Typing Liveness in Multiparty Communicating Systems

Luca Padovani, Vasco T. Vasconcelos, Hugo Torres Vieira

Typing Liveness in Multiparty Communicating Systems

Luca Padovani[1], Vasco Thudichum Vasconcelos[2] and Hugo Torres Vieira[2]

[1] Dipartimento di Informatica, Università di Torino, Italy
[2] LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

Abstract

Session type systems are an effective tool to prove that communicating programs do not go wrong, ensuring that the participants of a session follow the protocols described by the types. In previous work, we have introduced a typing discipline for the analysis of progress in binary sessions. In this paper, we generalize the approach to multiparty sessions following the conversation type discipline, while strengthening progress to liveness. Conversation types allow to discipline interaction in systems where a possibly unanticipated number of multiple participants interact using a single medium of communication. We combine the usual session-like fidelity analysis with the liveness analysis and devise an original treatment of recursive types allowing us to address challenging configurations that are out of the reach of existing approaches.

1 Introduction

The importance of error detection in the early cycles of software development, and the consequent savings arising from it, can never be overemphasized. The problem becomes even more acute when concurrency comes into play, for concurrency faults are notoriously hard to track down. This work concentrates on early error detection of concurrent, message passing systems, and addresses, apart from the usual communication safety and session fidelity, the static identification of states in which processes wait forever for messages that are never sent.

The setting in which we operate is that of (multi-party) sessions [2, 3, 4, 8, 10, 11, 12]. *Sessions* are private conversations occurring between two or more interacting participants. Each participant behaves according to a *session type* that describes the messages that the participant is supposed to send/receive and their relative order. One of the strengths of sessions is that they provide a structuring construct on top of which complex systems can be built in a modular way. The relatively simple typing discipline imposed by session types ensures strong properties such as *liveness*, that is the eventual completion of communication operations. This point in favor of sessions is also, somewhat paradoxically, a weakness: since session types describe only *intra*-session communications, but say nothing on *inter*-session dependencies, it may be the case that a well-typed participant simultaneously involved in two or more sessions finds itself in a deadlock because of mutual dependencies between sessions. We address this problem by capturing potentially dangerous dependencies between sessions, so that liveness is ensured also when communications on several different sessions are interleaved.

To illustrate the basic ingredients of our approach, consider the process

$$(\nu s)(\,\mathsf{rec}\,\mathcal{X}.s?\,x.\mathcal{X} \mid \mathsf{rec}\,\mathcal{X}.s?\,y.\mathcal{X} \mid \mathsf{rec}\,\mathcal{X}.s!\,5.s!\,\mathsf{true}.\mathcal{X}\,) \tag{1}$$

describing three participants (say `A`, `B`, and `C`, composed in parallel) that interact within the scope of a multiparty session s. The aim of `C` is to repeatedly send two messages (here exemplified as the constants 5 and true) respectively to `A` and `B`. Since all these participants interact within the same session s, however, the order of the synchronizations cannot be predicted and

it may well be the case that 5 messages are received by B and true messages are received by A or, in fact, that one of A or B does not receive any message at all! In order to recover the *linearity* of communications (there is at most one possible synchronization in a session channel at a given moment) we tag messages with *labels* following the approach of [4]. In particular, we refine (1) to

$$(\nu s)(\,\mathsf{rec}\,\mathcal{X}.s?\mathtt{l}\,x.\mathcal{X} \mid \mathsf{rec}\,\mathcal{X}.s?\mathtt{m}\,y.\mathcal{X} \mid \mathsf{rec}\,\mathcal{X}.s!\mathtt{l}\,5.s!\mathtt{m}\,\mathsf{true}.\mathcal{X}\,) \tag{2}$$

so that l- and m-tagged messages respectively and uniquely identify synchronizations with A and B. We are then able to characterize the overall protocol that takes place on session s as the type

$$T_s \triangleq \mu\alpha.\,\tau\,\mathtt{l}\,\mathsf{int}.\,\tau\,\mathtt{m}\,\mathsf{bool}.\alpha$$

saying that the conversation consists of an infinite exchange of alternated l- and m-tagged messages whose payload is described by the types int and bool, respectively. The occurrences of τ in the type denote *synchronizations* that are supposed to occur in a session typed by T_s. To specify the behavior of the participants involved in the conversation, we *split* T_s into "slices" which we distribute among the participants. First of all, we separate the behavior of C from the rest of the system, and obtain

$$T_s = T_{\mathtt{C}} \circ T' \quad \text{where} \quad T_{\mathtt{C}} \triangleq \mu\alpha.\,!\,\mathtt{l}\,\mathsf{int}.\,!\,\mathtt{m}\,\mathsf{bool}.\alpha \quad \text{and} \quad T' \triangleq \mu\alpha.\,?\,\mathtt{l}\,\mathsf{int}.\,?\,\mathtt{m}\,\mathsf{bool}.\alpha$$

In particular, $T_{\mathtt{C}}$ says that C repeatedly sends alternated l and m messages and T' says that the rest of the system should be ready to receive the very same messages, in this order. Then, we further split T' in the behaviors of A and B, thus:

$$T' = T_{\mathtt{A}} \circ T_{\mathtt{B}} \quad \text{where} \quad T_{\mathtt{A}} \triangleq \mu\alpha.\,?\,\mathtt{l}\,\mathsf{int}.\alpha \quad \text{and} \quad T_{\mathtt{B}} \triangleq \mu\alpha.\,?\,\mathtt{m}\,\mathsf{bool}.\alpha$$

Note that this splitting is valid *assuming* that the environment in which A and B execute guarantees that the synchronization on each l message occurs before the synchronization on each m message. This is indeed guaranteed by the sequential structure C. Since $T_{\mathtt{A}}$, $T_{\mathtt{B}}$, and $T_{\mathtt{C}}$ match the behaviors of A, B, and C with respect to s we can declare that process (2) is well typed and consequently that it enjoys *communication safety* (no message with wrong type is ever sent), *session fidelity* (the interactions follow the protocol described by T_s), and *liveness* (each interaction described in T_s eventually occurs).

Of all these properties, liveness is the most delicate one, in the sense that it may easily break up when two or more sessions are interleaved with each other. To illustrate the issue, consider the following refinement of (2)

$$(\nu s)(\,(\nu r)(\,\mathsf{rec}\,\mathcal{X}.\underline{r?\mathtt{m}\,y}.s?\mathtt{l}\,x.\mathcal{X} \mid \mathsf{rec}\,\mathcal{X}.s?\mathtt{m}\,y.\underline{r!\mathtt{m}\,y}.\mathcal{X}) \mid \mathsf{rec}\,\mathcal{X}.s!\mathtt{l}\,5.s!\mathtt{m}\,\mathsf{true}.\mathcal{X}\,) \tag{3}$$

in which A and B are engaged in another session r, different from s, while C behaves exactly as before. Now B forwards y in a m-tagged message to A, maybe so that A and B can double-check that they are given consistent information from C. Session s is still well typed according to T_s and session r is well typed according to $T_r \triangleq \mu\alpha.\,\tau\,\mathtt{m}\,\mathsf{bool}.\alpha$. Yet, (3) is stuck because A waits for the message from B *before* having received the message from C, but C sends its message to B only *after* it has successfully delivered the message to A. So, none of the synchronizations in T_s and T_r ever happens, although the structure of the participants in (3) agrees to these types.

One possibility for detecting the problem in (3) stems from the observation that the two sessions s and r are mutually dependent on each other. So, one may devise a static analysis technique that keeps track of inter-session dependencies and flags any system that gives rise to circularities as ill typed. This approach has been pursued, for instance, in [3, 6, 7]. The limit

of this approach is that, by considering sessions as atomic units, it is quite coarse grained when it comes to analyzing dependencies. For instance,

$$(\nu s)(\,(\nu r)(\,\mathsf{rec}\,\mathcal{X}.s?\mathtt{l}\,x.\underline{r?\mathtt{m}\,y.}\mathcal{X}\mid \mathsf{rec}\,\mathcal{X}.s?\mathtt{m}\,y.\underline{r!\mathtt{m}\,y.}\mathcal{X})\mid \mathsf{rec}\,\mathcal{X}.s!\mathtt{l}\,5.s!\mathtt{m}\,\mathsf{true}.\mathcal{X}\,) \tag{4}$$

is a simple variation of (3) where `A` performs the same two inputs, but in the "correct" order. Also in (4) there are actions on session s interleaving with actions on session r *and* vice versa, so the approach based on session dependencies also flags (4) as ill typed, which is unfortunate because (4), contrarily to (3), enjoys liveness.

The approach we pursue here is based on the idea of tracking the dependencies between *actions* instead of sessions. To this aim, we annotate each interaction in a type with an identifier—which we call *event*—and we keep track of the dependencies between events by means of *strict partial order* $\prec$. To get the flavor of the technique at work, let us apply it to the sessions s and r discussed above. First of all, we annotate the types of s and r with three events e, f, and g:

$$s : \mu\alpha.e\,\tau\,\mathtt{l}\,\mathsf{int}.f\,\tau\,\mathtt{m}\,\mathsf{bool}.\alpha \qquad r : \mu\alpha.g\,\tau\,\mathtt{m}\,\mathsf{bool}.\alpha$$

Then, we analyze the dependencies between the actions in the participants of (3): it must be $e \prec f$ (read, e precedes f) because `C` first sends the `l`-tagged message, and only then it sends the `m`-tagged message; it must be $g \prec e$ because `A` waits for the `m`-tagged message before waiting for the `l`-tagged one; finally, it must be $f \prec g$ by looking at the structure of `B`. Overall, $\prec$ is not a strict partial order because of the circularity in the relation $g \prec e \prec f \prec g$ between the two sessions s and r, hence (3) is ill typed.

Our approach builds on previous works [13, 14, 16, 21] that use analogous annotations for reasoning on the dependencies between actions. With respect to these works, our contributions are along two major vectors. First of all, we show that the techniques can be applied to sessions/conversations with an arbitrary (and possibly variable) number of participants. Second, we support complex recursive process structures. The latter aspect requires a non-trivial extension of the technique described in [21] because, in order to declare that a system like (4) is well typed, we must be able to distinguish occurrences of the same event that pertain to different iterations of a recursive process.

The next section formally describes our language. Section 3 introduces the type system and the main results. Section 4 concludes the paper including a more detailed comparison with related work and hints on future developments.

2 Process Model

We consider an infinite set of *names* ranged over by x, y, ... representing communication channels, an infinite set of *process variables* ranged over by $\mathcal{X}$, ..., a set of *message labels* l, ... Processes, ranged over by P, Q, ..., are the terms defined by the grammar in Fig. 1. The language is that of TyCO [19] that extends the π-calculus [15] by considering labelled communication. The terms $\mathbf{0}$, $P \mid Q$, and $(\nu x)P$ respectively denote the inactive process, the parallel composition of P and Q, and the restriction of name x in P. Terms $\mathsf{rec}\,\mathcal{X}.P$ and $\mathcal{X}$ are used to construct recursive processes. The term $x!l\,y.P$ denotes a process that sends a message on channel x and then continues as P. A *message* is made of a label l and an argument y. The term $x?\{l_i y_i.P_i\}_{i\in I}$ denotes a process that waits for a message from channel x and then continues as P_i according to the label of the received message. The argument of the received message replaces the name y_i in P_i. To keep the setting as simple as possible, we have not included conditional or non-deterministic processes. These constructs can be easily added.

The binders of the language are name restriction $(\nu x)P$, which binds the name x in P, the input prefix $x?l\,y.P$, which binds the name y in P, and the recursion $\mathsf{rec}\,\mathcal{X}.P$, which binds the

$$
\begin{array}{rcllll}
P, Q & ::= & \mathbf{0} & \text{(Inaction)} & \mid & x!l\,y.P \quad \text{(Output)} \\
& \mid & P \mid Q & \text{(Parallel)} & \mid & x?\{l_i y_i.P_i\}_{i \in I} \quad \text{(Input Summation)} \\
& \mid & (\nu x)P & \text{(Restriction)} & \mid & \mathsf{rec}\,\mathcal{X}.P \quad \text{(Recursion)} \\
& \mid & \mathcal{X} & \text{(Recursion Variable)} & &
\end{array}
$$

Figure 1: Syntax of processes

$$
\frac{k \in I}{x?\{l_i y_i.P_i\}_{i \in I} \mid x!l_k\, z.Q \to P_k\{z/y_k\} \mid Q} \qquad \frac{P \to Q}{(\nu x)P \to (\nu x)Q} \qquad \text{(R-Com,R-New)}
$$

$$
\frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q} \qquad \text{(R-Par,R-Cong)}
$$

Figure 2: Reduction relation

recursion variable $\mathcal{X}$ in P. The notions of free and bound names (as well as free and bound process variables) are defined in the usual way. We identify processes modulo renaming of bound names and of bound process variables.

The semantics of the language is defined via a structural congruence and a reduction relation. Structural congruence is standard, except that it includes the law $\mathsf{rec}\,\mathcal{X}.P \equiv P\{\mathsf{rec}\,\mathcal{X}.P/\mathcal{X}\}$ for unfolding recursive processes. Note that the unfolding of recursive processes is captured by structural congruence, where $P\{Q/\mathcal{X}\}$ denotes the capture-avoiding substitution of the free occurrences of $\mathcal{X}$ by process Q in P.

Reduction is defined by the rules in Fig. 2. Rule (R-Com) describes the synchronization of two processes exchanging a message: the sender emits a message with a label l_k that is among those accepted by the receiver and the argument z of the message replaces the bound input parameter in the appropriate continuation P_k of the receiver. The remaining rules close the relation under language contexts—name restriction and parallel composition—as well as under structural congruence.

Having presented our process model, which we consider to be the simplest possible setting that allows to discipline multiparty interaction, at least considering a session-type based approach, we are in a position to present our type system.

3 Type System

This section starts by introducing the notion of strict partial orders which allows to identify well-formed communication dependencies in processes. It then introduces types and operations on these, most notably type split which allows to separate a type in two disjoint "slices" of behavior. Processes are typed against a conventional typing context and an ordering for the events in the types. Finally it presents the main results of the paper, namely soundness of the type system (Theorem 1) and liveness (Theorem 2).

Strict partial orders. We consider an infinite set of event identifiers $\mathcal{E}$ and the set of natural numbers $\mathbb{N}$, and use $a, b, \ldots$ to range over $\mathcal{E}$ and $n, m, \ldots$ to range over $\mathbb{N}$. We use a^n to denote an element in set $\mathcal{E} \times \mathbb{N}$. We further introduce a distinguished event, $\top$, use $e, f, \ldots$ to range over $(\mathcal{E} \times \mathbb{N}) \cup \{\top\}$, and call this set the *set of events*. A *strict (or irreflexive) partial order* $\prec$ over the set of events is a binary relation that is asymmetric (hence irreflexive) and transitive. We write $e \prec f$ when the pair (e, f) is in $\prec$, and $\mathit{supp}(\prec)$ for the *support* of $\prec$, namely the set of events that occur in $\prec$.

Next we define two *partial* operations over strict partial orders. We write $e + \prec$ for the strict partial order obtained by *adding a least event* e to $\prec$, provided that e does not occur in

p	$::=$	$! \mid ? \mid \tau$	(Polarity)	B	$::=$	end	(Stop)
T	$::=$	$a^n\, p\, l\, T$	(Shared Type)		$\mid$	$B_1 \mid B_2$	(Parallel)
	$\mid$	$B^{\prec}$	(Linear Type)		$\mid$	$\mu\alpha.B$	(Recursion)
Γ	$::=$	$\cdot \mid \Gamma, x\colon T$	(Context)		$\mid$	α	(Variable)
Δ	$::=$	$\cdot \mid \Delta, X\colon (\Gamma; \prec)$	(Recursion Context)		$\mid$	$a^n\, p\{l_i\, T_i.B_i\}_{i\in I}$	(Prefix Summation)

Figure 3: Syntax of types and typing contexts

$supp(\prec)$. Formally, $e + \prec \triangleq \prec \cup \{(e, f) \mid f \in supp(\prec)\} \cup \{(e, \top)\}$, where we explicitly add the pair $(e, \top)$ since $\prec$ may be empty (in which case $e + \emptyset$ is defined as $\{(e, \top)\}$). We write $\prec_1 \uplus \prec_2$ for the least strict partial order that includes both $\prec_1$ and $\prec_2$, if it exists. We use $\uplus$ to gather the communication dependency structures of, e.g., two parallel processes.

Types. The syntax of types is given in Fig. 3. Our types are based on conversation types [4] extended with event annotations following the approach introduced in [21]. A polarity p describes a communication capability: ! specifies an output; ? specifies an input; and τ specifies a synchronization, i.e., a matched communication pair (cf., [4]). At the type level we distinguish two separate categories of channels: *shared* (or unrestricted) channels – described by $a^n\, p\, l\, T$ – are used for modeling (possibly persistent) services having a publicly known name, with which sessions can be established; *linear* channels – described by $B^{\prec}$ – are used for modeling the private conversations within sessions. Note that the distinction between shared and linear channels appears at the type level only, while they are treated uniformly in the process model. For the sake of simplicity we omit non-channel types (e.g., `Int`) which could be easily added.

A type $a^n\, p\, l\, T$ describes the behavior of a shared channel via an event a^n, a polarity p, a message label l and a type T describing the message argument. We associate shared types with events to temporally relate shared communications with others, in particular with the communications specified by the message type T. We include the exponent n for the sake of uniformity w.r.t. linear types (which exploit it) so as to simplify the presentation of the orderings, and omit them when specifying shared types. For example, the type $a\,?\texttt{service}\,T_1$ describes a channel which offers an input on label `service`, receiving a channel which is then used according to T_1, associated with event identifier a, while $b\,!\texttt{remoteservice}\,T_2$ describes a channel which outputs (with event b) on `remoteservice` a channel to be used accordingly to type T_2.

A type $B^{\prec}$ captures the linear usage of a channel: B specifies the *behavior* of a process w.r.t. the channel, whereas $\prec$ specifies the ordering of events expected from the external environment. Informally, $\prec$ is used in a type $B^{\prec}$ to represent the sequentiality information that B admits but does not impose. For example, when typing a process that concurrently sends messages `hello` and `bye` the type *may* specify that the outputs on `hello` and `bye` actually take place one after the other if such order is imposed by the corresponding inputs (present in the external process environment).

Behavioral types B include inaction end, parallel composition $B_1 \mid B_2$ of two independent behaviors B_1 and B_2, recursive types $\mu\alpha.B$, recursion variables α, and (prefixed) summation $a^n\, p\{l_i\, T_i.B_i\}_{i\in I}$. Sums capture communication capabilities associated with event a^n, polarity p, and a menu of synchronization options. Each entry in the menu is identified by a distinct label l_i, the type of the argument of the message T_i, and the behavior B_i that takes place after the synchronization. We say a linear type $B^{\prec}$ is well-formed if $supp(\prec)$ does not include events associated with communication actions of polarity τ in B (since no further ordering information can be provided for such actions by the external environment). In the remainder whenever we write $B^{\prec}$ we assume that $B^{\prec}$ is well-formed. We also identify α-equivalent (recursive) types

by convention.

Following the ideas presented in [21], we associate with each linear communication an event a^n so as to temporally relate the communication action described by the summation with respect to others, establishing an overall ordering of communications. In this work, we introduce the notion of *iteration*, by adding to events a natural number n, allowing to describe infinite chains of (related) events. Informally, the exponent allows to capture the several "stages" of a type by means of an *increment*, so, for example, $\mu\alpha.a^1\,\tau\,\mathtt{l}\,T.b^1\,\tau\,\mathtt{m}\,T'.\alpha$ unfolds to $a^1\,\tau\,\mathtt{l}\,T.b^1\,\tau\,\mathtt{m}\,T'.\mu\alpha.a^2\,\tau\,\mathtt{l}\,T.b^2\,\tau\,\mathtt{m}\,T'.\alpha$ so as to associate the first iteration (of l m synchronizations) with exponent 1 and the second iteration with exponent 2 and so on and so forth.

Operations on types. We write $\mathit{labels}(B)$ for the set of labels occurring in B. We say that B_1 and B_2 are *behaviorally independent*, notation $B_1 \# B_2$, if $\mathit{labels}(B_1) \cap \mathit{labels}(B_2) = \emptyset$, so that non-interference is captured by disjoint message label sets. We also need an operation to *remove* part of the partial order in a type, defined as $B^{\prec'} \setminus \prec \triangleq B^{\prec' \setminus \prec}$ for linear types, and $T \setminus \prec \triangleq T$ for shared types. Since linear types may contain sequentiality assumptions, we use this operation to clear hypotheses that are proved externally.

In order to capture the several iterations of a communication that may repeat itself in the context of recursion, we introduce an operator that *increments* the exponent associated with an event by a given factor, defined as $\mathit{inc}(a^n, m) \triangleq a^{n+m}$ and $\mathit{inc}(\top, m) \triangleq \top$. We then extend inc to strict partial orders, pointwise, and to behavior types so that $\mathit{inc}(a^n\,p\{l_i\,T_i.B_i\}_{i\in I}, m) \triangleq a^{n+m}\,p\{l_i\,\mathit{inc}(T_i, m).\mathit{inc}(B_i, m)\}_{i\in I}$. The increment of a behavior is an homomorphism for all other constructs. The increment operation on types affects only linear types, $\mathit{inc}(B^{\prec}, m) \triangleq \mathit{inc}(B, m)^{\mathit{inc}(\prec, m)}$, as events associated with shared communications (including the ones specified in the message type) are not considered to be repeated in different stages (and consequently ordered) but rather to be repeated always at the same stage, hence $\mathit{inc}(a^n\,p\,l\,T, m) \triangleq a^n\,p\,l\,T$. Essentially, we model shared communication repetition using replication (via recursion), so shared replicated communication actions have the same temporal ordering, while we model linear recursive repetition (necessarily) using a sequential chain of events. The exponent is then used to capture repetition (without cycles) in the orderings.

To simplify the typing rules, we define a type equivalence relation $\equiv$ that includes commutativity, associativity and neutral end for $\mid$, as well as iso-recursive equivalence for recursive types $\mu\alpha.B \equiv B\{\mu\alpha.\mathit{inc}(B, m)/\alpha\}$ for some $m > 0$, saying that the next iteration of the behavior is captured by the increment of the events (we use any positive m so as to support misalignment between processes and types, and otherwise use $m = 1$).

We now introduce operations that capture the temporal ordering prescribed by types. We write $\mathit{events}(B)$ for the set of elements of $\mathcal{E} \times \mathbb{N}$ occurring in a behavior B, not including the events in message types. Formally:

$$\mathit{events}(B) \triangleq \begin{cases} \emptyset & \text{if } B = \mathsf{end} \text{ or } B = \alpha \\ \mathit{events}(B_1) \cup \mathit{events}(B_2) & \text{if } B = B_1 \mid B_2 \\ \{e\} \cup \bigcup_{i\in I} \mathit{events}(B_i) & \text{if } B = e\,p\{l_i\,T_i.B_i\}_{i\in I} \\ \{\mathit{inc}(e, k) \mid e \in \mathit{events}(B') \text{ and } k \geq 0\} & \text{if } B = \mu\alpha.B' \end{cases}$$

Notice that $\mathit{events}(\mu\alpha.B)$ includes all the events in the body of the recursion, incremented zero or more times so as to capture the first and the following iterations. We extend the operation to types, by defining $\mathit{events}(B^{\prec}) \triangleq \mathit{events}(B)$, as we are only interested in linear types where $\mathit{supp}(\prec) \subseteq \mathit{events}(B)$, and by defining $\mathit{events}(e\,p\,l\,T) \triangleq \{e\}$.

We write $B{\downarrow}$ for the strict partial order over $\mathcal{E} \times \mathbb{N}$ induced by a type B. Notice that $B{\downarrow}$ is

a partial operator since it uses $\cup$ and $+$. Formally:

$$B{\downarrow} \triangleq \begin{cases} \emptyset & \text{if } B = \mathsf{end} \text{ or } B = \alpha \\ B_1{\downarrow} \cup B_2{\downarrow} & \text{if } B = B_1 \mid B_2 \\ e + (\cup_{i\in I} B_i{\downarrow}) & \text{if } B = e\,p\{l_i\,T_i.B_i\}_{i\in I} \\ \{(inc(e,k), inc(f,k)) \mid (e,f) \in B'{\downarrow} \text{ and } k \geq 0\} & \\ \quad \cup \{(inc(e,m), inc(f,n)) \mid e,f \in events(B') \text{ and } m < n\} & \text{if } B = \mu\alpha.B' \end{cases}$$

Notice that the operation adds a least event in the case of the prefix summation, and for recursion adds all pairs obtained from the body of the recursion (incremented zero or more times) and all pairs that pertain to different iterations. We extend the definition to types by taking $(e\,p\,l\,T){\downarrow} \triangleq (e, \top)$ and $B^{\prec}{\downarrow} \triangleq B{\downarrow} \setminus \prec$, where $\setminus$ denotes set difference. The definition for linear types considers the order obtained from the behavioral type removing the ordering expected from the external environment, so $B^{\prec}{\downarrow}$ characterizes exclusively the ordering imposed by the type.

To identify types that characterize channels that do not depend on the external environment to evolve, and hence are "self-sustained" communication wise, we introduce a predicate that is true of types containing no unmatched communication actions. We say that a behavioral type B is *matched* if it contains no top-level (i.e., excluding message types) input or output polarities. We extend the definition to linear types by considering $matched(B^{\prec}) \triangleq matched(B)$ (which, by well-formedness, implies $\prec = \emptyset$), and $matched(e\,p\,l\,T) \triangleq p = ?$. A message type of polarity ? says that a shared input is available. Since we are only interested in capturing continuously available shared inputs, ? shared types "absorb" (as will be clear from the definition of type splitting) ! shared types, so as to capture the fact that (replicated) shared inputs are still available after synchronization. Hence, the definition of *matched*() for shared types excludes solely (unmatched) shared outputs, and considers the (infinitely available) shared inputs to be matched (regardless whether they are used or not).

Typing contexts. The syntax of typing contexts is given in Fig. 3. We assume by convention that, in a typing context $\Gamma, x\colon T$ and in a recursion environment $\Delta, \mathcal{X}\colon (\Gamma; \prec)$, the name x and the process variable $\mathcal{X}$ do not occur in Γ and in Δ, respectively, as usual. Also, we consider contexts up to permutations of their entries.

We let Γ_{un} range over contexts that contain only outputs on shared channels and linear types with end behavior, that is, if $x\colon T$ is in Γ_{un}, then T is either $e\,!\,l\,T'$ or $\mathsf{end}^{\emptyset}$. We use such contexts to describe systems that only use resources that may be used exponentially, namely to describe (the continuation of) processes that input on shared channels. We exclude shared inputs from Γ_{un} in order to avoid "nested" shared inputs, so that inputs on shared channels are continuously active (cf. uniform receptiveness [1, 18]). Similarly, we let Γ_{lin} range over contexts that contain only linear types, that is, types of the form $B^{\prec}$.

We are interested in systems where all communications are matched, i.e., typed in *matched* contexts, defined as the pointwise extension of the *matched* predicate on types. We also lift the notions of type *increment*, *inc*, type *equivalence*, $\equiv$, and partial order difference, $\setminus$, pointwise to contexts.

Splitting and conformance. We now introduce two notions crucial to our development, namely *splitting* (inspired by [2] and by the *merge* operation of [4]) that explains how behaviors can be decomposed and safely distributed to distinct parts of a process (e.g., to the branches of a parallel composition), and *conformance* that captures the desired relation between typing contexts and strict partial orders.

We say type T conforms to order $\prec$, noted $conforms(T, \prec)$, if $T{\downarrow} \subseteq \prec$. Notice that since $T{\downarrow}$ excludes the ordering expected from the external environment, conformance focuses on

$$\frac{}{B^{\prec} = B^{\prec} \circ \mathsf{end}^{\emptyset}} \qquad \frac{B_1 \# B_2 \quad \forall_{i\in\{1,2\}}\ {B_i}^{\prec_i' \uplus \prec_i''} = {B_i'}^{\prec_i'} \circ {B_i''}^{\prec_i''}}{{B_1 \mid B_2}^{\uplus_{i\in\{1,2\}}(\prec_i' \uplus \prec_i'')} = {B_1' \mid B_2'}^{\uplus_{i\in\{1,2\}}\prec_i'} \circ {B_1'' \mid B_2''}^{\uplus_{i\in\{1,2\}}\prec_i''}} \quad \text{(L-End,L-Par)}$$

$$\frac{\forall_{j\in\{1,2\}}\ {T^j = e\, p_j\{l_i\, T_i.B_i^j\}_{i\in I}}^{\uplus_{i\in I}\prec_i^j} \quad p_1 = ? \quad p_2 = ! \quad \forall_{i\in I}\ {B_i}^{\prec_i^1 \uplus \prec_i^2} = {B_i^1}^{\prec_i^1} \circ {B_i^2}^{\prec_i^2}}{{e\,\tau\{l_i\, T_i.B_i\}_{i\in I}}^{\uplus_{i\in I}(\prec_i^1 \uplus \prec_i^2)\setminus(T^1\downarrow \uplus T^2\downarrow)} = T^1 \circ T^2} \quad \text{(L-Sync)}$$

$$\frac{T = e\, p\{l_i\, T_i.B_i'\}_{i\in I}^{\uplus_{i\in I}\prec_i} \quad \forall_{i\in I}\ {B_i}^{\prec_i \uplus \prec} = {B_i'}^{\prec_i} \circ B^{\prec} \quad e\, p\{l_i\, T_i.\mathsf{end}\}_{i\in I} \# B}{{e\, p\{l_i\, T_i.B_i\}_{i\in I}}^{(\uplus_{i\in I}\prec_i)\uplus\prec\uplus((e+events(B))\setminus(T\downarrow\uplus B^{\prec}\downarrow))} = T \circ B^{\prec}} \quad \text{(L-Break)}$$

$$\frac{B^{\prec_1 \uplus \prec_2} = {B_1}^{\prec_1} \circ {B_2}^{\prec_2}}{\mu\alpha.B^{\prec_1\uplus\prec_2} = \mu\alpha.{B_1}^{\prec_1} \circ \mu\alpha.{B_2}^{\prec_2}} \qquad \frac{}{\alpha^{\emptyset} = \alpha^{\emptyset} \circ \alpha^{\emptyset}} \quad \text{(L-Rec,L-Var)}$$

Figure 4: Linear type splitting

$$\frac{p \in \{?, !\}}{e\,?\,l\,T = e\,?\,l\,T \circ e\,p\,l\,T}\text{(S-In-L)} \qquad \frac{}{\cdot = \cdot \circ \cdot}\text{(C-Empty)} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon T = \Gamma_1, x\colon T \circ \Gamma_2}\text{(C-Left)}$$

$$\frac{}{e\,!\,l\,T = e\,!\,l\,T \circ e\,!\,l\,T}\text{(S-Out)} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \quad T = T_1 \circ T_2}{\Gamma, x\colon T\ = \Gamma_1, x\colon T_1 \circ \Gamma_2, x\colon T_2}\text{(C-Split)}$$

Figure 5: Shared type splitting

Figure 6: Context splitting

the order imposed by the types (which is the focus of the overall ordering). The *conforms* predicate is defined on typing contexts as the pointwise extension of the predicate on types, so $\mathit{conforms}(\Gamma, \prec)$ ensures that every communication action specified in Γ is ordered by $\prec$.

Splitting is defined both on types and on typing contexts. We write $T = T_1 \circ T_2$ to mean that type T is split in types T_1 and T_2, and likewise for $\Gamma = \Gamma_1 \circ \Gamma_2$. Linear type splitting, shared type splitting and context splitting are given by the rules in Figs. 4—6 (where we omit symmetric rules).

We briefly describe the rules in Fig 4. A behavioral type may be split in itself and in end, so as to allow, e.g., to give away the behavior completely to one branch of a parallel composition—rule (L-End). A parallel composition $B_1 \mid B_2$ (where B_1 and B_2 are apart $\#$) may be split in two parallel compositions, the components of which are obtained by decomposing B_1 and B_2 together with the ordering assumptions—rule (L-Par). A synchronized (τ) prefix summation may be split in prefix summations with dual polarities (? and !) whose continuations are obtained by splitting the synchronized prefix summation continuations—rule (L-Sync)—where further ordering assumptions can be present in (each) one of the splitted types, just as long such assumptions are proved by the other splitted type. We also ensure that there are no assumptions left regarding the event associated with the prefix e as the type is matched and no further ordering information can be provided by the external context.

A prefix summation may be split in an independent ($\#$) behavior, obtained by splitting (all) the continuations, and in the prefix summation whose continuations specify the remaining behavior—rule (L-Break)—where some extra care with the assumptions is required: since this rule may decompose a type in such a way that the overall ordering is not guaranteed by the splitted types (the split takes place at the level of the continuations), we ensure that such ordering assumptions are present in the overall type via a relation between the event of the prefix e with all events of the B type, where no assumption that is proved by the splitted

$$\frac{\Delta;\Gamma_1;\prec_1 \vdash P \quad \Delta;\Gamma_2;\prec_2 \vdash Q}{\Delta;\Gamma_1\circ\Gamma_2;\prec_1\cup\prec_2 \vdash P\,|\,Q} \qquad \frac{\Delta;\Gamma,x\colon T;\prec \vdash P \quad \mathit{matched}(T)}{\Delta;\Gamma;\prec \vdash (\nu x)P} \qquad \text{(T-Par,T-New)}$$

$$\frac{}{\Delta;\Gamma_{un};\emptyset \vdash \mathbf{0}} \qquad \frac{\Delta,\mathcal{X}\colon(\mathit{inc}(\Gamma_{lin},n);\mathit{inc}(\prec,n));\Gamma_{lin};\prec \vdash P \quad n\in\mathbb{N}}{\Delta;\Gamma_{lin};\prec \vdash \mathsf{rec}\,\mathcal{X}.P} \qquad \text{(T-Inact,T-LRec)}$$

$$\frac{\Delta,\mathcal{X}\colon(\Gamma;\prec);\Gamma;\prec \vdash P \quad \Gamma=\Gamma_{un},x\colon e\,?\,l\,T \quad \prec=(e+\prec'')\cup\prec' \quad e\notin \mathit{supp}(\prec')}{\Delta;\Gamma;\prec \vdash \mathsf{rec}\,\mathcal{X}.P} \qquad \text{(T-URec)}$$

$$\frac{\Delta(\mathcal{X})=(\Gamma;\prec) \quad \mathit{conforms}(\Gamma,\prec)}{\Delta;\Gamma;\prec \vdash \mathcal{X}} \qquad \frac{\Delta,\Gamma_2;\prec \vdash P \quad \Gamma_1\equiv\Gamma_2}{\Delta;\Gamma_1;\prec \vdash P} \qquad \text{(T-Var,T-Equiv)}$$

$$\frac{\forall_{i\in I}\ \Delta;\Gamma,x\colon {B_i}^{\prec'_i},y_i\colon T_i;\prec_i \vdash P_i}{\Delta;\Gamma,x\colon e\,?\{l_i\,T_i.B_i\}_{i\in I}{}^{\cup_{i\in I}\prec'_i};\ e+(\cup_{i\in I}\prec_i) \vdash x?\{l_i y_i.P_i\}_{i\in I}} \qquad \text{(T-LinIn)}$$

$$\frac{\Delta;\Gamma,x\colon {B_k}^{\prec'};\prec \vdash P \quad k\in I}{\Delta;(\Gamma,x\colon e\,!\{l_i\,T_i.B_i\}_{i\in I}{}^{\prec'})\circ y\colon T_k;\ e+(\prec\ \cup\ T_k\downarrow) \vdash x!l_k\,y.P} \qquad \text{(T-LinOut)}$$

$$\frac{\Delta;\Gamma_{un},x\colon e\,?\,l\,T,y\colon T;e+\prec\ \vdash P}{\Delta;\Gamma_{un},x\colon e\,?\,l\,T;e+\prec\ \vdash x?l\,y.P} \qquad \frac{\Gamma=(\Gamma_{un},x\colon e\,!\,l\,T)\circ y\colon T \quad \cdot;\cdot;\emptyset\vdash P}{\Delta;\Gamma;e+T\downarrow\ \vdash x!l\,y.P} \qquad \text{(T-UIn,T-UOut)}$$

Figure 7: Typing rules

types is present ($T\downarrow \cup\, B^{\prec}\downarrow$). A recursive type is split in two recursive types, the bodies of which are obtained by splitting the body of the incoming recursive type, as well as the ordering assumptions—rule (L-Rec). Also, a recursion variable (without assumptions) may be split in itself—rule (L-Var).

Shared type splitting (Fig. 5) decomposes shared communication capabilities in two distinct ways, depending on whether the polarity of the incoming type is ? or !. A shared input is split in a shared input and either in an output or another input, via rule (S-In-L). Essentially, the latter allows for typing processes that separately offer the input capability (e.g., a service that is provided by two distinct sites), and the former allows for typing processes that offer the dual communication capabilities (e.g., a service provider and a service client). A shared output is split in two shared outputs—rule (S-Out)—which allows for typing processes that offer the output capability separately (e.g., two clients of some service). Notice type splitting preserves the message types and event association so as to guarantee the dual communication actions agree on the type of what is communicated and on the ordering.

Context splitting (Fig. 6) allows to divide a context in two distinct ways: context entries either go into the left or the right outgoing contexts—(C-Left) and the omitted symmetric—or they go in both contexts—(C-Split). The latter form lifts the (type) behavior distribution to the context level, while the former allows to delegate the entire behavior to a part of the process, leaving no usage at all to the other part. To lighten notation we use $\Gamma_1\circ\Gamma_2$ to represent any Γ such that $\Gamma=\Gamma_1\circ\Gamma_2$ (if such Γ exists). Notice that, given Γ_1 and Γ_2, there may be more than one Γ such that $\Gamma=\Gamma_1\circ\Gamma_2$.

Typing system. We may now present our type system which characterizes processes according to typing assumptions for the free process variables (in Δ), for the names (in Γ), and an overall ordering of events $\prec$. We say process P is well-typed if $\Delta;\Gamma;\prec\vdash P$ is derivable using the rules in Fig. 7.

We briefly comment on the rules in Fig 7. In rule (T-Par) the parallel composition is typed if the branches are typed in splittings ($\circ$) of the context and a decomposition of the order (which is ensured to be sound via $\uplus$). In rule (T-New) the name restriction is typed if the process in the scope of the restriction is typed in the same contexts together with the typing assumption for the usage of the restricted name which must be a *matched* type (all communication prefixes are matched). Notice that the overall ordering $\prec$ is preserved, hence the ordering prescribed by name x is still present in the conclusion, even though the type T of x is not.

In rule (T-Inact) the inaction process is typed with any usage of recursion variables (Δ), and with only outputs on shared labels and end linear types (Γ_{un}) as these are the only resources that may be used exponentially, and an empty overall ordering ($\emptyset$). In rule (T-LRec) a recursive process is well typed if so is the body of the recursion in the same typing context Γ_{lin} (which only includes linear usages) and overall ordering $\prec$, and in the recursion environment augmented with an assumption for the recursion variable: the variable is assumed to have exactly the same usage and overall ordering *up to* an increment (for some n) of the natural exponent of the events. Rule (T-LRec) therefore captures, in a fairly intuitive way, subsequent iterations of a (linear) recursion: the point of the next iteration is characterized by an increment of the typing and ordering.

Rule (T-URec) addresses a recursive process that uses only shared resources where no increment is involved since shared communications do not have iterations (their repetition is considered to happen at the level of a single iteration). So the recursion environment is augmented with the same typing and ordering that types the body of the recursion. The typing mentions only shared exponential resources (Γ_{un}) together with a shared input (on x), as we intend to capture replicated shared inputs (a recursive process specifying only shared outputs can be typed separately in expected lines). In order to ensure that the shared input is an immediate action of the body of the recursion, the ordering makes e a minimal event. Given the above explanation, rule (T-Var) is straightforward: the assumption for the variable provides the context and ordering for the process. In rule (T-Equiv) we embed the notion of context equivalence in the type system, since we need to unfold recursive types when typing the body of a recursion.

Communication prefixes are also typed in separate rules, depending on the type of the subject of the communication. In rule (T-LinIn) the input on a channel x with linear usage is typed if the continuation processes are typed with the usages for x prescribed in the prefix summation type, together with a (sound) separation of the ordering assumptions; also, by adding a typing assumption for the usage of the received name (according to the corresponding message type), and a (sound) separation of the events *greater than* e in the overall ordering. The e is the event associated with the prefix summation (notice that we pick *fresh* events since $+$ is undefined otherwise). The fact that events in the continuation are of greater order ensures that the communications in the continuation are in fact prescribed to take place after the prefix itself. Notice that the overall ordering registered in the conclusion is a tree rooted in e. Further notice that the communication dependency structure of the received name is transparently kept in the conclusion (the ordering prescribed by the channel usages is *invariantly* registered in the overall ordering). This allows us to type systems where communications on received channels are interleaved with others, configurations out of reach of related approaches.

The reasoning is similar in rule (T-LinOut). The continuation is typed by considering the continuations of the prefix summation (any prefix summation containing the only label mentioned by the process) which is uniquely associated with event e, together with the same ordering assumptions $\prec'$ (as we are only interested that the environment guarantees the order of one branch). The typing context Γ is actually the result of a split of the context registered in the

conclusion, which also mentions the usage delegated in the communication for the sent name. Finally, the overall ordering in the conclusion also registers the ordering (of events greater than e) prescribed by the message type.

Rules (T-UIn) and (T-UOut) explain the typing for communications on shared channels. In rule (T-UIn) the input with shared usage is typed if the continuation process is typed adding the usage for the received name to the context. Notice that the fact that we type the continuation with the shared input usage implies that the continuation must offer again the shared input behavior (so shared inputs can be typed only in the context of a recursion). Notice also that the overall ordering in the conclusion is that of the premise, as expected in a replicated process, and specifies that the event associated with the shared input is minimal (so as to ensure it is immediately available). Furthermore, we require that the remaining context mentions exclusively shared outputs (Γ_{un}) so that no other shared inputs are defined in the continuation. This would be a problem for liveness since shared inputs on *free* names defined in the continuation might leave a matching output dangling. However, we may freely type processes in the continuation that specify shared inputs in restricted names (or even in the received name).

In rule (T-UOut) we type the output on a shared channel if the continuation is typed in the empty context and empty ordering. This means that our model for shared channel communications is an asynchronous one. There are (at least) two approaches to guarantee that shared inputs (that have matching outputs) are always active (*uniform receptiveness*): one is to exclude usage of the shared name in the continuations of *both* input and output prefixes [18] (we followed a similar approach in [21] excluding the corresponding *event* in the continuations); the other relies on an asynchronous model of communication [1]—which we adopt here for shared channels. The advantage of this approach is that it supports processes that specify in the continuation of a shared input a matching output (intuitively, think of a recursive "service" call). Also, looking at (T-UOut) and (T-Par) we argue that every process P that we have used in examples in the continuation of a shared output $x!l\,y.P$ can be specified (and typed) using the parallel composition $P \mid x!l\,y.\mathbf{0}$, essentially since the type delegated in the communication is obtained via a split of the context nonetheless. Notice that rule (T-UOut) says that the event associated with the output is minimal w.r.t. the message type in the conclusion. Notice also that the rules for communication prefixes make no distinction whatsoever on the type of the channel communicated.

We can now show that process (4) is well typed. We take

$$T_s \triangleq \mu\alpha.e^1\,\tau\,\mathtt{l}\,\mathsf{int}.f^1\,\tau\,\mathtt{m}\,\mathsf{bool}.\alpha \qquad T_r \triangleq \mu\alpha.g^1\,\tau\,\mathtt{m}\,\mathsf{bool}.\alpha$$

where we have added to each event an index denoting the iteration in the recursive behavior of the processes that implements these interactions. Each unfolding of a type increments the indexes of the events. The splitting of these behaviors produces

$$T_{\mathtt{A}s} \triangleq \mu\alpha.e^1\,?\,\mathtt{l}\,\mathsf{int}.\alpha \qquad T_{\mathtt{B}s} \triangleq \mu\alpha.f^1\,?\,\mathtt{m}\,\mathsf{bool}.\alpha \qquad T_{\mathtt{C}s} \triangleq \mu\alpha.e^1\,!\,\mathtt{l}\,\mathsf{int}.f^1\,!\,\mathtt{m}\,\mathsf{bool}.\alpha$$

$$T_{\mathtt{A}r} \triangleq \mu\alpha.g^1\,?\,\mathtt{m}\,\mathsf{bool}.\alpha \qquad T_{\mathtt{B}r} \triangleq \mu\alpha.f^1\,!\,\mathtt{m}\,\mathsf{bool}.\alpha$$

regarding sessions s and r. Now, looking at the structure of the participants in (4), we realize that the following relations must hold: the structure of `A` requires $e^1 \prec g^1 \prec e^2$; the structure of `B` requires $f^1 \prec g^1 \prec f^2$; finally, the structure of `C` requires $e^1 \prec f^1 \prec e^2$. Overall, the whole process is well typed by considering the strict partial order

$$\prec \triangleq T_s{\downarrow} \cup T_r{\downarrow} \cup \{(f^i, g^i), (g^i, e^{i+1}) \mid i \in \mathbb{N}\}$$

$$\frac{k \in I}{e\,\tau\{l_i\,T_i.B_i\}_{i\in I} \to B_k} \qquad \frac{B_1 \to B_1'}{B_1 \mid B_2 \to B_1' \mid B_2} \qquad \frac{B_1 \equiv B_1' \to B_2' \equiv B_2}{B_1 \to B_2}$$

$$\frac{}{\cdot \to \cdot} \qquad \frac{B_1 \to B_2}{\Gamma, x\colon {B_1}^{\prec} \to \Gamma, x\colon {B_2}^{\prec}} \qquad \frac{\Gamma_1 \to \Gamma_2}{\Gamma_1, x\colon T \to \Gamma_2, x\colon T}$$

Figure 8: Type and context reduction

$$\frac{}{\prec \to \prec} \qquad \frac{e \in \mathit{supp}(\prec)}{\prec \to \ \prec \setminus e} \qquad \frac{\Gamma_1 \to \Gamma_2 \quad \prec_1 \to \prec_2}{\Gamma_1; \prec_1 \to \Gamma_2; \prec_2}$$

Figure 9: Order and typing reduction

Results. We are now in a position to present our results, namely that typing is preserved under reduction (Theorem 1) and that a class of well-typed processes (those with matched communications) enjoys a liveness property (Theorem 2). We start by mentioning some auxiliary results, in particular that conformance between the typing context and the overall ordering is ensured for all derivations. This result may be viewed as a sanity check that says the conditions imposed by our rules are enough to keep conformance invariant in a derivation. We may also show that split is an associative relation, in particular for behavioral types. This result in particular ensures that the derivation (sub-)trees may be moved around, used in the proof of the following (standard) results.

Lemma 1 (Subject Congruence). *If* $\Delta; \Gamma; \prec \vdash P$ *and* $P \equiv Q$ *then* $\Delta; \Gamma; \prec \vdash Q$.

Lemma 2 (Substitution). *If* $\Delta; \Gamma_1, x\colon T; \prec \vdash P$ *and* $\Gamma_2 = \Gamma_1 \circ y\colon T$ *then* $\Delta; \Gamma_2; \prec \vdash P\{x/y\}$.

The proofs follow by induction on the structure of the process and on the length of the typing derivation (respectively) along non-surprising lines. Notice that substitution uses context splitting to characterize the context that types the resulting process, since name y may already be used by P and the soundness of the substitution is guaranteed by the split. Before presenting our first main result we need to introduce two auxiliary notions that characterize reduction of contexts and of strict partial orders. As expected from a behavioral type system, as processes evolve so must the types that characterize the processes. The reduction relations for behavioral types and contexts are given in Fig. 8. Hence, τ prefix summations (in "active contexts") may reduce and a context reduces if it has an entry on a linear type prefix that reduces. In such way contexts may mimic the corresponding behavior in processes. Also, the empty context reduces so as to mimic synchronizations on restricted channels and on shared channels (embedding context reduction with reflexivity) since no change in the types is required to capture such synchronizations.

Fig. 9 shows the reduction for orders and context/order pair. Strict partial order reduction is also reflexive to capture both shared synchronizations and communications that *depend* on shared communications (as they take place repeatedly for each of the continuation of the shared input). Reduction is also defined by removing an event of the ordering, so as to capture *one shot* synchronizations (which includes infinite chains of synchronizations). We may now present our first main result.

Theorem 1 (Preservation). *If* $\Delta; \Gamma_1; \prec_1 \vdash P_1$ *and* $P_1 \to P_2$ *then* $\Gamma_1; \prec_1 \to \Gamma_2; \prec_2$ *and* $\Delta; \Gamma_2; \prec_2 \vdash P_2$.

The proof follows by induction on the length of the derivation of $P_1 \to P_2$. The theorem says that typing is preserved under process reduction, up to a reduction in the context and

ordering. Fidelity is an immediate consequence of Theorem 1, as usual (cf. [2]), thanks to the precise correspondence between reduction in processes and in typing contexts. We now turn our attention to the liveness result, where we use $\rightarrow^n$ to denote a sequence of n reductions.

Theorem 2 (Liveness)**.** *Let* $\Delta;\Gamma_1;\prec_1 \vdash P_1$ *with matched*(Γ_1)*, and let* $x\colon {B_1}^{\prec_1'}$ *in* Γ_1*. If* $e \in \mathit{events}({B_1}^{\prec_1'})$ *then* $P_1 \rightarrow^n P_2$ *and* $(\Gamma_1;\prec_1) \rightarrow^n (\Gamma_2;\prec_2)$ *and* $\Delta;\Gamma_2;\prec_2 \vdash P_2$ *with* $x\colon {B_2}^{\prec_2'}$ *in* Γ_2 *and* $e \notin \mathit{events}({B_2}^{\prec_2'})$*, for some* $n > 0$*.*

In words, every event e occurring in the type of a linear channel used by a well-typed process can eventually disappear from the type environment. This means that either e is associated with an (inter)action that can eventually be performed by the process, or that e occurs in a branch of a choice which is not selected. This property is akin to *lock freedom* [13] or *progress* [3, 6, 12] except that e in Theorem 2 can be associated with an action that is arbitrarily deep within the process structure, whereas lock freedom and progress are usually formulated for top-level actions only. The proof follows by induction on $\prec$. The invariant is that for each linear synchronization prescribed by the types there is either an immediate corresponding synchronization in the process or there are preceding actions which necessarily are of "lesser" order. The fact that behaviors described by linear types have a correspondence with the communication capabilities of processes is a standard property of linear type theories.

Notice that we are not able to characterize shared usages in the same way, as the events associated with them are persistent. However, we may immediately conclude that since any linear synchronization that depends on a shared synchronization takes place then so does the shared synchronization (in fact, our proof relies on the fact that also shared synchronizations are live, along with communications in restricted channels with matched typings). Notice also that our type-based approach addresses processes with "unmatched" typing, just as long as we consider them up to the composition with any other processes for which the resulting typing is *matched*—in particular via rule (T-Par) of Fig. 7. An immediate consequence of Theorem 1 and Theorem 2 combined is that any reachable configuration also has the liveness property.

4 Concluding Remarks

We present a type system for multiparty session-based communication-centred systems that guarantees *liveness* in addition to session *fidelity* even when multiple sessions are interleaved. Compared to other models for multiparty session communication, our approach strives to achieve minimality of both language and type features. Regarding language features, we rely on message labels for preventing communication races on linear channels, whereas other approaches make use of *channel polarities* [9], of distinct *channel endpoints* [20], or *roles* [3, 6]. Moreover, we do not make use of dedicated session initialization primitives. Regarding type features, our work exploits notions introduced in [2, 4] (e.g., the τ and the splitting), allowing us to use the same type language for specifying both *global* and *local* types. This is in contrast with common multiparty session type theories such as [3, 6, 12], which introduce distinct languages for global and local types connected by a projection of the former into the latter.

A number of type-based techniques guaranteeing deadlock freedom, progress, or liveness properties have been proposed. Kobayashi [13, 14] presents type systems for *lock-free* and *deadlock-free* processes written in the pure π-calculus. Roughly, every top-level input/output prefix in a lock-free process is guaranteed to be eventually consumed, whereas a deadlock-free process is one that is always able to reduce, unless it has terminated. The type systems rely on *channel usages*, which are behavioral types resembling session types where actions are annotated with pairs of *obligation*/*capability* levels, roughly denoting the time at which actions begin/are

supposed to end. Top-level actions with a finite capability level are guaranteed to succeed in a finite amount of time (and possibly under some fairness assumption). For session-based languages, the relevant works on binary sessions are [8, 16], while [3, 6] deal with multiparty sessions. The basic idea of [3, 6, 8] is to devise a type system that detects the *dependency graph* between different sessions, where a dependency arises if a (blocking) action in one session guards an action pertaining a different session. Liveness is guaranteed if the dependency graph is acyclic. [16] leverages Kobayashi's technique described in [13] from channel usages to session types showing that such technique can achieve a greater accuracy, when compared to [3, 6, 8]. The present work differs from these in several minor and major ways. In particular, our process model is synchronous, while the ones in [3, 6, 16] is asynchronous. Asynchrony has a non-trivial impact in the type system for progress, mainly because output actions are *non-blocking*. The progress property considered in [3, 6] assumes that missing session participants can eventually join the system at any time. In practice, this assumption implies that any action on shared channels is considered non-blocking, because it is always possible to add some (well-typed) processes that provide for the missing messages. Also, [6] defines a syntax-directed type system and automatic inferences are known for the systems described in [13, 14]. In our case, the definition of a syntax-directed type system and of an inference algorithm remain open problems.

One major difference between our work and the aforementioned ones, which constitutes the main technical contribution, regards the treatment of recursive types. In all previous works, annotations such as obligation/capability levels in [13, 14], dependency graphs [3, 6], timestamps [16] are statically associated with types, regardless of their recursive structure. In our case, unfolding a recursive type has the effect to "freshen" the events occurring therein. This significantly increases the range of well-typed processes. In particular, none of the aforementioned works is able to prove progress for non-trivial recursive processes interleaving (blocking) actions on different channels. For example, the (appropriate encoding of the) (4) is ill typed according to all previous type systems. More recently, the first author has studied a type system for deadlock and lock freedom which is capable of addressing non-trivial recursive process configurations, albeit in the context of the linear π-calculus [17]. The type system in [17] can prove that a configuration such as (4) is (dead)lock free, but only encoding the multiparty session s in terms of several binary sessions which, in turn, can be encoded using linear channels. In the present work instead we consider a calculus with a primitive notion of multiparty session, addressing scenarios that cannot be compiled down to binary sessions.

Naturally a type-based approach is only relevant if it can be taken into practice, so decidability is a fundamental property. We may argue that we can extract a decidable type-checking procedure from our type system, if we annotate restricted names with their types (as usual) and process recursion variables with the increment factor (together with confining unfoldings to a "just-in-time" setting). Inference is also an important issue as it allows to save the programmer's effort to specify the types and increase the probability that such advanced type system can actually be used in practice. Although we believe these are very important questions to address, we decided to leave them to future clarification and concentrate on the principles of our approach for now, so as to make further efforts worthwhile. Furthermore, observing that types are becoming very rich characterizations of process behavior (in our case how and when channels are used), one may ask if it is possible to deduce processes from types (e.g., [5]) and spare the "programmer" the effort of writing programs and just ask him to write the types.

References

[1] R. M. Amadio, G. Boudol, and C. Lhoussaine. On message deliverability and non-uniform receptivity. *Fundam. Inform.*, 53(2):105–129, 2002.

[2] P. Baltazar, L. Caires, V. T. Vasconcelos, and H. T. Vieira. A type system for flexible role assignment in multiparty communicating systems. In *TGC'12*, volume 8191 of *LNCS*, pages 82–96. Springer, 2012.

[3] L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR'08*, LNCS 5201, pages 418–433. Springer, 2008.

[4] L. Caires and H. T. Vieira. Conversation types. *Theor. Comput. Sci.*, 411(51-52):4399–4440, 2010.

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

[6] M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida. Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. In *COORDINATION'13*, LNCS 7890, pages 45–59. Springer, 2013.

[7] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global progress for dynamically interleaved multiparty sessions. *MSCS*, to appear.

[8] M. Dezani-Ciancaglini, U. de'Liguoro, and N. Yoshida. On progress for structured communications. In *TGC'07*, LNCS 4912, pages 257–275. Springer, 2007.

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

[10] K. Honda. Types for dyadic interaction. In *CONCUR'93*, LNCS 715, pages 509–523. Springer, 1993.

[11] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, LNCS 1381, pages 122–138. Springer, 1998.

[12] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.

[13] N. Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002.

[14] N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR'06*, LNCS 4137, pages 233–247. Springer, 2006.

[15] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I and II. *Inf. Comput.*, 100(1):1–77, 1992.

[16] L. Padovani. From Lock Freedom to Progress Using Session Types. In *Proceedings of the 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'13)*, EPTCS 137, pages 3–19, 2013.

[17] L. Padovani. Deadlock and lock freedom in the linear π-calculus. Technical report, HAL, 2014.

[18] D. Sangiorgi. The name discipline of uniform receptiveness. *Theor. Comput. Sci*, 221(1-2):457–493, 1999.

[19] V. T. Vasconcelos. Typed concurrent objects. In *ECOOP'94*, LNCS 821, pages 100–117. Springer, 1994.

[20] V. T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.

[21] H. T. Vieira and V. T. Vasconcelos. Typing progress in communication-centred systems. In *COORDINATION'13*, volume 7890 of *LNCS*, pages 236–250. Springer, 2013.