright p$	::=	$(\mathscr{G}_1 \triangleright p) \parallel (\mathscr{G}_2 \triangleright p)$	(P-Paral)
$(\mathscr{G}_1 \,;\, \mathscr{G}_2) \triangleright p$	::=	$(\mathscr{G}_1 \triangleright p) \,;\, (\mathscr{G}_2 \triangleright p)$	(P-Seq)
$(\mathscr{G}_1 \otimes \mathscr{G}_2) \triangleright p$	::=	$(\mathscr{G}_1 \triangleright p) \otimes (\mathscr{G}_2 \triangleright p)$	(P-Shuffle)
$(\overset{n}{\underset{i=1}{\circledcirc}} \mathscr{G}_i) \triangleright p$	::=	$\overset{n}{\underset{i=1}{\circledcirc}} (\mathscr{G}_i \triangleright p)$	(P-Seq-N)
$(\overset{n}{\underset{i=1}{\bigoplus}} \mathscr{G}_i) \triangleright p$	::=	$\overset{n}{\underset{i=1}{\bigoplus}} (\mathscr{G}_i \triangleright p)$	(P-Choice-N)
$(\overset{n}{\underset{i=1}{\bigotimes}} \mathscr{G}_i) \triangleright p$	::=	$\overset{n}{\underset{i=1}{\bigotimes}} (\mathscr{G}_i \triangleright p)$	(P-Shuffle-N)
$(\overset{n}{\underset{i=1}{\parallel}} \mathscr{G}_i) \triangleright p$	::=	$\overset{n}{\underset{i=1}{\parallel}} (\mathscr{G}_i \triangleright p)$	(P-Paral-N)

For the lock/unlock example of Section 3, projecting onto a client c_k and the server s yields

$$
\begin{aligned}
\mathscr{G} \triangleright c_k &= \bigotimes_{i=1}^{n} (c_i \xrightarrow{lock} s \,;\, s \xrightarrow{ack} c_i \,;\, c_i \xrightarrow{unlock} s \triangleright c_k) && \text{(P-Shuffle-N)} \\
&= \bigotimes_{i \neq k} \varepsilon \otimes (s!lock \,;\, s?ack \,;\, s!unlock) && \text{(P-Interaction), (P-Seq)} \\
&= s!lock \,;\, s?ack \,;\, s!unlock, && \text{(eliminating } \varepsilon\text{)} \\
\mathscr{G} \triangleright s &= \bigotimes_{i=1}^{n} (c_i \xrightarrow{lock} s \,;\, s \xrightarrow{ack} c_i \,;\, c_i \xrightarrow{unlock} s \triangleright s) && \text{(P-Shuffle-N)} \\
&= \bigotimes_{i=1}^{n} (c_i?lock \,;\, c_i!ack \,;\, c_i?unlock)\,. && \text{(P-Interaction), (P-Seq)}
\end{aligned}
$$

Similarly, the projected local types for the resource sharing protocol of Section 3 are

$$
\begin{aligned}
\mathscr{L}_s &= \overset{n}{\underset{i=1}{\parallel}} (c_1?lock_i \,;\, c_1!ack_i \,;\, c_1?unlock_i \oplus c_2?lock_i \,;\, c_2!ack_i \,;\, c_2?unlock_i)^* \\
\mathscr{L}_{c_1} &= \overset{n}{\underset{i=1}{\parallel}} (s!lock_i \,;\, s?ack_i \,;\, s!unlock_i)^* \\
\mathscr{L}_{c_2} &= \overset{n}{\underset{i=1}{\parallel}} (s!lock_i \,;\, s?ack_i \,;\, s!unlock_i)^*\,.
\end{aligned}
$$

6 Type Checking

Given a global type, we need to be able to check the respective projections against the local types inferred from the program itself. This is possible due to the following properties of our language of local types:

Theorem 1 (Weak Normalization)**.** For any local type $\mathscr{L}$ in System-A, there exists a finite sequence of reduction steps which brings the type to a normal form.

We prove this in the extended version of this paper [13], where we provide the reduction semantics and a normalization process.

Corollary 1 (Strong Normalization)**.** For every local type $\mathscr{L}$ in System-A, all sequences of reduction steps are finite and lead to the same normal form.

In the extended version of this paper, we show that the aforementioned normalization process uniquely determines the reduction semantics, implying the uniqueness of normal forms.

Checking structural equivalence of the types derived from the program against the projections is decidable up to α-conversion. However, all that is required to overcome this issue is that names in the code are consistent with those in the supplied global type. In our opinion it is reasonable to expect programmers to adhere to such a naming convention.

7 Global Type Realization

In this section, we discuss the properties that a given global type must satisfy in order to be *projectable*. These properties are discussed while assuming actor semantics [1] for the messaging system; that is, asynchronous, unordered and eventual (guaranteed, albeit with arbitrary delay) delivery of messages. Applying the projection function to a *projectable* global type will result in local types for the participants, whose combined behavior is consistent with the global type—a fact we show in Section 8.

The subsequent discussion of projectability criteria uses the following definitions:

Definition 1 (Event)**.** An *event* is a single interaction $p_1 \xrightarrow{m} p_2$ in a global type.

We extend the projection function onto events and write $e \triangleright p$ to denote the projection of event e onto process p using rule (P-Interaction).

Definition 2 (Trace)**.** A *trace* is a sequence of events producible by a global type $\mathscr{G}$ and is of the form $e_1 ; e_2 ; \ldots ; e_k$. The set of traces a global type $\mathscr{G}$ can produce is denoted by $tr(\mathscr{G})$. The first and last events of a trace t are denoted $first(t)$ and $last(t)$ respectively. Abusing notation, the set of events that appear first in traces of $\mathscr{G}$ is denoted $first(\mathscr{G}) = \{first(t) \mid t \in tr(\mathscr{G})\}$. Similarly, the set of events that appear last in traces of $\mathscr{G}$ is denoted $last(\mathscr{G})$.

Since a trace is simply a sequence of events of the form $p \xrightarrow{m} q$, we extend the projection function onto traces in the natural way. We write $t \triangleright p$ to denote the projection of trace t onto process p using rules (P-Seq) and (P-Interaction).

7.1 Sequentiality Criterion

The purpose of this criterion is to ensure that the sequential constructs of a global type retain sequential semantics after projection. As an example problematic case, consider $\mathscr{G}_1 = a \xrightarrow{m_1} b \,;\, c \xrightarrow{m_2} d$. Without the use of some covert coordination channel (for example by implementing a barrier mechanism), it is impossible for c to know when b has received the message. The two events $a \xrightarrow{m_1} b$ and $c \xrightarrow{m_2} d$ are impossible to order using our projection function, as the resulting environment would be $\Delta_1 = \{a :$

$b!m_1, b : a?m_1, c : d!m_2, d : c?m_2\}$, which allows c to send m_2 to d before a sends m_1 to b. $\mathscr{G}_1$ does not satisfy the sequentiality criterion and thus is not projectable.

Another problematic case is $\mathscr{G}_2 = a \xrightarrow{m_1} b \,;\, a \xrightarrow{m_2} b$, where a cannot know when m_1 has been received so as to start transmitting m_2, hence $\mathscr{G}_2$ is not projectable either. The following definition captures the conditions under which events are guaranteed to respect the sequencing restrictions imposed in a global type, when the latter is projected onto individual processes.

Definition 3 (Sequentially Projectable Global Type)**.** The set of *sequentially projectable* (*SP*) global types is defined inductively as follows:

$$\begin{cases} p_1 \xrightarrow{t} p_2 \quad \in SP \quad \forall p_1, p_2 \in \Pi \\ p_1 \xrightarrow{m_1} p_2 \,;\, p_2 \xrightarrow{m_2} p_3 \quad \in SP \quad \forall p_1, p_2, p_3 \in \Pi \\ \\ \left(\forall e_1 \in last(\mathscr{G}_1), e_2 \in first(\mathscr{G}_2) \Rightarrow (e_1 \,;\, e_2) \in SP\right) & \Rightarrow \quad \mathscr{G}_1 \,;\, \mathscr{G}_2 \in SP \\ \left(\forall e_1 \in last(\mathscr{G}_i\{1/i\}), e_2 \in first(\mathscr{G}_i\{2/i\}) \Rightarrow (e_1 \,;\, e_2) \in SP\right) & \Rightarrow \quad \overset{n}{\underset{i=1}{\circledcirc}} \mathscr{G}_i \in SP \\ \left(\forall e_1 \in last(\mathscr{G}), e_2 \in first(\mathscr{G}) \Rightarrow (e_1 \,;\, e_2) \in SP\right) & \Rightarrow \quad \mathscr{G}^n \in SP \\ \left(\forall e_1 \in last(\mathscr{G}), e_2 \in first(\mathscr{G}) \Rightarrow (e_1 \,;\, e_2) \in SP\right) & \Rightarrow \quad \mathscr{G}^* \in SP \end{cases}$$

where Π denotes the set of processes.

Illustrating the third case of the definition above, the following global type is in *SP*:

$$\mathscr{G} = (a \xrightarrow{m} b \parallel c \xrightarrow{m} b) \,;\, (b \xrightarrow{m} l \parallel b \xrightarrow{m} k)$$

It is easy to see that $last\left(a \xrightarrow{m} b \parallel c \xrightarrow{m} b\right) = \{a \xrightarrow{m} b,\ c \xrightarrow{m} b\}$ and $first\left(b \xrightarrow{m} l \parallel b \xrightarrow{m} k\right)$ $= \{b \xrightarrow{m} l,\ b \xrightarrow{m} k\}$ so that all four sequences (e.g. $a \xrightarrow{m} b \,;\, b \xrightarrow{m} k$) are in *SP* according to the first two lines of the definition above.

7.2 Choice Criterion

The purpose of this criterion is to ensure that projecting $\mathscr{G}_1 \oplus \mathscr{G}_2$ maintains the choice semantics, meaning that all participants can recognize which branch of the choice operator they need to take during execution. As an example of a type that does not satisfy this criterion, consider

$$\mathscr{G} = (a \xrightarrow{m_1} b \,;\, b \xrightarrow{k} c \,;\, c \xrightarrow{t_1} d) \oplus (a \xrightarrow{m_2} b \,;\, b \xrightarrow{k} c \,;\, c \xrightarrow{t_2} d).$$

Here, a and b know which branch they are on, because on the left branch b receives a message of type m_1 from a, while on the branch on the right it receives a message of type m_2. However, from that point on, b behaves identically with respect to c, which has no way of telling whether the message to send to d should be of type t_1 or t_2. We call the first point at which two traces differ with respect to a given process the *distinctive point*, which can be ε if no such point exists. This notion is formalized in the following definition:

Definition 4 (Distinctive Point)**.** The *distinctive point* of a process p with respect to a pair of traces $t_1 = (e_1, \ldots, e_k) \in tr(\mathscr{G})$ and $t_2 = (f_1, \ldots, f_l) \in tr(\mathscr{G})$ is an index i given by

$$d_{t_1,t_2}(p) = \min\{i \mid (e_i \triangleright p) \neq (f_i \triangleright p)\}$$

where e_j, f_j denote events. In the case where $t_1 \triangleright p = \varepsilon$, or $t_2 \triangleright p = \varepsilon$, or $t_1 \triangleright p = t_2 \triangleright p$, no such i exists and the distinctive point is defined to be ε.

The definition that follows captures the conditions under which the choice semantics are maintained after projection. The first bullet deals with the non parameterized version of the choice operator $\oplus$. Item (i) captures the case where a process p is the first process acting on the two branches, in which case it must inform the others of the branch they are on. It does so by either sending a different message, or by sending to a different process in each case. Note that the same process must inform the others on both branches. Item (ii) captures the case where p is not the first process to act, in which case it must be informed of the branch it is on and the distinctive point should be a suitable receive event.

Notice how the second bullet deals with shuffling by means of choice. Clearly, if a process can tell whether it is on $\mathscr{G}_1$ or $\mathscr{G}_2$, it is also able to tell the order in which they appear.

The third bullet inductively uses the previous two to define choice-wise projectability in the parameterized cases of choice $\oplus$ and shuffle $\otimes$.

Definition 5 (Choice-Wise Projectable Global Type)**.** The set of *Choice-Wise Projectable* (*CP*) global types is defined inductively as follows:

- $\mathscr{G} = \mathscr{G}_1 \oplus \mathscr{G}_2 \in CP$ *iff* $\forall p \in \mathscr{G}$, either of the following is true:

  (i) $\mathscr{G} = \mathscr{G}_1 \oplus \mathscr{G}_2$ and

  $\forall e_1 \in first(\mathscr{G}_1),\ e_2 \in first(\mathscr{G}_2) : e_1 = p \xrightarrow{m} q,\ e_2 = p \xrightarrow{m'} q'$ where $p \neq q$ and $p \neq q'$ and $(q \neq q' \text{ or } m \neq m')$

  (ii) $\forall t_1 = (s_1, \ldots, s_{k_1}) \in tr(\mathscr{G}_1),\ t_2 = (u_1, \ldots, u_{k_2}) \in tr(\mathscr{G}_2)$, either $d_{t_1,t_2}(p) = \varepsilon$, or

  $d_{t_1,t_2}(p) = i$ and $s_i = q \xrightarrow{m} p,\ u_i = q' \xrightarrow{m'} p$ where $p \neq q$ and $p \neq q'$ and $(q \neq q' \text{ or } m \neq m')$

- $\mathscr{G} = \mathscr{G}_1 \otimes \mathscr{G}_2 \in CP \quad iff \quad \mathscr{G}_1 \oplus \mathscr{G}_2 \in CP$
- $\mathscr{G} = \bigoplus_{i=1}^{n} \mathscr{G}_i \in CP \quad iff \quad (\mathscr{G}_i\{1/i\} \oplus \mathscr{G}_i\{2/i\}) \in CP$

  $\mathscr{G} = \bigotimes_{i=1}^{n} \mathscr{G}_i \in CP \quad iff \quad (\mathscr{G}_i\{1/i\} \otimes \mathscr{G}_i\{2/i\}) \in CP$

The criterion for parameterized shuffling $\bigotimes_{i=1}^{n}$ is stricter than what one can derive if the value of n is given. However, it is hard to loosen up the constraint when it is dealt with as a parameter.

7.3 Parallel Composability Criterion

As an example of what can go wrong when composing two global types using the $\|$ operator, consider the example $\mathscr{G} = \left((a \xrightarrow{m_1} b\,;\, b \xrightarrow{k_1} c) \oplus (a \xrightarrow{m_2} b\,;\, b \xrightarrow{k_2} c)\right) \| a \xrightarrow{m_1} b$. The intended behavior of $\mathscr{G}$ is that a chooses whether to send a message of type m_1 or m_2 to b, which in turn decides whether to send c a message of type k_1 or k_2. Concurrently with this, an additional m_1 is sent from a to b. Assume that as far as $\oplus$ is concerned, a decides to send m_2 to b. It is then obvious how the additional parallel event $a \xrightarrow{m_1} b$ might confuse b to simultaneously take both branches of the choice operator.

In general, the problem appears when actions in one parallel branch affect choices made on another. Global types that do not exhibit this problem are *parallel projectable* (*PP*).

Definition 6 (Parallel Projectable Global Types)**.** For two global types $\mathscr{G}_1$ and $\mathscr{G}_2$, $\mathscr{G} = \mathscr{G}_1 \parallel \mathscr{G}_2$ is parallel projectable (*PP*) if there is no overlap between the distinctive points in $\mathscr{G}_1$ and events in $\mathscr{G}_2$. Formally,

- $\mathscr{G}_1 \parallel \mathscr{G}_2 \in PP \quad iff \quad \forall t_1 = (e_1,\ldots,e_k),\, t_1' = (e_1',\ldots,e_{k'}') \in tr(\mathscr{G}_1),\, t_2 \in tr(\mathscr{G}_2),\, p \in \Pi$ we have $d_{t_1,t_1'}(p) = i$ and one of the following is true:

  (a) $i = \varepsilon$

  (b) e_i, e_i' both have p as the sender

  (c) $e_i \notin t_2$ and $e_i' \notin t_2$

- $\parallel_{i=1}^{n} \mathscr{G}_i \in PP \quad iff \quad (\mathscr{G}_i\{1/i\} \parallel \mathscr{G}_i\{2/i\}) \in PP$

where Π denotes the set of processes. Notice how this definition incorporates parallel composability of two Kleene starred types (the Kleene Star entails a choice pertaining to loop entrance and exit).

7.4 Kleene Star Criterion

Use of the Kleene Star in global types can result in protocols whose projection is unsafe, that is, can result in execution traces that are not part of the original global type. To avoid this, a global type must be such that the entry and exit conditions to the starred type can be identified by all participants. Determining whether this is the case requires inspection of not only the starred type itself, but also of what comes after the starred section.

Definition 7 (Kleene Star Projectable Global Types)**.** For global types $\mathscr{G}$,$\mathscr{G}'$, we say that $\mathscr{G}^*\,;\mathscr{G}'$ is Kleene Star Projectable (KP) *iff* $\mathscr{G} \oplus \mathscr{G}' \in CP$.

As an example of a type that is not in *KP*, consider $\mathscr{G} = (a \xrightarrow{m} b\,;\, b \xrightarrow{m'} c)^*\,;\, c \xrightarrow{m''} d$ where c has no way of knowing whether it should wait for m' from b, or proceed immediately with sending d the message m''.

8 Correctness

The conditions discussed above are sufficient to ensure that the projection function generates local types which are functionally consistent with the global type. We call a global type that satisfies all of the above criteria *projectable*:

Definition 8 (Projectable Global Type)**.** The set of *projectable* (*PR*) global types is inductively defined in Table 4.

Theorem 2 formalizes our intuition that under the constraints mentioned above, the projection function is correct; that is, the projected environment is consistent with the global type. In what follows, $tr(\Delta)$ with $\Delta = \{p_i : \mathscr{L}_i\}_{i\in I}$ denotes the set of traces producible by environment Δ. Also, $\Delta_{\mathscr{G}}$ denotes the environment resulting from the projection of $\mathscr{G}$ onto the set of processes, i.e. $\Delta_{\mathscr{G}} = \{p : \mathscr{G} \triangleright p\}_{p\in\Pi}$.

Theorem 2. $\mathscr{G} \in PR \Rightarrow tr(\mathscr{G}) = tr(\Delta_{\mathscr{G}})$

We sketch the proof of this theorem in the extended version of this paper [13], where we inductively treat each of the cases in Table 4. What needs to be proved is essentially $\forall t \in tr(\mathscr{G}) \Leftrightarrow t \in tr(\Delta_{\mathscr{G}})$. While the forward direction is rather obvious, proving that the projected environment does not generate traces that are not part of the original global type is trickier and is why we need the criteria of Section 7.

Proving this theorem, we get a correctness proof of our projection function (given the premises discussed previously) for free.

Table 4: *The set PR*

$\varepsilon \in PR$				
$a \xrightarrow{m} b \in PR$				
$\mathscr{G} \in PR$	and	$\mathscr{G}^n \in SP$	$\Rightarrow$	$\mathscr{G}^n \in PR$
$\mathscr{G}_1, \mathscr{G}_2 \in PR$	and	$(\mathscr{G}_1 \,;\, \mathscr{G}_2) \in SP$	$\Rightarrow$	$(\mathscr{G}_1 \,;\, \mathscr{G}_2) \in PR$
$\mathscr{G}_1, \mathscr{G}_2 \in PR$	and	$(\mathscr{G}_1 \oplus \mathscr{G}_2) \in CP$	$\Rightarrow$	$(\mathscr{G}_1 \oplus \mathscr{G}_2) \in PR$
$\mathscr{G}_1, \mathscr{G}_2 \in PR$	and	$(\mathscr{G}_1 \otimes \mathscr{G}_2) \in CP$	$\Rightarrow$	$(\mathscr{G}_1 \otimes \mathscr{G}_2) \in PR$
$\mathscr{G}_1, \mathscr{G}_2 \in PR$	and	$(\mathscr{G}_1 \parallel \mathscr{G}_2) \in PP$	$\Rightarrow$	$(\mathscr{G}_1 \parallel \mathscr{G}_2) \in PR$
$\mathscr{G}_1, \mathscr{G}_2 \in PR$	and	$\mathscr{G}_1^* \in SP$ and $(\mathscr{G}_1^* \,;\, \mathscr{G}_2) \in SP \cap KP$	$\Rightarrow$	$(\mathscr{G}_1^* \,;\, \mathscr{G}_2) \in PR$
$\mathscr{G}_i\{1/i\}, \mathscr{G}_i\{2/i\} \in PR$	and	$(\mathscr{G}_i\{1/i\} \,;\, \mathscr{G}_i\{2/i\}) \in SP$	$\Rightarrow$	$\overset{n}{\underset{i=1}{\circledcirc}} \mathscr{G}_i \in PR$
$\mathscr{G}_i\{1/i\}, \mathscr{G}_i\{2/i\} \in PR$	and	$(\mathscr{G}_i\{1/i\} \oplus \mathscr{G}_i\{2/i\}) \in CP$	$\Rightarrow$	$\overset{n}{\underset{i=1}{\bigoplus}} \mathscr{G}_i \in PR$
$\mathscr{G}_i\{1/i\}, \mathscr{G}_i\{2/i\} \in PR$	and	$(\mathscr{G}_i\{1/i\} \oplus \mathscr{G}_i\{2/i\}) \in CP$	$\Rightarrow$	$\overset{n}{\underset{i=1}{\bigotimes}} \mathscr{G}_i \in PR$
$\mathscr{G}_i\{1/i\}, \mathscr{G}_i\{2/i\} \in PR$	and	$(\mathscr{G}_i\{1/i\} \parallel \mathscr{G}_i\{2/i\}) \in PP$	$\Rightarrow$	$\overset{n}{\underset{i=1}{\parallel}} \mathscr{G}_i \in PR$

9 Conclusions and Future Work

We introduced System-A which allows for parameterized parallelism, where the number of participants, the types of messages sent, as well as the number of such messages are controlled by type parameters. Choice among various execution paths can also be parameterized, so that the number and types of different paths to be taken is not known at compile time. System-A also introduces a shuffling operator, which expresses arbitrary reordering of its arguments, again in a parameterized fashion. A series of examples demonstrates the usefulness of these extensions, which allow us to specify and check previously inexpressible interactions such as the sliding window protocol and parallel resource locking/unlocking (Section 3). In System-A, we can statically verify—without instantiating the parameters—the compliance of implementations to protocols: we do this by first projecting (Section 5) the specification to parameterized types, and then comparing these projections against the types extracted from the program. An important result we obtain is that structural equivalence of types in System-A is decidable; we present this result in Section 6 by first showing weak and subsequently strong normalization of local types. Unlike other typing proposals, System-A does not depend on other theories (typed λ-calculus, system T, or system F) for type-checking. In Section 7 we discuss the conditions under which our projection function is correct and state their sufficiency in Section 8.

Future Work. Complete type checking with System-A is only decidable up to type inference; we do not provide an algorithm for inference of suitable types in an actor language. The design of a programming language along with the relevant type inference algorithm is the next step towards a practical implementation.

Another practical consideration includes semantic comparison of local types. Our normalization algorithm [13] already includes many cases of semantically equivalent, yet structurally differing types. Semantic comparison is unnecessary for the weak normalization proof, but would be useful in a prac-

tical setting where the user is interested in semantic adherence to a protocol. Specifically for the case where reordering of terms is possible as a result of operator commutativity, our suggested coding only serves as an existential proof. A more practical coding scheme could be developed, perhaps employing lexicographic ordering.

Deniélou et al. [14] propose a system where parameterization is achieved by means of quantification over *roles*. Roles are behavior specifications that are taken up by processes while they participate in a protocol, and processes are allowed to join and leave protocols (respectively, adopt and drop roles) dynamically. Their notation's expressiveness is limited when it comes to arbitrary, concurrency-induced interleavings of events. Nevertheless, incorporating their ideas in System-A would greatly expand the applicability of the ideas presented here, towards a different direction than what is addressed in the present paper.

Support for session delegation and exception handling (in the sense of Carbone et al. [10]) represents another opportunity for extension. Furthermore, it may be possible to transfer the recent, precise realizability results [4] for choreographies [33] to our parameterized specifications.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. This publication was made possible in part by sponsorships from the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement number FA8750-11-2-0084, as well as the Army Research Office under Award No. W911NF-09-1-0273. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Gul A. Agha (1990): *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence, MIT Press.

[2] Rajeev Alur, Pavol Cerný, P. Madhusudan & Wonhong Nam (2005): *Synthesis of interface specifications for Java classes*. In Jens Palsberg & Martín Abadi, editors: *POPL*, ACM, pp. 98–109. Available at `http://doi.acm.org/10.1145/1040305.1040314`.

[3] Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum & H. P. Barendregt (1992): *Lambda Calculi with Types*. In: *Handbook of Logic in Computer Science*, Oxford University Press, pp. 117–309.

[4] Samik Basu, Tevfik Bultan & Meriem Ouederni (2012): *Deciding choreography realizability*. In John Field & Michael Hicks, editors: *POPL*, ACM, pp. 191–202. Available at `http://doi.acm.org/10.1145/2103656.2103680`.

[5] Andi Bejleri (2010): *Practical Parameterised Session Types*. In Jin Song Dong & Huibiao Zhu, editors: *ICFEM*, *Lecture Notes in Computer Science* 6447, Springer, pp. 270–286. Available at `http://dx.doi.org/10.1007/978-3-642-16901-4_19`.

[6] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2008): *Global Progress in Dynamically Interleaved Multiparty Sessions*. In van Breugel & Chechik [8], pp. 418–433. Available at `http://dx.doi.org/10.1007/978-3-540-85361-9_33`.

[7] Eduardo Bonelli & Adriana B. Compagnoni (2007): *Multipoint Session Types for a Distributed Calculus*. In Gilles Barthe & Cédric Fournet, editors: *TGC*, *Lecture Notes in Computer Science* 4912, Springer, pp. 240–256. Available at `http://dx.doi.org/10.1007/978-3-540-78663-4_17`.

[8] Franck van Breugel & Marsha Chechik, editors (2008): *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*. Lecture Notes in Computer Science 5201, Springer.

[9] Marco Carbone, Kohei Honda & Nobuko Yoshida (2007): *Structured Communication-Centred Programming for Web Services*. In Rocco De Nicola, editor: *ESOP*, Lecture Notes in Computer Science 4421, Springer, pp. 2–17. Available at http://dx.doi.org/10.1007/978-3-540-71316-6_2.

[10] Marco Carbone, Kohei Honda & Nobuko Yoshida (2008): *Structured Interactional Exceptions in Session Types*. In van Breugel & Chechik [8], pp. 402–417. Available at http://dx.doi.org/10.1007/978-3-540-85361-9_32.

[11] Marco Carbone, Nobuko Yoshida & Kohei Honda (2009): *Asynchronous Session Types: Exceptions and Multiparty Interactions*. In Marco Bernardo, Luca Padovani & Gianluigi Zavattaro, editors: *SFM*, Lecture Notes in Computer Science 5569, Springer, pp. 187–212. Available at http://dx.doi.org/10.1007/978-3-642-01918-0_5.

[12] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini & Luca Padovani (2011): *On Global Types and Multiparty Sessions*. In Roberto Bruni & Jürgen Dingel, editors: *FMOODS/FORTE*, Lecture Notes in Computer Science 6722, Springer, pp. 1–28. Available at http://dx.doi.org/10.1007/978-3-642-21461-5_1.

[13] Minas Charalambides, Peter Dinges & Gul Agha (2012): *Parameterized Concurrent Multi-Party Session Types*. Technical Report, Department of Computer Science, University of Illinois at Urbana–Champaign. Available at http://osl.cs.illinois.edu. In preparation.

[14] Pierre-Malo Deniélou & Nobuko Yoshida (2011): *Dynamic multirole session types*. In Thomas Ball & Mooly Sagiv, editors: *POPL*, ACM, pp. 435–446. Available at http://doi.acm.org/10.1145/1926385.1926435.

[15] Pierre-Malo Deniélou & Nobuko Yoshida (2012): *Multiparty Session Types Meet Communicating Automata*. In Helmut Seidl, editor: *ESOP*, Lecture Notes in Computer Science 7211, Springer, pp. 194–213. Available at http://dx.doi.org/10.1007/978-3-642-28869-2_10.

[16] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou & Nobuko Yoshida (2006): *Bounded Session Types for Object Oriented Languages*. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf & Willem P. de Roever, editors: *FMCO*, Lecture Notes in Computer Science 4709, Springer, pp. 207–245. Available at http://dx.doi.org/10.1007/978-3-540-74792-5_10.

[17] Peter Dinges & Gul Agha (2012): *Scoped Synchronization Constraints for Large Scale Actor Systems*. In Marjan Sirjani, editor: *COORDINATION*, Lecture Notes in Computer Science 7274, Springer, pp. 89–103. Available at http://dx.doi.org/10.1007/978-3-642-30829-1_7.

[18] P. Eugster, T.F.S. Frischbier & S.A.A. Buchmann (2012): *Sound Transformations for Federated Objects*. In Matthew B. Dwyer, editor: *OOPSLA*, ACM. Available at http://www.dvs.tu-darmstadt.de/publications/pdf/transsem.pdf.

[19] Svend Frølund (1996): *Coordinating distributed objects - an actor-based approach to synchronization*. MIT Press.

[20] Svend Frølund & Gul Agha (1993): *A Language Framework for Multi-Object Coordination*. In Oscar Nierstrasz, editor: *ECOOP*, Lecture Notes in Computer Science 707, Springer, pp. 346–360. Available at http://dx.doi.org/10.1007/3-540-47910-4_18.

[21] Simon J. Gay & Malcolm Hole (2005): *Subtyping for session types in the pi calculus*. Acta Inf. 42(2-3), pp. 191–225. Available at http://dx.doi.org/10.1007/s00236-005-0177-z.

[22] Simon J. Gay & Vasco Thudichum Vasconcelos (2010): *Linear type theory for asynchronous session types*. J. Funct. Program. 20(1), pp. 19–50. Available at http://dx.doi.org/10.1017/S0956796809990268.

[23] Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert & Alexandre Z. Caldeira (2010): *Modular session types for distributed object-oriented programming*. In Manuel V. Hermenegildo & Jens Palsberg, editors: *POPL*, ACM, pp. 299–312. Available at http://doi.acm.org/10.1145/1706299.1706335.

[24] E. Giachino, C. Laneve & T. Lascu (2012): *Deadlocks and Livelocks in Concurrent Objects with Futures*. Available at http://www.cs.unibo.it/~laneve/papers/submLockAnalysis.pdf.

[25] Kurt Gödel (1958): *Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes*. Dialectica, p. 280287.

[26] Kohei Honda (1993): *Types for Dyadic Interaction*. In Eike Best, editor: CONCUR, Lecture Notes in Computer Science 715, Springer, pp. 509–523. Available at http://dx.doi.org/10.1007/3-540-57208-2_35.

[27] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: ESOP, Lecture Notes in Computer Science 1381, Springer, pp. 122–138. Available at http://dx.doi.org/10.1007/BFb0053567.

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

[29] Raymond Hu, Nobuko Yoshida & Kohei Honda (2008): *Session-Based Distributed Programming in Java*. In Jan Vitek, editor: ECOOP, Lecture Notes in Computer Science 5142, Springer, pp. 516–541. Available at http://dx.doi.org/10.1007/978-3-540-70592-5_22.

[30] Giuseppe Milicia & Vladimiro Sassone (2005): *Jeeg: temporal constraints for the synchronization of concurrent objects*. Concurrency - Practice and Experience 17(5-6), pp. 539–572. Available at http://dx.doi.org/10.1002/cpe.849.

[31] Dimitris Mostrous, Nobuko Yoshida & Kohei Honda (2009): *Global Principal Typing in Partially Commutative Asynchronous Sessions*. In Giuseppe Castagna, editor: ESOP, Lecture Notes in Computer Science 5502, Springer, pp. 316–332. Available at http://dx.doi.org/10.1007/978-3-642-00590-9_23.

[32] Matthias Neubauer & Peter Thiemann (2004): *Session Types for Asynchronous Communication*. Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.9.9995&rep=rep1&type=pdf.

[33] Chris Peltz (2003): *Web Services Orchestration and Choreography*. IEEE Computer 36(10), pp. 46–52. Available at http://doi.ieeecomputersociety.org/10.1109/MC.2003.1236471.

[34] Robert L. Probert & Kassem Saleh (1991): *Synthesis of Communication Protocols: Survey and Assessment*. IEEE Trans. Computers 40(4), pp. 468–476. Available at http://doi.ieeecomputersociety.org/10.1109/12.88466.

[35] Riccardo Pucella & Jesse A. Tov (2008): *Haskell session types with (almost) no class*. In Andy Gill, editor: Haskell, ACM, pp. 25–36. Available at http://doi.acm.org/10.1145/1411286.1411290.

[36] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou & Sergios Theodoridis, editors: PARLE, Lecture Notes in Computer Science 817, Springer, pp. 398–413. Available at http://dx.doi.org/10.1007/3-540-58184-7_118.

[37] Vasco Thudichum Vasconcelos, Simon J. Gay & António Ravara (2006): *Type checking a multithreaded functional language with session types*. Theor. Comput. Sci. 368(1-2), pp. 64–87. Available at http://dx.doi.org/10.1016/j.tcs.2006.06.028.

[38] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri & Raymond Hu (2010): *Parameterised Multiparty Session Types*. In C.-H. Luke Ong, editor: FOSSACS, Lecture Notes in Computer Science 6014, Springer, pp. 128–145. Available at http://dx.doi.org/10.1007/978-3-642-12032-9_10.

[39] Nobuko Yoshida & Vasco Thudichum Vasconcelos (2007): *Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication*. Electr. Notes Theor. Comput. Sci. 171(4), pp. 73–93. Available at http://dx.doi.org/10.1016/j.entcs.2007.02.056.

[40] M.C. Yuang (1988): *Survey of protocol verification techniques based on finite state machine models*. In: Computer Networking Symposium, 1988., Proceedings of the, pp. 164 –172, doi:10.1109/CNS.1988.4993.