Type Checking Liveness for Collaborative Processes with Bounded and Unbounded Recursion*

Søren Debois[1], Thomas Hildebrandt[1], Tijs Slaats[1,2], and Nobuko Yoshida[3]

[1] IT University of Copenhagen {debois,hilde,tslaats}@itu.dk
[2] Exformatics A/S
[3] Imperial College yoshida@doc.ic.ac.uk

Abstract. We present the first session typing system guaranteeing response liveness properties for possibly non-terminating communicating processes. The types augment the branch and select types of the standard binary session types with a set of required responses, indicating that whenever a particular label is selected, a set of other labels, its responses, must eventually also be selected. We prove that these extended types are strictly more expressive than standard session types. We provide a type system for a process calculus similar to a subset of collaborative BPMN processes with internal (data-based) and external (event-based) branching, message passing, bounded and unbounded looping. We prove that this type system is sound, i.e., it guarantees request-response liveness for dead-lock free processes. We exemplify the use of the calculus and type system on a concrete example of an infinite state system.

1 Introduction

Session types were originally introduced as typing systems for particular π-calculi, modelling the interleaved execution of two-party protocols. A well-typed process is guaranteed freedom from race-conditions as well as communication compatibility, usually referred to as session fidelity [15,26,24]. Session types have subsequently been studied intensely, with much work on applications, typically to programming languages, e.g., [11,17,14,20]. A number of generalisations of the theory has been proposed, notably to multi-party session types [16]. Multi-party session types have a close resemblance to choreographies as found in standards for business process modelling languages such as BPMN [21] and WS-CDL, and has been argued in theory to be able to provide typed BPMN processes [8].

Behavioral types usually furnish *safety* guarantees, notably progress and lock-freedom [3,1,5,10,25]. In contrast, in this paper we extend binary session types to allow specification of *liveness*—the property of a process eventually "doing something good". Liveness properties are usually verified by model-checking

(*) Work supported in part by the Computational Artifacts project (VELUX 33295, 2014-2017), by the Danish Agency for Science, Technology and Innovation, and by EPSRC EP/K034413/1, EP/K011715/1 and EP/L00058X/1.

techniques [6,2,4], requiring a state-space exploration. In the present paper we show that a fundamental class of liveness properties, so-called *request-response* properties, can be dealt with by type rules, that is, without resorting to states-pace exploration. As a consequence, we can deal statically with infinite state systems as exemplified below. Also, liveness properties specified in types can be understood and used as interface specifications and for compositional reasoning.

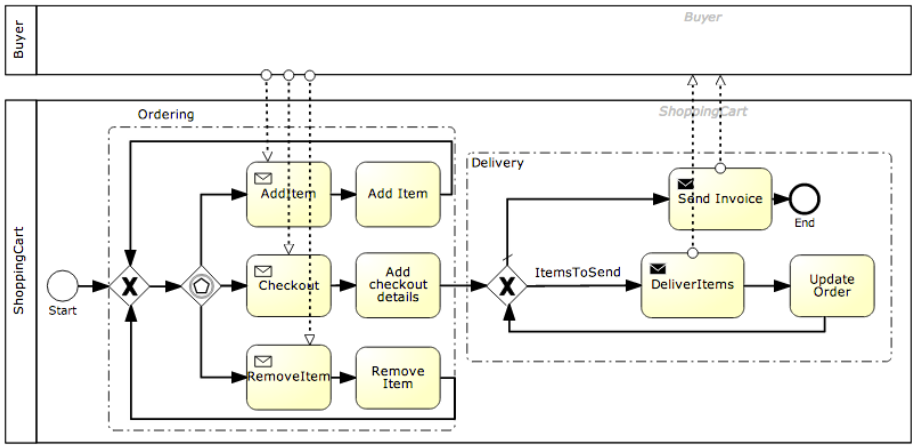

Fig. A. A Potentially Non-live Shopping Cart BPMN Process

As an example, the above diagram contains two pools: The Buyer and the ShoppingCart. Only the latter specifies a process, which has two parts: Ordering and Delivery. Ordering is a loop starting with an event-based gateway, branching on the message received by the customer. If it is AddItem or RemoveItem, the appropriate item is added or removed from the order, whereafter the loop repeats. If it is Checkout, the loop is exited, and the Delivery phase commences. This phase is again a loop, delivering the ordered items and then sending the invoice to the buyer.

A buyer who wants to communicate *safely* with the Shopping Cart, must follow the protocol described above, and in particular must be able to receive an unbounded number of items before receiving the invoice. Writing $\mathsf{AI}, \mathsf{RI}, \mathsf{CO}, \mathsf{DI}$, and SI for the actions "Add Items", "Remove Items", "Checkout", "Deliver Items" and "Send Invoice"; we can describe this protocol with a session type:

$$\mu t.\&\{\mathsf{AI}.?.t, \mathsf{RI}.?.t, \mathsf{CO}.?.\mu t'.\oplus\{\mathsf{DI}.!.t', \mathsf{SI}.!.\mathtt{end}\}\}\ .$$

This session type can be regarded as a *behavioral* interface, specifying that the process first expects to receive either an AI (AddItem), RI (RemoveItem) or a CO (CheckOut) event. The two first events must be followed by a message (indicated by "?"), which in the implementation provides the item to be added or removed, after which the protocol returns to the initial state. The checkout event is followed by a message (again indicated by a "?") after which the protocol

enters a new loop, either sending a DI (DeliverItem) event followed by a message (indicated by a "!") and repeating, or sending an SI (SendInvoice) event followed by a message (the invoice) and ending.

However, standard session types can not specify the very relevant *liveness* property, that a CheckOut event is *eventually* followed by an invoice event. This is an example of a so-called *response* property: an action (the request) must be followed by a particular response. In this paper we conservatively extend binary session types to specify such response properties, and we show that this extension is strictly more expressive than standard session types. We do so by annotating the checkout selection in the type with the required response:

$$\mu t.\&\{\mathsf{AI}.?.t, \mathsf{RI}.?.t, \mathsf{CO}[\{\mathsf{SI}\}].?.\mu t'.\oplus\{\mathsf{DI}.!.t', \mathsf{SI}.!.\mathtt{end}\}\}\ .$$

Intuitively: "if CO is selected, then subsequently also SI must be selected."

Determining from the flow graph alone if this response property is guaranteed is in general not possible: Data values dictate whether the second loop terminates. However, we can remove this data-dependency by replacing the loop with a bounded iteration. In BPMN this can be realised by a Sequential Multiple Instance Subprocess, which sequentially executes a (run-time determined) number of instances of a sub-process. With this, we may re-define Delivery as in Fig. B, yielding a re-defined Shopping Cart process which has the response property.

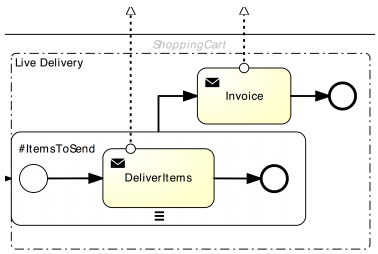

Fig. B. Live delivery with MI Sub-Process

In general, we need also be able to check processes where responses are requested within (potentially) infinite loops. The type system we present gives such guarantees, essentially by collecting all requested responses in a forward analysis, exploiting that potentially infinite loops can guarantee a particular response only if every path through the loop can; and that order (request-response vs response-request) is in this case irrelevant. We prove that, if the system is lock free, then the typing system indeed guarantees that all requested responses are eventually fulfilled. Lock-freedom is needed because, as is well known, collaborative processes with interleaved sessions may introduce dependency locks. Lock-freedom is well-studied for both π-calculus, e.g., [18], and binary session types [3,1,5,10,25], or may alternatively be achieved by resorting to global types [16].

In summary, our contributions are as follows.

- We extend binary session types with a notion of *required response*.
- We prove that this extension induces a strictly more expressive language class than standard session types.
- We give a typing system conservatively extending standard binary session types which gives the further guarantee that a lock-free well-typed process will, in any execution, provide all its required responses.

– We exemplify the use of these extended types to guarantee both safety and liveness properties for a non-trivial, infinite state collaborative process, which exhibits both possibly infinite looping and bounded iteration.

Related work. There is a vast amount of work on verification of collaborative processes. Most of the work take a model-checking approach, where the system under verification is represented as a kind of automaton or Petri Nets. An example that explicitly addresses collaborative business processes is [23], which however does not cover liveness properties. Live Sequence Charts (LSCs) [6] is a conservative extension of Message Sequence Charts allowing to distinguish possible (may) from required (must) behaviour, and thus the specification of liveness properties. LSCs can be mapped to symbolic timed automata [2] but relies as all model-checking approaches on abstraction techniques for reducing a large or even infinite state space to a tractable size. Here the work in [4] is interesting for the fact that the model-checking can be split on components. The work in [19] allows for model-checking of ML programs by a translation to higher-order recursion schemes. Interestingly, the model-checking problem is reduced to a type-checking problem, but rely on a technique for generation of a specific type system for the property of interest. In contrast, our approach is based on a single type system directly applicable for the process language at hand, where the (less general) liveness and safety properties of interest are specified as the type to be checked and can also be used as interface descriptions of processes. The fair subtyping of [22], the only work on session types addressing liveness we are aware of, details a liveness-preserving subtyping-relation for a session types-like CCS calculus. Here liveness is taken to mean the ability to always eventually output a special name, whereas in the present work, we consider the specification of fine-grained request-response liveness properties—"*if* something happens, something else must happen".

Overview of this paper. In Sec. 2 we define our calculus and its LTS-semantics. In Sec. 3 we extend binary session types with specification of response liveness properties, give transition semantics for types, and sketch a proof that the extended types induce a strictly larger class of languages than does standard types. In Sec. 4 we define exactly how types induce a notion of liveness on processes. In Sec. 5 we give our extended typing rules for sessions with responses and state its subject reduction result. In Sec. 6 we prove that the extended typing rules guarantees liveness for lock-free processes. Finally, in Sec. 7 we conclude. For want of space, this paper omits details and proofs; for these, refer to [7].

2 Process Terms and Semantics

Processes communicate only via named communication (session) channels by synchronizing send and receive actions or synchronizing select and branch events (as in standard session typed π-calculus). The session typing rules presented in the next section guarantees that there is always at most one active send and

receive action for a given channel. To distinguish dual ends of communication channels, we employ *polarised names* [13,26]: If c is a channel name, c^+ and c^- are the dual ends of the channel c. We call these *polarised channel names*, with "+" and "-" polarities. If k is a polarised channel name, we write $\overline{k}$ for its dual, e.g., $\overline{c^+} = c^-$. In general c ranges over channel names; p over polarities $+, -$; k, h over polarised channel names; x over data variables; i over recursion variables (explained below); v over data values including numbers, strings and booleans; e over data expressions; and finally X, Y over process variables.

$$\begin{aligned} P \ ::= \ & k!\langle e\rangle.P \ \mid \ k?(x).P \ \mid \ k!l.P \ \mid \ k?\{l_i.P_i\}_{i\in I} \ \mid \ \mathbf{0} \ \mid \ P \mid Q \\ & \mid \operatorname{rec} X.P \ \mid \ (\operatorname{rec}^e X(i).P; Q) \ \mid \ X[\tilde{k}] \ \mid \ \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \end{aligned}$$

The first four process constructors are for taking part in a communication. These are standard for session typed π-calculi, except that for simplicity of presentation, we only allow data to be sent (see Section 7). The process $k!\langle e\rangle.P$ sends data v over channel k when $e \Downarrow v$, and proceeds as P. Dually, $k?(x).P$ receives a data value over channel k and substitutes it for the x binding in P. A *branch* process $k?\{l_i.P_i\}_{i\in I}$ offers a choice between labels l_i, proceeding to P_i if the i'th label is chosen. The process $\mathbf{0}$ is the standard inactive process (termination), and $P \mid Q$ is the parallel composition of processes P and Q.

Recursion comes in two forms: a general, potentially non-terminating recursion $\operatorname{rec} X.P$, where X binds in P; and a primitive recursion, guaranteed to terminate, with syntax $(\operatorname{rec}^e X(i).P; Q)$. The latter process, when $e \Downarrow n+1$, executes $P\{n/i\}$ and repeats, and when $e \Downarrow 0$, evolves to Q. By convention in $(\operatorname{rec}^e X(i).P; Q)$ neither of $\mathbf{0}$, $\operatorname{rec} Y.P'$, $(\operatorname{rec}^{e'} Y(i).P'; P'')$ and $P' \mid P''$ occurs as subterms of P. These conventions ensure that the process $(\operatorname{rec}^e X(i).P; Q)$ will eventually terminate the loop and execute Q. Process variables $X[\tilde{k}]$ mention the channel names $\tilde{k}$ active at unfolding time for technical reasons.

We define the free polarised names $\mathsf{fn}(P)$ of P as usual, with $\mathsf{fn}(X[\tilde{k}]) = \tilde{k}$; substitution of process variables from $X[\tilde{k}]\{P/X\} = P$; and finally value substitution $P\{v/x\}$ in the obvious way, e.g., $k!\langle e\rangle.P\{v/x\} = k!\langle e\{v/x\}\rangle.(P\{v/x\})$. Variable substitution can never affect channels.

Example 2.1. We now show how to model the example BPMN process given in the introduction. To illustrate the possibility of type checking infinite state systems, we use a persistent data object represented by a process $\mathrm{DATA}(o)$ communicating on a session channel o.

$$\mathrm{DATA}(o) = \operatorname{rec} X.\ o^+?(x).\ \operatorname{rec} Y.\ o^+? \begin{cases} \mathsf{read}.\ o^+!\langle x\rangle.\ Y[o^+] \\ \mathsf{write}.\ X[o^+] \\ \mathsf{quit}.\ \mathbf{0} \end{cases}$$

After having received its initial value, this process repeatedly accepts commands read and write on the session channel o for respectively reading and writing its value, or the command quit for discarding the data object.

To make examples more readable, we employ the following shorthands. We write $\mathsf{init}(o, v).P$ for $o^-!\langle v\rangle.P$, which initializes the data object; we write $\mathsf{free}\, o.P$ for $o^-!\mathsf{quit}.P$, the process which terminates the data object session; we write

$\mathsf{read}\, o(x).P$ for $o^-!\mathsf{read}.\ o^-?(x).P$, the process which loads the value of the data object o into the process-local variable x; and finally, we write $o := e.P$ for $o^-!\mathsf{write}.o^-!\langle e\rangle.P$, the process which sets the value of the data-object o.

The shopping cart process can then be modelled as

$$P(Q) = \mathrm{DATA}(o) \ \mid\ \mathsf{init}(o,\epsilon).\ \mathrm{rec}\, X.k \begin{cases} \mathsf{AI}.\ k?(x).\ \mathsf{read}\, o(y).\ o := add(y,x).\ X[ko^-] \\ \mathsf{RI}.\ k?(x).\ \mathsf{read}\, o(y).\ o := rem(y,x).\ X[ko^-] \\ \mathsf{CO}.\ k?(x).\ \mathsf{read}\, o(y).\ o := add(y,x).\ Q \end{cases}$$

Here k is the session channel shared with the customer and o is the session channel for communicating with the data object modelling order data. We assume our expression language has suitable operators "add" and "rem", adding and removing items from the order. Finally, the process Q is a stand-in for either the (non live) delivery part of the BPMN process in Fig. A or the live delivery part shown in Fig. B. The non-live delivery loop can be represented by the process

$$D_0 = \mathrm{rec}\, Y.\ \mathsf{read}\, o(y).\ \mathsf{if}\ n(y) > 0 \begin{array}{l} \mathsf{then}\ k!\mathsf{DI}.\ k!\langle next(y)\rangle.\ o := update(y).\ Y[ko^-] \\ \mathsf{else}\ k!\mathsf{SI}.\ k!\langle inv(y)\rangle.\ \mathsf{free}\, o.\mathbf{0} \end{array}$$

where $n(y)$ is the integer expression computing from the order y the number of items to send, $next(y)$, $update(y)$ and $inv(y)$ are, respectively, the next item(s) to be sent; an update of the order to mark that these items have indeed been sent; and the invoice for the order. Whether this process terminates depends on the data operations. Using instead bounded iteration, live delivery becomes:

$$\begin{array}{l} D = \mathsf{read}\, o(y).\ (\mathrm{rec}^{n(y)}\, Y(i). \\ \qquad\qquad k!\mathsf{DI}.\mathsf{read}\, o(y).\ k!\langle pickitem(y,i)\rangle.Y[ko^-]; \\ \qquad\qquad\qquad\qquad k!\mathsf{SI}.\ \mathsf{read}\, o(y).\ k!\langle inv(y)\rangle.\ \mathsf{free}\, o.\mathbf{0}) \end{array}$$

(The second line is the body of the loop; the third line is the continuation.) Here $pickitem(y,i)$ is the expression extracting the ith item from the order y. □

Transition Semantics. We give a labelled transition semantics in Fig C. We assume a total evaluation relation $e \Downarrow v$; note the absence of a structural congruence. We assume τ is neither a channel nor a polarised channel. Define $\mathsf{subj}(k!v) = \mathsf{subj}(k?v) = \mathsf{subj}(k\&l) = \mathsf{subj}(k\oplus l) = k$ and $\mathsf{subj}(\tau) = \mathsf{subj}(\tau : l) = \tau$, and $\overline{\tau} = \tau$. We use these rules along with symmetric rules for [C-ParL] and [C-Com1/2]. Compared to standard CCS or π semantics, there are two significant changes: (1) In the [C-ParL], a transition λ of P is *not* preserved by parallel if the co-channel of the subject of λ is in P'; and (2) in prefix rules, the co-name of the subject cannot appear in the continuation. We impose (1) because if the co-channel of the subject of λ is in P', then $P \mid P'$ does not offer synchronisation on λ to its environment; the synchronisation is offered only to P'. E.g., the process $P = c^+!\langle v\rangle.Q \mid c^-?(x).R$ does not have a transition $c^+!\langle v\rangle.Q \mid c^-?(x).R \xrightarrow{c^+!v} Q \mid c^-?(x).R$. If it had such a transition, no environment U able to receive on c^- could be put in parallel with P and form a

$$[\text{C-Out}]\quad \frac{e \Downarrow v}{k!\langle e\rangle.P \xrightarrow{k!v} P}\ \overline{k} \notin \mathsf{fn}(P) \qquad [\text{C-In}]\quad \frac{}{k?(x).P \xrightarrow{k?v} P\{v/x\}}\ \overline{k} \notin \mathsf{fn}(P)$$

$$[\text{C-Sel}]\quad \frac{}{k!l.P \xrightarrow{k\oplus l} P}\ \overline{k} \notin \mathsf{fn}(P) \qquad [\text{C-Bra}]\quad \frac{}{k?\{l_i.P_i\}_{i\in I} \xrightarrow{k\&l_i} P_i}\ \overline{k} \notin \mathsf{fn}(P)$$

$$[\text{C-ParL}]\quad \frac{P \xrightarrow{\lambda} Q}{P \mid P' \xrightarrow{\lambda} Q \mid P'}\qquad \overline{\mathsf{subj}(\lambda)} \notin \mathsf{fn}(P')$$

$$[\text{C-Com1}]\quad \frac{P \xrightarrow{\overline{k}!v} P' \qquad Q \xrightarrow{k?v} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad [\text{C-Com2}]\quad \frac{P \xrightarrow{\overline{k}\oplus l} P' \qquad Q \xrightarrow{k\&l} Q'}{P \mid Q \xrightarrow{\tau:l} P' \mid Q'}$$

$$[\text{C-Rec}]\quad \frac{P\{\mathrm{rec}\, X.P/X\} \xrightarrow{\lambda} Q}{\mathrm{rec}\, X.P \xrightarrow{\lambda} Q} \qquad [\text{C-PRec0}]\quad \frac{e \Downarrow 0 \qquad Q \xrightarrow{\lambda} R}{(\mathrm{rec}^e\, X(i).P; Q) \xrightarrow{\lambda} R}$$

$$[\text{C-PRecN}]\quad \frac{e \Downarrow n+1 \qquad P\{n/i\}\{(\mathrm{rec}^n\, X(i).P; Q)/X\} \xrightarrow{\lambda} R}{(\mathrm{rec}^e\, X(i).P; Q) \xrightarrow{\lambda} R}$$

$$[\text{C-CondT}]\quad \frac{e \Downarrow \mathsf{true} \qquad P \xrightarrow{\lambda} P'}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\lambda} P'} \qquad [\text{C-CondF}]\quad \frac{e \Downarrow \mathsf{false} \qquad Q \xrightarrow{\lambda} Q'}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\lambda} Q'}$$

Fig. C. Transition semantics for terms

well-typed process, since both U and $c^-?(d).R$ would then contain the name c^- free. The reason for (2) is similar: If a process $k!\langle e\rangle.P \xrightarrow{k!v} P$, and P contains $\overline{k}$, again no well-typed environment for that process can contain $\overline{k}$.

3 Session Types with Responses

In this section, we generalise binary session types to *session types with responses*. In addition to providing the standard communication safety properties, these also allow us to specify response liveness properties.

Compared to standard session types, we do not consider delegation (name passing). Firstly, as illustrated by our example calculus, the types are already expressive enough to cover a non-trivial subset of collaborative processes. Secondly, as we show in the end of the section, session types with responses are already strictly more expressive than standard session types with respect to the languages they can express. Thus, as we also address in Sec. 7, admitting delegation and answering the open question about how response obligations can be safely exchanged with the environment, is an interesting direction for future work which is beyond the scope of the present paper.

We first define request/response liveness in the abstract. In general, we shall take it to be the property that "a request is eventually followed by a response".

Definition 3.1. *A* request/response structure *is a tuple* $(A, R, \mathsf{req}, \mathsf{res})$ *where* A *is a set of actions,* R *is a set of responses, and* $\mathsf{req} : A \to R$ *and* $\mathsf{res} : A \to R$ *are maps defining the set of responses requested respectively performed by an action.*

Notation. We write ϵ for the empty string, we let ϕ, ψ range over finite strings, and we let α, β, γ range over finite or infinite sequences. We write sequence concatenation by juxtaposition, i.e., $\phi\alpha$.

Definition 3.2. *Suppose* $(A, R, \mathsf{req}, \mathsf{res})$ *is a request/response structure and* α *a sequence over* A*. Then the* responses $\mathsf{res}(\alpha)$ *of* α *is defined by* $\mathsf{res}(\alpha) = \cup\{\mathsf{res}(a) \mid \exists\varphi, \beta.\ \alpha = \varphi a \beta\}$*. Moreover,* α *is* live *iff* $\alpha = \phi a \beta \implies \mathsf{req}(a) \subseteq \mathsf{res}(\beta)$*.*

Definition 3.3 (LTS with requests/responses). *Let* $(S, L, \to)$ *be an LTS. When the set of labels* L *is the set of actions of a request/response structure, we say that* $(S, L, \to)$ *is an* LTS with requests/responses*, and that a transition sequence of this LTS is* live *when its underlying sequence of labels is.*

Next, syntax of types. Let l range over labels and L sets of labels.

$$S, T \ ::= \ \&\{l_i[L_i].T_i\}_{i\in I} \ \mid \ \oplus\{l_i[L_i].T_i\}_{i\in I} \ \mid \ !.T \ \mid \ ?.T \ \mid \ \mu t.T \ \mid \ t \ \mid \ \mathtt{end}$$

By convention, the l_i in each $\&\{l_i[L_i].T_i\}_{i\in I}$ resp. $\oplus\{l_i[L_i].T_i\}_{i\in I}$ are distinct.

A session type is a (possibly infinite) tree of actions permitted for one partner of a two-party communication. The type $\&\{l_i[L_i].T_i\}_{i\in I}$, called *branch*, is the type of *offering* a choice between different continuations. If the partner chooses the label l_i, the session proceeds as T_i. Compared to standard session types, making the choice l_i also requests a subsequent response on every label mentioned in the set of labels L_i; we formalise this in the notion of *responsive trace* below. Dual to branch is *select* $\oplus\{l_i[L_i].T_i\}_{i\in I}$: the type of *making* a choice between different continuations. Like branch, making a choice l_i requests every label in L_i as future responses. The type $!.T$ and $?.T$ are the types of sending and receiving data values. As mentioned above, channels cannot be communicated. Also, we have deliberately omitted types of values (e.g. integers, strings, booleans) being sent, since this can be trivially added and we want to focus on the behavioural aspects of the types. Finally, session types with responses include recursive types. We take the equi-recursive view, identifying a type T and its unfolding into a potentially infinite tree. We define the central notion of *duality* between types as the symmetric relation induced coinductively by the following rules.

$$\frac{}{\mathtt{end} \bowtie \mathtt{end}} \qquad \frac{T \bowtie T'}{!.T \bowtie\ ?.T'} \qquad \frac{T_i \bowtie T_i' \quad J \subseteq I}{\&\{l_i[L_i].T_i\}_{i\in I} \bowtie \oplus\{l_j[L_j'].T_j'\}_{j\in J}} \tag{1}$$

The first rule says that dual processes agree on when communication ends; the second that if a process sends a message, its dual must receive; and the third says that if one process offers a branch, its dual must choose among the offered choices. However, required responses do not need to match: the two participants in a session need not agree on the notion of liveness for the collaborative session.

Example 3.4. Recall from Ex. 2.1 the processes DATA(o) encoding data-object and $P(D)$ encoding the (live) shopping-cart process. The former treats the channel o as $T_D = \mu t.?.\mu s.\&\{\mathsf{read}.!.s,\ \mathsf{write}.t,\ \mathsf{quit}.\mathtt{end}\ \}$. The latter treats its channel k to the buyer as $T_P = \mu t.\&\{\mathsf{AI}.?.t,\ \mathsf{RI}.?.t,\ \mathsf{CO}[\{\mathsf{SI}\}].?.\mu t'.\oplus\{\mathsf{DI}.!.t',\ \mathsf{SI}.!.\mathtt{end}\}\}$. To illustrate both responses in unbounded recursion and duality of disparate responses, note that the $P(D)$ actually treats its data object channel o^- according to the type $T_E = \mu t.!.\mu s.\oplus\{\mathsf{read}.?.s,\ \mathsf{write}[\{\mathsf{read}\}].t,\ \mathsf{quit}.\mathtt{end}\ \}$, i.e., every write is eventually followed by a read. However, $T_D \bowtie T_E$: the types T_E and T_D are nonetheless dual. □

Having defined the syntax of session types with responses, we proceed to give their semantics. The meaning of a session type is the possible sequences of communication actions it allows, requiring that pending responses eventually be done. Formally, we equip session types with a labeled transition semantics in Fig. D. We emphasise that under the equi-recursive view of session types, the

$$\begin{array}{rl} \text{Type transition labels:} & \rho ::= \ ! \ \mid \ ? \ \mid \ \&l[L] \ \mid \ \oplus l[L] \\ \text{Type transition label duality:} & ! \bowtie \ ? \quad \text{and} \quad \&l[L] \bowtie \oplus l[L'] \end{array}$$

$$\text{[D-Out]}\ \frac{}{!.T \xrightarrow{!} T} \qquad \frac{}{?.T \xrightarrow{?} T}\ \text{[D-In]}$$

$$\text{[D-Bra]}\ \frac{i \in I}{\&\{l_i[L_i].T_i\}_{i\in I} \xrightarrow{\&l_i[L_i]} T_i} \qquad \frac{i \in I}{\oplus\{l_i[L_i].T_i\}_{i\in I} \xrightarrow{\oplus l_i[L_i]} T_i}\ \text{[D-Sel]}$$

Fig. D. Transitions of types (1)

transition system of a recursive type T may in general be infinite.

Taking actions A to be the set of labels ranged over by ρ, and recalling that $\mathcal{L}$ is our universe of labels for branch/select, we obtain a request/response structure $(A, \mathcal{P}(\mathcal{L}), \mathsf{req}, \mathsf{res})$ with the latter two operators defined as follows.

$$\begin{array}{ll} \mathsf{res}(!) = \mathsf{res}(?) = \emptyset & \mathsf{res}(\&l[L]) = \mathsf{res}(\oplus l[L]) = \{l\} \\ \mathsf{req}(!) = \mathsf{req}(?) = \emptyset & \mathsf{req}(\&l[L]) = \mathsf{req}(\oplus l[L]) = L \end{array}$$

Selecting a label l performs the response l; pending responses L associated with that label are conversely requested. The LTS of Fig. D is thus one with responses, and we may speak of its transition sequences being live or not.

Definition 3.5. *Let T be a type. We define:*

1. *The* traces $\mathsf{tr}(T) = \{(\rho_i)_{i\in I} \mid (T_i, \rho_i)_{i\in I}$ *transition sequence of* $T\ \}$
2. *The* responsive traces $\mathsf{tr}_R(T) = \{\alpha \in \mathsf{tr}(T) \mid \alpha$ *live* $\}$.

That is, in responsive traces any request is followed by a response.

Definition 3.6. *A type T is a* standard *session type if it requests no responses, that is, every occurrence of L in it is has $L = \emptyset$. Define $\mathsf{sel}(\rho) = l$ when $\rho = \&l[L]$ or $\rho = \oplus l[L]$, otherwise ϵ; lift $\mathsf{sel}(-)$ to sequences by union. We then define:*

1. *The* selection traces $\mathsf{str}(T) = \{\mathsf{sel}(\alpha) \mid \alpha \in \mathsf{tr}(T)\}$

2. *The* responsive selection traces $\mathsf{str}_R(T) = \{\mathsf{sel}(\alpha) \mid \alpha \in \mathsf{tr}_R(T)\}$.
3. *The* language of standard session types
   $\mathcal{T} = \{\alpha \mid \alpha \in \mathsf{str}(T), T \textit{ is a standard session type}\}$.
4. *The* language of responsive session types
   $\mathcal{R} = \{\alpha \mid \alpha \in \mathsf{str}_R(T), T \textit{ is a session type with responses}\}$.

That is, we compare standard session types and session types of responses by considering the sequences of branch/select labels they admit. This follows recent work on multi-party session types and automata [8,9].

Example 3.7. The type T_P of Example 3.4 has (amongst others) the two selection traces: $t = \mathsf{AI}\,\mathsf{CO}\,\mathsf{DI}\,\mathsf{DI}\,\mathsf{SI}$ and $u = \mathsf{AI}\,\mathsf{CO}\,\mathsf{DI}\,\mathsf{DI}\,\mathsf{DI}\cdots$. Of these, only t is responsive; u is not, since it never selects SI as required by its CO action. That is, $t, u \in \mathsf{str}(T_P)$ and $t \in \mathsf{str}_R(T_P)$, but $u \notin \mathsf{str}_R(T_P)$. □

Theorem 3.8. *The language of session types with responses $\mathcal{R}$ is strictly more expressive than that of standard session types $\mathcal{T}$; that is, $\mathcal{T} \subset \mathcal{R}$.*

Proof (sketch). The non-strict inclusion is immediate by definition; it remains to prove it strict. For this consider the session type with responses $T = \mu t. \oplus \{a[b].t; b[a].t\}$, which has as responsive traces all strings with both infinitely many as and bs. We can find every sequence a^n as a *prefix* of such a trace. But, (by regularity) any *standard* session type that has all a^n as finite traces must also have the trace a^ω, which is not a responsive trace of T, and thus the responsive traces of T can not be expressed as the traces of a standard session type.

4 Session Typing

Recall that the standard typing system [15,26] for session types has judgements $\Theta \vdash_{\mathsf{std}} P \triangleright \Delta$. We use this typing system without restating it; refer to either [15,26] or the full version of this paper [7]. In this judgement, Θ takes process variables to session type environments; in turn, a *session typing environment* Δ is a finite partial map from channels to types. We write Δ, Δ' for the union of Δ and Δ', defined when their domains are disjoint. We say Δ is *completed* if $\Delta(T) = \mathsf{end}$ when defined; it is *balanced* if $k : T, \overline{k} : U \in \Delta$ implies $T \bowtie U$.

We generalise transitions of types (Fig. D) to session typing environments in Fig. E, with transitions $\delta ::= \tau \mid \tau : l, L \mid k : \rho$. We define $\mathsf{subj}(k : \rho) = k$ and $\mathsf{subj}(\tau : l, L) = \mathsf{subj}(\tau) = \tau$. We lift $\mathsf{sel}(-), \mathsf{req}(-)$, and $\mathsf{res}(-)$ to actions δ in

$$\text{[E-Lift]}\quad \frac{T \xrightarrow{\rho} T'}{k : T \xrightarrow{k:\rho} k : T'} \qquad \text{[E-Par]}\quad \frac{\Delta \xrightarrow{\delta} \Delta'}{\Delta, \Delta'' \xrightarrow{\delta} \Delta', \Delta''}$$

$$\text{[E-Com1]}\quad \frac{\Delta_1 \xrightarrow{k:!} \Delta_1' \qquad \Delta_2 \xrightarrow{\overline{k}:?} \Delta_2'}{\Delta_1, \Delta_2 \xrightarrow{\tau} \Delta_1', \Delta_2'} \qquad \text{[E-Com2]}\quad \frac{\Delta_1 \xrightarrow{k:\oplus l[L]} \Delta_1' \qquad \Delta_2 \xrightarrow{\overline{k}:\&l[L']} \Delta_2'}{\Delta_1, \Delta_2 \xrightarrow{\tau:l, L\cup L'} \Delta_1', \Delta_2'}$$

Fig. E. Transitions of types (2)

the obvious way, e.g., $\mathsf{req}(\tau : l, L) = L$. The type environment transition is thus an LTS with responses, and we may speak of its transition sequences being live.

Definition 4.1. *We define a binary relation on type transition labels δ and transition labels λ, written $\delta \simeq \lambda$, as follows. $\tau \simeq \tau$, $k : \&l[L] \simeq k\&l$, $k : ! \simeq k!v$, $\tau : l, L \simeq \tau : l$, $k : \oplus l[L] \simeq k \oplus l$, $k : ? \simeq k?x$.*

Theorem 4.2. *If $\Gamma \vdash_{\mathsf{std}} P \triangleright \Delta$ and $P \xrightarrow{\lambda} Q$, then there exists $\delta \simeq \lambda$ s.t. $\Delta \xrightarrow{\delta} \Delta'$ and $\Gamma \vdash_{\mathsf{std}} Q \triangleright \Delta'$.*

Definition 4.3. *The* typed transition system *is the transition system which has states $\Gamma \vdash_{\mathsf{std}} P \triangleright \Delta$ and transitions $\Gamma \vdash_{\mathsf{std}} P \triangleright \Delta \xrightarrow{\lambda,\delta} \Gamma \vdash_{\mathsf{std}} P' \triangleright \Delta'$ whenever there exist transitions $P \xrightarrow{\lambda} P'$ and $\Delta \xrightarrow{\delta} \Delta'$ with $\delta \simeq \lambda$.*

We can now say what it means for a process to be live (relying on the definition of maximal transition sequences given in Def. 6.2 below).

Definition 4.4 (Live process). *A well-typed process $\Theta \vdash_{\mathsf{std}} P \triangleright \Delta$ is* live *wrt. Θ, Δ iff for any maximal transition sequence $(P_i, \lambda_i)_i$ of P there exists a live type transition sequence $(\Delta_i, \delta_i)_i$ of Δ s.t. $((P_i, \Delta_i), (\lambda_i, \delta_i))_i$ is a typed transition sequence of $\Theta \vdash_{\mathsf{std}} P \triangleright \Delta$.*

Example 4.5. Wrt. the standard session typing system, *both* of the processes $P(D_0)$ and $P(D)$ of Example 2.1 are typable wrt. the types we postulated for them in Example 3.4. Specifically, we have $\cdot \vdash_{\mathsf{std}} P(D_0) \triangleright k : T_P, o^+ : T_D, o^- : \overline{T_D}$ and similarly for $P(D)$. The judgement means that the process $P(D)$ treats k according to T_P and the (two ends of) the data object according to T_D and its syntactic dual $\overline{T_D}$. The standard session typing system of course does not act on our liveness annotations, and so does not care that $P(D_0)$ is not live.

5 Typing System for Liveness

We now give our extended typing system for session types with responses. The central judgement will be $\Gamma; L \vdash P \triangleright \Delta$, with the intended meaning that "with process variables Γ and pending responses L, the process P conforms to Δ." We shall see in the next section that a well-typed lock-free P is live and will eventually perform every response in L. We need:

1. *Session typing environments* Δ defined at the start of Section 4.
2. *Response environments* L are simply sets of branch/select labels.
3. *Process variable environments* Γ are finite partial maps from process variables X to tuples (L, L, Δ) or (L, Δ). We write these (A, I, Δ) for (A)ccumulated selections and request (I)nvariant. We define $(\Gamma + L)(X) = (A \cup L, I, \Delta)$ when $\Gamma(X) = (A, I, \Delta)$ and $\Gamma(X)$ otherwise, writing $\Gamma + l$ instead of $\Gamma + \{l\}$.

$$\text{[F-Out]}\quad \frac{\Gamma; L \vdash P \triangleright \Delta, k : T}{\Gamma; L \vdash k!\langle e\rangle.P \triangleright \Delta, k : !.T} \qquad \text{[F-In]}\quad \frac{\Gamma; L \vdash P \triangleright \Delta, k : T}{\Gamma; L \vdash k?(x).P \triangleright \Delta, k : ?.T}$$

$$\text{[F-Bra]}\quad \frac{\forall i \in I : \quad \Gamma + l_i; (L \setminus l_i) \cup L_i \vdash P_i \triangleright \Delta, k : T_i}{\Gamma; L \vdash \ k?\{l_i.P_i\}_{i\in I} \triangleright \Delta, k : \&\{l_i[L_i].T_i\}_{i\in I}}$$

$$\text{[F-Sel]}\quad \frac{\Gamma + l_j; (L \setminus l_j) \cup L_j \vdash P \triangleright \Delta, k : T_j}{\Gamma; L \vdash k!l_j.P \triangleright \Delta, k : \oplus\{l_i[L_i].T_i\}_{i\in I}} \quad (j \in I)$$

$$\text{[F-Par]}\quad \frac{\Gamma; L_1 \vdash P_1 \triangleright \Delta_1 \qquad \Gamma; L_2 \vdash P_2 \triangleright \Delta_2}{\Gamma; L_1 \cup L_2 \vdash P_1 \mid P_2 \triangleright \Delta_1, \Delta_2} \qquad \text{[F-Inact]}\quad \frac{\Delta \text{ completed}}{\Gamma; \emptyset \vdash \mathbf{0} \triangleright \Delta}$$

$$\text{[F-Var]}\quad \frac{L \subseteq I \subseteq A \qquad \mathsf{dom}(\Delta) = \tilde{k}}{\Gamma, X : (A, I, \Delta); L \vdash X[\tilde{k}] \triangleright \Delta} \qquad \text{[F-VarP]}\quad \frac{L \subseteq L' \qquad \mathsf{dom}(\Delta) = \tilde{k}}{\Gamma, X : (L', \Delta); L \vdash X[\tilde{k}] \triangleright \Delta}$$

$$\text{[F-RecP]}\quad \frac{\Gamma, X : (L', \Delta); L' \vdash P \triangleright \Delta \qquad \Gamma; L' \vdash Q \triangleright \Delta \qquad L \subseteq L'}{\Gamma; L \vdash (\mathrm{rec}^e\, X(i).P; Q) \triangleright \Delta}$$

$$\text{[F-Rec]}\quad \frac{\Gamma, X\colon (\emptyset, I, \Delta); I \vdash P \triangleright \Delta \quad L \subseteq I}{\Gamma; L \vdash \mathrm{rec}\, X.P \triangleright \Delta} \qquad \text{[F-Cond]}\quad \frac{\Gamma; L \vdash P \triangleright \Delta \quad \Gamma; L \vdash Q \triangleright \Delta}{\Gamma; L \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta}$$

Fig. F. Typing System

Our typing system is in Fig. F. The rules [F-Bra]/[F-Sel] types branch/select. To type $k!l.P$ wrt. $k : \oplus l[L'].T$, P must do every response in L'. For this we maintain an environment L of pending responses. In the hypothesis, when typing P, we add to this the new pending responses L'. But selecting l performs the response l, so altogether, to support pending responses L in the conclusion, we must have pending responses $L \setminus \{l\} \cup L'$ in the hypothesis. Branching is similar.

For finite processes, liveness is ensured if the inactive process can be typed with the empty request environment. For infinite processes there is no point at which we can insist on having no pending responses. Consider $\mathrm{rec}\, X.\ k!a.\ k!b.\ X[k]$, typeable under $k : \mu t.\ \oplus\{a[b].t; b[a].t\}$. This process has the single transition sequence $P \xrightarrow{k\oplus a} k!b.P \xrightarrow{k\oplus b} P \xrightarrow{k\oplus a} \cdots$. At each state but the initial one either b or a is pending. Yet the process is live: *any response requested in the body of the recursion is also discharged in the body*, although not in order. Since infinite behaviour arises as of unfolding of recursion, responses are ensured if the body of every recursion discharges the requests of that body, even if out of order.

For general recursion, [F-Rec] and [F-Var], we thus find for each recursion a set of responses, such that at most that set is requested in its body and that it reponds with at least that set. In the process variable environment Γ we record this response invariant for each variable, along with a tally of the responses performed since the start of the recursion. The tally is updated by the rules [F-Sel]/[F-Bra] for select and branch. The rule for process variable [F-Var] typing then checks that the tally includes the invariant, and that the invariant includes every currently pending response.

Definition 5.1. *We define* $\mathsf{std}(\Gamma)$*, the* standard *process variable environment of* Γ *by* $\mathsf{std}(\Gamma)(X) = \Delta$ *when* $\Gamma(X) = (A, I, \Delta)$ *or* $\Gamma(X) = \Delta$*.*

Theorem 5.2. *If* $\Gamma; L \vdash P \triangleright \Delta$ *then also* $\mathsf{std}(\Gamma) \vdash_{\mathsf{std}} P \triangleright \Delta$*.*

Theorem 5.3 (Subject reduction). *Suppose that* $\cdot; L \vdash P \triangleright \Delta$ *with and* $P \xrightarrow{\lambda} Q$*. Then there exists a type transition* $\Delta \xrightarrow{\delta} \Delta'$ *with* $\delta \simeq \lambda$*, such that* $\cdot; (L \setminus \mathsf{res}(\delta)) \cup \mathsf{req}(\delta) \vdash Q \triangleright \Delta'$*. Moreover, if* Δ *balanced then also* Δ' *balanced.*

Example 5.4. With the system of Figure F, the process $P(D)$ is typable wrt. the types given Example 3.4. The process $P(D_0)$ on the other hand is not: We have $\cdot; \emptyset \vdash P(D) \triangleright k : T_P, o^+ : T_D, o^- : \overline{T_D}$, but the same does *not* hold for $P(D_0)$. We also exemplify a typing judgment with non-trivial guaranteed responses. The process D, the order-fulfillment part of $P(D)$, can in fact be typed

$$\cdot; \{\mathsf{SI}\} \vdash D \triangleright k : \mu t'. \oplus\{\mathsf{DI}.!.t', \mathsf{SI}.!.\mathtt{end}\},\ o^- : \overline{T_D}$$

Note the left-most $\{\mathsf{SI}\}$, indicating intuitively that this process will eventually select SI in *any* execution. The process D has this property essentially because it is implemented by bounded recursion. □

6 Liveness

We now prove that a lock-free process well-typed under our liveness typing system is indeed live as defined in Def. 4.4. To define lock-freedom and fairness, we must track occurrences of prefixes across transitions. This is straightforward in the absence of a structural congruence; refer to [12] for a formal treatment. Our notion of lock-freedom is derived from [18].

Definition 6.1. *A* prefix M *is a process on one of the forms* $k!\langle e\rangle.P$*,* $k?(x).P$*,* $k?\{l_i.P_i\}$*, or* $k!l.P$*. An* occurence *of a prefix* M *in a process* P *is a path in the abstract syntax tree of* P *to a subterm on the form* M *(see [12] for details). An occurrence of a prefix* P *in* M *where* $P \xrightarrow{\lambda} Q$ *is* preserved *by the latter if* M *has the same occurrence in* Q*;* executed *otherwise. It is* enabled *if it is executed by some transition, and* top-level *if it is not nested in another prefix.*

Definition 6.2. *An infinite transition sequence* $s = (P_i, \lambda_i)_{i\in\mathbb{N}}$ *is* fair *iff whenever a prefix* M *occurs enabled in* P_n *then some* $m \geq n$ *has* $P_m \xrightarrow{\lambda_m} P_{m+1}$ *executing that occurence. A transition sequence* s *is* terminated *iff it has length* n *and* $P_n \not\longrightarrow$*. It is* maximal *iff it is finite and terminated or infinite and fair. A maximal transition sequence* (P_i, λ_i) *is lock-free iff whenever there is a top-level occurence of a prefix* M *in* P_i*, then there exists some* $j \geq i$ *s.t.* $P_j \xrightarrow{\lambda_j} P_{j+1}$ *executes that occurrence. A process is lock-free iff all its transition sequences are.*

Definition 6.3. *For a process transition label* λ*, define* $\mathsf{sel}(\lambda)$ *by* $\mathsf{sel}(k!v) = \mathsf{sel}(k?v) = \mathsf{sel}(\tau) = \emptyset$ *and* $\mathsf{sel}(k\&l) = \mathsf{sel}(k \oplus l) = \mathsf{sel}(\tau : l) = l$*. Given a trace* α *we lift* $\mathsf{sel}(-)$ *pointwise, that is,* $\mathsf{sel}(\alpha) = \{\mathsf{sel}(\lambda) \mid \alpha = \phi\lambda\alpha'\}$*.*

Proposition 6.4. *Suppose* $\cdot\,; L \vdash P \triangleright \Delta$ *with* P *lock-free, and let* $s = (P_i, \alpha_i)_i$ *be a maximal transition sequence of* P*. Then* $L \subseteq \mathsf{sel}(\alpha)$*.*

Example 6.5. We saw in Example 5.4 that the process D of Example 2.1 is typable $\cdot; \{\mathsf{SI}\} \vdash D \triangleright \cdots$. By Proposition 6.4 above, noting that D is clearly lock-free, every maximal transition sequence of D must eventually select SI.

Theorem 6.6. *Suppose* $\cdot\,; L \vdash P \triangleright \Delta$ *with* P *lock-free. Then* P *is live for* $\cdot\,, \Delta$*.*

Example 6.7. We saw in Example 5.4 that $P(D)$ is typable as $\cdot; \emptyset \vdash P(D) \triangleright k : T_P, o^+ : T_D, o^- : \overline{T_D}$. Noting $P(D)$ lock-free, by the above Theorem it is live, and so will uphold the liveness guarantee in T_P: if CO is selected, then eventually also SI is selected. Or in the intuition of the example: If the buyer performs "Checkout", he is guaranteed to subsequently receive an invoice.

7 Conclusion and Future Work

We introduced a conservative generalization of binary session types to *session types with responses*, which allows to specify response liveness properties. We showed that session types with responses are strictly more expressive (wrt. the classes of behaviours they can express) than standard binary session types. We provided a typing system for a process calculus similar to a non-trivial subset of collaborative BPMN processes with possibly infinite loops and bounded iteration and proved that lock-free, well typed processes are live.

We have identified several interesting directions for future work: Firstly, the present techniques could be lifted to multi-party session types, which guarantees lock-freedom. Secondly, investigate more general liveness properties. Thirdly, channel passing is presently omitted for simplicity of presentation and not needed for our expressiveness result (Theorem 3.8). Introducing it, raises the question of wether one can delegate the responsibility for doing responses or not? If *not*, then channel passing does not affect the liveness properties of a lock-free process, and so is not really interesting for the present paper. If one *could*, it must be ensured that responses are not forever delegated without ever being fulfilled, which is an interesting challenge for future work. We hope to leverage existing techniques for the π-calculus, e.g., [18]. Finally, and more speculatively, we plan to investigate relations to fair subtyping [22] and Live Sequence Charts [6].

References

1. Bettini, L., M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini and N. Yoshida, *Global progress in dynamically interleaved multiparty sessions*, in: *CONCUR*, 2008, pp. 418–433.
2. Brill, M., W. Damm, J. Klose, B. Westphal and H. Wittke, *Live sequence charts: An introduction to lines, arrows, and strange boxes in the context of formal verification*, in: *SoftSpez Final Report*, LNCS **3147** (2004), pp. 374–399.

3. Carbone, M. and S. Debois, *A graphical approach to progress for structured communication in web services*, in: *ICE*, 2010, pp. 13–27.
4. Cheung, S.-C., D. Giannakopoulou and J. Kramer, *Verification of liveness properties using compositional reachability analysis*, in: *ESEC/SIGSOFT FSE*, Lecture Notes in Computer Science **1301** (1997), pp. 227–243.
5. Coppo, M., M. Dezani-Ciancaglini, L. Padovani and N. Yoshida, *Inference of global progress properties for dynamically interleaved multiparty sessions*, in: *COORDINATION*, 2013, pp. 45–59.
6. Damm, W. and D. Harel, *Lscs: Breathing life into message sequence charts*, Formal Methods in System Design **19** (2001), pp. 45–80.
7. Debois, S., T. Hildebrandt, T. Slaats and N. Yoshida, *Type checking liveness for collaborative processes with bounded and unbounded recursion (full version)*. URL `http://www.itu.dk/~hilde/liveness-full.pdf`
8. Deniélou, P.-M. and N. Yoshida, *Multiparty session types meet communicating automata*, in: *ESOP*, 2012, pp. 194–213.
9. Deniélou, P.-M. and N. Yoshida, *Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types*, in: *ICALP*, 2013, pp. 174–186.
10. Dezani-Ciancaglini, M., U. de'Liguoro and N. Yoshida, *On progress for structured communications*, in: *TGC*, 2007, pp. 257–275.
11. Dezani-Ciancaglini, M., S. Drossopoulou, D. Mostrous and N. Yoshida, *Objects and session types*, Inf. Comput. **207** (2009), pp. 595–641.
12. Fossati, L., K. Honda and N. Yoshida, *Intensional and extensional characterisation of global progress in the π-calculus*, in: *CONCUR*, 2012, pp. 287–301.
13. Gay, S. J. and M. Hole, *Subtyping for session types in the pi calculus*, Acta Inf. **42** (2005), pp. 191–225.
14. Honda, K., A. Mukhamedov, G. Brown, T.-C. Chen and N. Yoshida, *Scribbling interactions with a formal foundation*, in: *ICDCIT*, 2011, pp. 55–75.
15. Honda, K., V. Vasconcelos and M. Kubo, *Language primitives and type discipline for structured communication-based programming*, in: *ESOP*, 1998, pp. 122–138.
16. Honda, K., N. Yoshida and M. Carbone, *Multiparty asynchronous session types*, in: *POPL*, 2008, pp. 273–284.
17. Hu, R., N. Yoshida and K. Honda, *Session-based distributed programming in Java*, in: J. Vitek, editor, *ECOOP '08*, LNCS **5142**, 2008 pp. 516–541.
18. Kobayashi, N., *A type system for lock-free processes*, I&C **177** (2002), pp. 122–159.
19. Kobayashi, N. and C.-H. L. Ong, *A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes*, in: *LICS* (2009), pp. 179–188.
20. Mostrous, D. and V. T. Vasconcelos, *Session typing for a featherweight Erlang*, in: *COORDINATION*, 2011, pp. 95–109.
21. Object Management Group BPMN Technical Committee, *Business Process Model and Notation, v2.0*, Webpage (2011), `http://www.omg.org/spec/BPMN/2.0/PDF`.
22. Padovani, L., *Fair subtyping for open session types*, in: *ICALP*, 2013, pp. 373–384.
23. Roa, J., O. Chiotti and P. D. Villarreal, *A verification method for collaborative business processes*, in: *Business Process Management Workshops (1)*, Lecture Notes in Business Information Processing **99** (2011), pp. 293–305.
24. Vasconcelos, V., *Fundamentals of session types*, I&C **217** (2012), pp. 52–70.
25. Vieira, H. T. and V. T. Vasconcelos, *Typing progress in communication-centred systems*, in: *COORDINATION*, 2013, pp. 236–250.
26. Yoshida, N. and V. T. Vasconcelos, *Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication*, ENTCS **171** (2007), pp. 73–93.