Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, Nobuko Yoshida

Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions

Mario Coppo[1], Mariangiola Dezani-Ciancaglini[1],
Luca Padovani[1], and Nobuko Yoshida[2]

[1] Università di Torino, Dipartimento di Informatica
[2] Imperial College London, Department of Computing

Abstract. Conventional session type systems guarantee progress within single sessions, but do not usually take into account the dependencies arising from the interleaving of simultaneously active sessions and from session delegations. As a consequence, a well-typed system may fail to have progress, even assuming that helper processes can join the system after its execution has started. In this paper we develop a static analysis technique, specified as a set of syntax-directed inference rules, that is capable of verifying whether a system of processes engaged in simultaneously active multiparty sessions has the progress property.

1 Introduction

A system of multiparty sessions has the *global progress property* if all processes in the system that are involved in ongoing sessions do not get stuck waiting for a message that is never sent and if every message sent is eventually consumed. On the one hand, this notion of progress is stronger than requiring that a non-terminated system can always reduce. For example, a system containing two processes engaged in an "infinite chatter" (like two non terminating threads which communicate with each other) does not have the progress property if some other process involved in an open session is stuck and unable to complete its own task. On the other hand, this notion of progress is weaker than requiring that all processes in the system must be able to reduce. For example, a system with an incomplete session, i.e. a session that has not been initiated and for which some participants are missing, does have the progress property if it can be completed with the missing participants to a system that has the progress property.

Communication type systems such as those introduced in [12,6] can check that processes behave correctly with respect to the protocols associated with the single sessions. The same type systems can also assure a local progress property *within the single sessions*, but they fall short in assuring the global progress property when several multiparty sessions are interleaved with each other or the communication topology of the system changes as a consequence of *delegations* across these sessions.

In previous work [6] we have defined an *interaction type system* that, when used in conjunction with the communication type system, can assure the global progress property for processes in a calculus of asynchronous multiparty sessions. The interaction type system pivots around three different typing rules for service initiations. To build the type deduction for a process, provided that one exists, it is crucial, for each service

occurring in the process, to choose the right typing rule. In practice, this means that the interaction type system can be efficiently used only for *verifying* whether a given process has a given type. A naive type inference algorithm based directly on the rules of the type system would require backtracking, resulting in an exponential explosion of the search space. The contribution of the present paper is the definition of a deterministic, compositional inference algorithm which is proved to be sound and complete with respect to the interaction type system of [6]. The algorithm is presented in a "natural deduction" style, as a set of inference rules that can be evaluated in a single-pass analysis according to the structure of processes. The complexity is quadratic in the size of processes, since the application of the rules requires evaluations of linear functions. The basic idea is to devise a suitable data structure that stores the information about all the possible ways a service initiation can be typed in the interaction type system, postponing the commitment to a specific typing rule as long as possible. The inference algorithm refines the information in this data structure discarding the typing rules of service initiations that are found to be incompatible with the structure of the processes being analyzed.

In §2 we define syntax and reduction semantics of the calculus of multiparty sessions. In §3 we illustrate, through a number of smaller examples, various behavioral patterns that we want to consider and how and when these may cause deadlocks. This tutorial informally hints at the information available to the inference algorithm that helps preventing deadlocks and how such information can be inferred from the structure of processes. The inference algorithm and the data structures it uses are described in §4, which ends by showing the algorithm at work on a few examples. Related work is discussed in §5, while §6 concludes with a summary of the results and an account of ongoing and future work.

2 The Calculus of Multiparty Sessions

Syntax. We begin by fixing some notation for the following sets: *service names* are ranged over by a, b, ...; *value variables* are ranged over by x, x', ...; *identifiers*, i.e., service names and variables, are ranged over by u, w, ...; *channel variables* are ranged over by y, z, t, ...; *labels*, functioning like method selectors, are ranged over by l, l', ...; we write $\mathscr{S}$ for the set of all service names and $\mathscr{V}$ for the set of all channel variables. *Processes*, ranged over by P, Q, ..., and *expressions*, ranged over by e, e', ..., are given by the grammar in Table 1, where the syntax occurring only at runtime appears shaded.

The process $\overline{u}[\mathtt{p}](y).P$ initiates a new session through an identifier u with the other participants, each of the form $u[\mathtt{q}](y).Q_{\mathtt{q}}$ where $1 \leq \mathtt{q} \leq \mathtt{p}-1$. The (bound) variable y is the channel used for the private communications inside the session. We call $\mathtt{p}$, $\mathtt{q}$, ... (ranging over natural numbers) the *participants* of the session and we use Π, Π' to denote finite, non-empty sets of participants.

Communications that take place inside an established session are represented using the next three pairs of primitives: the sending and receiving of a value; the sending and receiving of a session channel (where the sender delegates the receiver to participate in a session by passing a channel associated with the session); selection and branching

Table 1. Calculus of multiparty sessions (syntax).

$P ::= \mathbf{0}$	Inaction	$v ::= a \mid \mathsf{true} \mid \mathsf{false}$	Value
$\mid \overline{u}[\mathtt{p}](y).P$	Service request		
$\mid u[\mathtt{p}](y).P$	Service accept	$e ::= x \mid v \mid \cdots$	Expression
$\mid c!\langle \Pi, e\rangle.P$	Send value		
$\mid c?(\mathtt{p}, x).P$	Receive value	$c ::= y \mid s[\mathtt{p}]$	Channel
$\mid c!\langle\langle \mathtt{p}, c'\rangle\rangle.P$	Send channel		
$\mid c?((\mathtt{q}, y)).P$	Receive channel	$m ::= (\mathtt{q}, \Pi, v)$	Value in transit
$\mid c \oplus \langle \Pi, l\rangle.P$	Select	$\mid (\mathtt{q}, \mathtt{p}, s[\mathtt{p}'])$	Session in transit
$\mid c\&(\mathtt{p}, \{l_i : P_i\}_{i \in I})$	Branch	$\mid (\mathtt{q}, \Pi, l)$	Label in transit
$\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Conditional		
$\mid P \mid Q$	Parallel	$h ::= \varnothing \mid h \cdot m$	Queue
$\mid (\nu a : G)P$	Restricted service		
$\mid (\nu s)P$	Restricted session		
$\mid s : h$	Named queue		

(where the former chooses one of the branches offered by the latter). All these operations specify the channel and the index of the sender or the receiver. Thus, $c!\langle \Pi, e\rangle$ sends a value on channel c to all the participants in Π, while $c?(\mathtt{p}, x)$ denotes the intention of receiving a value on channel c from the participant $\mathtt{p}$. The same holds for delegation/reception (but the receiver is only one) and for selection/branching. We write $c!\langle \mathtt{p}, e\rangle.P$ and $c \oplus \langle \mathtt{p}, l\rangle.P$ in place of $c!\langle \{\mathtt{p}\}, e\rangle.P$ and $c \oplus \langle \{\mathtt{p}\}, l\rangle.P$. An *output action* is a value sending, session sending or label selection. An *input action* is a value reception, session reception or label branching; an *input process* is a process prefixed by an input action. The service restrictions are decorated with the global types of the services. Global types describe the communication protocol followed by the session participants; we omit their syntax and refer the interested reader to [6] for the details. Conditional processes and parallel composition are standard.

Queues and channels with role are generated by the operational semantics (see Table 2). A *channel with role* is a pair $s[\mathtt{p}]$ representing the runtime endpoint of session s used by participant $\mathtt{p}$. As in [12], we model TCP-like asynchronous communications (where the message order is preserved and send actions are non-blocking) with unbounded queues of messages in a session, denoted by h. A message in a queue can be a value message $(\mathtt{q}, \Pi, v)$, indicating that the value v was sent by participant $\mathtt{q}$ to the recipients in Π; a channel message (delegation) $(\mathtt{q}, \mathtt{p}, s[\mathtt{p}'])$, indicating that $\mathtt{q}$ delegates to $\mathtt{p}$ the role of $\mathtt{p}'$ on the session s (represented by the channel with role $s[\mathtt{p}']$); and a label message $(\mathtt{q}, \Pi, l)$ (similar to a value message). By $\varnothing$ and $h \cdot m$ we respectively denote the empty queue and the queue obtained by concatenating m to the queue h. With some abuse of notation we will also write $m \cdot h$ to denote the queue with head element m. By $s : h$ we denote the queue h of the session s. In $(\nu s)P$ all occurrences of $s[\mathtt{p}]$ and the queue name s are bound.

We write $\mathsf{fs}(P)$, $\mathsf{fc}(P)$ respectively for the sets of service names and channel names occurring free in P. We define $\mathsf{fn}(P) = \mathsf{fs}(P) \cup \mathsf{fc}(P)$. A *user process* is a process which does not contain runtime syntax.

Table 2. Reduction (selected rules).

$$\prod_{i=1}^{n} a[i](y).P_i \mid \overline{a}[n+1](y).P_{n+1} \to (\nu s)(\prod_{i=1}^{n} P_i\{s[i]/y\} \mid P_{n+1}\{s[n+1]/y\} \mid s : \varnothing) \quad [\textsc{Init}]$$

$$s[\mathtt{p}]!\langle \Pi, e\rangle.P \mid s : h \to P \mid s : h \cdot (\mathtt{p}, \Pi, v) \quad (e \downarrow v) \quad [\textsc{Send}]$$

$$s[\mathtt{p}]!\langle\langle \mathtt{q}, s'[\mathtt{p}']\rangle\rangle.P \mid s : h \to P \mid s : h \cdot (\mathtt{p}, \mathtt{q}, s'[\mathtt{p}']) \quad [\textsc{Deleg}]$$

$$s[\mathtt{p}] \oplus \langle \Pi, l\rangle.P \mid s : h \to P \mid s : h \cdot (\mathtt{p}, \Pi, l) \quad [\textsc{Sel}]$$

$$s[\mathtt{p}]?(\mathtt{q}, x).P \mid s : (\mathtt{q}, \mathtt{p}, v) \cdot h \to P\{v/x\} \mid s : h \quad [\textsc{Rcv}]$$

$$s[\mathtt{p}]?((\mathtt{q}, y)).P \mid s : (\mathtt{q}, \mathtt{p}, s'[\mathtt{p}']) \cdot h \to P\{s'[\mathtt{p}']/y\} \mid s : h \quad [\textsc{SRcv}]$$

$$s[\mathtt{p}] \& (\mathtt{q}, \{l_i : P_i\}_{i \in I}) \mid s : (\mathtt{q}, \mathtt{p}, l_k) \cdot h \to P_k \mid s : h \quad (k \in I) \quad [\textsc{Branch}]$$

Operational Semantics. The operational semantics is defined as the combination of reduction rules expressing actual computation steps and structural equivalence rules that rearrange terms so as to enable reductions. Structural equivalence is almost standard (and therefore omitted). The only peculiar rules allow rearranging the order of messages in a queue when the senders or the receivers are not the same and for splitting a message targeted to multiple recipients. Table 2 shows a selection of the relevant rules for the process reduction relation $P \to P'$. We briefly comment the rules in what follows.

Rule [Init] describes the initiation of a new session involving $n+1$ participants that synchronize over the service name a. Here we use $\prod_{i=1}^{n} P_i$ to denote $P_1 \mid \cdots \mid P_n$. The last participant $\overline{a}[n+1](y).P_{n+1}$, distinguished by the overbar on the service name, determines the number $n+1$ of participants. After the initiation, the participants share a private session name s and the queue associated with s, which is initially empty. The variable y in each participant p is replaced by the corresponding channel with role $s[\mathtt{p}]$. The output rules [Send], [Deleg], and [Sel] respectively push values, channels and labels into the queue of the session s (in rule [Send], the side condition $e \downarrow v$ denotes the evaluation of the expression e to the value v). The input rules [Rcv], [SRcv] and [Branch] perform the corresponding complementary operations. Note that these operations check that the sender of the message matches the expected one so that the message is actually meant for the receiver. Reduction is closed under evaluation contexts, which are special terms with holes $[\,]$ generated by the grammar below:

$$\mathscr{E} \;::=\; [\,] \;\mid\; P \;\mid\; (\nu a : G)\mathscr{E} \;\mid\; (\nu s)\mathscr{E} \;\mid\; (\mathscr{E} \mid \mathscr{E})$$

We write $\mathscr{E}[P_1, \ldots, P_n]$ for $\mathscr{E}$ where the i-th (left-to-right) hole has been filled with P_i.

The Communication Type System. The communication type system checks that processes use service names and channels according to the global types associated with them. It ensures that messages are exchanged in the right order and have the right types within sessions. The communication type system also guarantees progress within a single session, if this session is not interleaved with other sessions, but it cannot guarantee progress when multiple sessions are interleaved. We omit the specification of the communication type system because it is well understood (see [12,6] for details). In fact all processes in this paper are (assumed to be) well typed with respect to the communication type system.

Progress. Informally, we intend that a process has the progress property if each session, once started, is guaranteed to satisfy all the requested interactions. A formal definition of the progress property is not straightforward and the definition in [1] is unsatisfactory in presence of infinite computations. We explain the key ideas and problems separately.

A natural requirement for progress in the case of communication protocols is that an input process can always read a message in the expected queue and vice versa a message in a queue is always read by an input process. Hence, we must assure that any request of interaction on a session channel will always be satisfied. For instance, take the processes:

$$P_1 = a[1](y).b[1](z).y?(2,x).z!\langle 2,x\rangle \quad Q_1 = \overline{a}[2](y).\overline{b}[2](z).z?(1,x').y!\langle 1,x'\rangle$$

The problem of $P_1 \mid Q_1$ is that it reduces to a process in which the output actions of both sessions are prefixed by input actions of the other session. Indeed, $P_1 \mid Q_1$ reduces to

$$(\nu s)(\nu s')(s[1]?(2,x).s'[1]!\langle 2,x\rangle \mid s'[2]?(1,x').s[2]!\langle 1,x'\rangle)$$

where the private sessions s and s' respectively established for the a and b services have replaced the channel variables y and z in P_1 and Q_1. This configuration is stuck because the two processes are blocked mutually waiting for a message from restricted channels. Instead, the process $P_1 \mid Q_1'$ where:

$$Q_1' = \overline{a}[2](y).\overline{b}[2](z).y!\langle 1,\mathsf{true}\rangle.z?(1,x).\mathbf{0}$$

has progress and reduces to $\mathbf{0}$.

Building on Kobayashi's definition of lock-freedom [13] and on the definition of communication safety of [8] we require that each input process will always be able to receive an appropriate message along some computation and that each message in a queue will always be received by an appropriate input process along some computation. However, we must also consider that an incomplete session (i.e., without all the required participants) on service a occurring in a process P can always be allowed to start by composing P with a process containing the missing participants for a. For this reason, we use *catalyser processes* to provide the missing participants to sessions and to make sure that rule [INIT] can always be applied, so that session accept and session request prefixes are never blocking. We omit here the precise definition of catalysers which requires a number of auxiliary definitions (see [6] for the details). Intuitively, a catalyser is a parallel composition of processes where each process implements the behavior of a *single* participant. In particular, in a catalyser it is never the case that actions pertaining to different sessions are interleaved with each other in the same sequential thread. Therefore, catalysers cannot generate deadlocks.

The last notion we need before defining progress is a natural duality between input processes and message queues, which only takes into account top inputs in processes and leftmost messages in queues.

Definition 2.1 (Duality). *The* duality *between input processes and message queues is the least symmetric relation defined by:*

$$\begin{gathered} s[\mathtt{p}]?(\mathtt{q},x).P \bowtie s:(\mathtt{q},\mathtt{p},v)\cdot h \\ s[\mathtt{p}]?((\mathtt{q},y)).P \bowtie s:(\mathtt{q},\mathtt{p},s'[\mathtt{p}'])\cdot h \\ s[\mathtt{p}]\&(\mathtt{q},\{l_i:P_i\}_{i\in I}) \bowtie s:(\mathtt{q},\mathtt{p},l_k)\cdot h \qquad (k\in I) \end{gathered}$$

We are now able to define progress as follows:

Definition 2.2 (Progress). *A process P* has the progress property *if for all catalysers Q such that $P \mid Q$ is well typed in the communication system, if $P \mid Q \to^* \mathscr{E}[R]$, where R is an input process or a non-empty message queue, then there are a catalyser Q', and $\mathscr{E}', R'$ such that $\mathscr{E}[R] \mid Q' \to^* \mathscr{E}'[R, R']$ and $R \bowtie R'$.*

3 A Tutorial to Progress Inference

Service dependencies. The basic idea for preventing deadlocks is to forbid mutual dependencies between services. A dependency between two services originates when an input action pertaining to one of the services guards (hence potentially blocks) any action of the other service. A paradigmatic example of process without progress is $P_1 \mid Q_1$ that we have already examined in §2. Observe that in process P_1 we have an input action on service a that guards an output action on service b. This dependency can be recorded as the relation $a \prec b$ associated with process P_1. In process Q_1 the situation is reversed, determining $b \prec a$. If we take P_1 and Q_1 in isolation, then no circular dependency is detected. However, when considering $P_1 \mid Q_1$, the relations associated with this composition result into the circular dependency $a \prec b \prec a$.

The idea of avoiding circular dependencies between services breaks apart as soon as service names are first-class entities that can be sent as messages. When this happens, the actual dependencies between services may dynamically change as the system evolves and it might happen that a system without circular dependencies *turns into* one with circular dependencies. To illustrate the issue, consider the processes

$$P_2 = c[1](t).t?(2,x).x[1](y).b[1](z).y?(2,x').z!\langle 2,x'\rangle \qquad Q_2 = \overline{c}[2](z).z!\langle 1,a\rangle$$

and observe that Q_2 sends to P_2 the name of service a. The analysis of process P_2 may determine the relation $x \prec b$, because there is an action pertaining to service x that blocks another action pertaining to service b. However, since x is a *bound variable* in P_2, there is no obvious way to associate this dependency with P_2. On the other hand, the analysis of process Q_2 yields no apparent dependencies for a. Overall, no dependency is inferred for $P_2 \mid Q_2$, even though at runtime the system will reduce to a configuration that yields the relation $a \prec b$. Then, if $P_2 \mid Q_2$ is composed with a process that yields the inverse dependency $b \prec a$, a deadlock can occur. Indeed $P_2 \mid Q_2 \mid Q_1$ reduces to $P_1 \mid Q_1$ which leads to a deadlock, as we have seen in §2.

The idea then is to identify a class of services which do not cause deadlocks even when they are involved into circular dependencies, and to allow a service name to be sent as a message only if it refers to a service in this class. A practically relevant class of services with this property is that of *nested* ones, which are characterized by the fact that they can only be blocked by actions pertaining to nested invocations of services that are themselves nested. As an example, consider the processes

$$\begin{aligned} P_3 &= \overline{a}[2](y).y?(1,x).\overline{a}[2](z).z?(1,x').z!\langle 1,\mathsf{true}\rangle.y!\langle 1,\mathsf{false}\rangle \\ Q_3 &= a[1](y).y!\langle 2,\mathsf{false}\rangle.a[1](z).z!\langle 2,\mathsf{true}\rangle.z?(2,x').y?(2,x) \\ R_3 &= a[1](y).y!\langle 2,\mathsf{false}\rangle.a[1](z).y?(2,x).z!\langle 2,\mathsf{true}\rangle.z?(2,x') \end{aligned}$$

and observe that P_3 represents the request of two nested invocations of service a. Observe also that in P_3 there is an input action on channel z that guards an output action on channel y and that both actions pertain to the service a. As a consequence, these dependencies result in the relation $a \prec a$ that denotes a circular dependency. However, P_3 has a peculiar structure in that all the actions related to the innermost invocation of a are completely nested within the ones related to the outermost invocation of a. More generally, there is no blocking action of the outermost invocation of a that is interleaved with actions of the innermost invocation of a. In fact, this interaction structure closely resembles an ordinary function call of a sequential programming language, where a caller function is suspended until the callee has terminated. The point is that if all request and accept operations concerning service a follow this pattern (i.e., they are not interleaved with blocking actions from other sessions), then the process P_3 cannot deadlock even if its structural analysis establishes the circular dependency $a \prec a$. For example, also Q_3 gives rise to the same circular dependency, but it follows the same structure as P_3 and the composition $P_3 \mid Q_3$ is deadlock free. By contrast, in R_3 we notice that, after the innermost invocation of a has been accepted, there is an input action on y, which pertains to the outermost invocation, blocking the actions pertaining to the innermost one. Indeed, the composition $P_3 \mid R_3$ yields a deadlock.

Relative and Nested services. To promote P_3 (and Q_3) among the safe processes, we associate services with different *features* and we impose different constraints on the structure of services depending on the features they have. We say that a service that is never involved in circular dependencies with other services has the R (for Relative) feature. A service a where no action from other sessions can block the sessions initiated on a has the N (for Nested) feature. This is precisely the case of the innermost invocation of a in P_3 and Q_3. But there is more: if the innermost session cannot deadlock, it becomes "unobservable" as far as the dependency analysis is concerned so we can say that also the outermost invocation of a in P_3 and Q_3 is not blocked by actions of other sessions. As a consequence, the outermost service a has the N feature as well.

The N feature may also be used for dealing with circular dependencies between *different* services. As an example, consider the processes

$$P_4 = \overline{a}[2](y).\overline{b}[2](z).z?(1,x).y?(1,x') \qquad Q_4 = \overline{b}[2](z).\overline{a}[2](y).y?(1,x).z?(1,x')$$

representing two clients which, for unspecified reasons, request the two services a and b in different orders. If P_4 and Q_4 run within the same system, then they immediately yield the circular dependencies $a \prec b \prec a$. Still, if the processes implementing a and b (not shown here) are independent, in the sense that they do not rely on each other, then there is no danger of deadlock. The fundamental observation here is that neither service seems to have the N feature if considered in isolation: each service request is blocked by an action from the other service. However, if b is *assumed* to have the N feature, then a has the N feature also, and vice versa. In other words, the circular dependency $a \prec b \prec a$ identifies a clique of services that is safe (i.e., deadlock-free) if *every* service in the clique has the N feature under the hypothesis that all the others do as well.

In general, the same service may have both the R and the N features at the same time. This is the case of a and b in P_4 and Q_4 above *when each process is considered*

in isolation. However, note that the b service in P_1 has the R feature but not the N one, while neither a nor b has the R feature in $P_4 \mid Q_4$. This observation is crucial for the inference algorithm because the fact that a service *a does not have* a particular feature may affect other services related to a by the dependency relation. In particular, if $a \prec b$ and b does not have the R feature (hence it has the N one), then a cannot have the R feature (and it must have the N one). Dually, if a does not have the N feature, then b cannot have the N feature.

Bounded services. The next usage pattern that we wish to consider concerns private services. Take for example the process

$$b[1](y).(\nu a : 1 \to 2 : \langle \mathsf{bool} \rangle)(\overline{a}[2](z).z?(1,x).y!\langle 2, \mathsf{false} \rangle)$$

where the a service has been restricted and is therefore inaccessible from the outside. Even if the a service has both the R and N features, the fact that it is restricted makes it observably equivalent to the idle process. This has severe consequences on the outer service b, because the output action on channel y cannot be executed. In essence, we devise a third feature B (for Bounded) associated with services that can be restricted and that prevents them to be followed by *any* communication action on free channels.

Wrap up. To summarize, when we analyze a system of interleaved multiparty sessions we associate services in the system with (a combination of) three features R, N, and B:

- A service has the R feature if it never generates circular dependencies with other services it is interleaved with.
- A service has the N feature if it is never interleaved with blocking actions from other services not having the N feature.
- Finally, a service has the B feature if it has the N feature and it is never followed by any action on free communication channels.

Overall there are eight feature combinations. One of these corresponds to the fact that a service has none of the features outlined above. In this case, the service will be rejected by our system as being ill typed. Furthermore, having the B feature implies having the N feature. Therefore, each well-typed service may be classified into one of five feature combinations. Note that, in the informal definitions above, "never" means both "for no occurrence of the service in the system" and "at any time during the evolution of the system". The inference algorithm has to find a trade off between flexibility (the number of systems for which progress can be guaranteed) and feasibility (the analysis is solely based on the initial state of the system). In fact, when discussing first-class service names we have already seen a case in which the algorithm is forced to act conservatively due to the lack of precise information about the runtime evolution of a system.

The inference of the progress property performs an analysis on the structure of processes, keeping track of the dependencies between services and incrementally refining the features associated with services, making sure that each service has at least one of the features described above. Initially, each service has every feature. As the analysis proceeds bottom up on the structure of processes, features are removed from services

that are found to be incompatible with them. In a nutshell, the most relevant refinement steps taken by the algorithm occur at the following events:

- As soon as a circular dependency is detected, all processes involved in the circularity (and those preceding them in the dependency relation) lose the R feature.
- When a process of shape $\tilde{a}[\mathtt{p}](y).P$ is encountered, where $\tilde{a}$ is either an accept action a or a request action $\overline{a}$, a loses the N feature if it is not minimal in the dependency relation (meaning that it may be blocked by another session of a service not having the N feature). Also, a loses the B feature if P has free channels other than y.
- When a process of shape $y?(\mathtt{p},x).P$ is encountered, the dependencies are enriched with relations $y \prec z$ for every channel z that occurs free in P. The same happens for session receives and branching processes, since these are all blocking actions.
- When a process of shape $P \mid Q$ is encountered, the dependencies computed for P and those computed for Q are merged together, while the features for every service in the overall process are those in common between P and Q. Similar operations are performed when analyzing branching and conditional processes, where multiple processes come together.
- When a process of shape $y!\langle \mathtt{p}, a\rangle.P$ is encountered, the service a loses the R feature.
- Special measures must be taken when channels are communicated. These will be detailed shortly.

The next section is devoted to formalizing all the concepts and procedures outlined in this tutorial.

4 Progress Inference

In this section we introduce a deterministic, compositional type inference algorithm, defined via a set natural semantics rules, assuring that a given *user* process has the progress property. As we have anticipated in §3, the basic idea of the inference algorithm is to keep track of dependencies between services.

A *service qualifier* is either a service name a or a channel variable y; we write $\Lambda = \mathscr{S} \cup \mathscr{V}$ for the set of all service qualifiers; we let λ range over elements of Λ and $\mathscr{L}$ over subsets of Λ.

A *dependency relation* is a transitive relation $\mathsf{D} \subseteq \Lambda \times \Lambda$. We denote with $\lambda \prec \lambda'$ the elements of $\Lambda \times \Lambda$. The meaning of $\lambda \prec \lambda'$ is, roughly, that an input action on the channel (or on the channel bound by service) λ can block a communication action on the channel (or on the channel bound by service) λ'.

The inference algorithm makes use of some auxiliary operators for D that are introduced below:

- $\mathsf{D} \downarrow \lambda \stackrel{\text{def}}{=} \{\lambda\} \cup \{\lambda' \mid \lambda' \prec \lambda \in \mathsf{D}\}$ is the set of elements that are smaller than or equal to λ in D, namely the set of service qualifiers having an input action that can block a communication action on λ, plus λ itself.
- $\mathsf{D} \uparrow \lambda \stackrel{\text{def}}{=} \{\lambda\} \cup \{\lambda' \mid \lambda \prec \lambda' \in \mathsf{D}\}$ is the symmetric operation that determines the set of service qualifiers that may be blocked by an input action on λ, plus λ itself.
- $\mathsf{D} \setminus \mathscr{L} \stackrel{\text{def}}{=} \{\lambda \prec \lambda' \in \mathsf{D} \mid \lambda \notin \mathscr{L} \wedge \lambda' \notin \mathscr{L}\}$ is the subset of D pertaining to all the service qualifiers not occurring in $\mathscr{L}$.

Table 3. Inference algorithm for the interaction type system.

$$\{\textsc{Inact-I}\}\quad \mathbf{0} \Mapsto \emptyset;\mathscr{S};\mathscr{S};\mathscr{S}$$

$$\{\textsc{Init*-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}{\tilde{a}[\mathsf{p}](y).P \Mapsto \mathfrak{F}(\mathsf{D}\{a/y\}^+,\mathsf{R},\mathsf{N},\mathsf{B}\setminus\{a \mid \mathsf{fc}(P) \not\subseteq \{y\}\})}$$

$$\{\textsc{InitV-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B} \qquad \mathsf{fc}(P)\subseteq\{y\}}{\tilde{x}[\mathsf{p}](y).P \Mapsto \mathfrak{F}(\mathsf{D}\setminus\{y\},\mathsf{R}\setminus(\mathsf{D}\downarrow y),\mathsf{N},\mathsf{B})}$$

$$\{\textsc{NRes-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B} \qquad a\in\mathsf{B}}{(\nu a:G)P \Mapsto \mathsf{D}\setminus\{a\};\mathsf{R}\setminus\{a\};\mathsf{N}\setminus\{a\};\mathsf{B}\setminus\{a\}}$$

$$\{\textsc{Send-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}{y!\langle\Pi,e\rangle.P \Mapsto \mathfrak{F}(\mathsf{D},\mathsf{R}\setminus\{e\},\mathsf{N},\mathsf{B})}$$

$$\{\textsc{Rcv-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}{y?(\mathsf{q},x).P \Mapsto (\mathsf{pre}(y,\mathsf{fc}(P))\cup\mathsf{D})^+;\mathsf{R};\mathsf{N};\mathsf{B}}$$

$$\{\textsc{Deleg-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}{y!\langle\langle\mathsf{p},z\rangle\rangle.P \Mapsto (\{y\prec z\}\cup\mathsf{D})^+;\mathsf{R};\mathsf{N};\mathsf{B}}$$

$$\{\textsc{SRcv-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B} \qquad \mathsf{D}\setminus\mathscr{S}\subseteq\{y\prec z\}}{y?((\mathsf{q},z)).P \Mapsto \mathsf{D}\setminus\{z\};\mathsf{R};\mathsf{N};\mathsf{B}}$$

$$\{\textsc{Sel-I}\}\quad \frac{P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}{y\oplus\langle\Pi,l\rangle.P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}}$$

$$\{\textsc{Branch-I}\}\quad \frac{P_i \Mapsto \mathsf{D}_i;\mathsf{R}_i;\mathsf{N}_i;\mathsf{B}_i\ {}^{(i\in I)} \qquad \mathsf{D}=(\mathsf{pre}(y,\bigcup_{i\in I}\mathsf{fc}(P_i))\cup\bigcup_{i\in I}\mathsf{D}_i)^+}{y\&(\mathsf{p},\{l_i:P_i\}_{i\in I}) \Mapsto \mathfrak{F}(\mathsf{D},\bigcap_{i\in I}\mathsf{R}_i,\bigcap_{i\in I}\mathsf{N}_i,\bigcap_{i\in I}\mathsf{B}_i)}$$

$$\{\textsc{Par-I}\}\quad \frac{P_i \Mapsto \mathsf{D}_i;\mathsf{R}_i;\mathsf{N}_i;\mathsf{B}_i\ {}^{(i=1,2)} \qquad \mathsf{D}=(\mathsf{D}_1\cup\mathsf{D}_2)^+}{P_1 \mid P_2 \Mapsto \mathfrak{F}(\mathsf{D},\mathsf{R}_1\cap\mathsf{R}_2,\mathsf{N}_1\cap\mathsf{N}_2,\mathsf{B}_1\cap\mathsf{B}_2)}$$

$$\{\textsc{If-I}\}\quad \frac{P_i \Mapsto \mathsf{D}_i;\mathsf{R}_i;\mathsf{N}_i;\mathsf{B}_i\ {}^{(i=1,2)} \qquad \mathsf{D}=(\mathsf{D}_1\cup\mathsf{D}_2)^+}{\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \Mapsto \mathfrak{F}(\mathsf{D},\mathsf{R}_1\cap\mathsf{R}_2,\mathsf{N}_1\cap\mathsf{N}_2,\mathsf{B}_1\cap\mathsf{B}_2)}$$

– $\mathsf{D}^\infty \stackrel{\text{def}}{=} \{\lambda \mid \lambda \prec \lambda \in \mathsf{D}\}$ is the set of service qualifiers involved in circular dependencies in D.

We extend $\downarrow$ and $\uparrow$ to sets $\mathscr{L}$ of service qualifiers in the natural way. We also write $\mathsf{D}\{a/y\}$ for the relation obtained from D where every occurrence of y has been replaced by a and $\mathfrak{R}^+$ for the transitive closure of a generic relation $\mathfrak{R}$.

The inference rules prove judgments of the form $P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}$, where D is a dependency relation and R, N, and B are sets of service names. As a first approximation, we can think of the services in these sets as those that respectively have the R, N, and B feature. However, for services that are communicated in messages it is not easy to *statically* guarantee that they will not be involved in a circular dependency *at runtime*. Therefore, we conservatively remove communicated services from the R set even if they are not explicitly involved in circular dependencies.

A judgment $P \Mapsto \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}$ is *well formed* if:

1. If a service a has the R feature, then all the services following a in D have R feature. Also, no service involved in a circular dependency can have the R feature. This is formally expressed as $\mathsf{D} \uparrow \mathsf{R} \subseteq \mathsf{R} \setminus \mathsf{D}^{\infty} \cup \mathscr{V}$.
2. If a service a has the N feature, then all service qualifiers preceding a in D must be services with the N feature. That is, $\mathsf{D} \downarrow \mathsf{N} \subseteq \mathsf{N}$.
3. The set of services having the B feature is included in those having the N feature. That is, $\mathsf{B} \subseteq \mathsf{N}$.
4. All services occurring free in P have at least the R or the N feature. If some service in P has neither the R nor the N feature, then our inference algorithm does not guarantee the progress property for P. That is, $\mathsf{fs}(P) \subseteq \mathsf{R} \cup \mathsf{N}$.

In general, the inference rules add dependencies to the D relation and remove service names from the R, N, B sets when these services lose features. To be sure that the quadruple resulting from the application of an inference rule still satisfies the conditions (1–3) above, we define a function $\mathfrak{F}$ that, given a quadruple D, R, N, B, computes a new one where services are removed from the sets R, N, B whenever they are found to be incompatible with the corresponding feature:

$$\mathfrak{F}(\mathsf{D},\mathsf{R},\mathsf{N},\mathsf{B}) \stackrel{\text{def}}{=} \mathsf{D};\mathsf{R}';\mathsf{N}';\mathsf{B} \cap \mathsf{N}'$$

where $\mathsf{R}' = \{a \in \mathsf{R} \mid \mathsf{D} \uparrow a \subseteq \mathsf{R} \setminus \mathsf{D}^{\infty} \cup \mathscr{V}\}$ and $\mathsf{N}' = \{a \in \mathsf{N} \mid \mathsf{D} \downarrow a \subseteq \mathsf{N}\}$.

Table 3 defines the inference for the interaction type system. We implicitly assume that an inference rule can be applied only if the judgment in the conclusion is well formed. In the next paragraphs we describe each inference rule in detail.

{INACT-I} is by far the simplest inference rule, which yields no dependencies and poses no constraints on the features of services. In particular, D is $\emptyset$ and the R, N, and B components are the full set $\mathscr{S}$ of service names.

{INIT*-I} is used for typing accept and request operations on a known service name a (recall that we use $\tilde{a}$ for either a or $\overline{a}$). The rule computes a new quadruple $\mathfrak{F}(\mathsf{D}\{a/y\}^{+},\mathsf{R},\mathsf{N},\mathsf{B} \setminus \{a \mid \mathsf{fc}(P) \setminus \{y\} \neq \emptyset\})$ from the one obtained by typing the continuation process P, where $\mathsf{D}\{a/y\}^{+}$ replaces the channel variable y with a in D so that all the dependencies already established for a are enriched with those computed for y. Also, a loses the B feature if P contains free channels other than y.

{INITV-I} is analogous to {INIT*-I}, but considers the case in which the session is initiated on an unknown service x. Because nothing is known on the service a that will replace x at runtime, the rule acts conservatively assuming that a has both the N and the B features. In particular, the continuation process P is required to have no free channel other than y (this is necessary if a has the B feature) and all services preceding y in D lose the R feature (this is necessary if a has the N feature but not the R one). Note that it is not possible to keep track, in D, of all the dependencies related to y as we did in {INIT*-I}. In fact, any dependency related to y in D is removed. This may prevent the inference algorithm from statically detecting circular dependencies for services that are communicated in messages. For this reason, we will require that all service names communicated by rule {SEND-I} must have the N feature (Example 4.2 shows that this is necessary for communicated services to prevent deadlocks).

When a service name a is restricted in a process P, rule {NRES-I} checks that a has the B feature. Then, all dependencies related to a and a itself are removed from all the components of the quadruple in the conclusion of the rule.

Rules {SEND-I} and {SEL-I} do not change the dependency relation because send operations are non-blocking. In the case of {SEND-I}, however, we must check that if the message sent e is a service name, then it cannot have the R feature. The application of the $\mathfrak{F}$ function makes sure that all the components of the quadruple remain consistent after this removal.

Rule {RCV-I} is used for typing value receptions. In this case, only the dependency relation is changed to record the fact that the input action on channel y may block subsequent actions on the free channels occurring in P. The function $\mathsf{pre}(y, \mathsf{fc}(P))$ creates the dependency relation that contains the pairs $y \prec z$ for all $z \in \mathsf{fc}(P)$. Note that no dependency is recorded between y and the free service names possibly occurring in P. This is because these services can always be unblocked by adding suitable catalysers (see Definition 2.2) provided that the communication occurring on y does not reach a deadlock.

Rule {BRANCH-I} is a natural generalization of rule {RCV-I} to a process with multiple branches. In this case, the dependencies inferred for each branch are merged together and services lose those features that are not present in every branch.

Rule {DELEG-I} is similar to {SEND-I} and {SEL-I} in that it deals with a non-blocking send operation. However, in this case the process is sending a channel variable z over channel y, meaning that an action blocking a communication on y may also block a communication on z, because z cannot be used by the receiver process until delegation happens. Consequently, the dependency relation is enriched with the $y \prec z$ dependency.

Rule {SRCV-I} is similar to {RCV-I}, except that it is used for typing the reception of a session channel. The rule is particularly restrictive because it is meant to prevent a dangerous phenomenon called self-delegation, which happens when one process ends up owning two (or more) endpoints of the same session. An example of this phenomenon is shown in the processes

$$P_5 = b[1](z).a[1](y).y!\langle\langle 2, z\rangle\rangle \qquad Q_5 = \overline{b}[2](z).\overline{a}[2](y).y?((1,x)).x?(2,w).z!\langle 1, \mathsf{false}\rangle$$

which, when executed in parallel, open two sessions on services a and b. Then, P_5 sends the channel z related to the session on b over the channel y, which is related to the session on a. At this point, Q_5 owns both endpoints of the session on b and tries to use them in an order that causes a deadlock. Indeed, $P_5 \mid Q_5$ reduces to

$$(\nu s)(s[1]?(2,w).s[2]!\langle 1, \mathsf{false}\rangle)$$

which is stuck. Remarkably, the process $P_5 \mid Q_5$ is typable in the communication type system hence it is the interaction type system that must detect the problem in this case. The premise $\mathsf{D} \setminus \mathscr{S} \subseteq \{y \prec z\}$ requires that the continuation process P_5 cannot perform any potentially blocking action on any channel other than y, and that if a potentially blocking action is performed on y then it must necessarily block a communication action on z. This restriction prevents self-delegation and, in general, suffices to guarantee progress. Note that P_5 is still allowed to open new sessions on other services.

{PAR-I} and {IF-I} conclude the inference system by suitably combining dependencies and features, similarly to what we have already seen for the {BRANCH-I} rule.

The algorithm is quadratic in the size of processes, being defined on their structure, if we use appropriate data structures to represent the dependency relation and the service sets, getting linear complexity for the evaluation of the required functions.

The algorithm is sound, namely:

Theorem 4.1. *If $P \Vdash \mathsf{D};\mathsf{R};\mathsf{N};\mathsf{B}$, then P has the progress property.*

This theorem can be proved by showing that the inference algorithm is sound and complete with respect to the interaction type system defined in [6].

We end with the application of the inference algorithm on two examples used earlier.

Example 4.1. Below are two executions of the inference algorithm on P_1 and Q_1 of §2. For the sake of readability, we develop the inference bottom up assuming $\mathscr{S} = \{a,b\}$.

P_1	D	R	N	B	
$z!\langle 2,x\rangle$	$\emptyset$	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	{SEND-I}
$y?(2,x)$	$\{y \prec z\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	{RCV-I}
$b[1](z)$	$\{y \prec b\}$	$\{a,b\}$	$\{a\}$	$\{a\}$	{INIT*-I}
$a[1](y)$	$\{a \prec b\}$	$\{a,b\}$	$\{a\}$	$\{a\}$	{INIT*-I}

Q_1	D	R	N	B	
$y!\langle 1,x'\rangle$	$\emptyset$	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	{SEND-I}
$z?(1,x')$	$\{z \prec y\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	{RCV-I}
$\overline{b}[2](z)$	$\{b \prec y\}$	$\{a,b\}$	$\{a,b\}$	$\{a\}$	{INIT*-I}
$\overline{a}[2](y)$	$\{b \prec a\}$	$\{a,b\}$	$\{a,b\}$	$\{a\}$	{INIT*-I}

From the above table it turns out that both P_1 and Q_1 are well typed in isolation, in particular we have $P_1 \Vdash \{a \prec b\};\{a,b\};\{a\};\{a\}$ and $Q_1 \Vdash \{b \prec a\};\{a,b\};\{a,b\};\{a\}$ but the application of rule {PAR-I} fails since $\mathfrak{F}(\mathsf{D},\{a,b\},\{a\},\{a\}) = (\mathsf{D},\emptyset,\emptyset,\emptyset)$ where $\mathsf{D} = \{a \prec b, b \prec a\}^+$, and the resulting judgment would not satisfy condition 4 of the definition of well formedness. In particular the circular dependency removes the R feature from both a and b and the N feature is removed from b in P_1 and then also from a in the composition $P_1 \mid Q_1$ because of $b \prec a$ (see the definition of $\mathfrak{F}$). ■

Example 4.2. The inference algorithm is not always able to statically determine a violation of the R feature, therefore it is unsafe to leave service names that are sent as messages in the R set. Below is the result of the inference algorithm on the processes P_2 and Q_2 of §3 assuming $\mathscr{S} = \{a,b,c\}$:

P_2	D	R	N	B	
$z!\langle 2,x'\rangle$	$\emptyset$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	{SEND-I}
$y?(2,x')$	$\{y \prec z\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	{RCV-I}
$b[1](z)$	$\{y \prec b\}$	$\{a,b,c\}$	$\{a,c\}$	$\{a,c\}$	{INIT*-I}
$x[1](y)$	$\emptyset$	$\{a,b,c\}$	$\{a,c\}$	$\{a,c\}$	{INITV-I}
$t?(2,x)$	$\emptyset$	$\{a,b,c\}$	$\{a,c\}$	$\{a,c\}$	{RCV-I}
$c[1](t)$	$\emptyset$	$\{a,b,c\}$	$\{a,c\}$	$\{a,c\}$	{INIT*-I}

Q_2	D	R	N	B	
$t!\langle 1,a\rangle$	$\emptyset$	$\{b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	{SEND-I}
$\overline{c}[2](t)$	$\emptyset$	$\{b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	{INIT*-I}

Note that the dependency $y \prec b$ in P_2 is erased because it concerns an unknown service x that is bound in P_2. This means that b is actually involved in dependencies $a \prec b$ for every service a that is sent to P_2, which is precisely what Q_2 does. Indeed we have $P_2 \mid Q_2 \Vdash \emptyset;\{b,c\};\{a,c\};\{a,c\}$ but $P_2 \mid Q_2 \mid Q_1$, where Q_1 is defined in Example 4.1, cannot be typed. In fact, adding c to the set of services we get immediately $Q_1 \Vdash \{b \prec a\};\{a,b,c\};\{a,b,c\};\{a,c\}$ but rule {PAR-I} cannot be applied since $\mathfrak{F}(\{b \prec a\},\{b,c\},\{a,c\},\{a,c\}) = (\{b \prec a\},\{b,c\},\{c\},\{c\})$ does not satisfy the condition 4 of the definition of well-formedness for service a. Indeed we have the reduction $P_2 \mid Q_2 \mid Q_1 \rightarrow^* P_1 \mid Q_1$ which leads to a deadlock, as we have seen in §3. ■

5 Related Work

Our notion of progress is strongly related to, and partly inspired from, the notion of *lock-freedom* in [13], where Kobayashi develops a type system to enforce it. Intuitively, a process is lock-free if, no matter how it reduces, every top-level prefix can be eventually consumed. In our case this roughly corresponds to the property that no process gets stuck on an input action and that every message in a queue can be received. Kobayashi's type system seems capable of a much more fine-grained analysis than our type system. However, despite the similarities between progress and lock-freedom, the two type systems are difficult to compare, because of several major differences in both processes and types. In addition to the fact that we consider progress modulo the availability of catalysers, our type system is given for an asynchronous language with a native notion of (multiparty) session, while Kobayashi's type system is defined for a basic variant of the synchronous, pure π-calculus. A natural way for comparing these analysis techniques would require compiling a session-based process into the pure π-calculus [7], and then using Kobayashi's type system for reasoning on progress of the original process in terms of lock-freedom of the one resulting from the compilation.

A strategy that is alternative to compiling/encoding session-based processes is to lift the technique underlying Kobayashi's type system to a session type system for reasoning directly on the progress properties of processes. Some preliminary experiments in this sense are reported in [14].

Most papers on service-oriented calculi only assure that clients are never stuck inside a *single* session [12,9,8]. The first papers considering progress for interleaved sessions required the nesting of sessions in Java [11,5].

The papers more related to the present one are [10] and [3]. In both these papers there are constructions of processes providing missing participants, which are simpler than our catalysers since sessions are dyadic.

[2] proposes a sophisticated proof system which builds a well-founded ordering on events to enforce progress for processes of the Conversation Calculus [15], also in presence of dynamic join and leave of participants. Their progress is guaranteed under the assumption that all communications are matched with sufficient joiners.

Formal theories of contracts using multiparty interaction structures are studied in [4]. Contracts record the overall behaviour of a process, and typable processes themselves may not always satisfy properties such as progress: it is proved *later* by checking whether a whole contract satisfies a certain form. Proving properties with contracts requires an exploration of all possible interleaved or non-deterministic paths of a protocol.

6 Conclusions and Future Work

We have presented a sound and complete inference algorithm for the interaction type system defined in [6] restricted to finite processes. This system guarantees progress of interleaved multiparty sessions with session delegation and service communication.

There is a number of extensions stemming from this work, we focus on two of them. First of all, it appears that the algorithm can be easily adapted to deal with recursive processes, although soundness and completeness of such extension remain to be formally

established. Second, we plan to investigate how the approach can be applied to concrete programming languages. The point is that the inference algorithm (and the interaction type system as well) makes the fundamental assumption that a process can be examined in terms of the complete sequence of input/output operations it performs. In practice, programs are made of opaque structures (higher-order functions, methods, modules, etc.) and it is currently unclear whether such structures can be faithfully encoded as processes in our calculus, or if instead it is necessary to devise richer type constructs to describe them and to reason on global progress of systems in a modular way.

Acknowledgments. The authors are grateful to the reviewers for their useful comments and to Naoki Kobayashi for discussions on the notion of lock-freedom. This work was partially supported by EPSRC EP/G015635/1 and EP/K011715/1, NSF Ocean Observatories Initiative, MIUR Project CINA and Ateneo/CSP Project SALT.

References

1. Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR'08*, LNCS 5201, pages 418–433. Springer, 2008.
2. Luís Caires and Hugo Torres Vieira. Conversation types. *Theoretical Computer Science*, 411(51-52):4399–4440, 2010.
3. Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In *ICE*, volume 38 of *EPTCS*, pages 13–27, 2010.
4. Giuseppe Castagna and Luca Padovani. Contracts for Mobile Processes. In *CONCUR'09*, LNCS 5710, pages 211–228. Springer, 2009.
5. Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31. Springer, 2007.
6. Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*. to appear.
7. Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP'12*, pages 139–150. ACM Press, 2012.
8. Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic Multirole Session Types. In *POPL'11*, pages 435–446. ACM Press, 2011.
9. Mariangiola Dezani-Ciancaglini and Ugo de' Liguoro. Sessions and Session Types: an Overview. In *WS-FM'09*, volume 6194 of *LNCS*, pages 1–28. Springer, 2010.
10. Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Nobuko Yoshida. On Progress for Structured Communications. In *TGC'07*, LNCS 3912, pages 257–275. Springer, 2008.
11. Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.
12. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM Press, 2008.
13. Naoki Kobayashi. A Type System for Lock-Free Processes. *Information and Computation*, 177:122–159, 2002.
14. Luca Padovani. From Lock Freedom to Progress Using Session Types. In *PLACES'13*, 2013. to appear.
15. Hugo Torres Vieira, Luis Caires, and Joao C. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In *ESOP'08*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.