Multiparty Session Types Meet Communicating Automata

Pierre-Malo Deniélou and Nobuko Yoshida

Department of Computing, Imperial College London

Abstract. Communicating finite state machines (CFSMs) represent processes which communicate by asynchronous exchanges of messages via FIFO channels. Their major impact has been in characterising essential properties of communications such as freedom from deadlock and communication error, and buffer boundedness. CFSMs are known to be computationally hard: most of these properties are undecidable even in restricted cases. At the same time, multiparty session types are a recent typed framework whose main feature is its ability to efficiently enforce these properties for mobile processes and programming languages. This paper ties the links between the two frameworks to achieve a two-fold goal. On one hand, we present a generalised variant of multiparty session types that have a direct semantical correspondence to CFSMs. Our calculus can treat expressive forking, merging and joining protocols that are absent from existing session frameworks, and our typing system can ensure properties such as safety, boundedness and liveness on distributed processes by a polynomial time type checking. On the other hand, multiparty session types allow us to identify a new class of CFSMs that automatically enjoy the aforementioned properties, generalising Gouda et al's work [13] (for two machines) to an arbitrary number of machines.

1 Introduction

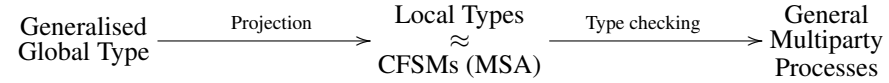
Multiparty Session Types The importance that distributed systems are taking today underlines the necessity for precise specifications and full correctness guarantees for interactions (protocols) between distributed components. To that effect, multiparty session types [3, 15] are a type discipline that can enforce strong communication safety for distributed processes [3, 15], via a choreographic specification (called *global type*) of the interaction between several peers. Global types are then projected to end-point types (called *local types*), against which processes can be statically type-checked. Well-typed processes are guaranteed to interact correctly, following the global protocol. The tool chain (projection and type-checking) is decidable in polynomial time and automatically guarantees properties such as type safety, deadlock freedom, and progress. Multiparty session types are thus directly applicable to the design and implementation of real distributed programming languages. They are used for structured protocol programming in contexts such as security [9, 24], protocol optimisations for distributed objects [23] and parallel algorithms [19], and have recently lead to industrial projects [21, 22].

Communicating Automata or Communicating Finite State Machines (CFSMs) [5], are a classical model for protocol specification and verification. Before being used in many industrial contexts, CFSMs have been a pioneer theoretical formalism in which

distributed safety properties could be formalised and studied. Building a connection between communicating automata and session types allows to answer some open questions in session types which have been asked since [14]. The first question is about expressiveness: to which class of CFSMs do session types correspond? The second question concerns the semantical correspondence between session types and CFSMs: how do the safety properties that session types guarantee relate to those of CFSMs? The third question is about efficiency: why do session types provide polynomial algorithms while general CFSMs are undecidable?

A First Answer to these questions has been recently given in the *binary* case: a two-machine subclass (which had been studied by Gouda et al. in 1984 [13] and later by Villard [25]) of half-duplex systems [7] (defined as systems where at least one of the two communication buffers between two parties is always empty) has been found to correspond to *binary* session types [14]. This subclass, compatible deterministic two-machine without mixed states [13] (see § 3 and § 7), automatically satisfies the safety properties that binary session types can guarantee. It also explains why binary session types offer a tractable framework since, in two-machine half-duplex systems, safety properties and buffer boundedness are decidable in polynomial time [7]. However, in half-duplex systems with three machines or more, these problems are undecidable (Theorem 36 [7]). This shows that an extension to multiparty is very challenging, leading to two further questions. Can we use a multiparty session framework [15] to define a new class of deadlock-free CFSMs with more than two machines? How far can we extend global session type languages to capture a wider class of well-behaved CFSMs, still preserving expected properties and enabling type-checking processes and languages?

Our Answer is a *theory of generalised multiparty session types*, which can automatically generate, through projection and translation, a new class of safe CFSMs, which we call *multiparty session automata* (MSA). We use MSA as a semantical interpretation of types to prove the safety and liveness of expressive multiparty session mobile processes, allowing complexly structured protocols, including the Alternating Bit Protocol, to be simply represented. Our generalised multiparty session type framework can be summarised by the following diagram:



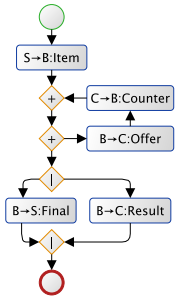
Generalised Global Types This paper proposes a new global type syntax which encompasses previous systems [3, 15] with extended constructs (join and merge) and generalised graph syntax. Its main feature is to explicitly distinguish the branching points (where choices are made) from the forking points (where concurrent, interleaved interaction can take place). Such a distinction is critical to avoid the state explosion and to directly and efficiently type session-based languages and processes.

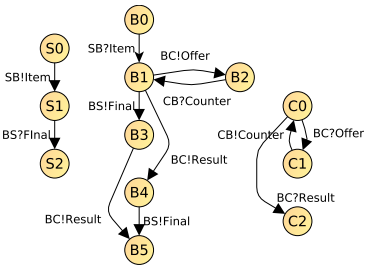
Fig. 1 illustrates our new syntax on a running example, named Trade. For the intuition, Trade is also represented as a BPMN-like [4] activity diagram, where '+' is for exclusive gateways and '|' for parallel ones, following session type conventions.

This scenario (from [6, § 7.3]) comprehensively combines recursion, fork, join, choice and merge. It models a protocol where a seller `S` relies on a broker `B` to negotiate and sell an item to a client `C`. The seller sends a message *Item* to the broker, the

Fig. 1. Trade Example: Global Type and CFSM

$$
\begin{aligned}
\mathbf{G}_{\mathrm{Trade}} &= \mathsf{def} \\
\mathbf{x}_0 &= \mathtt{S} \to \mathtt{B} : \mathit{Item}\langle \mathsf{string} \rangle ; \mathbf{x}_1 \\
\mathbf{x}_5 + \mathbf{x}_1 &= \mathbf{x}_2 \\
\mathbf{x}_2 &= \mathbf{x}_3 + \mathbf{x}_6 \\
\mathbf{x}_3 &= \mathtt{B} \to \mathtt{C} : \mathit{Offer}\langle \mathsf{nat} \rangle ; \mathbf{x}_4 \\
\mathbf{x}_4 &= \mathtt{C} \to \mathtt{B} : \mathit{Counter}\langle \mathsf{nat} \rangle ; \mathbf{x}_5 \\
\mathbf{x}_6 &= \mathbf{x}_7 \mid \mathbf{x}_8 \\
\mathbf{x}_7 &= \mathtt{B} \to \mathtt{S} : \mathit{Final}\langle \mathsf{nat} \rangle ; \mathbf{x}_9 \\
\mathbf{x}_8 &= \mathtt{B} \to \mathtt{C} : \mathit{Result}\langle \mathsf{nat} \rangle ; \mathbf{x}_{10} \\
\mathbf{x}_9 \mid \mathbf{x}_{10} &= \mathbf{x}_{11} \\
\mathbf{x}_{11} &= \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0
\end{aligned}
$$





broker then has a choice between entering the negotiation loop *Offer-Counter* with the client as many times as he chooses, or finishing the protocol by concurrently sending *both* messages *Final* and *Result* to the seller and the client respectively.

$\mathbf{G}_{\mathrm{Trade}}$ is called a *global type* as it represents the choreography of the interactions and not just a collection of local behaviours. It is of the form $\mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ where $\widetilde{G}$ represents the transitions between states, and where $\mathbf{x}_0$ is the initial state of all the participants. A transition of the form $\mathbf{x}_0 = \mathtt{S} \to \mathtt{B} : \mathit{Item}\langle \mathsf{string} \rangle ; \mathbf{x}_1$ corresponds to the emission of a message *Item* carrying a value of type string from S to B, followed by the interactions that happen in $\mathbf{x}_1$. A transition $\mathbf{x}_2 = \mathbf{x}_3 + \mathbf{x}_6$ denotes a choice (done by one of the participants, here B) between following with $\mathbf{x}_3$ or $\mathbf{x}_6$. A transition $\mathbf{x}_6 = \mathbf{x}_7 \mid \mathbf{x}_8$ describes that the interaction should continue concurrently with the actions of $\mathbf{x}_7$ and of $\mathbf{x}_8$. In a symmetric way, a transition $\mathbf{x}_5 + \mathbf{x}_1 = \mathbf{x}_2$ merges two branches that are mutually exclusive, while a transition $\mathbf{x}_9 \mid \mathbf{x}_{10} = \mathbf{x}_{11}$ joins two concurrent interaction threads reaching points $\mathbf{x}_9$ and $\mathbf{x}_{10}$ into a single thread starting from $\mathbf{x}_{11}$.

Local Types and CFSMs We build the formal connection between multiparty session types, CFSMs and processes by first projecting a global type to the local type of each end-point. We then show that the local types are *implementable* as CFSMs. This defines a new subclass of CFSMs, named Multiparty Session Automata, or MSA, that are not limited to two machines or to half-duplex communications, and that automatically satisfy distributed safety and progress.

To illustrate this relationship between local types and MSA, we give in Fig. 1 the CFSM representation of Trade: on the left is the seller S, at the centre the broker B, on the right the client C. These communicating automata correspond to the collection of local behaviours represented by the local types (shown later in Ex. 3.1). Each automaton starts from an initial state $\mathtt{S}_0$, $\mathtt{B}_0$ or $\mathtt{C}_0$ and allows some transitions to be activated. Transitions can either be outputs of the form SB!*Item* where SB indicates the channel between the seller S and the broker B and where *Item* is the message label; or inputs of the symmetric form SB?*Item*. When a sending action happens, the message label is appended to the channel's FIFO queue. Activating an input action requires the expected label to appear on top of the specified queue.

The connection between local types and CFSM gives a formal semantics to global types and creates a correspondence between CFSM and session type properties.

Fig. 2. Generalised Global Types

$\mathbf{G}$	::=	$\mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}$	Global type	U	::=	$\langle \mathbf{G} \rangle \mid \mathsf{bool} \mid \mathsf{nat} \mid \cdots$ Sorts
G	::=	$\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle ; \mathbf{x}'$	Labelled messages			$\mathbf{x} \mid \mathbf{x}' = \mathbf{x}''$ Join
	$\mid$	$\mathbf{x} = \mathbf{x}' \mid \mathbf{x}''$	Fork			$\mathbf{x} + \mathbf{x}' = \mathbf{x}''$ Merge
	$\mid$	$\mathbf{x} = \mathbf{x}' + \mathbf{x}''$	Choice			$\mathbf{x} = \mathsf{end}$ End

Our Contributions are listed below, with the corresponding section number:

- We introduce new generalised multiparty (global and local) session types that solve open problems of expressiveness and algorithmic projection posed in [6] (§ 2).
- We give a CFSM interpretation of local types that defines a formal semantics for global types and allows the standardisation of distributed safety properties between session type systems and communicating automata (§ 3).
- We define multiparty session automata, a new communicating automata subclass that automatically satisfy strong distributed safety properties, solving open questions from [7, 25] (§ 3).
- We develop a new typing system for multiparty session mobile processes generalised with choice, fork, merge and join constructs (§ 4, § 5), and prove that typed processes conform the safety and liveness properties defined in CFSMs (§ 6).
- We compare our framework with existing session type theories and CFSMs results (§ 7). Our framework (global type well-formedness checking, projection, typechecking) is notably polynomial in the size of the global type or mobile processes.

The appendix provides proofs, auxiliary definitions and examples.

2 Generalised Multiparty Sessions

2.1 Global Types for Generalised Multiparty Sessions

This subsection introduces new *generalised global types*, whose expressiveness encompasses previous session frameworks. The syntax is defined in Fig. 2.The new features are flexible fork, choice, merge and join operations for precise thread management. A global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ describes an interaction between a fixed number of participants. The prescribed interaction starts from $\mathbf{x}_0$, which we call the *initial state*, and proceeds according to the transitions specified in $\widetilde{G}$. The *state variables* $\mathbf{x}$ in $\widetilde{G}$ represent the successive distributed states of the interaction. Transitions can be *labelled message exchanges* $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle ; \mathbf{x}'$ where $\mathtt{p}$ and $\mathtt{p}'$ denote the sending and receiving *participants* (process identities), U is the payload type of the message and l its label. This transition specifies that $\mathtt{p}$ can go from $\mathbf{x}$ to the continuation $\mathbf{x}'$ by sending message l, while $\mathtt{p}'$ goes from $\mathbf{x}$ to $\mathbf{x}'$ by receiving it. All other participants can go from $\mathbf{x}$ to $\mathbf{x}'$ for free. Sort types § include shared channel types $\langle \mathbf{G} \rangle$ or base types. Message types U are either value types S or local types $\mathbf{T}$ (which correspond to the behaviour of one of the session participants) for delegation, which is defined later.$\mathbf{x} = \mathbf{x}' + \mathbf{x}''$ represents the choice (made by exactly one participant) between continuing with $\mathbf{x}'$ or $\mathbf{x}''$ and

$\mathbf{x} = \mathbf{x}' \mid \mathbf{x}''$ represents forking the interactions, allowing the interleaving of actions at $\mathbf{x}'$ and $\mathbf{x}''$. These forking threads are eventually collected by joining construct $\mathbf{x}' \mid \mathbf{x}'' = \mathbf{x}$. Similarly choices are closed by merging construct $\mathbf{x}' + \mathbf{x}'' = \mathbf{x}$, where two mutually exclusive paths share a continuation. $\mathbf{x} = \mathsf{end}$ denotes session termination.

The motivation behind this choice of graph syntax is to support general graphs. A traditional global type syntax tree, with operators fork $\mid$ and choice $+$, even with recursion [3, 6, 11, 15], is limited to series-parallel graphs.

Example 2.1 (Generalised Global Types)). We now give several examples in Fig. 3, with their graph representation. We keep this representation informal throughout this paper (although there is an exact match with the syntax: variables are edges and transitions are nodes). The examples are numbered 1–7, with increasing complexity.

1. A simple one-message (*Msg* of type nat) is exchanged between `Alice` and `Bob`.
2. A protocol with a simple choice between messages *Book* and *Film*.
3. `Alice` and `Bob` concurrently exchange the messages *Book* and *Film*.
4. A protocol where `Alice` keeps sending successive messages to `Bob` (recursion is written using merging).
5. The Trade example from § 1 (Fig. 1) shows how choice, recursion and parallelism can be integrated to model a three party protocol.
6. $\mathbf{G}_6$ features an initial choice between directly contacting `Carol` or to do it through `Bob`. Note that without the last interaction from `Carol` to `Bob` (in $\mathbf{x}_6$), if the chosen path leads to $\mathbf{x}_3$, `Bob` enters a deadlock, waiting forever for a message from `Alice`.
7. $\mathbf{G}_{AB}$ in Fig. 3 gives a representation of ***the Alternating Bit Protocol***. `Alice` repeatedly sends to `Bob` alternating messages Msg_1 and Msg_2 but will always concurrently wait for the acknowledgement Ack_i to send Msg_i. This interaction structure requires a general graph syntax and is thus not representable in any existing session type framework, and is difficult in other formalisms (see § 7). We emphasise the fact that, not only it is representable in our syntax, but our framework is able to demonstrate its progress and safety and enforce it on realistic processes.

2.2 Well-formed Global Types

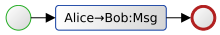
This subsection defines three well-formedness conditions for global types.

Sanity Conditions within global types prevent possible syntactic confusions about which continuations to follow at any given point. A global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ satisfies the *sanity* conditions if it satisfies the following conditions.

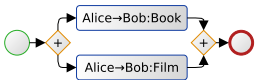
1. (**Unambiguity**) Every state variable $\mathbf{x}$ except $\mathbf{x}_0$ should appear exactly once on the left-hand side and once on the right-hand side of the transitions in $\widetilde{G}$.
2. (**Unique start**) $\mathbf{x}_0$ appears exactly once, on the left-hand side.
3. (**Unique end**) end appears at most once.
4. (**Thread correctness**) The transitions $\widetilde{G}$ define a connected graph where threads are always collected by joins.

Fig. 3. Examples of Global Types

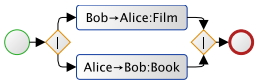
1. $\mathbf{G}_1 = \mathsf{def}\ \mathbf{x}_0 = \mathtt{Alice} \to \mathtt{Bob} : \mathit{Msg}\langle \mathsf{nat} \rangle ; \mathbf{x}_1$
   $\mathbf{x}_1 = \mathsf{end}\ \mathsf{in}\ \mathbf{x}_0$



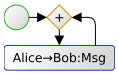
2. $\mathbf{G}_2 = \mathsf{def}\ \mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_2$
   $\mathbf{x}_1 = \mathtt{Alice} \to \mathtt{Bob} : \mathit{Book}\langle \mathsf{string} \rangle ; \mathbf{x}_3$
   $\mathbf{x}_2 = \mathtt{Alice} \to \mathtt{Bob} : \mathit{Film}\langle \mathsf{string} \rangle ; \mathbf{x}_4$
   $\mathbf{x}_3 + \mathbf{x}_4 = \mathbf{x}_5$
   $\mathbf{x}_5 = \mathsf{end}\ \mathsf{in}\ \mathbf{x}_0$



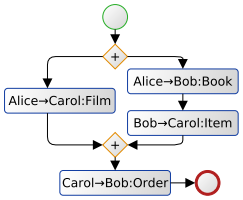
3. $\mathbf{G}_3 = \mathsf{def}\ \mathbf{x}_0 = \mathbf{x}_1 \mid \mathbf{x}_2$
   $\mathbf{x}_1 = \mathtt{Alice} \to \mathtt{Bob} : \mathit{Book}\langle \mathsf{string} \rangle ; \mathbf{x}_3$
   $\mathbf{x}_2 = \mathtt{Bob} \to \mathtt{Alice} : \mathit{Film}\langle \mathsf{string} \rangle ; \mathbf{x}_4$
   $\mathbf{x}_3 \mid \mathbf{x}_4 = \mathbf{x}_5$
   $\mathbf{x}_5 = \mathsf{end}\ \mathsf{in}\ \mathbf{x}_0$



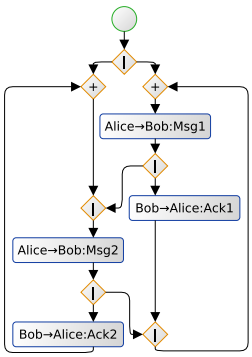
4. $\mathbf{G}_4 = \mathsf{def}\ \mathbf{x}_0 + \mathbf{x}_2 = \mathbf{x}_1$
   $\mathbf{x}_1 = \mathtt{Alice} \to \mathtt{Bob} : \mathit{Msg}\langle \mathsf{string} \rangle ; \mathbf{x}_2\ \mathsf{in}\ \mathbf{x}_0$



6. $\mathbf{G}_6 = \mathsf{def}\ \mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_3$
   $\mathbf{x}_1 = \mathtt{Alice} \to \mathtt{Bob} : \mathit{Book}\langle \mathsf{string} \rangle ; \mathbf{x}_2$
   $\mathbf{x}_2 = \mathtt{Bob} \to \mathtt{Carol} : \mathit{Item}\langle \mathsf{nat} \rangle ; \mathbf{x}_4$
   $\mathbf{x}_3 = \mathtt{Alice} \to \mathtt{Carol} : \mathit{Film}\langle \mathsf{string} \rangle ; \mathbf{x}_5$
   $\mathbf{x}_4 + \mathbf{x}_5 = \mathbf{x}_6$
   $\mathbf{x}_6 = \mathtt{Carol} \to \mathtt{Bob} : \mathit{Order}\langle \mathsf{string} \rangle ; \mathbf{x}_7$
   $\mathbf{x}_7 = \mathsf{end}\ \mathsf{in}\ \mathbf{x}_0$



7. $\mathbf{G}_{AB} = \mathsf{def}\ \mathbf{x}_0 = \mathbf{x}_1 \mid \mathbf{x}_2$
   $\mathbf{x}_1 + \mathbf{x}_3 = \mathbf{x}_4$
   $\mathbf{x}_2 + \mathbf{x}_5 = \mathbf{x}_6$
   $\mathbf{x}_4 = \mathtt{Alice} \to \mathtt{Bob} : \mathit{Msg}_1\langle \mathsf{string} \rangle ; \mathbf{x}_7$
   $\mathbf{x}_7 = \mathbf{x}_8 \mid \mathbf{x}_9$
   $\mathbf{x}_8 = \mathtt{Bob} \to \mathtt{Alice} : \mathit{Ack}_1\langle \mathsf{unit} \rangle ; \mathbf{x}_{10}$
   $\mathbf{x}_6 \mid \mathbf{x}_9 = \mathbf{x}_{11}$
   $\mathbf{x}_{11} = \mathtt{Alice} \to \mathtt{Bob} : \mathit{Msg}_2\langle \mathsf{string} \rangle ; \mathbf{x}_{12}$
   $\mathbf{x}_{12} = \mathbf{x}_{13} \mid \mathbf{x}_{14}$
   $\mathbf{x}_{13} = \mathtt{Bob} \to \mathtt{Alice} : \mathit{Ack}_2\langle \mathsf{unit} \rangle ; \mathbf{x}_5$
   $\mathbf{x}_{10} \mid \mathbf{x}_{14} = \mathbf{x}_3\ \mathsf{in}\ \mathbf{x}_0$



The conditions (1–3) are self-explanatory. (Thread correctness) aims at verifying connexity, the ability to reach end (liveness) and that global types should always join states that occur concurrently and only them: this prevents both deadlocks and state explosion (see Appendix B.2 for the polynomial verification algorithm). In $\mathbf{G}_{\neg\mathrm{thr}}$ in Fig. 4, an illegal join waits for two mutually exclusive messages: as a consequence, Bob is in a deadlock, waiting for both *Book* and *Film* to arrive from Alice.

Fig. 4. Incorrect Global Types

$$\begin{array}{l} \mathbf{G}_{\neg\text{thr}} = \mathsf{def}\ \mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_2 \\ \quad \mathbf{x}_1 = \texttt{Alice} \to \texttt{Bob} : \mathit{Book}\langle\mathsf{string}\rangle;\mathbf{x}_3 \\ \quad \mathbf{x}_2 = \texttt{Alice} \to \texttt{Bob} : \mathit{Film}\langle\mathsf{string}\rangle;\mathbf{x}_4 \\ \mathbf{x}_3 \mid \mathbf{x}_4 = \mathbf{x}_5 \\ \quad \mathbf{x}_5 = \texttt{Bob} \to \texttt{Alice} : \mathit{Price}\langle\mathsf{nat}\rangle;\mathbf{x}_6 \\ \quad \mathbf{x}_6 = \mathsf{end\ in}\ \mathbf{x}_0 \end{array} \qquad \begin{array}{l} \mathbf{G}_{\neg\text{loc}} = \mathsf{def}\ \mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_2 \\ \quad \mathbf{x}_1 = \texttt{Alice} \to \texttt{Bob} : \mathit{Book}\langle\mathsf{string}\rangle;\mathbf{x}_3 \\ \quad \mathbf{x}_2 = \texttt{Bob} \to \texttt{Alice} : \mathit{Film}\langle\mathsf{string}\rangle;\mathbf{x}_4 \\ \mathbf{x}_3 + \mathbf{x}_4 = \mathbf{x}_5 \\ \quad \mathbf{x}_5 = \mathsf{end\ in}\ \mathbf{x}_0 \end{array}$$

Local Choice is essential for the consistency of a global type with respect to choice (branching). For $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$, we need to check that each choice is clearly labelled, local to a participant (the choice of which branch to follow should be made by a unique participant) and propagated to the others. To this effect, we define a function $\mathit{Rcv}(\widetilde{G})(\mathbf{x})$ in Fig. 5, which computes the set of all the participants that will be expecting at least one message starting from state $\mathbf{x}$. Additionally, $\mathit{Rcv}(\widetilde{G})(\mathbf{x})$ returns the label l of the received message and the merging points $\tilde{\mathbf{x}}$ encountered. We say that the equality $\mathit{Rcv}(\widetilde{G})(\mathbf{x}_1) = \mathit{Rcv}(\widetilde{G})(\mathbf{x}_2)$ holds if $\forall(\mathtt{p} : l_1 : \tilde{\mathbf{x}}_1) \in \mathit{Rcv}(\widetilde{G})(\mathbf{x}_1), \forall(\mathtt{p} : l_2 : \tilde{\mathbf{x}}_2) \in \mathit{Rcv}(\widetilde{G})(\mathbf{x}_2), l_1 \neq l_2 \vee \tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2$ share a non-null suffix (i.e. the two branches have merged). Note that $\mathbf{G}_6$ in Ex. 2.1 satisfies this condition (the *Rcv* sets of both branches contain `Bob` and `Carol`).
To guarantee that choices are local to a participant, we also define a function that asserts that, for a choice $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2 \in \widetilde{G}$, a unique sender $\mathtt{p}$ is active in each branch $\mathbf{x}_1$ and $\mathbf{x}_2$. This is written $\mathit{ASend}(\widetilde{G})(\mathbf{x}) = \mathtt{p}$ and is undefined if there is more than one active sender (i.e. if the choice is not localised at a unique participant $\mathtt{p}$) (the definition is in Appendix B.3). As an example, Fig. 4 gives an illegal global type $\mathbf{G}_{\neg\text{loc}}$ where `Alice` and `Bob` are respectively the active sender of branches $\mathbf{x}_1$ and $\mathbf{x}_2$: as both branches do not agree, the mutual exclusion of *Book* and *Film* can be violated.

Definition 2.1 (Local Choice). A global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ satisfies the local choice conditions if for every transition $\mathbf{x} = \mathbf{x}' + \mathbf{x}'' \in \widetilde{G}$, we have **(1)** (**Choice awareness**) $\mathit{Rcv}(\widetilde{G})(\mathbf{x}') = \mathit{Rcv}(\widetilde{G})(\mathbf{x}'')$; and **(2)** (**Unique sender**) $\exists \mathtt{p}, \mathit{ASend}(\widetilde{G})(\mathbf{x}) = \mathtt{p}$.

Linearity In order to avoid processes with race-conditions, we impose that no participant can be faced with two concurrent receptions where messages can have the same label. This condition, *linearity*, is enforced by comparing the results of $\mathit{Lin}(\widetilde{G})(\mathbf{x}_1)$ and $\mathit{Lin}(\widetilde{G})(\mathbf{x}_2)$ whenever a forking transition $\mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2$ is in $\widetilde{G}$. The *Lin* function works in a similar way on message labels as the *Rcv* function on message receivers (linearity is to forks what choice awareness is to choice) and it thus omitted here. As an example, linearity would prevent the labels $\mathit{Msg}1$ and $\mathit{Msg}2$ from both being renamed $\mathit{Msg}0$ in $\mathbf{G}_{AB}$ (since they can be received concurrently and thus confused), but would allow the two labels of $\mathbf{G}_3$ to be identical (they are received by two different parties). Note that the linearity condition incidentally prevents the unbounded creation of threads.

Definition 2.2 (Linearity). *A global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ satisfies the linearity condition if, for every transition $\mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \in \widetilde{G}$, we have $\mathit{Lin}(\widetilde{G})(\mathbf{x}') = \mathit{Lin}(\widetilde{G})(\mathbf{x}'')$.*

Fig. 5. Receiver Computation (up to permutation of $|$ and $+$)

$$Rcv(\widetilde{G})(\mathbf{x}) = Rcv(\widetilde{G},\emptyset,\emptyset)(\mathbf{x}) \quad \text{(remembers recursive calls and receivers)}$$
$$Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}) = Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}') \quad \text{if } \mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle; \mathbf{x}' \in \widetilde{G} \wedge \mathtt{p}' \in \tilde{\mathtt{p}} \text{ or if } \mathbf{x} \mid \mathbf{x}'' = \mathbf{x}' \in \widetilde{G}$$
$$Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}) = \{\mathtt{p}' : l : \tilde{\mathbf{x}}\} \cup Rcv(\widetilde{G},\tilde{\mathbf{x}},\mathtt{p}'\tilde{\mathtt{p}})(\mathbf{x}') \quad \text{if } \mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle; \mathbf{x}' \in \widetilde{G} \wedge \mathtt{p}' \notin \tilde{\mathtt{p}}$$
$$Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}) = Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}') \cup Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}'') \quad \text{if } \mathbf{x} = \mathbf{x}' + \mathbf{x}'' \in \widetilde{G} \text{ or } \mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \in \widetilde{G}$$
$$Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}) = \emptyset \quad \text{if } \mathbf{x}+\mathbf{x}' = \mathbf{x}'' \in \widetilde{G} \wedge \mathbf{x}'' \in \tilde{\mathbf{x}} \text{ or if } \mathbf{x} = \mathsf{end} \in \widetilde{G}$$
$$Rcv(\widetilde{G},\tilde{\mathbf{x}},\tilde{\mathtt{p}})(\mathbf{x}) = Rcv(\widetilde{G},\tilde{\mathbf{x}}\mathbf{x}'',\tilde{\mathtt{p}})(\mathbf{x}'') \quad \text{if } \mathbf{x}' + \mathbf{x} = \mathbf{x}'' \in \widetilde{G} \wedge \mathbf{x}'' \notin \tilde{\mathbf{x}}$$

Well-formedness We say that a global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ is *well-formed*, if it satisfies the *sanity*, *local choice* and *linearity* conditions. These conditions are related to similar CFSM properties, as discussed in § 3.2. We can easily check that global types from Ex. 2.1 are well-formed. Since *Rcv*, *ASend* and *Lin* can be computed in polynomial time in the size of $\mathbf{G}$ by a simple syntax graph traversal, we have:

Proposition 2.1 (Well-formedness Verification). *Given* $\mathbf{G}$*, we can determine whether* $\mathbf{G}$ *is well-formed or not in polynomial time.*

3 Multiparty Session Automata (MSA) and their Properties

This section starts by defining local types, details the translation from local types into CFSMs, and shows that these CFSMs guarantee the properties given in § 3.3. We call this class of communicating systems *multiparty session automata* (MSA).

3.1 Local Types and the Projection Algorithm

Local types are defined in Fig. 6. They represent the actions of session end-points that each process implementation must follow. As for global types, a local type $\mathbf{T}$ follows the shape of a state machine definition: local types are of the form $\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_0$.

The local type for send $(!\langle \mathtt{p}, l\langle U\rangle\rangle)$ corresponds to the action of sending to $\mathtt{p}$ a message with label l and type U, while receive $(?\langle \mathtt{p}, l\langle U\rangle\rangle)$ is the action of receiving from $\mathtt{p}$ a message with label l and type U. Other behaviours are the indirection (nop), internal choice, external choice, merge, fork, join and end. Note that merge is used for both internal and external choices.

We define the projection of a well-formed global type $\mathbf{G}$ to the local type of participant $\mathtt{p}$ (written $\mathbf{G} \upharpoonright \mathtt{p}$) in Fig. 7. The projection is straightforward: $\mathbf{x} = \mathtt{p} \to \mathtt{q} : l\langle U\rangle; \mathbf{x}'$ is an output from $\mathtt{p}$'s viewpoint and an input from $\mathtt{q}$'s viewpoint; otherwise it creates an indirection link from $\mathbf{x}$ to $\mathbf{x}'$ (i.e. this message exchange is invisible). Choice $\mathbf{x} = \mathbf{x}' + \mathbf{x}''$ is projected to the internal choice if $\mathtt{p}$ is the unique (thanks to the local choice well-formedness condition of definition 2.1) participant deciding on which branch to choose; otherwise the projection gives an external choice. For local types, we also define a congruence relation $\equiv$ over $\widetilde{T}$ which eliminates the indirections ($\widetilde{T}, \mathbf{x} = \mathbf{x}' \equiv \widetilde{T}[\mathbf{x}/\mathbf{x}']$) and locally irrelevant choices, and removes the unused local threads. Appendix C.2 gives the definition.

Thanks to the simplicity of projection, we have:

Fig. 6. Generalised Local Types

$\mathbf{T}$	::=	$\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}$	local type				
T	::=	$\mathbf{x} = !\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}'$	send	$\mathbf{x} = \mathbf{x}' \oplus \mathbf{x}''$	internal choice	$\mathbf{x} = \mathbf{x}' \mid \mathbf{x}''$	fork
		$\mathbf{x} = ?\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}'$	receive	$\mathbf{x} = \mathbf{x}' \,\&\, \mathbf{x}''$	external choice	$\mathbf{x} \mid \mathbf{x}' = \mathbf{x}''$	join
		$\mathbf{x} = \mathbf{x}'$	indirection	$\mathbf{x} + \mathbf{x}' = \mathbf{x}''$	merge	$\mathbf{x} = \mathsf{end}$	end

Proposition 3.1 (Projection). *Given a well-formed* $\mathbf{G}$*, the computation of* $\mathbf{G} \upharpoonright \mathtt{p}$ *is linear in the size of* $\mathbf{G}$*.*

Example 3.1 (Trade Example). We illustrate our projection algorithm by showing the result of the projection of the global type $\mathbf{G}_{\text{Trade}}$ from § 1 to the three local types of the seller $\mathbf{T}_{\text{TradeS}}$, the broker $\mathbf{T}_{\text{TradeB}}$ and the client $\mathbf{T}_{\text{TradeC}}$. Local type congruence rules are used to simplify the result. When comparing with the CFSMs of Fig. 1, one can observe the similarities but also that local types make the interaction structure clearer and more compact thanks to more precise type constructs ($\oplus$, & and $|$).

3.2 Communicating Finite State Machines

In this subsection, we give some preliminary notations (following [7]) and definitions that are relevant to establishing the CFSM connection to local types.

Definitions ε is the empty word. $\mathbb{A}$ is a finite alphabet and $\mathbb{A}^*$ is the set of all finite words over $\mathbb{A}$. $|x|$ is the length of a word x and $x.y$ or xy the concatenation of two words x and y. Let $\mathcal{P}$ be a set of process identities fixed throughout the paper: $\mathcal{P} \subseteq \{\mathtt{Alice}, \mathtt{Bob}, \mathtt{Carol}, \ldots, \mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots, \mathtt{S}, \ldots\}$.

Definition 3.1 (CFSM). A communicating finite state machine is a finite transition system given by a 5-tuple $M = (Q, C, q_0, \mathbb{A}, \delta)$ where (1) Q is a finite set of *states*; (2) $C = \{\mathtt{pq} \in \mathcal{P}^2 \mid \mathtt{p} \neq \mathtt{q}\}$ is a set of channels; (3) $q_0 \in Q$ is an initial state; (4) $\mathbb{A}$ is a finite *alphabet* of messages, and (5) $\delta \subseteq Q \times (C \times \{!, ?\} \times \mathbb{A}) \times Q$ is a finite set of *transitions*.

In transitions, $\mathtt{pq}!a$ denotes the *sending* action of a from process $\mathtt{p}$ to process $\mathtt{q}$, and $\mathtt{pq}?a$ denotes the *receiving* action of a from $\mathtt{p}$ by $\mathtt{q}$. $\pi, \pi', \ldots$ range over actions. A state $q \in Q$ whose outgoing transitions are all labelled with sending (resp. receiving) actions is called a *sending* (resp. *receiving*) state. A state $q \in Q$ which does not have any outgoing transition is called a *final* state. If q has both sending and receiving outgoing transitions, then q is called *mixed*.

A *path* in M is a finite sequence of $q_0, \ldots, q_n$ ($n \geq 1$) such that $(q_i, \pi, q_{i+1}) \in \delta$ ($0 \leq i \leq n-1$), and we write $q \xrightarrow{\pi} q'$ if $(q, \pi, q') \in \delta$. M is *connected* if for every state $q \neq q_0$, there is a path from q_0 to q. Hereafter we assume each CFSM is connected.

A CFSM $M = (Q, C, q_0, \mathbb{A}, \delta)$ is *deterministic* if for all states $q \in Q$ and all actions π, $(q, \pi, q'), (q, \pi, q'') \in \delta$ imply $q' = q''$.[1]

[1] "Deterministic" often means the same channel should carry a unique value, i.e. if $(q, c!a, q') \in \delta$ and $(q, c!a', q'') \in \delta$ then $a = a'$ and $q' = q''$. Here we follow a different definition [7] in order to represent branching type constructs.

Fig. 7. Projection Algorithm

$$
\begin{array}{rcl}
\mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x} \upharpoonright \mathtt{p} & = & \mathsf{def}\ \widetilde{G} \upharpoonright_{\widetilde{G}} \mathtt{p}\ \mathsf{in}\ \mathbf{x} \\
\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle; \mathbf{x}' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} = !\langle \mathtt{p}', l\langle U \rangle \rangle.\mathbf{x}' \\
\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle; \mathbf{x}' \upharpoonright_{\widetilde{G}} \mathtt{p}' & = & \mathbf{x} = ?\langle \mathtt{p}, l\langle U \rangle \rangle.\mathbf{x}' \\
\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle; \mathbf{x}' \upharpoonright_{\widetilde{G}} \mathtt{p}'' & = & \mathbf{x} = \mathbf{x}' \ (\mathtt{p} \notin \{\mathtt{p}, \mathtt{p}'\}) \\
\mathbf{x} \mid \mathbf{x}' = \mathbf{x}'' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} \mid \mathbf{x}' = \mathbf{x}'' \\
\mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} = \mathbf{x}' \mid \mathbf{x}''
\end{array}
\qquad
\begin{array}{rcl}
\mathbf{x} = \mathbf{x}' + \mathbf{x}'' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} = \mathbf{x}' \oplus \mathbf{x}'' \\
 & & (\text{if } \mathtt{p} = \mathit{ASend}(\widetilde{G})(\mathbf{x})) \\
\mathbf{x} = \mathbf{x}' + \mathbf{x}'' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} = \mathbf{x}' \ \& \ \mathbf{x}'' \\
 & & (\text{otherwise}) \\
\mathbf{x} + \mathbf{x}' = \mathbf{x}'' \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} + \mathbf{x}' = \mathbf{x}'' \\
\mathbf{x} = \mathsf{end} \upharpoonright_{\widetilde{G}} \mathtt{p} & = & \mathbf{x} = \mathsf{end}
\end{array}
$$

Definition 3.2 (CS). A (communicating) system S is a tuple $S = (M_\mathtt{p})_{\mathtt{p} \in \mathcal{P}}$ of CFSMs such that $M_\mathtt{p} = (Q_\mathtt{p}, C, q_{0\mathtt{p}}, \mathbb{A}, \delta_\mathtt{p})$.

Let $S = (M_\mathtt{p})_{\mathtt{p} \in \mathcal{P}}$ such that $M_\mathtt{p} = (Q_\mathtt{p}, C, q_{0\mathtt{p}}, \mathbb{A}, \delta_\mathtt{p})$ and $\delta = \uplus_{\mathtt{p} \in \mathcal{P}} \delta_\mathtt{p}$. A configuration of S is a tuple such that $s = (\vec{q}; \vec{w})$ with $\vec{q} = (q_\mathtt{p})_{\mathtt{p} \in \mathcal{P}}$ with $q_\mathtt{p} \in Q_\mathtt{p}$ and $\vec{w} = (w_{\mathtt{pq}})_{\mathtt{p} \neq \mathtt{q} \in \mathcal{P}}$ with $w_{\mathtt{pq}} \in \mathbb{A}^*$. A configuration $s' = (\vec{q}'; \vec{w}')$ is *reachable* from another configuration $s = (\vec{q}; \vec{w})$ by the *firing of the transition* t, written $s \to s'$ or $s \xrightarrow{t} s'$, if there exists $a \in \mathbb{A}$ such that either:

1. $t = (q_\mathtt{p}, \mathtt{pq}!a, q'_\mathtt{p}) \in \delta_\mathtt{p}$ and (a) $q'_{\mathtt{p}'} = q_{\mathtt{p}'}$ for all $\mathtt{p}' \neq \mathtt{p}$; and (b) $w'_{\mathtt{pq}} = w_{\mathtt{pq}}.a$ and $w'_{\mathtt{p}'\mathtt{q}'} = w_{\mathtt{p}'\mathtt{q}'}$ for all $\mathtt{p}'\mathtt{q}' \neq \mathtt{pq}$; or
2. $t = (q_\mathtt{q}, \mathtt{pq}?a, q'_\mathtt{q}) \in \delta_\mathtt{q}$ and (a) $q'_{\mathtt{p}'} = q_{\mathtt{p}'}$ for all $\mathtt{p}' \neq \mathtt{q}$; and (b) $w_{\mathtt{pq}} = a.w'_{\mathtt{pq}}$ and $w'_{\mathtt{p}'\mathtt{q}'} = w_{\mathtt{p}'\mathtt{q}'}$ for all $\mathtt{p}'\mathtt{q}' \neq \mathtt{pq}$.

The condition (1-b) puts the content a to a channel $\mathtt{pq}$, while (2-b) gets the content a from a channel $\mathtt{pq}$. The reflexive and transitive closure of $\to$ is $\to^*$. For a transition $t = (s, \pi, s')$, we write $\ell(t) = \pi$. We write $s_1 \xrightarrow{t_1 \cdots t_m} s_{m+1}$ for $s_1 \xrightarrow{t_1} s_2 \cdots \xrightarrow{t_m} s_{m+1}$. We use the metavariable φ to designate sequences of transitions of the form $t_1 \cdots t_m$. The *initial configuration* of the system is $s_0 = (\vec{q}_0; \vec{\varepsilon})$ with $\vec{q}_0 = (q_{0\mathtt{p}})_{\mathtt{p} \in \mathcal{P}}$. A *final configuration* of the system is $s_f = (\vec{q}; \vec{\varepsilon})$ with all $q_\mathtt{p} \in \vec{q}$ final. A configuration s is *reachable* if $s_0 \to^* s$ and we define the *reachable set* of S as $RS(S) = \{s \mid s_0 \to^* s\}$.

Properties Let S be a communicating system, t one of its transitions and $s = (\vec{q}; \vec{w})$ one of its configurations. The following definitions follow [7, Definition 12].

1. s is *stable* if all its buffers are empty, i.e., $\vec{w} = \vec{\varepsilon}$.
2. s is a *deadlock configuration* if $\vec{w} = \vec{\varepsilon}$ and each $q_\mathtt{p}$ is a receiving state, i.e. all machines are blocked, waiting for messages.
3. s is an *orphan message configuration* if all $q_\mathtt{p} \in \vec{q}$ are final but $\vec{w} \neq \emptyset$, i.e. there is at least an orphan message in a buffer.
4. s is an *unspecified reception configuration* if there exists $\mathtt{q} \in \mathcal{P}$ such that $q_\mathtt{q}$ is a receiving state and $(q_\mathtt{q}, \mathtt{pq}?a, q'_\mathtt{q}) \in \delta$ implies that $|w_{\mathtt{pq}}| > 0$ and $w_{\mathtt{pq}} \notin a\mathbb{A}^*$, i.e $q_\mathtt{q}$ is prevented from receiving any message from buffer $\mathtt{pq}$.

The set of *receivers* of transitions $s_1 \xrightarrow{t_1 \cdots t_m} s_{m+1}$ is defined as $Rcv(t_1 \cdots t_m) = \{\mathtt{q} \mid \exists i \leq m, t_i = (s_i, \mathtt{pq}?a, s_{i+1})\}$. The set of *active senders* are defined as $\mathit{ASend}(t_1 \cdots t_m) = \{\mathtt{p} \mid \exists i \leq m, t_i = (s_i, \mathtt{pq}!a, s_{i+1}) \wedge \forall k < i.\ t_k \neq (s_k, \mathtt{p}'\mathtt{p}?b, s_{k+1})\}$ and represent the participants who could immediately send from state s_1. These definitions match the global

Fig. 8. Trade Example: Local Types

$$\mathbf{T}_{\mathrm{TradeS}} = \begin{array}{l}\mathsf{def}\ \mathbf{x}_0 = !\langle \mathtt{SB}, \mathit{Item}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_1 \\ \quad \mathbf{x}_1 = ?\langle \mathtt{BS}, \mathit{Final}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_{10} \\ \quad \mathbf{x}_{10} = \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0\end{array}$$

$$\mathbf{T}_{\mathrm{TradeB}} = \begin{array}{l}\mathsf{def}\ \mathbf{x}_0 = ?\langle \mathtt{SB}, \mathit{Item}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_1 \\ \quad \mathbf{x}_5 + \mathbf{x}_1 = \mathbf{x}_2 \\ \quad\quad \mathbf{x}_2 = \mathbf{x}_3 \oplus \mathbf{x}_6 \\ \quad\quad \mathbf{x}_3 = !\langle \mathtt{BC}, \mathit{Offer}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_4 \\ \quad\quad \mathbf{x}_4 = ?\langle \mathtt{CB}, \mathit{Counter}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_5 \\ \quad\quad \mathbf{x}_6 = \mathbf{x}_7 \mid \mathbf{x}_8 \\ \quad\quad \mathbf{x}_7 = !\langle \mathtt{BS}, \mathit{Final}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_9 \\ \quad\quad \mathbf{x}_8 = !\langle \mathtt{CB}, \mathit{Result}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_{10} \\ \quad \mathbf{x}_9 \mid \mathbf{x}_{10} = \mathbf{x}_{11} \\ \quad\quad \mathbf{x}_{11} = \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0\end{array}$$

$$\mathbf{T}_{\mathrm{TradeC}} = \begin{array}{l}\mathsf{def}\ \mathbf{x}_5 + \mathbf{x}_0 = \mathbf{x}_2 \\ \quad \mathbf{x}_2 = \mathbf{x}_3 \ \&\ \mathbf{x}_6 \\ \quad \mathbf{x}_3 = ?\langle \mathtt{BC}, \mathit{Offer}\langle \mathit{nat}\rangle\rangle.\mathbf{x}_4 \\ \quad \mathbf{x}_4 = !\langle \mathtt{CB}, \mathit{Counter}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_5 \\ \quad \mathbf{x}_6 = ?\langle \mathtt{BC}, \mathit{Result}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_{10} \\ \quad \mathbf{x}_{10} = \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0\end{array}$$

types ones. A sequence of transitions (an execution) $s_1 \xrightarrow{t_1} s_2 \cdots s_m \xrightarrow{t_m} s_{m+1}$ is said to be *k-bounded* if all channels of all intermediate configurations s_i do not contain more than k messages.

Definition 3.3 (properties). Let S be a communicating system.

1. S satisfies the *local choice* property if, for all $s \in RS(S)$ and $s \xrightarrow{\varphi_1} s_1$ and $s \xrightarrow{\varphi_2} s_2$, there exists $\varphi_1', \varphi_2', s_1', s_2'$ such that $s_1 \xrightarrow{\varphi_1'} s_1'$ and $s_2 \xrightarrow{\varphi_2'} s_2'$ with $Rcv(\varphi_1\varphi_1') = Rcv(\varphi_2\varphi_2')$ and $ASend(\varphi_1\varphi_1') = ASend(\varphi_2\varphi_2')$.
2. S is *deadlock-free* (resp. *orphan message-free*, *reception error-free*) if $s \in RS(S)$, s is not a deadlock (resp. orphan message, unspecified reception) configuration.
3. S is *strongly bounded* if the contents of buffers of all reachable configurations form a finite set.
4. S satisfies the *progress property* if for all $s \in RS(S)$, $s \longrightarrow^* s'$ implies s' is either final or $s' \longrightarrow s''$; and S satisfies the *liveness property*[2] if for all $s \in RS(S)$, there exists $s \longrightarrow^* s'$ such that s' is final.

3.3 Multiparty session automata (MSA)

We now give a translation from local types to CFSMs, specifying the sequences of actions in a local type as transitions of a CFSM. We use the following notation to keep track of local states:

$$\mathtt{X} \ ::= \ \mathbf{x} \ \mid \ \mathtt{X} \mid \mathtt{X} \qquad \mathtt{X}[\_] \ ::= \ \_ \ \mid \ \mathtt{X}[\_] \mid \mathtt{X} \ \mid \ \mathtt{X} \mid \mathtt{X}[\_]$$

We also define in Fig. 9 an equivalence relation $\equiv_{\widetilde{T}}$ that identifies two states if one of them allows the actions of the other:

Definition 3.4 (translation from local types to MSA). Let $\mathbf{T} = \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_0$ be the local type of participant $\mathtt{p}$ projected from $\mathbf{G}$. The automaton corresponding to $\mathbf{T}$ is $\mathcal{A}(\mathbf{T}) = (Q, C, q_0, \mathbb{A}, \delta)$ where:

[2] The terminology follows [6].

Fig. 9. Local State Equivalence for Local State Automata

$$\mathtt{X} \mid \mathtt{X}' \equiv_{\widetilde{T}} \mathtt{X}' \mid \mathtt{X} \qquad \mathtt{X} \mid (\mathtt{X}' \mid \mathtt{X}'') \equiv_{\widetilde{T}} (\mathtt{X} \mid \mathtt{X}') \mid \mathtt{X}''$$

$$\frac{\mathbf{x} = \mathbf{x}' \in \widetilde{T}}{\mathtt{X}[\mathbf{x}] \equiv_{\widetilde{T}} \mathtt{X}[\mathbf{x}']} \quad \frac{\mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \in \widetilde{T}}{\mathtt{X}[\mathbf{x}] \equiv_{\widetilde{T}} \mathtt{X}[\mathbf{x}' \mid \mathbf{x}'']} \quad \frac{\mathbf{x} \mid \mathbf{x}' = \mathbf{x}'' \in \widetilde{T}}{\mathtt{X}[\mathbf{x} \mid \mathbf{x}'] \equiv_{\widetilde{T}} \mathtt{X}[\mathbf{x}'']} \quad \frac{\mathbf{x} = \mathbf{x}' \,\&\, \mathbf{x}'' \in \widetilde{T}}{\mathtt{X}[\mathbf{x}] \equiv_{\widetilde{T}} \mathtt{X}[\mathbf{x}']} \quad \frac{\mathbf{x} = \mathbf{x}' \,\&\, \mathbf{x}'' \in \widetilde{T}}{\mathtt{X}[\mathbf{x}] \equiv_{\widetilde{T}} \mathtt{X}[\mathbf{x}'']}$$

$$\frac{\mathbf{x} = \mathbf{x}' \oplus \mathbf{x}'' \in \widetilde{T}}{\mathtt{X}[\mathbf{x}] \equiv_{\widetilde{T}} \mathtt{X}[\mathbf{x}']} \quad \frac{\mathbf{x} = \mathbf{x}' \oplus \mathbf{x}'' \in \widetilde{T}}{\mathtt{X}[\mathbf{x}] \equiv_{\widetilde{T}} \mathtt{X}[\mathbf{x}'']} \quad \frac{\mathbf{x} + \mathbf{x}' = \mathbf{x}'' \in \widetilde{T}}{\mathtt{X}[\mathbf{x}] \equiv_{\widetilde{T}} \mathtt{X}[\mathbf{x}'']} \quad \frac{\mathbf{x} + \mathbf{x}' = \mathbf{x}'' \in \widetilde{T}}{\mathtt{X}[\mathbf{x}'] \equiv_{\widetilde{T}} \mathtt{X}[\mathbf{x}'']}$$

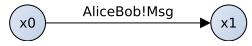
- Q is defined as the set of states $\mathtt{X}$ built from the recursion variables $\{\mathbf{x}_i\}$ of $\mathbf{T}$. Q is defined up to the equivalence relation $\equiv_{\widetilde{T}}$ (Fig. 9).
- $C = \{\mathtt{pq} \mid \mathtt{p},\mathtt{q} \in \mathbf{G}\}$; $q_0 = \mathbf{x}_0$; and $\mathbb{A}$ is the set of $\{l \in \mathbf{G}\}$
- δ is defined by: $(\mathtt{X}[\mathbf{x}], (\mathtt{pp}'!l), \mathtt{X}[\mathbf{x}']) \in \delta$ if $\mathbf{x} = !\langle \mathtt{p}', l\langle U\rangle\rangle.\mathbf{x}' \in \widetilde{T}$
  $(\mathtt{X}[\mathbf{x}], (\mathtt{p}'\mathtt{p}?l), \mathtt{X}[\mathbf{x}']) \in \delta$ if $\mathbf{x} = ?\langle \mathtt{p}', l\langle U\rangle\rangle.\mathbf{x}' \in \widetilde{T}$

We call ***Multiparty Session Automata (MSA)***, communicating systems S of the form $(\mathcal{A}(\mathbf{G} \upharpoonright \mathtt{p}))_{\mathtt{p} \in \mathbf{G}}$ when $\mathbf{G}$ is a well-formed global type.

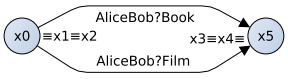
The generation of an MSA from a global type $\mathbf{G}$ is exponential in the size of $\mathbf{G}$. It is however polynomial in the absence of parallel composition. Note that neither well-formedness nor type-checking requires the explicit generation of MSAs.

MSA Examples The following shows local types (projections from Ex. 2.1) and their corresponding automata. The Trade example from Fig. 1 and Ex. 3.1 is another complete example of MSA.

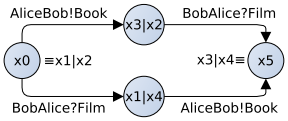
1. $\mathbf{G}_1 \upharpoonright \mathtt{Alice} = \mathsf{def}\ \mathbf{x}_0 = !\langle \mathtt{Bob}, \mathit{Msg}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_1$, $\mathbf{x}_1 = \mathsf{end}$ in $\mathbf{x}_0$



2. $\mathbf{G}_2 \upharpoonright \mathtt{Bob} = \mathsf{def}\ \mathbf{x}_0 = \mathbf{x}_1 \& \mathbf{x}_2$, $\mathbf{x}_1 = ?\langle \mathtt{Alice}, \mathit{Book}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_3$, $\mathbf{x}_2 = ?\langle \mathtt{Alice}, \mathit{Film}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_4$, $\mathbf{x}_3 + \mathbf{x}_4 = \mathbf{x}_5$, $\mathbf{x}_5 = \mathsf{end}$ in $\mathbf{x}_0$



3. $\mathbf{G}_3 \upharpoonright \mathtt{Alice} = \mathsf{def}\ \mathbf{x}_0 = \mathbf{x}_1 \mid \mathbf{x}_2$, $\mathbf{x}_1 = !\langle \mathtt{Bob}, \mathit{Book}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_3$, $\mathbf{x}_2 = ?\langle \mathtt{Bob}, \mathit{Film}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_4$, $\mathbf{x}_3 \mid \mathbf{x}_4 = \mathbf{x}_5$, $\mathbf{x}_5 = \mathsf{end}$ in $\mathbf{x}_0$



1. The MSA of the projection of $\mathbf{G}_1$ to $\mathtt{Alice}$ has two states and one transition.
2. Since $\mathtt{Bob}$ is receiving $\mathtt{Alice}$'s messages, the projection of $\mathbf{G}_2$ to $\mathtt{Bob}$ gives an external choice. The automaton has two nodes $\mathbf{x}_0$ (equivalent to $\mathbf{x}_1$ and $\mathbf{x}_2$) and $\mathbf{x}_5$ (equivalent to $\mathbf{x}_3$ and $\mathbf{x}_4$), and two transitions between these nodes.
3. $\mathbf{G}_3$ has two concurrent communications. It results in an automaton for $\mathtt{Alice}$ with four nodes, reflecting the interleavings of the concurrent interactions.

3.4 Properties of MSAs

This subsection proves that MSA satisfy the properties defined in definition 3.3. We qualify executions of the form $s \xrightarrow{\varphi_1} s_1 \xrightarrow{\varphi_2} s_2$ with $s \in RS(S)$ such that φ_1 is an alternation of sending and corresponding receive actions (i.e. the action pq!a is immediately followed by pq?a) and φ_2 is only sending actions as being *stable-outputs*. The key property is Lemma 3.1(3), whose proof is non-trivial and relies on Lemma 3.1(2) and well-formed conditions of global types (except choice awareness in definition 2.1). Then Lemma 3.1(4) (the existence of *stable executions* [7]) directly leads to unspecified reception error-freedom and orphan message freedom. For the deadlock-freedom, we require choice awareness of Lemma 3.1(1), ensured by the same condition in definition 2.1. Theorem 3.2 uses the results from [10, § 3]; in Theorem 3.3, progress is proved from Theorem 3.1, while liveness directly uses the thread correctness condition.

Lemma 3.1 (Properties of MSAs). *Suppose S is a MSA.*

1. (local choice) *S satisfies a local choice condition.*
2. (diamond property) *Suppose $s \in RS(S)$ and $s \xrightarrow{t_1} s_1$ and $s \xrightarrow{t_2} s_2$ where (1) t_1 and t_2 are both inputs; or (2) t_1 is an output and t_2 is an input, then there exists s' such that $s_1 \xrightarrow{t_1'} s'$ and $s_2 \xrightarrow{t_2'} s'$ where $\ell(t_1) = \ell(t_2')$ and $\ell(t_2) = \ell(t_1')$.*
3. (stable-outputs decomposition) *Suppose $s \in RS(S)$. Then there exists $s_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_n} s$ where each φ_i is stable-outputs.*
4. (stable) *Suppose $s_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_n} s$ with φ_i stable-outputs. Then there exists an execution $\xrightarrow{\varphi'}$ such that $s \xrightarrow{\varphi'} s_3$ and s_3 is stable, and there is a 1-buffer execution $s_0 \xrightarrow{\varphi''} s_3$.*

Theorem 3.1 (Safety Properties). *A MSA S is free from unspecified reception errors, orphan messages and deadlock.*

Theorem 3.2 (Strong Boundedness). *Consider a MSA S, generated from the local types of* $\mathbf{G}$. *If all actions that are within a cycle in* $\mathbf{G}$ *are also part of causal input-output cycle (IO-causality) [10, 15],*[3] *then S is strongly bounded.*

Theorem 3.3 (Progress and Liveness). *A MSA S satisfies the progress property. If a MSA S is generated from the local types of* $\mathbf{G}$ *and* $\mathbf{G}$ *contains* end, *then S satisfies the liveness property.*

4 General Multiparty Session Processes

This section introduces *general multiparty session processes* which are designed following the shape of the multiparty global types presented in § 2.1. Our new system handles (1) new external and internal choice operators that allow branching with different receivers and merging with different senders; and (2) forking and joining threads which are not verifiable by standard session type systems [3, 6, 15].

Fig. 10. Process and Network Syntax

$v ::= a \mid s[\mathtt{p}] \mid \mathtt{true} \mid \mathtt{false} \mid \ldots$	values		$e ::= v \mid x \mid e \wedge e \mid \ldots$		expression
$\mathbf{P} ::= \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X}$	definition		$h ::= \emptyset \mid h \cdot (\mathtt{p}, \mathtt{q}, l\langle v \rangle)$		messages
$P ::=$	process	transition	$\mathbf{X} ::=$		state
$\mid \mathbf{x}(\tilde{x}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e})$		init	$\mid \mathbf{x}(\tilde{v})$		thread
$\mid \mathbf{x}(\tilde{x}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e})$		request	$\mid \mathbf{X} \mid \mathbf{X}$		parallel
$\mid \mathbf{x}(\tilde{x}) = x!\langle \mathtt{p}, l\langle e \rangle \rangle.\mathbf{x}'(\tilde{e})$		send	$\mid (\nu a)\mathbf{X}$		restriction
$\mid \mathbf{x}(\tilde{x}) = x?\langle \mathtt{p}, l(y) \rangle.\mathbf{x}'(\tilde{e})$		receive	$\mid \mathbf{0}$		null
$\mid \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{y}) \mid \mathbf{x}''(\tilde{z})$		parallel	$\mathbf{N} ::=$		network
$\mid \mathbf{x}(\tilde{x}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}'(\tilde{e}')\ \mathtt{else}\ \mathbf{x}''(\tilde{e}'')$		conditional	$\mid \mathbf{P}$		def
$\mid \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{x}) \,\&\, \mathbf{x}''(\tilde{x})$		external choice	$\mid \mathbf{N} \mid\mid \mathbf{N}$		parallel
$\mid \mathbf{x}(\tilde{y}) \mid \mathbf{x}'(\tilde{z}) = \mathbf{x}''(\tilde{x})$		join	$\mid (\nu a)\mathbf{N}$		new name
$\mid \mathbf{x}(\tilde{x}) + \mathbf{x}'(\tilde{x}) = \mathbf{x}''(\tilde{x})$		merge	$\mid \mathbf{0}$		null
$\mid \mathbf{x}(\tilde{x}) = (\nu a)\ \mathbf{x}'(a\tilde{x})$		new name	$\mid (\nu s)\mathbf{N}$		new session
$\mid \mathbf{x}(\tilde{x}) = \mathbf{0}$		null	$\mid s : h$		queue
			$\mid a\langle s \rangle[\mathtt{p}]$		invitation

Syntax The syntax of processes is defined in Fig. 10. While it follows some standard constructs [3], the control flow and functional flavour are new in session calculi and allow a simple type checking verification technique. Note that this syntax is not meant to be directly written by programmers, but rather abstracts the control flow of any standard programming language equipped with fork and join constructs. A process always starts from a definition $\mathbf{P} = \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{x}(\tilde{v})$, where the parameters of $\mathbf{x}$ in $\widetilde{P}$ are to be instantiated by $\tilde{v}$. The form of process actions $\widetilde{P}$ follows global and local types and rely on a functional style to pass values around continuations. Variables $\tilde{x}$ in $\mathbf{x}(\tilde{x})$ occurring on the left-hand side of a process action are binding variables on the right-hand side. Variables y in request and receive are also binding (e.g. in $\mathbf{x}(x,z) = z?\langle \mathtt{p}, l(y) \rangle.\mathbf{x}'(x,y,z)$, the final z is bound by z in $\mathbf{x}(x,z)$, while y is bound by the input).

A session is initialised by a transition of the form $\mathbf{x}(\tilde{x}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e})$ where $\mathbf{G}$ is a global type. It attributes a global interaction pattern defined in $\mathbf{G}$ to the shared channel a that x gets substituted to. The variables in $\tilde{e}$ are all bound by $\tilde{x}$. After a session initialisation, participants can accept the session with $\mathbf{x}(\tilde{x}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e})$ (as long as x is substituted by the same share channel a as the initialisation), starting the interaction: the variables in $\tilde{e}$ are bound by $\tilde{x}$ and by y, which, at run-time, receives the session channel.

The sending action $x!\langle \mathtt{p}, l\langle e \rangle \rangle$ allows in session x to send to $\mathtt{p}$ a value e labelled by a constant l. The reception $x?\langle \mathtt{p}, l(y) \rangle.\mathbf{x}'(\tilde{e})$ expects from $\mathtt{p}$ a message with a label l. The message payload is then received in variable y, which binds in $\mathbf{x}'(\tilde{e})$.

$\mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{y}) \mid \mathbf{x}''(\tilde{z})$ represent forking threads (i.e. $P \mid Q$): $\tilde{y}$ and $\tilde{z}$ are subsets of $\tilde{x}$. The conditional ($\mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}'(\tilde{e}')\ \mathtt{else}\ \mathbf{x}''(\tilde{e}'')$) and the external choices ($\mathbf{x}'(\tilde{x}) \,\&\, \mathbf{x}''(\tilde{x})$) are extensions of the traditional selection and branching actions of session types. The join action collects parallel threads, while the merge action collects internal and external choices. Note that external choice, fork, join and merge only allow a restricted use of

[3] It is formally defined in [10, 15] and Appendix C.4.

bound variables for continuations. $\mathbf{x}(\tilde{x}) = (\nu a)\mathbf{x}'(a\tilde{x})$ creates a new shared name a. $\mathbf{0}$ is an inactive agent. For simplicity, we omit the action of leaving a session.

The process states $\mathbf{X}$ are defined from the state variables present in $\widetilde{P}$. The network $\mathbf{N}$ is a parallel composition of definition agents, with restrictions of the form $(\nu a)\mathbf{N}$.

Once a session is running, our operational semantics uses run-time syntax not directly accessible to the programmer. $\mathbf{X} \mid \mathbf{X}'$ and $(\nu a)\mathbf{X}$ are for example only accessible at run-time. Session instances are represented by session restriction $(\nu s)P$. The message buffer $s : h$ stores the messages in transit for the session instance s. A session invitation $a[\mathtt{p}]\langle s\rangle$ invites participant $\mathtt{p}$ to start the session s announced on channel a.

A network which only consists of shared name restrictions and parallel compositions of $\mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{x}(\vec{v})$ is called *initial*.

Operational Semantics We define the operational semantics for processes and networks in Fig. 11. We use the following labels to organise the reduction of processes.

$$\alpha,\beta \ ::= \ \tau \ \mid \ s[\mathtt{p},\mathtt{q}]!l\langle v\rangle \ \mid \ s[\mathtt{p},\mathtt{q}]?l\langle v\rangle \ \mid \ a\langle\mathbf{G}\rangle \ \mid \ a\langle\mathtt{p}\rangle[s]$$

The rules are divided into two parts. The first part corresponds to a transition relation of the form $\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$ representing that a process in a state $\mathbf{X}$ can move to state $\mathbf{X}'$ with action α. The second part defines reductions within networks (with unlabelled transitions $\mathbf{N} \to \mathbf{N}'$). $e \downarrow v$ denotes the evaluation of expression e to v.

Fig. 11. Operational Semantics (selected rules)

$$\frac{x[\tilde{v}/\tilde{x}] = a \qquad \tilde{e}[\tilde{v}/\tilde{x}] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x\langle\mathbf{G}\rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{a\langle\mathbf{G}\rangle} \mathbf{x}'(\tilde{v}')}\ [\textsc{Init}] \qquad \frac{x[\tilde{v}/\tilde{x}] = a \qquad \tilde{e}[\tilde{v}/\tilde{x}][s/y] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{a\langle s\rangle[\mathtt{p}]} \mathbf{x}'(\tilde{v}')}\ [\textsc{Acc}]$$

$$\frac{x[\tilde{v}/\tilde{x}] = s[\mathtt{q}] \qquad e[\tilde{v}/\tilde{x}] \downarrow v \qquad \tilde{e}[\tilde{v}/\tilde{x}] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x!\langle\mathtt{p}, l\langle e\rangle\rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{s[\mathtt{q},\mathtt{p}]!l\langle v\rangle} \mathbf{x}'(\tilde{v}')}\ [\textsc{Send}]$$

$$\frac{x[\tilde{v}/\tilde{x}] = s[\mathtt{q}] \qquad \tilde{e}[\tilde{v}/\tilde{x}][v/y] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = x?\langle\mathtt{p}, l(y)\rangle.\mathbf{x}'(\tilde{e}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle v'\rangle} \mathbf{x}'(\tilde{v}')}\ [\textsc{Rcv}] \qquad \frac{a \notin \tilde{v}}{\mathbf{x}(\tilde{x}) = (\nu a)\mathbf{x}'(a\tilde{x}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\tau} (\nu a)\mathbf{x}'(a\tilde{v})}\ [\textsc{New}]$$

$$\frac{e[\tilde{v}/\tilde{x}] \downarrow \mathsf{true} \qquad \tilde{e}'[\tilde{v}/\tilde{x}] \downarrow \tilde{v}'}{\mathbf{x}(\tilde{x}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}'(\tilde{e}')\ \mathtt{else}\ \mathbf{x}''(\tilde{e}'') \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\tau} \mathbf{x}'(\tilde{v}')}\ [\textsc{IfT}]$$

$$\frac{\widetilde{P}, \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{x})\ \&\ \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}'(\tilde{v}) \xrightarrow{\alpha} \mathbf{X}}{\widetilde{P}, \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{x})\ \&\ \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\alpha} \mathbf{X}}\ [\textsc{Ext}] \qquad \frac{\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'}{\mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X} \xrightarrow{\alpha} \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X}'}\ [\textsc{Def}] \qquad \frac{\mathbf{P} \xrightarrow{\tau} \mathbf{P}'}{\mathbf{P} \to \mathbf{P}'}\ [\textsc{Tau}]$$

$$\frac{\mathbf{P} \xrightarrow{s[\mathtt{p},\mathtt{q}]!l\langle v\rangle} \mathbf{P}'}{\mathbf{P} \mid\mid s : h \to \mathbf{P}' \mid\mid s : h \cdot (\mathtt{p},\mathtt{q},l\langle v\rangle)}\ [\textsc{Put}] \qquad \frac{\mathbf{P} \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle v\rangle} \mathbf{P}'}{\mathbf{P} \mid\mid s : (\mathtt{p},\mathtt{q},l\langle v\rangle) \cdot h \to \mathbf{P}' \mid\mid s : h}\ [\textsc{Get}]$$

$$\frac{\mathbf{P} \xrightarrow{a\langle\mathbf{G}\rangle} \mathbf{P}' \quad \mathtt{p}_0, \ldots, \mathtt{p}_k \in \mathbf{G} \quad s \notin \mathsf{fn}(\mathbf{P}')}{\mathbf{P} \to (\nu s)(\mathbf{P}' \mid\mid s : \varepsilon \mid\mid a\langle s\rangle[\mathtt{p}_0] \mid\mid \ldots \mid\mid a\langle s\rangle[\mathtt{p}_k])}\ [\textsc{Init}_N] \qquad \frac{\mathbf{P} \xrightarrow{a\langle s\rangle[\mathtt{p}]} \mathbf{P}'}{\mathbf{P} \mid\mid a\langle s\rangle[\mathtt{p}] \to \mathbf{P}'}\ [\textsc{Acc}_N]$$

Rule [Send] emits a message from $\mathtt{p}$ to $\mathtt{q}$, substituting variables $\tilde{x}$ by $\tilde{v}$ and evaluating e to v. Rule [Rcv] inputs a message and instantiates y to the received value v. Rule [Init] initiates a session, while rule [Acc] emits a signal which signifies the process's readiness to participate in a session. Rule [IfT] internally selects the first branch with respect to the

Fig. 12. Trade Example: Processes

$$
\begin{array}{l}
\mathbf{P}_{\mathtt{S}} = \mathsf{def} \quad \mathbf{x}(x,y) = x\langle \mathbf{G}_{\mathrm{Trade}} \rangle.\mathbf{x}'(x,y) \\
\qquad\qquad \mathbf{x}'(x,y) = x[\mathtt{S}](z).\mathbf{x}_0(y,z) \\
\qquad\qquad \mathbf{x}_0(y,z) = z\,!\,\langle \mathtt{B}, \mathit{Item}\langle y \rangle \rangle.\mathbf{x}_1(z) \\
\qquad\qquad \mathbf{x}_1(z) = z\,?\,\langle \mathtt{B}, \mathit{Final}(y) \rangle.\mathbf{x}_{10}(z,y) \\
\qquad\qquad \mathbf{x}_{10}(z,y) = \mathbf{0} \qquad \mathsf{in}\ \mathbf{x}(a, \text{“}\mathtt{HGG}\text{”})
\end{array}
$$

$$
\begin{array}{l}
\mathbf{P}_{\mathtt{C}} = \mathsf{def} \quad \mathbf{x}(x,i) = x[\mathtt{C}](z).\mathbf{x}_0(i,z) \\
\qquad \mathbf{x}_5(i,z) + \mathbf{x}_0(i,z) = \mathbf{x}_2(i,z) \\
\qquad\qquad \mathbf{x}_2(i,z) = \mathbf{x}_3(i,z)\ \&\ \mathbf{x}_6(i,z) \\
\qquad\qquad \mathbf{x}_3(i,z) = z\,?\,\langle \mathtt{B}, \mathit{Offer}(y) \rangle.\mathbf{x}_4(i,z,y) \\
\qquad\qquad \mathbf{x}_4(i,z,y) = z\,!\,\langle \mathtt{B}, \mathit{Counter}\langle i \rangle \rangle.\mathbf{x}_5(i+5,z) \\
\qquad\qquad \mathbf{x}_6(i,z) = z\,?\,\langle \mathtt{B}, \mathit{Result}(y) \rangle.\mathbf{x}_{10}(y,z) \\
\qquad\qquad \mathbf{x}_{10}(y,z) = \mathbf{0} \qquad \mathsf{in}\ \mathbf{x}(a,50)
\end{array}
$$

value of e ([IFF] is similarly defined). Rule [NEW] creates a new shared name. Rule [EXT] is the external choice, which invokes either the left or right state variable, depending on which label α is received.

Rules [DEF] and [TAU] promote processes to the network level. [INIT$_N$] is used in combination with [INIT]. It creates an empty queue $s : \varepsilon$ together with invitations for each participant. Rule [ACC$_N$] consumes an invitation to participate to the session if someone has been signalled ready (via [ACC]). Other contextual rules are standard (we omit the structure rules, $\equiv$). We write $\longrightarrow^*$ for the multi-step reduction.

We write here an implementation of the Trade example from § 1. The reader can refer to Fig. 1 and Ex. 3.1 for the global and local types.
In Fig. 12, $\mathbf{P}_{\mathtt{S}}$ and $\mathbf{P}_{\mathtt{C}}$, respectively correspond to the seller $\mathtt{S}$ and client $\mathtt{C}$. $\mathbf{P}_{\mathtt{S}}$ initiates the session by announcing $\mathbf{G}_{\mathrm{Trade}}$ on shared name a. According to rule [INIT$_N$], it creates a session name s, a message buffer and invitations for $\mathtt{S}$, $\mathtt{B}$ and $\mathtt{C}$. $\mathbf{P}_{\mathtt{S}}$ then joins the session as the seller $\mathtt{S}$, the variable z being used to contain the session name. $\mathbf{P}_{\mathtt{S}}$ proceeds with $\mathbf{x}_0(y,z)$ where y is the string “$\mathtt{HGG}$” and z the session name. The execution of $\mathbf{x}_0(y,z)$ sends a message *Item* with payload “$\mathtt{HGG}$” in the message buffer. $\mathbf{P}_{\mathtt{C}}$ starts in $\mathbf{x}(a,50)$ where a is the shared name and 50 the price it is ready to offer initially. It joins the session as the client $\mathtt{C}$, gets in variable z the session name s and continues with $\mathbf{x}_0(i,z)$. The message *Offer* is then countered as many times needed with a slowly increased proposed price.

5 Typing Multiparty Interactions

This section introduces the typing system. There is one main difference with existing multiparty typing system: to type a process $\mathbf{P}$, we need to gather for every session the typing constraints of the transitions $\widetilde{P}$ in $\mathbf{P}$, keeping track of associations such as $\mathbf{x}_1 = !\langle \mathtt{p}, l\langle U \rangle \rangle.\mathbf{x}_2$. We rely on an effective use of “matching” between local types and inferred transitions to keep the typing system for initial processes simple.

Environments We use u to denote a shared channel a and its variable x and c to denote a session channel $s[\mathtt{p}]$ or its variable. The grammar of environments are defined as:

$$\Gamma \ ::= \ \emptyset \mid \Gamma, u : U \qquad \Delta \ ::= \ \emptyset \mid \Delta, c : \mathbf{T} \qquad \Sigma \ ::= \ \emptyset \mid \Sigma, \mathbf{x} : \tilde{U}$$

Γ is the *standard environment* which associates variables to sort types and shared names to global types. Δ is the *session environment* which associates channels to session types. Σ keeps tracking state variable associations. We write $\Gamma, u : U$ only if $u \notin \mathsf{dom}(\Gamma)$. Similarly for other variables.

Fig. 13. Typing System for Initial State Processes (selected rules)

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \tilde{y}:\tilde{U}\vdash x:\langle\mathbf{G}\rangle \quad \forall i,\mathbf{T}_i=\mathbf{T}_i'\uplus\mathbf{x}=\mathbf{x}'}{\vdash \mathbf{x}(\tilde{y}\tilde{z})=x\langle\mathbf{G}\rangle.\mathbf{x}'(\tilde{e}\tilde{z})\rhd\mathbf{x}:\tilde{U}\,\widetilde{\mathbf{T}} \;[\!]\; \mathbf{x}':\tilde{U}'\,\widetilde{\mathbf{T}}'}\text{[INIT]}$$

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \tilde{y}:\tilde{U}\vdash x:\langle\mathbf{G}\rangle \quad \forall i,\mathbf{T}_i=\mathbf{T}_i'\uplus\mathbf{x}=\mathbf{x}' \quad \mathbf{T}=\mathbf{G}\upharpoonright \mathtt{p}}{\vdash \mathbf{x}(\tilde{y}\tilde{z})=x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}\tilde{z}y)\rhd\mathbf{x}:\tilde{U}\,\widetilde{\mathbf{T}} \;[\!]\; \mathbf{x}':\tilde{U}'\,\widetilde{\mathbf{T}}'\mathbf{T}}\text{[REQ]}$$

$$\frac{\tilde{y}:\tilde{U}\vdash e:U \quad \tilde{y}:\tilde{U}\vdash\tilde{e}:\tilde{U}' \quad \mathbf{T}_i=\mathbf{T}_i'\uplus\mathbf{x}=!\,\langle\mathtt{p},l\langle U\rangle\rangle.\mathbf{x}' \quad \forall j\neq i,\mathbf{T}_j=\mathbf{T}_j'\uplus\mathbf{x}=\mathbf{x}'}{\vdash \mathbf{x}(\tilde{y}\tilde{z})=z_i\,!\,\langle\mathtt{p},l\langle e\rangle\rangle.\mathbf{x}'(\tilde{e}\tilde{z})\rhd\mathbf{x}:\tilde{U}\,\widetilde{\mathbf{T}} \;[\!]\; \mathbf{x}':\tilde{U}'\,\widetilde{\mathbf{T}}'}\text{[SEND]}$$

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{y_1}:\tilde{U}_1 \quad \tilde{y}:\tilde{U}\vdash\tilde{y_2}:\tilde{U}_2 \quad \forall i,\mathbf{T}_i=(\mathbf{T}_{1i}\cup\mathbf{T}_{2i})\uplus\mathbf{x}=\mathbf{x}_1\mid\mathbf{x}_2}{\vdash \mathbf{x}(\tilde{y}\tilde{z})=\mathbf{x}_1(\tilde{y}_1\tilde{z})\mid\mathbf{x}_2(\tilde{y}_2\tilde{z})\rhd\mathbf{x}:\tilde{U}\,\widetilde{\mathbf{T}} \;[\!]\; \mathbf{x}_1:\tilde{U}_1\,\widetilde{\mathbf{T}}_1,\mathbf{x}_2:\tilde{U}_2\,\widetilde{\mathbf{T}}_2}\text{[PAR]}$$

$$\frac{\tilde{y}:\tilde{U}\vdash\tilde{y_1}:\tilde{U}_1 \quad \tilde{y}:\tilde{U}\vdash\tilde{y_2}:\tilde{U}_2 \quad \forall i,\mathbf{T}_i=\mathbf{T}_i'\uplus\mathbf{x}_1\mid\mathbf{x}_2=\mathbf{x}}{\vdash \mathbf{x}_1(\tilde{y}_1\tilde{z})\mid\mathbf{x}_2(\tilde{y}_2\tilde{z})=\mathbf{x}(\tilde{y}\tilde{z})\rhd \;\mathbf{x}_1:\tilde{U}_1\,\widetilde{\mathbf{T}},\mathbf{x}_2:\tilde{U}_2\,\widetilde{\mathbf{T}} \;[\!]\; \mathbf{x}:\tilde{U}\,\widetilde{\mathbf{T}}'}\text{[JOIN]}$$

$$\frac{\vdash P_i\rhd\Sigma_i \;[\!]\; \Sigma_i' \quad comp(\{\Sigma_i \;[\!]\; \Sigma_i'\}_i) \quad \mathbf{x}_0:\tilde{U}=(\cup\Sigma_i)\setminus(\cup\Sigma_i') \quad \Gamma\vdash\tilde{v}:\tilde{U}}{\Gamma\vdash \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{x}_0(\tilde{v})}\text{[DEF]}$$

Judgements The different judgements that are used are:

$\Gamma\vdash e:U$	Expression e has type U under Γ
$\Gamma\vdash P\rhd\Sigma \;[\!]\; \Sigma'$	Left/right variables in P have types Σ/Σ' under Γ
$\Gamma\vdash\mathbf{P}$	Process $\mathbf{P}$ is typed under Γ $\quad$ $\Gamma\vdash\mathbf{N}$ Network $\mathbf{N}$ is typed under Γ

Typing Rules We give the selected typing system for processes in Fig. 13. Apart from inferring local types through constraint gathering, the essence of the typing system is the same as the original system of [3, 15]. In the rules, $\tilde{y}$ and $\tilde{z}$ correspond to sorts and session types, respectively.

Rule [INIT] types the initialisation. $\tilde{y}$ should cover x and variables in $\tilde{e}$ appearing in the right hand side. The type system records that every z_i should have type $\mathbf{T}\uplus\mathbf{x}=\mathbf{x}'$, which means that we record $\mathbf{x}=\mathbf{x}'$ at the head of $\mathbf{T}$ (formally defined as: $\mathsf{def}\ \mathbf{x}=\mathbf{x}',\widetilde{T}\ \mathsf{in}\ \mathbf{x}$ if $\mathbf{T}=\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}'$). Rule [REQ] is similar except we record the introduced projected session type $\mathbf{T}=\mathbf{G}\upharpoonright\mathtt{p}$. Rule [SEND] records the send type for z_i ($T_i=\mathbf{x}=!\,\langle\mathtt{p},l\langle U\rangle\rangle.\mathbf{x}'$) and $\mathbf{x}=\mathbf{x}'$ for all other sessions. The rule for the input is symmetric.

Rule [PAR] introduces the parallel composition. The operation $(\mathbf{T}_1\cup\mathbf{T}_2)\uplus\mathbf{x}=\mathbf{x}_1\mid\mathbf{x}_2$ is defined as $\mathsf{def}\ \mathbf{x}=\mathbf{x}_1\mid\mathbf{x}_2,\widetilde{T_1}\cup\widetilde{T_2}\ \mathsf{in}\ \mathbf{x}$ where $\mathbf{T}_i=\mathsf{def}\ \widetilde{T_i}\ \mathsf{in}\ \mathbf{x}_i$ (we record $\mathbf{x}=\mathbf{x}_1\mid\mathbf{x}_2$ at the head of a union of $\mathbf{T}_1$ and $\mathbf{T}_2$). Rule [JOIN] is a symmetric rule.

In [DEF], we write $comp(\{\Sigma_i;\Sigma_i'\}_i)$ (environments are *complete*) if (1) there exists unique $\mathbf{x}_0\in(\cup_i\Sigma_i)\setminus(\cup_i\Sigma_i')$; (2) for all $\mathbf{x}\neq\mathbf{x}_0$, $\mathbf{x}$ appears exactly once in Σ_i and Σ_j' for a unique i,j. (3) if $\Sigma_i(\mathbf{x})=\tilde{U};\tilde{\mathbf{T}}$ and $\Sigma_j'(\mathbf{x})=\tilde{U}';\tilde{\mathbf{T}}'$, then $\tilde{U}=\tilde{U}'$ and $\tilde{\mathbf{T}}=\tilde{\mathbf{T}}'$. The conditions (1,2) corresponds to (Unambiguity) condition in the well-formed global types defined in § 2.2, while (3) simply checks the matching of the argument types of each state variable appears in right Σ_i and left Σ_j'. Other rules (for delegation, choice, condition, restriction and networks) are similar or straightforward, following the structure of terms.

Since checking well-formedness and $comp(\{\Sigma_i;\Sigma_i'\})$ are decidable in polynomial time, following the standard method [15, § 4], we have:

Fig. 14. Type Transition System, Structure Rules and Environment Communication Rule

$$\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x} \equiv \mathsf{def}\ \widetilde{T}'\ \mathsf{in}\ \mathbf{x} \quad (\widetilde{T} = \widetilde{T}') \qquad \lfloor \textsc{Eq} \rfloor$$

$$\mathsf{def}\ \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_i \equiv \mathsf{def}\ \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x} \quad (i = 1 \text{ or } i = 2) \qquad \lfloor \textsc{Merge} \rfloor$$

$$\mathsf{def}\ \mathbf{x}_1 \mid \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1 \mid \mathbf{x}_2 \equiv \mathsf{def}\ \mathbf{x}_1 \mid \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x} \qquad \lfloor \textsc{Join} \rfloor$$

$$\mathsf{def}\ \mathbf{x} = !\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x} \xrightarrow{!\langle \mathtt{p}, l\langle U\rangle\rangle} \mathsf{def}\ \mathbf{x} = !\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x}' \qquad \lfloor \textsc{Send}_\ell \rfloor$$

$$\mathsf{def}\ \mathbf{x} = ?\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x} \xrightarrow{?\langle \mathtt{p}, l\langle U\rangle\rangle} \mathsf{def}\ \mathbf{x} = ?\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x}' \qquad \lfloor \textsc{Recv}_\ell \rfloor$$

$$\mathsf{def}\ \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2, \widetilde{T}\ \mathsf{in}\ \mathbf{x} \xrightarrow{\tau} \mathsf{def}\ \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_i \quad (i = 1 \text{ or } i = 2) \qquad \lfloor \textsc{Cond} \rfloor$$

$$\frac{\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1 \xrightarrow{\ell} \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1'}{\mathsf{def}\ \mathbf{x} = \mathbf{x}_1\ \&\ \mathbf{x}_2, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1 \xrightarrow{\ell} \mathsf{def}\ \mathbf{x} = \mathbf{x}_1\ \&\ \mathbf{x}_2, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1'} \lfloor \textsc{Choice} \rfloor$$

$$\frac{\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1 \xrightarrow{\ell} \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1'}{\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1 \mid \mathbf{X}_2 \xrightarrow{\ell} \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1' \mid \mathbf{X}_2} \lfloor \textsc{Par} \rfloor \qquad \frac{\mathbf{T}_1 \xrightarrow{!\langle \mathtt{q}, l\langle U\rangle\rangle} \mathbf{T}_1' \qquad \mathbf{T}_2 \xrightarrow{?\langle \mathtt{p}, l\langle U\rangle\rangle} \mathbf{T}_2'}{(s[\mathtt{p}] : \mathbf{T}_1, s[\mathtt{q}] : \mathbf{T}_2, \Delta) \to (s[\mathtt{p}] : \mathbf{T}_1', s[\mathtt{q}] : \mathbf{T}_2', \Delta)} [\textsc{Com}]$$

Proposition 5.1 (Decidability). *Assuming the new and bound names and variables in* $\mathbf{N}$ *are annotated by types, type checking of* $\Gamma \vdash \mathbf{N}$ *terminates in polynomial time.*

Typing for Run-time Processes We have four judgement to type run-time processes:

$$\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta \quad \Gamma \vdash \mathbf{P} \triangleright \Delta \quad \Gamma \vdash_{\mathscr{S}} \mathbf{N} \triangleright \Delta \quad \Gamma \vdash_{\mathscr{S}} s : h \triangleright \Delta$$

where $\mathscr{S}$ denotes the set of session names of queues. We also extend the type $\mathbf{T}$ to $\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathtt{X}$ to type states $\mathbf{X}$ and message types to type queues [3]. The rest can be understood without these typing systems, hence we leave them to Appendix E.

6 Properties of Typed Multiparty Session Processes

This section shows that typed processes enjoy the same properties as MSAs defined in definition 3.3. The correspondence with CFSMs makes the statements of the properties of processes formally rigorous and eases the proofs. For full proofs, see Appendix F.

6.1 Safety and Boundedness

Let ℓ range over transition labels for types: $\ell ::= \tau \mid !\langle \mathtt{p}, l\langle U\rangle\rangle \mid ?\langle \mathtt{p}, l\langle U\rangle\rangle$. Figure 14 defines a labelled transition relation between types $\mathbf{T} \xrightarrow{\ell} \mathbf{T}'$, defined modulo structure rules (for join and merge) and type equality. The sending and receiving actions occur when the state variable $\mathbf{x}$ points to sending and receiving types (Rules $\lfloor \textsc{Send}_l \rfloor$ and $\lfloor \textsc{Recv}_l \rfloor$). Others are contextual rules. We also use the labelled transition relation between environments, denoted by $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma', \Delta')$ where the main rule is $\lfloor \textsc{Com} \rfloor$ in Fig. 14 which represents the reduction between a message queue and a process at the network level. Other omitted rules are straightforward.

Lemma 6.1 (Subject Congruence). *Suppose* $\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$ *and* $\widetilde{P} \vdash \mathbf{X} \equiv \mathbf{X}'$. *Then* $\Gamma, \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta$. *Similarly for* $\mathbf{P}$ *and* $\mathbf{N}$.

The following theorem, which is often called *type soundness*, states that if a process (resp. network) emits a label (resp. performs a reduction), then the environment can do the corresponding action, and the resulting process and the environment match.

Theorem 6.1 (Subject Transition and Reduction).

1. $\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$ *and* $\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$ *imply* $\Gamma', \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta'$ *with* $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma', \Delta')$.
2. $\Gamma \vdash \mathbf{P} \triangleright \Delta$ *and* $\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'$ *imply* $\Gamma' \vdash \mathbf{P}' \triangleright \Delta'$ *with* $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma', \Delta')$.
3. $\Gamma \vdash_{\mathscr{S}} \mathbf{N} \triangleright \Delta$ *and* $\mathbf{N} \longrightarrow \mathbf{N}'$ *imply* $\Gamma \vdash_{\mathscr{S}} \mathbf{N}' \triangleright \Delta'$ *with* $\Delta \longrightarrow^* \Delta'$.

We also use the following one-to-one correspondence between local state automata and local types. We write $\xrightarrow{\tilde{\ell}}$ for $\xrightarrow{\ell_1} \cdots \xrightarrow{\ell_n}$. We use the notation $\stackrel{\ell}{\Longrightarrow}$ for $(\xrightarrow{\tau})^* \xrightarrow{\ell} (\xrightarrow{\tau})^*$ and similarly for $\stackrel{\tilde{\ell}}{\Longrightarrow}$. The proof is straightforward by the definition in § 3.3.

Theorem 6.2 (CFSMs and Local Types). $(\mathbf{G} \upharpoonright \mathtt{p}) \stackrel{\tilde{\ell}}{\Longrightarrow}$ *iff* $\mathcal{A}(\mathbf{G} \upharpoonright \mathtt{p}) \xrightarrow{\tilde{\ell}}$.

We say P has ***a type error*** if expressions in P contain either a type error for a value or constant in the standard sense (e.g. $(\mathsf{true} + 7)$) or ***a reception error*** (e.g. the sender sends a value with label l_0 while the receiver does not expect label l_0).

Theorem 6.3 (Type Safety). *Suppose* $\Gamma \vdash \mathbf{N}$. *For any* $\mathbf{N}'$ *such that* $\mathbf{N} \longrightarrow^* \mathbf{N}'$, $\mathbf{N}'$ *has no type error.*

Proof. Suppose $\Gamma \vdash \mathbf{N}$ has a type error. Then there are reductions such that $\mathbf{N} \longrightarrow^* (\nu s)(s : (\mathtt{p}, \mathtt{q}, l\langle v \rangle) \cdot h \mid\mid \mathbf{P}_1 \mid\mid \cdots \mid\mid \mathbf{P}_n) \mid\mid \mathbf{N}'$ and $\mathbf{P}_j \xrightarrow{s[\mathtt{p},\mathtt{q}]?\langle v \rangle} \mathbf{P}_j$ where v does not meet the specified type. Suppose $\Gamma \vdash \mathbf{P}_j \triangleright \Delta, s[\mathtt{p}] : \mathbf{T}_j$. By Theorem 6.2, $\mathbf{T}_j \xrightarrow{?\langle \mathtt{q}, l\langle U \rangle \rangle} \mathbf{T}'_j$. Since $\mathbf{P}_j$ has a type error, $\Gamma \not\vdash v : U$, which contradicts Theorem 6.1. □

Using Theorem 3.2, boundedness is derived as Theorem 6.4.

Theorem 6.4 (Boundedness). *Suppose for all occurrences of* $\mathbf{G}$ *in* Γ, $\mathcal{A}(\{\mathbf{G} \upharpoonright \mathtt{p}_i\}_{1 \le i \le n})$ *with* $\mathtt{p}_1, \ldots, \mathtt{p}_n \in \mathbf{G}$ *is strongly bounded. Then for all* $\mathbf{N}'$ *such that* $\Gamma \vdash \mathbf{N}$ *and* $\mathbf{N} \longrightarrow^* \mathbf{N}'$, *the reachable contents of a given channel buffer is finite.*

This result can be extended to other variants such as existential boundedness or K-boundedness [13] by applying the global buffer analysis on $\langle \mathbf{G} \rangle$ from [10].

6.2 Advanced Properties in a Single Multiparty Session

We now focus on advanced properties guaranteed when only a single multiparty session executes. We say $\mathbf{N}$ is *simple* [15, 26] if $\mathbf{N}_0 \longrightarrow^* \mathbf{N}$ such that $\mathbf{N}_0 \equiv \mathbf{P}_1 \mid\mid \cdots \mid\mid \mathbf{P}_n$ and $\Gamma \vdash \mathbf{N}_0$ where each $\mathbf{P}_i$ is either an initiator $\mathsf{def}\ \mathbf{x}_0(x) = x\langle \mathbf{G} \rangle.\mathbf{x}_1, \mathbf{x}_1 = \mathbf{0}\ \mathsf{in}\ \mathbf{x}_0(a)$ or an acceptor $\mathsf{def}\ \mathbf{x}_0(x) = x[\mathtt{p}](y).\mathbf{x}_1, \widetilde{P}\ \mathsf{in}\ \mathbf{x}_0(a)$ where $\widetilde{P}$ does not contain any initiator, acceptor, name creator, delegation nor catch (sending and receiving session channels as arguments). This means that, once the session is started, all processes continue within that session without any interference by other sessions. In a simple network, we can guarantee the following completeness result (the reverse direction of Theorem 6.1).

Theorem 6.5 (Completeness). *Below we assume* $\mathbf{X}$, $\mathbf{P}$ *and* $\mathbf{N}$ *are sub-terms of derivations from a simple network. Then:* $\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$ *and* $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma', \Delta')$ *imply* $\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$ *with* $\Gamma', \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta'$. *Similarly* $\mathbf{P}$ *and* $\mathbf{N}$ *satisfy the reversed direction of Theorem 6.1.*

We say $\mathbf{N}$ is a ***deadlock*** if all processes are blocked, waiting for messages. Formally $\mathbf{N}$ is a *deadlock* if there exists $\mathbf{N}'$ such that $\mathbf{N} \longrightarrow^* \mathbf{N}' = (\nu s)(s : \emptyset \mid\mid \mathbf{P}'_1 \mid\mid \cdots \mid\mid \mathbf{P}'_n) \mid\mid \mathbf{N}''$ and for all $1 \leq j \leq n$, if $\mathbf{P}'_j \xrightarrow{\alpha_j} \mathbf{P}''_j$ then $\alpha_j = s[\mathtt{p}, \mathtt{q}]?l\langle v \rangle$ (i.e., $\mathbf{P}'_j$ is an input process).

Theorem 6.6 (Deadlock Freedom). *Suppose* $\Gamma \vdash \mathbf{N}$ *is simple. Then there is no reduction such that* $\mathbf{N} \longrightarrow^* \mathbf{N}'$ *and* $\mathbf{N}'$ *is a deadlock.*

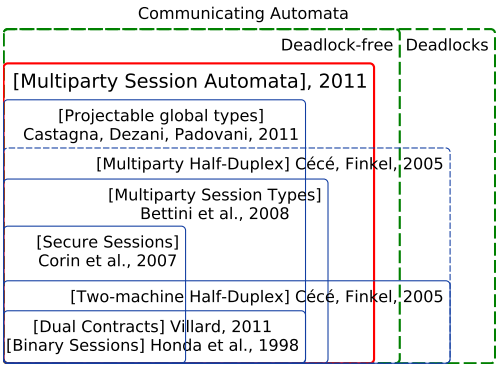
Proof. Suppose there is a deadlock network $\mathbf{N}'$ such that $\mathbf{N} \longrightarrow^* \mathbf{N}' \equiv (\nu s)(s : \emptyset \mid\mid \mathbf{P}_1 \mid\mid \cdots \mid\mid \mathbf{P}_n) \mid\mid \mathbf{N}''$ and all $\mathbf{P}_j$ is an input. By Completeness (Theorem 6.5), $\mathbf{N}' \not\xrightarrow{\alpha}$ implies $(s[\mathtt{p}_1] : \mathbf{T}_1, \cdots, s[\mathtt{p}_n] : \mathbf{T}_n) \not\xrightarrow{\alpha}$ where $\Gamma \vdash \mathbf{P}_j \triangleright \Delta, s[\mathtt{p}_j] : \mathbf{T}_j$. Since its $(\{\mathcal{A}(\mathbf{G} \upharpoonright \mathtt{p}_i)\}_i)$ is deadlock-free, for all i, $\mathbf{T}_i = \mathsf{end}$. This contradicts that $\mathbf{N}'$ is a deadlock. □

Theorem 6.7. (1) (**Progress**) *Suppose* $\Gamma \vdash \mathbf{N}$ *is simple. Then for all* $\mathbf{N} \longrightarrow^* \mathbf{N}'$, *either* $\mathbf{N}' \equiv \mathbf{0}$ *or* $\mathbf{N}' \longrightarrow \mathbf{N}''$. (2) (**Liveness**) *Suppose* $a : \langle \mathbf{G} \rangle \vdash \mathbf{N}$ *and* $\mathcal{A}(\{\mathbf{G} \upharpoonright \mathtt{p}_i\}_{1 \leq i \leq n})$ *satisfies liveness with* $\mathtt{p}_1, \ldots, \mathtt{p}_n \in \mathbf{G}$. *Assume* $\mathbf{N} \longrightarrow^* (\nu s)(s : h \mid\mid \mathbf{P}_1 \mid\mid \mathbf{P}_2 \mid\mid \cdots \mid\mid \mathbf{P}_n)$ *such that* $a : \langle \mathbf{G} \rangle \vdash \mathbf{P}_j \triangleright s[\mathtt{p}_j] : \mathbf{T}_j$. *Then there exits a reduction such that* $\mathbf{N} \longrightarrow^* \mathbf{0}$.

Proof. (1) By Theorem 6.5 with $(\nu s)(s : \emptyset) \equiv \mathbf{0}$. (2) By Theorems 6.2 and 6.5 with (1).

Thanks to the strong correspondence that typing enforces between processes behaviours and automata, we have proved that all the good properties enjoyed by MSA generated by a global type $\mathbf{G}$ also hold in the processes typed by the same $\mathbf{G}$.

7 Related Work



The relationship with other session types and CFSMs is summarised in the diagram. The outside box represents communicating automata, with the undecidable separation between deadlock-free and deadlocking machines. Within it, we represent the known inclusions between session and CFSMs systems. First, *binary* (two party) session types [14] correspond to the set of compatible half-duplex deterministic two-machine systems without mixed states [13, 25] (compatible means that each send is matched by a receive, and vice-versa). This is not the case for the MSA generated from secure session specifications [9], which satisfy strong sequentiality properties and are multiparty. They can however be shown to be *restricted half-duplex* in [7, § 4.1.2] (i.e. at most one queue is non-empty). The original multiparty session types [3, 15],

which correspond to our system when parallel composition is disallowed, are a subset of the *natural multiparty extension of half-duplex system* [7, § 4.1.2] where each pair of machines is linked by two buffered channels, one in each direction, such that at most one is non-empty. Our MSA can have mixed states and are not half-duplex, as shown in $\mathbf{G}_3$ (Ex. 2.1 (3), both `Alice` and `Bob` can fill both buffers concurrently). From this picture are omitted Gouda et al.'s pioneering work [13] and Villard's extension [18] of [25] to unreliable systems, which proves that safety properties and boundedness are still decidable. These works [13, 18, 25] only treat the two-machine case.

Finally, we mention two related works by Castagna et al. [6] and Bultan et al. [1, 2]. The first two papers [1, 6] focus on proving the semantical correspondence between global and local descriptions. In Castagna et al. [6], global choreographies are described by a language of types with general fork ($\wedge$), choice ($\vee$) and repetition $(G)^*$ (which represents a finite loop of zero or more interactions of G). Note that these global types of [6] use series-parallel syntax trees and are thus limited by the lack of support for general joins and merges. This prevents many examples, such as the Alternating Bit Protocol $\mathbf{G}_{AB}$ in Ex. 2.1 (7), the Trade example from § 1 and $\mathbf{G}_6$ in Ex. 2.1 (6), from being algorithmically projectable (i.e. implementable). In [1], on the other hand, global specifications are given by a finite state machine with no special support for parallel composition. In both cases, their systems do not treat the extended causality between sends and receives (the OO-causality and II-causality at different channels [15]). They also do not give a practical (language-based) framework, from types to processes to tackle real programs. In terms of results, [6] proposes well-formedness conditions under which local types correspond to global types, while [1] describes a sound and complete decision algorithm for realising (i.e. projecting) a choreography specification. Our work avoid this theoretical completeness question by using sufficient well-formedness conditions and by directly giving a global type semantics in terms of local automata. Recently, [2] extends [1] to tackle the synchronisability problem (equivalent to our Lemma 3.1 (3)). They however do not go as far as deadlock-freedom, progress and liveness.

When comparing these works with ours, the main differences are: (1) unlike [25] and ours, [1, 6] only investigate the relationship between global and local specifications, not from types (contracts) to programs or processes to ensure safety properties; (2) while the semantical tools are close (formal languages, finite state machines), there are subtle differences concerning buffer-boundedness [1, 2], finite recursion [6] and causality [1, 2, 6]; (3) Bultan et al. [1, 2] do not propose any global description language, while Castagna et al.'s language [6] is not rich enough compared to ours; and (4) the algorithmic projectability in [6] is more limited than ours, and [1, 2] only propose exponential decision results, limiting their applicability.

Message sequence graphs (MSGs) In terms of expressiveness, a very comparable system is the extension of Message sequence charts (MSCs) to *Message Sequence Graphs* (MSGs). MSGs are finite transition systems where each state embeds a single MSC. Many variants of MSGs are investigated in the literature [12] in order to provide efficient conditions for verification and implementability, i.e. projectability to CFSMs. Some of these conditions in MSGs are similar to ours: for example, our local choice condition corresponds to the local choice condition with additional data of [12, Def. 2]. A detailed comparison between MSGs and global types is given in [6, § 7.1].

In general MSGs are however incomparable with our framework because MSGs' transition system is global and non-deterministic. We aim our global type language to be more compact, precise and suitable for programming. For example, extending the Alternating Bit Protocol $\mathbf{G}_{AB}$ to three parties can be easily done in our system ($\mathbf{G}_9$ in Fig. 15 in Appendix B.1), while it can only be written in a complex extension of MSGs, called Compositional MSGs (CMSGs). The main benefit of our type-based approach is that there is no gap between specifications and programs: we can instantly check the properties of programs by static type-checking. More investigation on global types and MSGs properties would however bring mutual benefits by identifying the expressiveness differences.

8 Conclusion and Future Work

We have introduced a new framework of multiparty session types which is tightly linked to CFSMs, and showed that a new class of CFSMs, that we called multiparty session automata (MSA), generated from global types, automatically satisfy safety and liveness properties, extending the results in [13] to multiple machines. We use MSA to define and prove precise safety and liveness properties for well-typed mobile processes. The syntax of our session types and processes brings expressiveness to new levels (general fork, choice, merging and joining) that have not been reached by existing systems [3, 6, 15], while keeping a polynomial tool chain. Our general choice is already included into Scribble 1.0 [22], an industrial language to describe application-level protocols among communicating systems based on the multiparty session type theory.

Future work include finding a characterisation of MSA that is independent of session types, investigating model checking for MSA to justify typed bisimulations [17], relating MSA with models of true concurrency, including Mazurkiewicz traces, extending MSA to parameterisation [26], multiroles [11] and multiparty contracts [18, 25].

Acknowledgments We are grateful to the anonymous reviewers, Kohei Honda, Raymond Hu, Étienne Lozes, Rumyana Neykova and Jules Villard for their helpful comments. This work was supported by EPSRC EP/F003757/01 and G015635/01.

References

1. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. In: POPL'12. ACM (2012), to appear
2. Basu, S., Bultan, T., Ouederni, M.: Synchronizability for verification of asynchonously communicating systems. In: VMCAI'12. LNCS, Springer (2012)
3. Bettini, L., et al.: Global progress in dynamically interleaved multiparty sessions. In: CONCUR. LNCS, vol. 5201, pp. 418–433 (2008)
4. Business Process Model and Notation, `http://www.bpmn.org`
5. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM 30, 323–342 (April 1983)
6. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multi-party sessions. In: FMOODS/FORTE. LNCS, vol. 6722, pp. 1–28 (2011)
7. Cécé, G., Finkel, A.: Verification of programs with half-duplex communication. Inf. Comput. 202(2), 166–190 (2005)

8. Cheng, A., Esparza, J., Palsberg, J.: Complexity results for 1-safe nets. Theoretical Computer Science 147(1-2), 117–136 (1995)
9. Corin, R., Deniélou, P.M., Fournet, C., Bhargavan, K., Leifer, J.: Secure implementations for typed session abstractions. In: CSF. pp. 170–186 (2007)
10. Deniélou, P.M., Yoshida, N.: Buffered communication analysis in distributed multiparty sessions. In: CONCUR'10. LNCS, vol. 6269, pp. 343–357. Springer (2010)
11. Deniélou, P.M., Yoshida, N.: Dynamic multirole session types. In: POPL. pp. 435–446. ACM (2011), full version, Prototype at `http://www.doc.ic.ac.uk/~pmalo/dynamic`
12. Genest, B., Muscholl, A., Peled, D.: Message sequence charts. In: Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 537–558 (2004)
13. Gouda, M., Manning, E., Yu, Y.: On the progress of communication between two finite state machines. Information and Control. 63, 200–216 (1984)
14. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: ESOP'98. LNCS, vol. 1381, pp. 22–138. Springer (1998)
15. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL'08. pp. 273–284. ACM (2008)
16. Jones, N., Landweber, L., Edmund Lien, Y.: Complexity of some problems in petri nets. Theoretical Computer Science 4(3), 277–299 (1977)
17. Kouzapas, D., Yoshida, N., Honda, K.: On asynchronous session semantics. In: FMOODS/-FORTE. LNCS, vol. 6722, pp. 228–243 (2011)
18. Lozes, E., Villard, J.: Reliable contracts for unreliable half-duplex communications. In: WS-FM. Springer (2011), to appear
19. Ng, N., Yoshida, N., Pernet, O., Hu, R., Kryftis, Y.: Safe Parallel Programming with Session Java. In: COORDINATION. LNCS, vol. 6721, pp. 110–126. Springer (2011)
20. Ocean Observatories Initiative (OOI), `http://www.oceanobservatories.org/`
21. Savara JBoss Project, `http://www.jboss.org/savara`
22. Scribble JBoss Project, `http://www.jboss.org/scribble`
23. Sivaramakrishnan, K.C., Nagaraj, K., Ziarek, L., Eugster, P.: Efficient session type guided distributed interaction. In: COORDINATION. LNCS, vol. 6116, pp. 152–167 (2010)
24. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bharagavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: ICFP. pp. 266–278. ACM (2011)
25. Villard, J.: Heaps and Hops. Ph.D. thesis, ENS Cachan (2011)
26. Yoshida, N., Deniélou, P.M., Bejleri, A., Hu, R.: Parameterised multiparty session types. In: FoSSaCs. LNCS, vol. 6014, pp. 128–145 (2010)

Table of Contents

Appendix

A Implementation

We have implemented the session type framework described in this paper. We use generalised multiparty session types as an internal representation (with a friendlier user syntax with fork, join, choice and merge) and use the algorithms described here to check for well-formedness. For the typing system part, we use a technique developed in [9]. We generate for each endpoint a typed API and rely on the Ocaml typing system to verify user's session conformance.

Our general choice is already included into Scribble 1.0 [22], a language to describe application-level protocols among communicating systems based on the multiparty session type theory. In another application, we use MSAs for run-time safety enforcement for large-scale, cross-language distributed applications developed in [20]. We generate MSAs against protocol use-case [20] and use them for the dynamic monitoring of incoming and outgoing messages of each end-point. Since MSAs specify all traces, monitors can locally validate every message efficiently with a small memory footprint.

B Appendix for Section 2

In this appendix section, we start by some additional global type examples, numbered 8 to 10, followed by details about the three well-formedness conditions.

B.1 Additional Global Type Examples

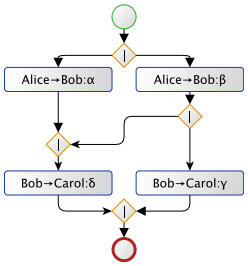
In figure 15, we give several additional global type examples that illustrate some particular features that were not developed in the main sections.

8. This global type illustrates two things. First, it shows a simple example of the use of forking followed by immediate joining. This behaviour is always local, thanks to well-formedness conditions, and, therefore, at process level can be simply and efficiently implemented.
   Second, the traces represented by this global type are not easily representable by syntax representations based on trees. The possible traces of $\mathbf{G}_8$ are $\alpha\gamma\beta\delta$, $\beta\alpha\gamma\delta$, $\alpha\beta\gamma\delta$, $\beta\alpha\delta\gamma$ and $\alpha\beta\delta\gamma$. It's easy to write with a standard syntax a representation for the relation between messages α, β and δ: $(\alpha \mid \beta);\delta$; but γ should always follow α without being linked with β or δ. Only a graph-like syntax can represent that set of traces without introducing a choice-based enumeration which breaks determinism and implementability.
9. This global type represent the extension of the Alternating-Bit protocol to three messages. As we can see the global type only linearly increases in size, while the CFSM it represents double its number of states.
10. This global type follows the shape of the Alternating bit protocol, but replacing all forks by choices and joins by merges. Such a transformation preserves well-formedness global type. The inverse is not true.

Fig. 15. Examples of Global Types (2)

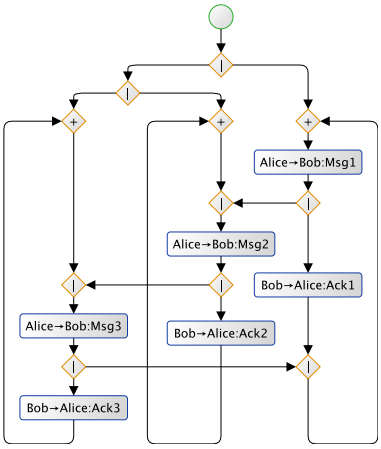
8.

$$
\begin{aligned}
\mathbf{G}_8 = \mathsf{def}\ \mathbf{x}_0 &= \mathbf{x}_1 \mid \mathbf{x}_2 \\
\mathbf{x}_1 &= \mathtt{A} \to \mathtt{B} : \alpha\langle U\rangle;\mathbf{x}_3 \\
\mathbf{x}_2 &= \mathtt{A} \to \mathtt{B} : \beta\langle U\rangle;\mathbf{x}_4 \\
\mathbf{x}_3 &= \mathbf{x}_5 \mid \mathbf{x}_6 \\
\mathbf{x}_4 \mid \mathbf{x}_5 &= \mathbf{x}_7 \\
\mathbf{x}_6 &= \mathtt{A} \to \mathtt{B} : \gamma\langle U\rangle;\mathbf{x}_8 \\
\mathbf{x}_7 &= \mathtt{A} \to \mathtt{B} : \delta\langle U\rangle;\mathbf{x}_9 \\
\mathbf{x}_8 \mid \mathbf{x}_9 &= \mathbf{x}_{10} \\
\mathbf{x}_{10} &= \mathsf{end\ in}\ \mathbf{x}_0
\end{aligned}
$$



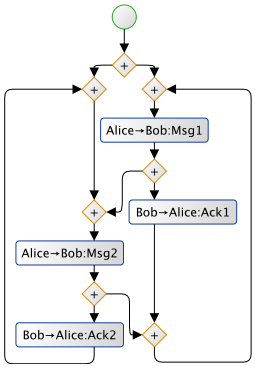
9.

$$
\begin{aligned}
\mathbf{G}_9 = \mathsf{def}\ \mathbf{x}_0 &= \mathbf{x}_1 \mid \mathbf{x}_2 \\
\mathbf{x}_1 &= \mathbf{x}_3 \mid \mathbf{x}_4 \\
\mathbf{x}_2 + \mathbf{x}_5 &= \mathbf{x}_6 \\
\mathbf{x}_3 + \mathbf{x}_7 &= \mathbf{x}_8 \\
\mathbf{x}_4 + \mathbf{x}_9 &= \mathbf{x}_{10} \\
\mathbf{x}_6 &= \mathtt{Alice} \to \mathtt{Bob} : Msg_1\langle \mathsf{string}\rangle;\mathbf{x}_{11} \\
\mathbf{x}_{11} &= \mathbf{x}_{12} \mid \mathbf{x}_{13} \\
\mathbf{x}_{13} &= \mathtt{Bob} \to \mathtt{Alice} : Ack_1\langle \mathsf{unit}\rangle;\mathbf{x}_{14} \\
\mathbf{x}_{12} \mid \mathbf{x}_8 &= \mathbf{x}_{15} \\
\mathbf{x}_{15} &= \mathtt{Alice} \to \mathtt{Bob} : Msg_2\langle \mathsf{string}\rangle;\mathbf{x}_{16} \\
\mathbf{x}_{16} &= \mathbf{x}_{17} \mid \mathbf{x}_{18} \\
\mathbf{x}_{17} &= \mathtt{Bob} \to \mathtt{Alice} : Ack_2\langle \mathsf{unit}\rangle;\mathbf{x}_7 \\
\mathbf{x}_{18} \mid \mathbf{x}_{10} &= \mathbf{x}_{19} \\
\mathbf{x}_{19} &= \mathtt{Alice} \to \mathtt{Bob} : Msg_3\langle \mathsf{string}\rangle;\mathbf{x}_{20} \\
\mathbf{x}_{20} &= \mathbf{x}_{21} \mid \mathbf{x}_{22} \\
\mathbf{x}_{21} &= \mathtt{Bob} \to \mathtt{Alice} : Ack_3\langle \mathsf{unit}\rangle;\mathbf{x}_9 \\
\mathbf{x}_{22} \mid \mathbf{x}_{14} &= \mathbf{x}_5\ \mathsf{in}\ \mathbf{x}_0
\end{aligned}
$$



10.

$$
\begin{aligned}
\mathbf{G}_{10} = \mathsf{def}\ \mathbf{x}_0 &= \mathbf{x}_1 + \mathbf{x}_2 \\
\mathbf{x}_1 + \mathbf{x}_3 &= \mathbf{x}_4 \\
\mathbf{x}_2 + \mathbf{x}_5 &= \mathbf{x}_6 \\
\mathbf{x}_4 &= \mathtt{Alice} \to \mathtt{Bob} : Msg_1\langle \mathsf{string}\rangle;\mathbf{x}_7 \\
\mathbf{x}_7 &= \mathbf{x}_8 + \mathbf{x}_9 \\
\mathbf{x}_8 &= \mathtt{Bob} \to \mathtt{Alice} : Ack_1\langle \mathsf{unit}\rangle;\mathbf{x}_{10} \\
\mathbf{x}_6 + \mathbf{x}_9 &= \mathbf{x}_{11} \\
\mathbf{x}_{11} &= \mathtt{Alice} \to \mathtt{Bob} : Msg_2\langle \mathsf{string}\rangle;\mathbf{x}_{12} \\
\mathbf{x}_{12} &= \mathbf{x}_{13} + \mathbf{x}_{14} \\
\mathbf{x}_{13} &= \mathtt{Bob} \to \mathtt{Alice} : Ack_2\langle \mathsf{unit}\rangle;\mathbf{x}_5 \\
\mathbf{x}_{10} + \mathbf{x}_{14} &= \mathbf{x}_3\ \mathsf{in}\ \mathbf{x}_0
\end{aligned}
$$



B.2 Sanity Condition

As the (Unambiguity), (Unique start) and (Unique end) are trivial, we do not provide additional details. We therefore focus our explanations on the (Thread correctness) condition.

Motivation The thread correctness condition aims at verifying some graph properties of the global type. This first to be ensured is connexity: each transition can be executable from the initial state. The second property is the ability to reach end: it is essential to achieve the liveness property. The third property that is checked is that global types should only join states that are always occurring concurrently: this prevents deadlocks.

Petri net representation We state well-threadedness as a Petri net property. We therefore first define how global types can be seen as a special subclass of free-choice Petri nets.

Definition B.1 (Petri net representation). *Given a global type* $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$*, we define the Petri net* $\mathbb{P}(\mathbf{G})$ *by:*

- *Each state variable* $\mathbf{x} \in \widetilde{G}$ *is a place in* $\mathbb{P}(\mathbf{G})$.
- *All the places are initially empty, except* $\mathbf{x}_0$*, which initially has one token.*
- *Transitions in* $\widetilde{G}$ *are translated according to their kind:*
  - *If* $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle; \mathbf{x}' \in \widetilde{G}$ *then their is a transition in* $\mathbb{P}(\mathbf{G})$*, whose unique input arc comes from* $\mathbf{x}$ *and whose unique output arc goes to* $\mathbf{x}'$.
  - *If* $\mathbf{x}_1 = \mathbf{x}_2 \mid \mathbf{x}_3 \in \widetilde{G}$ *then there is a transition in* $\mathbb{P}(\mathbf{G})$*, whose unique input arc comes from* $\mathbf{x}_1$ *and whose two outputs arcs go to* $\mathbf{x}_2$ *and* $\mathbf{x}_3$.
  - *If* $\mathbf{x}_1 = \mathbf{x}_2 + \mathbf{x}_3 \in \widetilde{G}$ *then there are two transitions in* $\mathbb{P}(\mathbf{G})$*, that each have an input arc from* $\mathbf{x}_1$ *and that respectively have an output arc to* $\mathbf{x}_2$ *and* $\mathbf{x}_3$.
  - *If* $\mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}_3 \in \widetilde{G}$ *then there are two transitions in* $\mathbb{P}(\mathbf{G})$*, that respectively have an input arc from* $\mathbf{x}_1$ *and* $\mathbf{x}_2$ *and that both have an output arc to* $\mathbf{x}_3$.
  - *If* $\mathbf{x}_1 \mid \mathbf{x}_2 = \mathbf{x}_3 \in \widetilde{G}$ *then there is a transition in* $\mathbb{P}(\mathbf{G})$*, whose two input arcs respectively come from* $\mathbf{x}_1$ *and* $\mathbf{x}_2$ *and whose unique output arc goes to* $\mathbf{x}_3$.

By construction, a global type $\mathbf{G}$ gives a free-choice Petri net $\mathbb{P}(\mathbf{G})$.

Theoretical well-threadedness We state well-threadedness as a Petri net property.

Definition B.2 (Theoretical well-threadedness). *A global type* $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ *satisfies the well-threadedness condition if* $\mathbb{P}(\mathbf{G})$ *is:*

1. *weakly-connected, and*
2. *safe (i.e.* 1*-bounded), and*
3. *deadlock-free, and*
4. *if a transition* $\mathbf{x} = \mathsf{end}$ *is in* $\widetilde{G}$*, then the marking where all places are empty except for* $\mathbf{x}$ *with one token, is reachable.*

Well-threadedness and polynomial verification algorithm We define well-threadedness by a polynomial verification algorithm. It relies on the Petri net connection of Def. B.1.

We call a Petri-net an ST-system if it is an S-system or a T-systems, or, recursively, if, by replacing a S-system or T-system subnet that has a single entry and a single exit by a single transition, we get a ST-system.

Definition B.3 (Well-threadedness). *A global type* $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ *is said to be well-threaded if, once applied:*

Fig. 16. Thread Correctness Reduction on global types

$$\widetilde{G}, \mathbf{x} = \mathbf{x}' \longrightarrow \widetilde{G}[\mathbf{x}/\mathbf{x}'] \quad \text{[Sub]}$$
$$\widetilde{G}, \mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle; \mathbf{x}' \longrightarrow \widetilde{G}, \mathbf{x} = \mathbf{x}' \quad \text{[Trans]}$$
$$\widetilde{G}, \mathbf{x}_1 = \mathbf{x}_2 \mid \mathbf{x}_3, \mathbf{x}_2 \mid \mathbf{x}_3 = \mathbf{x}_4 \longrightarrow \widetilde{G}, \mathbf{x}_1 = \mathbf{x}_4 \quad \text{[Par]}$$
$$\widetilde{G}, \mathbf{x}_1 = \mathbf{x}_2 + \mathbf{x}_3, \mathbf{x}_2 + \mathbf{x}_3 = \mathbf{x}_4 \longrightarrow \widetilde{G}, \mathbf{x}_1 = \mathbf{x}_4 \quad \text{[Bra]}$$
$$\widetilde{G}, \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}_3, \mathbf{x}_3 = \mathbf{x}_4 + \mathbf{x}_2 \longrightarrow \widetilde{G}, \mathbf{x}_1 = \mathbf{x}_4 \quad \text{[Rec]}$$

– *the rewriting rules of Fig. 16, followed by*
– *the transformation into a Petri net of Def. B.1,*

the resulting Petri net is an ST-system that satisfies theoretical well-threadedness.

Each of these actions is polynomial, as S-systems and T-systems are well-known polynomial subclasses of Petri nets (S-systems correspond to global types without fork and join; T-systems correspond to global types without choice).

Finding a polynomial algorithm which checks exactly the theoretical well-threadedness properties is an open problem. Existing results show that for general free-choice Petri nets, checking 1-boundedness is PSPACE-complete [16]. Then, for 1-bounded free-choice Petri nets, deadlock-freedom is NP-complete [8]. For 1-bounded Petri nets, it is also known that reachability is PSPACE-complete [8]. We however believe that global types represent a smaller class of Petri nets with polynomial time properties. The solution presented here is to use an algorithm which works polynomially, but does not cover all theoretical well-threaded global types.

B.3 Local Choice Condition

We list here some omitted definitions and examples from the main section.

Active Sender The active sender computation is not as much a counting algorithm as a verification method that checks whether a given choice has a unique (i.e. local) responsible participant. In particular, the deduction rules gather the active senders on each branch and restrict it to be a singleton p.

Local Choice Example (Local choice) is essential for the consistency of the distributed interaction with respect to choice.

Consider the following incorrect global type:

$$\begin{aligned}
\mathbf{G}_{\neg\text{ch.a.}} = \mathsf{def}\ \mathbf{x}_0 &= \mathbf{x}_1 + \mathbf{x}_3 \\
\mathbf{x}_1 &= \mathtt{Alice} \to \mathtt{Bob} : \mathit{Item}\langle \mathsf{string} \rangle; \mathbf{x}_2 \\
\mathbf{x}_2 &= \mathtt{Bob} \to \mathtt{Dave} : \mathit{Book}\langle \mathsf{string} \rangle; \mathbf{x}_4 \\
\mathbf{x}_3 &= \mathtt{Carol} \to \mathtt{Bob} : \mathit{Film}\langle \mathsf{string} \rangle; \mathbf{x}_5 \\
\mathbf{x}_4 + \mathbf{x}_5 &= \mathbf{x}_6 \\
\mathbf{x}_6 &= \mathsf{end\ in}\ \mathbf{x}_0
\end{aligned}$$

This global type is incorrect because both `Alice` and `Carol` are supposed to make a local choice between *Item* and *Film* (prevented by $\mathit{ASend}(\widetilde{G}_{\neg\text{ch.a.}})(\mathbf{x}_0)$). Even if `Alice`

Fig. 17. Active Sender Computation

$$\frac{\mathbf{x} = \mathsf{end} \in \widetilde{G}}{\widetilde{G} \vdash \mathbf{x} : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x} : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2} \qquad \frac{\mathbf{x}_1 = \mathsf{end} \in \widetilde{G} \quad \mathbf{x}_2 \in \tilde{\mathbf{x}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\mathbf{x}_1 \in \tilde{\mathbf{x}}_1 \quad \mathbf{x}_2 \in \tilde{\mathbf{x}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2} \qquad \frac{\mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x} \in \widetilde{G}}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{(\mathbf{x}_1 \mid \mathbf{x} = \mathbf{x}' \in \widetilde{G} \vee \mathbf{x} \mid \mathbf{x}_1 = \mathbf{x}' \in \widetilde{G}) \quad \widetilde{G} \vdash \mathbf{x}' : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{(\mathbf{x}_1 + \mathbf{x} = \mathbf{x}' \in \widetilde{G} \vee \mathbf{x} + \mathbf{x}_1 = \mathbf{x}' \in \widetilde{G}) \quad \widetilde{G} \vdash \mathbf{x}' : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\mathbf{x}_1 = \mathbf{x} \mid \mathbf{x}' \in \widetilde{G} \quad \widetilde{G} \vdash \mathbf{x} : \mathbf{x}\tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2 \quad \widetilde{G} \vdash \mathbf{x}' : \mathbf{x}\tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\mathbf{x}_1 = \mathbf{x} + \mathbf{x}' \in \widetilde{G} \quad \widetilde{G} \vdash \mathbf{x} : \mathbf{x}\tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2 \quad \widetilde{G} \vdash \mathbf{x}' : \mathbf{x}\tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\mathbf{x}_1 = \mathrm{p} \to \mathrm{p}' : l\langle U \rangle; \mathbf{x}_1' \in \widetilde{G} \quad \mathrm{p} \in \tilde{\mathrm{p}}_1 \quad \widetilde{G} \vdash \mathbf{x}_1' : \mathbf{x}\tilde{\mathbf{x}}_1, \mathrm{p}'\tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\widetilde{G} \vdash \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2 \,[\!]\, \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1}{\widetilde{G} \vdash \mathbf{x}_1 : \tilde{\mathbf{x}}_1, \tilde{\mathrm{p}}_1 \,[\!]\, \mathbf{x}_2 : \tilde{\mathbf{x}}_2, \tilde{\mathrm{p}}_2}$$

$$\frac{\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2 \in \widetilde{G} \quad \widetilde{G} \vdash \mathbf{x}_1 : \emptyset, \mathrm{p} \,[\!]\, \mathbf{x}_2 : \emptyset, \mathrm{p}}{\mathit{ASend}(\widetilde{G})(\mathbf{x}) = \mathrm{p}}$$

were to send message *Film*, it forgets to inform `Dave` of the choice, which may result in him waiting forever for a message *Book* that will never come (it is prevented by the condition $\mathit{Rcv}(\widetilde{G}_{\neg \text{ch.a.}})(\mathbf{x}_1) \neq \mathit{Rcv}(\widetilde{G}_{\neg \text{ch.a.}})(\mathbf{x}_3)$). Note that $\mathbf{G}_5$ satisfies the local choice property.

The following incorrect examples illustrates how local choice can also prevents two identical labels from being in the two branches of the same choice (it confuses `Bob`):

$$\begin{aligned}
\mathbf{G}_{\neg \text{lin}} = \mathsf{def}\ \mathbf{x}_0 &= \mathbf{x}_1 + \mathbf{x}_3 \\
\mathbf{x}_1 &= \texttt{Alice} \to \texttt{Bob} : \mathit{Film}\langle \mathsf{string} \rangle; \mathbf{x}_2 \\
\mathbf{x}_2 &= \texttt{Bob} \to \texttt{Alice} : \mathit{Book}\langle \mathsf{string} \rangle; \mathbf{x}_4 \\
\mathbf{x}_3 &= \texttt{Alice} \to \texttt{Bob} : \mathit{Film}\langle \mathsf{string} \rangle; \mathbf{x}_5 \\
\mathbf{x}_4 + \mathbf{x}_5 &= \mathbf{x}_6 \\
\mathbf{x}_6 &= \mathsf{end}\ \mathsf{in}\ \mathbf{x}_0
\end{aligned}$$

B.4 Linearity Condition

The *linearity* condition is enforced by comparing the results of the $\mathit{Lin}(\widetilde{G})(\mathbf{x}_1)$ and $\mathit{Lin}(\widetilde{G})(\mathbf{x}_2)$ whenever a forking transition $\mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2$ is in $\widetilde{G}$. The *Lin* function is defined in figure 18.

The *Lin* function works in a similar way on message labels as the *Rcv* function on message receivers (linearity is to forks what choice awareness is to choice) except that the *Lin* function also collects joining points in order to establish which labelled message are possibly conflicting. As with *Rcv*, the equality $\mathit{Lin}(\widetilde{G})(\mathbf{x}_1) = \mathit{Lin}(\widetilde{G})(\mathbf{x}_2)$

Fig. 18. Linearity Computation (up to permutation of $|$ and $+$)

$$
\begin{array}{ll}
Lin(\widetilde{G})(\mathbf{x}) = Lin(\widetilde{G},\emptyset,\emptyset)(\mathbf{x}) & \text{(remembers merges and joins)} \\
Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = \emptyset & \text{if } \mathbf{x}+\mathbf{x}' = \mathbf{x}'' \in \widetilde{G} \wedge \mathbf{x}'' \in \tilde{\mathbf{x}} \text{ or } \mathbf{x} = \mathsf{end} \in \widetilde{G} \\
Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = \{\mathtt{p}\mathtt{p}' : l : \tilde{\mathbf{x}}_j\} \cup Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}') & \text{if } \mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle;\mathbf{x}' \in \widetilde{G} \wedge \mathtt{p}' \notin \tilde{\mathtt{p}} \\
Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}') \cup Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}'') & \text{if } \mathbf{x} = \mathbf{x}' + \mathbf{x}'' \in \widetilde{G} \\
Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}') \cup Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}'') & \text{if } \mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \in \widetilde{G} \\
Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = Lin(\widetilde{G},\tilde{\mathbf{x}}_m\mathbf{x}'',\tilde{\mathbf{x}}_j)(\mathbf{x}'') & \text{if } \mathbf{x}' + \mathbf{x} = \mathbf{x}'' \in \widetilde{G} \wedge \mathbf{x}'' \notin \tilde{\mathbf{x}} \\
Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j)(\mathbf{x}) = Lin(\widetilde{G},\tilde{\mathbf{x}}_m,\tilde{\mathbf{x}}_j\mathbf{x}'')(\mathbf{x}'') & \text{if } \mathbf{x}' \mid \mathbf{x} = \mathbf{x}'' \in \widetilde{G}
\end{array}
$$

holds if $\forall(\mathtt{p}\mathtt{p}' : l_1 : \tilde{\mathbf{x}}_1) \in Lin(\widetilde{G})(\mathbf{x}_1), \forall(\mathtt{p}\mathtt{p}' : l_2 : \tilde{\mathbf{x}}_2) \in Lin(\widetilde{G})(\mathbf{x}_2), l_1 \neq l_2 \vee \tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2$ share a non-null suffix.

C Appendix for Section 3

C.1 Well-formedness for Local Types

Fig. 19. Local Thread Collection Rules

$$
\frac{\chi \equiv \chi_0 \quad \chi_0 \to_{\widetilde{T}} \chi_0' \quad \chi_0' \equiv \chi'}{\chi \to_{\widetilde{T}} \chi'}
\qquad
\frac{\mathbf{x} = \mathbf{x}' \in \widetilde{T}}{\chi[\mathbf{x}] \to_{\widetilde{T}} \chi[\mathbf{x}']}
\qquad
\frac{\mathbf{x} = !\,\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}' \in \widetilde{T}}{\chi[\mathbf{x}] \to_{\widetilde{T}} \chi[\mathbf{x}']}
$$

$$
\frac{\mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2 \in \widetilde{T}}{\chi[\mathbf{x}] \to_{\widetilde{T}} \chi[\mathbf{x}_1 \mid \mathbf{x}_2]}
\qquad
\frac{\mathbf{x}_1 \mid \mathbf{x}_2 = \mathbf{x} \in \widetilde{T}}{\chi[\mathbf{x}_1 \mid \mathbf{x}_2] \to_{\widetilde{T}} \chi[\mathbf{x}]}
\qquad
\frac{\mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2 \in \widetilde{T}}{\chi[\mathbf{x}] \to_{\widetilde{T}} \chi[\mathbf{x}_1 + \mathbf{x}_2]}
\qquad
\frac{\mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x} \in \widetilde{T}}{\chi[\mathbf{x}_1 + \mathbf{x}_2] \to_{\widetilde{T}} \chi[\mathbf{x}]}
$$

$$
\frac{\mathbf{x}_1 + \mathbf{x} = \mathbf{x}_2,\ \mathbf{x}_1' = \mathbf{x}' \oplus \mathbf{x}_2' \in \widetilde{T} \quad \chi[\mathbf{x}_2] \to_{\widetilde{T}}^* \chi'[\mathbf{x}_1'] \quad \chi'[\mathbf{x}_2'] \to_{\widetilde{T}}^* \chi[\mathbf{x}_1]}{\chi[\mathbf{x}] \to_{\widetilde{T}} \chi'[\mathbf{x}']}
$$

$$
\frac{\mathbf{x}_1 + \mathbf{x} = \mathbf{x}_2,\ \mathbf{x}_1' = \mathbf{x}_2' \oplus \mathbf{x}' \in \widetilde{T} \quad \chi[\mathbf{x}_2] \to_{\widetilde{T}}^* \chi'[\mathbf{x}_1'] \quad \chi'[\mathbf{x}_2'] \to_{\widetilde{T}}^* \chi[\mathbf{x}_1]}{\chi[\mathbf{x}] \to_{\widetilde{T}} \chi'[\mathbf{x}']}
$$

$$
\frac{\mathbf{x} + \mathbf{x}_1 = \mathbf{x}_2,\ \mathbf{x}_1' = \mathbf{x}' \oplus \mathbf{x}_2' \in \widetilde{T} \quad \chi[\mathbf{x}_2] \to_{\widetilde{T}}^* \chi'[\mathbf{x}_1'] \quad \chi'[\mathbf{x}_2'] \to_{\widetilde{T}}^* \chi[\mathbf{x}_1]}{\chi[\mathbf{x}] \to_{\widetilde{T}} \chi'[\mathbf{x}']}
$$

$$
\frac{\mathbf{x} + \mathbf{x}_1 = \mathbf{x}_2,\ \mathbf{x}_1' = \mathbf{x}_2' \oplus \mathbf{x}' \in \widetilde{T} \quad \chi[\mathbf{x}_2] \to_{\widetilde{T}}^* \chi'[\mathbf{x}_1'] \quad \chi'[\mathbf{x}_2'] \to_{\widetilde{T}}^* \chi[\mathbf{x}_1]}{\chi[\mathbf{x}] \to_{\widetilde{T}} \chi'[\mathbf{x}']}
$$

$$
\frac{\mathbf{x} = \mathsf{end} \in \widetilde{T}}{\widetilde{T} \vdash \mathbf{x} \triangleright \mathsf{ok}}
\qquad
\frac{\mathbf{x} = \mathsf{end} \notin \widetilde{T} \quad \{\mathbf{x}_i + \mathbf{x}_i' = \mathbf{x}_i''\}_{0<i\leq n} \subset \widetilde{T} \quad \mathcal{T}[\mathbf{x}_1'', \ldots, \mathbf{x}_n''] \to_{\widetilde{T}}^* \mathcal{T}[\mathbf{x}_1', \ldots, \mathbf{x}_n']}{\widetilde{T} \vdash \mathcal{T}[\mathbf{x}_1, \ldots, \mathbf{x}_n] \triangleright \mathsf{ok}}
$$

$$
\frac{\mathbf{x}_0 \to_{\widetilde{T}}^* \chi \quad \widetilde{T} \vdash \chi \triangleright \mathsf{ok}}{\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_0 \triangleright \mathsf{ok}}
$$

The rules for the external choice and receive are defined as the internal choice and send.

We first define the well-formed local types in figure 19. Well-formedness conditions for local types are similarly defined as the ones for global types, including inductive

rules for thread collections. Definitions for local versions of χ and $\mathcal{T}$ are similar to their global counterparts.

Proposition C.1 (Well-formedness). *If* $\mathbf{G}$ *is well-formed, then the projected types* $\{\mathbf{G} \upharpoonright \mathtt{p}\}_i$ *are well-formed.*

C.2 Congruence Rules for Local Types

For local types of the form $\mathbf{T} = \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_0$, we define in figure 20 a congruent relation $\equiv$ over $\widetilde{T}$.

Fig. 20. Local Type Congruence (up to commutativity of $|$ and $+$)

$$\begin{aligned}
\widetilde{T}, \mathbf{x} = \mathbf{x}' &\equiv \widetilde{T}[\mathbf{x}/\mathbf{x}'] \\
\mathbf{x}_1 = \mathbf{x}_2 \oplus \mathbf{x}_3, \mathbf{x}_2 + \mathbf{x}_3 = \mathbf{x}_4 &\equiv \mathbf{x}_1 = \mathbf{x}_4 \\
\mathbf{x}_1 = \mathbf{x}_2 \,\&\, \mathbf{x}_3, \mathbf{x}_2 + \mathbf{x}_3 = \mathbf{x}_4 &\equiv \mathbf{x}_1 = \mathbf{x}_4 \\
\mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}_3, \mathbf{x}_3 = \mathbf{x}_2 \,\&\, \mathbf{x}_4 &\equiv \mathbf{x}_1 = \mathbf{x}_4 \\
\mathbf{x}_1 = \mathbf{x}_2 \mid \mathbf{x}_3, \mathbf{x}_2 \mid \mathbf{x}_3 = \mathbf{x}_4 &\equiv \mathbf{x}_1 = \mathbf{x}_4
\end{aligned}$$

It eliminates the indirections and locally irrelevant choices, and removes the unused local threads that can be created by projection. It also eases type-checking by allowing the projected types to be slightly different from the local types that can be inferred from processes.

C.3 From Global Types to Global State Automata

This subsection defines the global state automata which are used as the intermediate automata for the proofs in the main section in § 3.

We can represent the sequences of actions that a global type specifies as a communicating state machine. This allows to interpret some well-formedness conditions of global types as standard properties of communicating finite state machines.

We write $\mathtt{X}(\mathbf{G})$ the set of well-formed states built from the recursion variables of a given global type $\mathbf{G}$. We define local state contexts in a standard way.

Definition C.1 (Global State Automaton). We start from a global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ and define the automaton $\mathcal{A}(\mathbf{G}) = (Q, C, q_0, \mathbb{A}, \delta)$ in the following way:

- $\mathbb{A}$ is the set of $\{l \in T\}$
- Q is defined as a subset of $(\mathtt{X}(\mathbf{G}))^p \times (\mathbb{A}^*)^{p(p-1)}$, with p the number of participants in $\mathbf{G}$.
  We write, for $q \in Q$, $q.\mathtt{p}$ to designate the component $\mathtt{X}_\mathtt{p}$ of q at $\mathtt{p}$'s position, and $q.\mathtt{pq}$ for the content of the channel between $\mathtt{p}$ and $\mathtt{q}$. We use the notation $q[\mathtt{p} := \mathtt{X}]$ to represent a state q' for which, $\forall \mathtt{p}' \neq \mathtt{p}, q'.\mathtt{p}' = q.\mathtt{p}'$ and $q'.\mathtt{p} = \mathtt{X}$. We similarly define $q[\mathtt{pp}' := \omega]$ for $\omega \in \mathbb{A}^*$.
  Q is defined up to an equivalence relation on local states $\equiv_{\widetilde{G}}$ defined in figure 21 and by $q \equiv_{\widetilde{G}} q[\mathtt{p} := \mathtt{X}]$ if $\mathtt{p} \vdash q.\mathtt{p} \equiv_{\widetilde{G}} \mathtt{X}$.

- $C = \{\mathtt{pq} \mid \mathtt{p},\mathtt{q} \in G\}$
- $q_0 = (\mathbf{x}_0,\ldots,\mathbf{x}_0)$
- δ is defined by:
  - $(q,(\mathtt{pp}'!l),q') \in \delta$ if $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle.\mathbf{x}' \in G$ and $q.\mathtt{p} = \mathtt{X}[\mathbf{x}]$ and $q.\mathtt{pp}' = \omega$ and $q'.\mathtt{p} = \mathtt{X}[\mathbf{x}']$ and $q'.\mathtt{pp}' = \omega \cdot l$ and all the other components of q and q' are identical.
  - $(q,(\mathtt{pp}'?l),q') \in \delta$ if $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle.\mathbf{x}' \in G$ and $q.\mathtt{pp}' = l \cdot \omega$ and $q.\mathtt{p}' = \mathtt{X}'[\mathbf{x}]$ and $q'.\mathtt{p}' = \mathtt{X}'[\mathbf{x}']$ and $q'.\mathtt{pp}' = \omega$ and all the other components of q and q' are identical.

Fig. 21. Local State Equivalence for Global State Automata

$$\frac{}{\mathtt{p} \vdash \mathtt{X} \mid \mathtt{X}' \equiv_{\widetilde{G}} \mathtt{X}' \mid \mathtt{X}} \qquad \frac{}{\mathtt{p} \vdash \mathtt{X} \mid (\mathtt{X}' \mid \mathtt{X}'') \equiv_{\widetilde{G}} (\mathtt{X} \mid \mathtt{X}') \mid \mathtt{X}''}$$

$$\frac{\mathbf{x} = \mathtt{p}' \to \mathtt{p}'' : l\langle U\rangle.\mathbf{x}' \in \widetilde{G} \quad \mathtt{p} \notin \{\mathtt{p}',\mathtt{p}''\}}{\mathtt{p} \vdash \mathtt{X}[\mathbf{x}] \equiv_{\widetilde{G}} \mathtt{X}[\mathbf{x}']}$$

$$\frac{\mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \in \widetilde{G}}{\mathtt{p} \vdash \mathtt{X}[\mathbf{x}] \equiv_{\widetilde{G}} \mathtt{X}[\mathbf{x}' \mid \mathbf{x}'']} \qquad \frac{\mathbf{x} \mid \mathbf{x}' = \mathbf{x}'' \in \widetilde{G}}{\mathtt{p} \vdash \mathtt{X}[\mathbf{x} \mid \mathbf{x}'] \equiv_{\widetilde{G}} \mathtt{X}[\mathbf{x}'']} \qquad \frac{\mathbf{x} = \mathbf{x}' + \mathbf{x}'' \in \widetilde{G}}{\mathtt{p} \vdash \mathtt{X}[\mathbf{x}] \equiv_{\widetilde{G}} \mathtt{X}[\mathbf{x}']} \qquad \frac{\mathbf{x} = \mathbf{x}' + \mathbf{x}'' \in \widetilde{G}}{\mathtt{p} \vdash \mathtt{X}[\mathbf{x}] \equiv_{\widetilde{G}} \mathtt{X}[\mathbf{x}'']}$$

$$\frac{\mathbf{x} + \mathbf{x}' = \mathbf{x}'' \in \widetilde{G}}{\mathtt{p} \vdash \mathtt{X}[\mathbf{x}] \equiv_{\widetilde{G}} \mathtt{X}[\mathbf{x}'']} \qquad \frac{\mathbf{x} + \mathbf{x}' = \mathbf{x}'' \in \widetilde{G}}{\mathtt{p} \vdash \mathtt{X}[\mathbf{x}'] \equiv_{\widetilde{G}} \mathtt{X}[\mathbf{x}'']}$$

C.4 Causal Ordering in Global Types

Now that both local and global automata have been defined, we need some reasoning methods to link their behaviour to the global and local types.

Causal Ordering We write $\mathbf{x} <_{\widetilde{G}} \mathbf{x}'$ if $\mathbf{x} \in \mathsf{fv}_L(G_i)$ and $\mathbf{x}' \in \mathsf{fv}_R(G_i)$ for some $G_i \in \widetilde{G}$, i.e. $\mathbf{x}$ appears on the left and $\mathbf{x}'$ on the right of a single transition G_i. For example, $\mathbf{x} <_{\widetilde{G}} \mathbf{x}'$ if $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle;\mathbf{x}' \in \widetilde{G}$, and $\mathbf{x} <_{\widetilde{G}} \mathbf{x}'$ and $\mathbf{x} <_{\widetilde{G}} \mathbf{x}''$ if $\mathbf{x} = \mathbf{x}' \mid \mathbf{x}'' \in \widetilde{G}$ or $\mathbf{x} = \mathbf{x}' + \mathbf{x}'' \in \widetilde{G}$. Set $<^+_{\widetilde{G}}$ as the transitive closure of $<_{\widetilde{G}}$. The relation $<^+_{\widetilde{G}}$ is not an order because of recursion.

We say $\mathbf{x}_0$ and $\mathbf{x}_1$ are *consecutive* (written $\mathbf{x}_0 \prec_{\widetilde{G}} \mathbf{x}_1$) if $\mathbf{x}_0 = \mathtt{p}_0 \to \mathtt{p}'_0 : l_0\langle U_0\rangle;\mathbf{x}'_0 \in \widetilde{G}$ and $\mathbf{x}_1 = \mathtt{p}_1 \to \mathtt{p}'_1 : l_1\langle U_1\rangle;\mathbf{x}'_1 \in \widetilde{G}$ such that $\mathbf{x}_0 <^+_{\widetilde{G}} \mathbf{x}_1$ and if there is no $\mathbf{x}_2 = \mathtt{p}_2 \to \mathtt{p}'_2 : l_2\langle U_2\rangle;\mathbf{x}'_2 \in \widetilde{G}$ such that $\mathbf{x}_0 <^+_{\widetilde{G}} \mathbf{x}_2$ and $\mathbf{x}_2 <^+_{\widetilde{G}} \mathbf{x}_1$.

We denote by $\mathbf{x}_1 \prec_{\widetilde{T}} \mathbf{x}_2$ the same consecutiveness relation adapted to local types ($\mathbf{x}_1$ and $\mathbf{x}_2$ then correspond to sending or receiving transitions).

Executions Before proceeding, we need to elaborate on the definition of stable-outputs executions. We use the term *stable-outputs sequence* (SOS) execution to refer to an execution φ that is such that $\varphi = \varphi_1 \ldots \varphi_n$ where all $\varphi_1,\ldots,\varphi_n$ are stable-outputs.

An execution $s_0 \xrightarrow{\varphi} s_n$ in $\mathcal{A}(\mathbf{G})$ is said to be *globally ordered* if, for any transition t such that $\varphi = \varphi_0 t \varphi_1$, either t is of the form $\mathtt{pq}!l$, with corresponding global transition $\mathbf{x}_1 = \mathtt{p} \to \mathtt{q} : l_1\langle U_1\rangle; \mathbf{x}_1' \in \widetilde{G}$, and there exists in φ_0 an action of the form $\mathtt{p'q'}?l'$, with corresponding global transition $\mathbf{x}_2 = \mathtt{p}' \to \mathtt{q}' : l'\langle U'\rangle; \mathbf{x}' \in \widetilde{G}$ such that $\mathbf{x}_2 \prec_{\widetilde{G}} \mathbf{x}_1$; or t is of the form $\mathtt{pq}?l$ and φ_0 is of the form $\varphi_0' t'$ with $t' = \mathtt{pq}!l$.

Note that globally ordered executions always start from the initial state and are thus 1-buffer.

Transition Causality We now define causality ordering between transitions in global automata. For any reachable state s of a global automata $\mathcal{A}(\mathbf{G})$ such that $s \xrightarrow{t_1} s_1 \xrightarrow{\varphi} s_2 \xrightarrow{t_2} s_3$, we say that t_2 depends on t_1, or t_1 causes t_2, written $t_1 \lhd t_2$ if either:

- (OO) $t_1 = \mathtt{pq}_1!l_1$ and $t_2 = \mathtt{pq}_2!l_2$ and $\mathbf{x}_1 = \mathtt{p} \to \mathtt{q}_1 : l_1\langle U_1\rangle; \mathbf{x}_1' \in \widetilde{G}$ and $\mathbf{x}_2 = \mathtt{p} \to \mathtt{q}_2 : l_2\langle U_2\rangle; \mathbf{x}_2' \in \widetilde{G}$ and $\mathbf{x}_1 \prec_{\widetilde{T}} \mathbf{x}_2$ (with $\widetilde{T} = \widetilde{G} \upharpoonright \mathtt{p}$) and $s.\mathtt{p} = \mathtt{X}[\mathbf{x}_1]$;
- (II) $t_1 = \mathtt{p}_1\mathtt{q}?l_1$ and $t_2 = \mathtt{p}_2\mathtt{q}?l_2$ and $\mathbf{x}_1 = \mathtt{p}_1 \to \mathtt{q} : l_1\langle U_1\rangle; \mathbf{x}_1' \in \widetilde{G}$ and $\mathbf{x}_2 = \mathtt{p}_2 \to \mathtt{q} : l_2\langle U_2\rangle; \mathbf{x}_2' \in \widetilde{G}$ and $\mathbf{x}_1 \prec_{\widetilde{T}} \mathbf{x}_2$ (with $\widetilde{T} = \widetilde{G} \upharpoonright \mathtt{q}$) and $s.\mathtt{q} = \mathtt{X}[\mathbf{x}_1]$;
- (IO) $t_1 = \mathtt{p}_1\mathtt{q}?l_1$ and $t_2 = \mathtt{qp}_2!l_2$ and $\mathbf{x}_1 = \mathtt{p}_1 \to \mathtt{q} : l_1\langle U_1\rangle; \mathbf{x}_1' \in \widetilde{G}$ and $\mathbf{x}_2 = \mathtt{q} \to \mathtt{p}_2 : l_2\langle U_2\rangle; \mathbf{x}_2' \in \widetilde{G}$ and $\mathbf{x}_1 \prec_{\widetilde{T}} \mathbf{x}_2$ (with $\widetilde{T} = \widetilde{G} \upharpoonright \mathtt{q}$) and $s.\mathtt{q} = \mathtt{X}[\mathbf{x}_1]$;
- (OI) $t_1 = \mathtt{pq}_1!l_1$ and $t_2 = \mathtt{q}_2\mathtt{p}?l_2$ and $\mathbf{x}_1 = \mathtt{p} \to \mathtt{q}_1 : l_1\langle U_1\rangle; \mathbf{x}_1' \in \widetilde{G}$ and $\mathbf{x}_2 = \mathtt{q}_2 \to \mathtt{p} : l_2\langle U_2\rangle; \mathbf{x}_2' \in \widetilde{G}$ and $\mathbf{x}_1 \prec_{\widetilde{T}} \mathbf{x}_2$ (with $\widetilde{T} = \widetilde{G} \upharpoonright \mathtt{q}$) and $s.\mathtt{q} = \mathtt{X}[\mathbf{x}_1]$;
- (Comm) $t_1 = \mathtt{pq}!l$ and $t_2 = \mathtt{pq}?l$ and $\mathbf{x}_1 = \mathtt{p} \to \mathtt{q} : l_1\langle U_1\rangle; \mathbf{x}_1' \in \widetilde{G}$ and $s.\mathtt{p} = \mathtt{X}[\mathbf{x}_1]$ and $s.\mathtt{q} = \mathtt{X}'[\mathbf{x}_1]$;

Note that the (OO), (II), (IO) and (OI) causalities correspond to local ordering, while the (Comm) causality is the result of communication. We write $t_1 \bowtie t_2$ if $t_1 \lhd t_2$ does not hold.

We say that an execution φ is a *causal chain* if, for any $t \in \varphi$, there exists a $t' \in \varphi$ and $\varphi_0, \varphi_1, \varphi_2$ such that $\varphi = \varphi_0 t' \varphi_1 t \varphi_2$ and $t' \lhd t$.

Finally, we characterise an SOS execution φ as being a *SOS causal chain* if, for any $t \in \varphi$, there exists a $t' \in \varphi$ and $\varphi_0, \varphi_1, \varphi_2$ such that $\varphi = \varphi_0 t' \varphi_1 t \varphi_2$ and $t' \lhd t$, or t is an send and t is immediately followed in φ by the corresponding receive.

C.5 Global State Automata Properties

We now state some essential properties of global state automata.

Lemma C.1 (Commutativity). *For any well-formed global type* $\mathbf{G}$*, in the MSA* $\mathcal{A}(\mathbf{G})$*, if there is a reachable state* s *such that* $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2$ *and* $t_1 \bowtie t_2$*, then there exists a state* s_1' *such that* $s \xrightarrow{t_2} s_1' \xrightarrow{t_1} s_2$*.*

Proof. By case analysis on the transitions t_1 and t_2.

Proposition C.2. *For any well-formed global type* $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$*, if* $\mathtt{X}_1[\mathbf{x}] \equiv_{\widetilde{G}} \mathtt{X}_2[\mathbf{x}]$ *and* $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle.\mathbf{x}'$*, then* $\mathtt{X}_1'[\mathbf{x}'] \equiv_{\widetilde{G}} \mathtt{X}_2'[\mathbf{x}']$ *for all* $\mathtt{X}_1', \mathtt{X}_2'$ *such that* $\mathtt{X}_1 \equiv_{\widetilde{G}} \mathtt{X}_1'$ *and* $\mathtt{X}_2 \equiv_{\widetilde{G}} \mathtt{X}_2'$*.*

Proof. By definition of $\equiv$.

Proposition C.3 (Determinism). *If* $\mathbf{G}$ *is well-formed, then* $\mathcal{A}(\mathbf{G})$ *is deterministic.*

Proof. Suppose $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ is well-formed. We assume by contradiction that there exists a global state $q \in Q$ from which there are two transitions $(q, (\mathtt{pp}'!l), q_1'')$ and $(q, (\mathtt{pp}'!l), q_2'')$ with $q_1'' \not\equiv_{\widetilde{G}} q_2''$.

From the linearity condition and the definition of $\mathcal{A}(\mathbf{G})$, we know that there is a unique element of $\widetilde{G}$ of the form $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle.\mathbf{x}'$. Then by definition of $\mathcal{A}(\mathbf{G})$, there exists q_1, q_2, q_1', q_2' such that $q_1.\mathtt{p} = \mathtt{X}_1[\mathbf{x}]$, $q_2.\mathtt{p} = \mathtt{X}_2[\mathbf{x}]$, $q_1.\mathtt{pp}' = \omega_1$, $q_2.\mathtt{pp}' = \omega_2$, $q_1' = q_1[\mathtt{p} := \mathtt{X}_1[\mathbf{x}'], \mathtt{pp}' := \omega_1 \cdot l]$, $q_2' = q_2[\mathtt{p} := \mathtt{X}_2[\mathbf{x}'], \mathtt{pp}' := \omega_2 \cdot l]$ and $q \equiv_{\widetilde{G}} q_1 \equiv_{\widetilde{G}} q_2$ and $q_1'' \equiv_{\widetilde{G}} q_1'$ and $q_2'' \equiv_{\widetilde{G}} q_2'$.

First, by definition of $\equiv_{\widetilde{G}}$, we have $\omega_1 = \omega_2$. Second, by well-formedness, we know that $\mathtt{X}_1$ and $\mathtt{X}_2$ are contexts which do not contain either $\mathbf{x}$ or $\mathbf{x}'$. Then by Proposition C.2, $\mathtt{X}_1[\mathbf{x}'] \equiv_{\widetilde{G}} \mathtt{X}_2[\mathbf{x}']$. Hence it contradicts the assumption $q_1'' \not\equiv_{\widetilde{G}} q_2''$.

Proposition C.4 (Choice Awareness). *If* $\mathbf{G}$ *is well-formed, then* $\mathcal{A}(\mathbf{G})$ *is choice aware.*

Proof. By definition of well-formedness of $\mathbf{G}$.

Proposition C.5 (Correct Ending). *If* $\mathbf{G} = \mathit{def}\ \widetilde{G}\ \mathit{in}\ \mathbf{x}_0$ *is well-formed and if there is a transition* $\mathbf{x}_{\mathit{end}} = \mathit{end} \in \widetilde{G}$*, then* $\mathcal{A}(\mathbf{G})$ *does not have any transition from* $q = (\mathbf{x}_{\mathit{end}}, \ldots \mathbf{x}_{\mathit{end}}, \varepsilon, \ldots, \varepsilon)$.

Proof. There exists a unique $\mathbf{x}_{\mathsf{end}} = \mathsf{end} \in \widetilde{G}$. Suppose there is a transition from the end state $q_{\mathsf{end}} = (\mathbf{x}_{\mathsf{end}}, \ldots, \mathbf{x}_{\mathsf{end}}, \varepsilon, \ldots, \varepsilon)$. It means there exists a transition $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle.\mathbf{x}' \in \widetilde{G}$ for which there is a $q' \equiv q_{\mathsf{end}}$ such that $q.\mathtt{p} = \mathtt{X}[\mathbf{x}]$. All the equivalence rules have to be read from right to left with the important fact that no message rule implying $\mathtt{p}$ can be traversed. Only the rule $\mathbf{x}_1 = \mathbf{x}_2 \,\&\, \mathbf{x}_3$ and $\mathbf{x}_1 = \mathbf{x}_2 \oplus \mathbf{x}_3$ allows to be equivalent to a recursion variable that is on the left-hand side of a rule. But the (choice awareness rule) implies that active senders on both branches of the choice must be identical, which contradicts the fact that $\mathtt{p}$ is able to send a message.

Lemma C.2 (Parallel). *Assume* $\mathbf{G}$ *is well-formed. Suppose* $s \in RS(\mathcal{A}(\mathbf{G}))$ *and* $s \xrightarrow{t_1} s_1$ *and* $s \xrightarrow{t_2} s_2$.

1. *If* t_1 *and* t_2 *are both inputs at* $\mathtt{q}$*, then the state of* $\mathtt{q}$ *in* s *is* $\mathtt{X}[\mathbf{x}_1 \mid \mathbf{x}_2]$ *with* $\mathbf{x}_i = \mathtt{p}_\mathtt{i} \to \mathtt{q} : l_i\langle U_i\rangle.\mathbf{x}_i' \in \mathbf{G}$ *(*$i = 1, 2$*).*
2. *If* t_1 *is an input at* $\mathtt{q}$*, and* t_2 *is an output to* $\mathtt{q}$*, then the state of* $\mathtt{q}$ *in* s *is* $\mathtt{X}[\mathbf{x}_1 \mid \mathbf{x}_2]$ *with* $\mathbf{x}_1 = \mathtt{p}_1 \to \mathtt{q} : l_1\langle U_1\rangle.\mathbf{x}_1' \in \mathbf{G}$ *and* $\mathbf{x}_2 = \mathtt{q} \to \mathtt{q}_2 : l_2\langle U_2\rangle.\mathbf{x}_2' \in \mathbf{G}$.

Proof. For (1), by definition of δ of $\mathcal{A}(\mathbf{G})$, there is a state $\mathtt{X}$ such that $\mathtt{X} \equiv \mathtt{X}_1[\mathbf{x}_1] \equiv \mathtt{X}_2[\mathbf{x}_2]$. By assumption, we have $t_1 \bowtie t_2$. The statement means that we prove $\mathtt{X} \equiv \mathtt{X}'[\mathbf{x}_1 \mid \mathbf{x}_2]$ if $t_1 \bowtie t_2$ and there is no case $\mathtt{X} \equiv \mathtt{X}'[\mathbf{x}_1 + \mathbf{x}_2]$.

We prove by induction on the number of applications of $\equiv$ (the size of the $\equiv$ proof). The base case is obvious by $\mathtt{X} = \mathtt{X}_1[\mathbf{x}_1] = \mathtt{X}_2[\mathbf{x}_2] = \mathtt{X}'[\mathbf{x}_1 \mid \mathbf{x}_2]$.

For inductive case, suppose $\mathtt{X}_1[\mathbf{x}_1] \equiv \mathtt{X}_2[\mathbf{x}_2]$. We apply the $\equiv$ rule once to $\mathtt{X}_1[\mathbf{x}_1]$.

Then $\mathtt{X}_1[\mathbf{x}_1] \equiv \mathtt{X}_1[\mathbf{x}_3 + \mathbf{x}_4]$ with $\mathbf{x}_1 = \mathbf{x}_3 + \mathbf{x}_4 \in \mathbf{G}$ or $\mathbf{x}_3 + \mathbf{x}_4 = \mathbf{x}_1 \in \mathbf{G}$. In this case, $\mathtt{X}_1 = \mathbf{x}_2 \mid \mathtt{X}''$ for some $\mathtt{X}''$.

It is obvious that there is no case such that $\mathtt{X} \equiv \mathtt{X}'[\mathbf{x}_1'' + \mathbf{x}_2'']$ with $\mathbf{x}_1'' <_{\widetilde{G}}^{+} \mathbf{x}_1$ and $\mathbf{x}_2'' <_{\widetilde{G}}^{+} \mathbf{x}_2$ because if so, only one of t_i can be executed from s, which contradicts to the assumption.

If we apply the fifth rule to $\mathbf{x}_1 \mid \mathbf{x}_2$ in $\mathtt{X}'[\mathbf{x}_1 \mid \mathbf{x}_2]$ to obtain $\mathtt{X}'[\mathbf{x}']$, then we apply back to obtain $\mathtt{X}'[\mathbf{x}_1 \mid \mathbf{x}_2]$. If we apply other rules to $\mathtt{X}_1[\mathbf{x}_1]$, then obviously $\mathtt{X}_1 = \mathbf{x}_2 \mid \mathtt{X}''$. Applying this repeatedly, we can check if $\mathtt{X}[\mathbf{x}_1 \mid \mathbf{x}_2] \equiv \mathtt{X}_1'[\mathbf{x}_1' \mid \mathbf{x}_2]$ and $\mathtt{X}[\mathbf{x}_1 \mid \mathbf{x}_2] \equiv \mathtt{X}_2'[\mathbf{x}_1 \mid \mathbf{x}_2']$, then there exits $\mathtt{X}'$ such that $\mathtt{X}[\mathbf{x}_1 \mid \mathbf{x}_2] \equiv \mathtt{X}'[\mathbf{x}_1' \mid \mathbf{x}_2']$; or $\mathtt{X}'[\mathbf{x}']$ such that $\mathtt{X}[\mathbf{x}_1 \mid \mathbf{x}_2] \equiv \mathtt{X}'[\mathbf{x}_1' \mid \mathbf{x}_2']$ with $\mathbf{x}_1' \mid \mathbf{x}_2' = \mathbf{x}' \in \mathbf{G}$.

The proof for (2) is similar to the one for (1).

Lemma C.3 (Diamond Property). *Assume* $\mathbf{G}$ *is well-formed. Suppose* $s \in RS(\mathcal{A}(\mathbf{G}))$ *and* $s \xrightarrow{t_1} s_1$ *and* $s \xrightarrow{t_2} s_2$ *where (1)* t_1 *and* t_2 *are both inputs; or (2)* t_1 *is an output and* t_2 *is an input, then there exists* s' *such that* $s_1 \xrightarrow{t_1'} s'$ *and* $s_2 \xrightarrow{t_2'} s'$ *where* $\ell(t_1) = \ell(t_2')$ *and* $\ell(t_2) = \ell(t_1')$.

Proof. (1) Suppose $t_1 = \mathtt{p}_1\mathtt{q}_1?l_1$ and $t_2 = \mathtt{p}_2\mathtt{q}_2?l_2$. By linearity, $l_1 \neq l_2$. By definition of a receiving transition, we have $\mathtt{p}_1\mathtt{q}_1 \neq \mathtt{p}_2\mathtt{q}_2$ (to allow both receptions at the same time).

If $\mathtt{q}_1 \neq \mathtt{q}_2$, then s contains local states for $\mathtt{q}_1$ and $\mathtt{q}_2$ that each allow a reception. Since t_1 does not modify $\mathtt{q}_2$'s state (and vice-versa), the property holds.

If $\mathtt{q}_1 = \mathtt{q}_2 = \mathtt{q}$, let $\mathtt{X}$ be the state of $\mathtt{q}$ in s. Let $\mathbf{x}_1 = \mathtt{p}_1 \to \mathtt{q} : l_1\langle U_1\rangle.\mathbf{x}_1'$ and $\mathbf{x}_2 = \mathtt{p}_2 \to \mathtt{q} : l_2\langle U_2\rangle.\mathbf{x}_2'$ be the relevant global transitions in $\mathbf{G}$. We know that $\mathtt{X} \equiv \mathtt{X}_1[\mathbf{x}_1] \equiv \mathtt{X}_2[\mathbf{x}_2]$. By Lemma C.2(1), we know that $\mathtt{X} \equiv \mathtt{X}_3[\mathbf{x}_1 \mid \mathbf{x}_2]$. Hence the diamond property holds by definition of $\mathcal{A}(\mathbf{G})$.

Concerning (2), we suppose $t_1 = \mathtt{p}_1\mathtt{q}_1?l_1$ and $t_2 = \mathtt{p}_2\mathtt{q}_2!l_2$.

If $\mathtt{q}_1 \neq \mathtt{p}_2$, then s contains local states for $\mathtt{q}_1$ and $\mathtt{p}_2$ that allow a reception and a send, respectively. Since t_2 does not modify $\mathtt{q}_2$'s state (and vice-versa), the diamond property holds.

If $\mathtt{q}_1 = \mathtt{p}_2 = \mathtt{q}$, let $\mathtt{X}$ be the state of $\mathtt{q}$ in s. Let $\mathbf{x}_1 = \mathtt{p}_1 \to \mathtt{q} : l_1\langle U\rangle.\mathbf{x}_1'$ and $\mathbf{x}_2 = \mathtt{q} \to \mathtt{q}_2 : l_2\langle U\rangle.\mathbf{x}_2'$ be the relevant global transitions in $\mathbf{G}$. We know that $\mathtt{X} \equiv \mathtt{X}_1[\mathbf{x}_1] \equiv \mathtt{X}_2[\mathbf{x}_2]$. Then by Lemma C.2(2), we have $\mathtt{X} \equiv \mathtt{X}_3[\mathbf{x}_1 \mid \mathbf{x}_2]$. Hence the diamond property holds by definition of $\mathcal{A}(\mathbf{G})$.

Proposition C.6 (Stable-Outputs Decomposition). *Assume* $\mathbf{G}$ *is well-formed and* $s \in RS(\mathcal{A}(\mathbf{G}))$. *Then there exists* $s_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_n} s$ *where each* φ_i *is stable-outputs, i.e.* s *is reachable by an SOS execution.*

Proof. By induction on the number of transitions leading to the state s.

If $s = s_0$, we conclude trivially. If the last transition to s is a send, then we can conclude by induction. We therefore examine the only interesting case: the last transition is of the form $t = (s_n, \mathtt{pp}'?l, s)$. The corresponding global type transition is $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U\rangle.\mathbf{x}' \in \mathbf{G}$.

We call φ an SOS execution from s_0 to s_n such that the sending action, of the form $t' = (s_i, \mathtt{pp}'!l, s_{i+1})$, corresponding to t is the closest to s_n. We are thus in the following situation (with φ_0 SOS, and a minimal φ_1):

$$s_0 \xrightarrow{\varphi_0} s_i \xrightarrow{t'} s_{i+1} \xrightarrow{\varphi_1} s_n \xrightarrow{t} s$$

assume φ_0 to be stable-outputs.

By minimality of φ_1, we deduce that $t'\varphi_1$ is an SOS causal chain (meaning that each action is either causally related to a previous one, or is a sending action preceding a reception). Since $t \notin \varphi_1$, the SOS causality chain φ_1 only involves transitions that correspond to global type transitions that are successors of $\mathbf{x}$. Therefore, we know that φ_1 does not contain any action by $\mathtt{p}'$ and thus that t is not causally related to any transition in φ_1. By Lemma C.1, our reordered situation is the following:

$$s_0 \xrightarrow{\varphi_0} s_i \xrightarrow{t'} s_{i+1} \xrightarrow{t} s'_{i+1} \xrightarrow{\varphi_1} s$$

Since $\varphi_0 t' \varphi_1$ was SOS, $\varphi_0 t' t \varphi_1$ is SOS.

Proposition C.7 (Reception Error Freedom). *If the global type* $\mathbf{G}$ *is well-formed, then* $\mathcal{A}(\mathbf{G})$ *is reception-error-free.*

Proof. By definition of $\mathcal{A}(\mathbf{G})$, only transitions with correct labels can exist.

Proposition C.8 (Stable Configurations). *Suppose* $s_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_n} s$ *with* φ_i *stable-outputs. Then there exists an execution* $\xrightarrow{\varphi'}$ *such that* $s \xrightarrow{\varphi'} s_3$ *and* s_3 *is stable, and there is a 1-buffer execution* $s_0 \xrightarrow{\varphi''} s_3$.

Proof. We proceed by induction on the total number of messages that are in the buffers in reachable configuration s and on the size of the non-globally ordered suffix of an execution to reach s.

By Proposition C.6, we have the existence of SOS executions reaching s from s_0. We take the one with the maximal (i.e. longest) globally ordered prefix. If this prefix reaches s, we are done. Otherwise we are in the following situation:

$$s_0 \xrightarrow{\varphi_0} s_i \xrightarrow{t} s_{i+1} \xrightarrow{\varphi_1} s$$

with φ_0 maximal, globally ordered and 1-buffer, and s_i stable.

By definition of an SOS and globally ordered execution, t is of the form $\mathtt{pq}!l$,with $\mathbf{x} = \mathtt{p} \to \mathtt{q} : l\langle U \rangle; \mathbf{x}' \in \widetilde{G}$. Since φ_0 is maximal, we know that $\mathbf{x}$ is not the consecutive transition of any in φ_0. It implies that there exists a chain of consecutive transitions $\mathbf{x}_1, \ldots, \mathbf{x}_k$ from some point in φ_0, with $\mathbf{x}_k \prec_{\widetilde{G}}^{\sim} \mathbf{x}$. By definition of $\mathcal{A}(\mathbf{G})$, the alternation of send and receive transitions from $\mathbf{x}_1, \ldots, \mathbf{x}_k$, written φ^*, are executable from s_i. These transitions φ^* are such that $\varphi_0 \varphi^*$ is globally ordered and that no transition of φ^* are in φ_1. It is therefore possible to apply Lemma C.3 in the following way.

$$s_0 \xrightarrow{\varphi_0} s_i \xrightarrow{t} s_{i+1} \xrightarrow{\varphi_1} s \xrightarrow{\varphi^*} s^* \quad \text{and} \quad s_0 \xrightarrow{\varphi_0} s_i \xrightarrow{\varphi^*} \xrightarrow{t} s'_{i+1} \xrightarrow{\varphi_1} s^*$$

There are now two cases:

- If t is immediately followed in φ_1 by the corresponding $t' = \mathtt{pq}?l$, i.e. $\varphi_1 = t'\varphi_2$, we use the induction hypothesis on s^* defined by $s_0 \xrightarrow{\varphi_0} s_i \xrightarrow{\varphi^*} \xrightarrow{t} s'_{i+1} \xrightarrow{t'} \xrightarrow{\varphi_2} s^*$, which is an SOS execution with the same buffer size as the one for s, but with a smaller non-globally ordered suffix (a suffix of φ_2). We therefore get the existence of φ' such that $s^* \xrightarrow{\varphi'} s''$ with s'' stable. For s, the execution leading to s'' is thus $s \xrightarrow{\varphi^*} s^* \xrightarrow{\varphi'} s''$.

– If φ_1 does not contain the transition $t' = \mathtt{pq}?l$, it is possible to execute it from state s^* and state s'_{i+1}, and it commutes with φ_1. We thus have:

$$s_0 \xrightarrow{\varphi_0} s_i \xrightarrow{t} s_{i+1} \xrightarrow{\varphi_1} s \xrightarrow{\varphi^*} s^* \xrightarrow{t'} s' \quad \text{and} \quad s_0 \xrightarrow{\varphi_0} s_i \xrightarrow{\varphi^*} \xrightarrow{t} s'_{i+1} \xrightarrow{t'} s' \xrightarrow{\varphi_1} s'$$

The state s' is thus accessible by an SOS execution and its buffer usage is inferior to s's buffer usage. We can thus apply the induction hypothesis and get the existence of φ' such that $s' \xrightarrow{\varphi'} s''$ with s'' stable. For s, the execution leading to s'' is thus $s \xrightarrow{\varphi^*} s^* \xrightarrow{t'} s' \xrightarrow{\varphi'} s''$.

Proposition C.9 (Deadlock-Freedom). *If* $\mathbf{G}$ *is well-formed, then* $\mathcal{A}(\mathbf{G})$ *is deadlock-free.*

Proof. By (reception error freedom) and (correct ending), together with (stable-output decomposition), we only have to check, there is no input is waiting with an empty queue forever. Suppose by contradiction, there is $s \in RS(\mathcal{A}(\mathbf{G}))$ such that $s = (\vec{q}; \vec{\varepsilon})$ and there exists input state $q_\mathtt{p} \in \vec{q}$ and no output transition from q_k such that $k \neq \mathtt{q}$.

Then by the shape of global types, there exists at least one $G = \mathbf{x} = \mathtt{q} \to \mathtt{p} : l\langle U \rangle.\mathbf{x}' \in \widetilde{G}$ for some $\mathtt{q}$. By well-formedness of $\mathbf{G}$, there is a 1-buffer execution φ which corresponds to this G.

Since φ is not taken (if so, by (linearity), $q_\mathtt{p}$ can perform an input), then there is another execution φ' such that it leads to state s which is deadlock at $q_\mathtt{p}$.

Case (1) Suppose φ does not include input actions at $\mathtt{q}$ except l, i.e. l is the first input action at $\mathtt{q}$ in φ. We let φ_0 for the prefix before the actions of $\mathtt{qp}!l \cdot \mathtt{qp}?l$.

By (choice awareness), we know $\mathtt{p} \in Rcv(\varphi')$.

By the assumption, the corresponding input action has a different label from l, i.e. $\mathtt{q}'\mathtt{p}?l' \in \varphi'$. By the same argument of the proofs in Lemma C.3 and Proposition C.6, $\mathtt{q}'\mathtt{p}?l'$ and $\mathtt{qp}?l$ are originated from the parallel composition (i.e. $t_1 \bowtie t_2$ with $\mathtt{q}'\mathtt{p}?l' = \ell(t_1)$ and $\mathtt{qp}?l = \ell(t_2)$). Hence the both corresponding outputs $\mathtt{q}'\mathtt{p}!l'$ and $\mathtt{qp}!l$ can be always fired if one of them is (since they are also originated from the parallel composition by definition of $\mathcal{A}(\mathbf{G})$). This contradicts the assumption that $q_\mathtt{p}$ is deadlock with label l.

Case (2) Suppose φ includes other input actions at $\mathtt{q}$ before $\mathtt{qp}?l$, i.e. $\mathtt{p} \in Rcv(\varphi_0)$. Let $\mathtt{q}'\mathtt{p}?l'$ the action which first occurs in φ_0. By $\mathtt{p} \in Rcv(\varphi')$, there exists $\mathtt{q}''\mathtt{p}?l'' \in \varphi'$. If $\mathtt{q}''\mathtt{p}?l'' \neq \mathtt{q}'\mathtt{p}?l'$, by the same reasoning as (1), the both corresponding outputs are available. Hence we assume the case $\mathtt{q}''\mathtt{p}?l'' = \mathtt{q}'\mathtt{p}?l'$. Let s is the first state from which a transition in φ_0 and a transition in φ' are separated. Then by assumption, if $s \xrightarrow{\varphi_0 \cdot \mathtt{q}'\mathtt{p}!l' \cdot \mathtt{q}'\mathtt{p}?l'} s_1$ and $s \xrightarrow{\varphi_1 \cdot \mathtt{q}'\mathtt{p}!l' \cdot \mathtt{q}'\mathtt{p}?l'} s_2$, by assumption $l' \notin \varphi_0 \cup \varphi_1$, hence $s \xrightarrow{\mathtt{q}'\mathtt{p}!l' \cdot \mathtt{q}'\mathtt{p}?l'} s'_1 \xrightarrow{\varphi'_0} s_1$ and $s \xrightarrow{\mathtt{q}'\mathtt{p}!l' \cdot \mathtt{q}'\mathtt{p}?l'} s'_2 \xrightarrow{\varphi'_1} s_2$ by Lemma C.1. Since s_1 can perform an input at $\mathtt{q}$ by the assumption (because of $\mathtt{qp}?l$), φ'_1 should contain an input at $\mathtt{q}$ by the choice awareness. If it contains the input to $\mathtt{q}$ in φ'_1, then we repeat Case (2); else we use Case (1) to lead the contradiction; otherwise if it contains the same input as $\mathtt{qp}?l$, then it contradicts the assumption that $q_\mathtt{p}$ is deadlock.

Proposition C.10 (Strong Boundedness). *If* $\mathbf{G}$ *is well-formed and all cycles in* $\mathcal{A}(\mathbf{G})$ *has an IO-causality between uses of the same channel, then* $\mathbf{G}$ *is strongly bounded.*

Proof. We follow exactly [10, § 3].

Proposition C.11 (Progress). *If* $\mathbf{G}$ *is well-formed, then* $\mathcal{A}(\mathbf{G})$ *satisfies the progress property.*

Proof. By Propositions C.5, C.7 and C.9.

Proposition C.12 (Liveness). *If* $\mathbf{G}$ *is well-formed and* $\mathbf{x} = \mathsf{end} \in \mathbf{G}$*, then* $\mathcal{A}(\mathbf{G})$ *satisfies the liveness property.*

Proof. By Propositions C.11 and C.5.

C.6 Equivalence between Global State Automata and MSA

We now state that the collection of the local state automata obtained after projection from a global type is actually the same automaton as the global state automaton that can be defined directly.

Proposition C.13 (Automaton Equivalence). *For any well-formed global type* $\mathbf{G}$*, the MSA* $\mathcal{A}(\mathbf{G})$ *is isomorphic to the collection* $(\mathcal{A}(\mathbf{G} \upharpoonright \mathtt{p}_1), \ldots, \mathcal{A}(\mathbf{G} \upharpoonright \mathtt{p}_p))$.

Proof. The sets of states, channels and alphabets are identical.

The state equivalences are identical as well, since projection for $\mathtt{p}''$ of $\mathbf{x} = \mathtt{p} \to \mathtt{p}' : l\langle U \rangle; \mathbf{x}'$ gives $\mathbf{x} = \mathbf{x}'$ and the state equivalences for global state automata and MSAs are the same in this case. Thus, the equivalence relation for states for $\mathcal{A}(\mathbf{G} \upharpoonright \mathtt{p})$ is the same as the one that can be proved under $\mathtt{p}$ in $\mathcal{A}(\mathbf{G})$.

The transitions are also identical by definition.

C.7 Proofs for Properties of MSA

We use Proposition C.13.

Proofs for Lemma 3.1

1. (Determinism) By Proposition C.3 and Proposition C.13.
2. (Choice Awareness) By Proposition C.4 and Proposition C.13.
3. (Diamond Property) By Lemma C.3 and Proposition C.13.
4. (Stable-Outputs Decomposition) By Proposition C.6 and Proposition C.13.
5. (Stable Configurations) By Proposition C.8 and Proposition C.13.

Proofs for Theorem 3.1 By Proposition C.9 and Proposition C.13.

Proofs for Theorem 3.2 By Proposition C.10 and Proposition C.13.

Proofs for Theorem 3.3 By Proposition C.11, Proposition C.12 and Proposition C.13.

D Appendix for Section 4

D.1 Additional Process Examples

Broker B in the Trade Example

$$\begin{array}{rl}
\mathbf{P}_{\mathtt{B}} = \mathsf{def} & \mathbf{x}(x,t,o) = x[\mathtt{B}](z).\mathbf{x}_0(t,o,z) \\
& \mathbf{x}_0(t,o,z) = z?\langle \mathtt{B}, \mathit{Item}(x)\rangle.\mathbf{x}_1(t,o,0,z) \\
& \mathbf{x}_5(t,o,i,z) + \mathbf{x}_1(t,o,i,z) = \mathbf{x}_2(t,o,i,z) \\
& \mathbf{x}_2(t,o,i,z) = \mathtt{if}\ t < i\ \mathtt{then}\ \mathbf{x}_3(t,o,z) \\
& \qquad\qquad\qquad\qquad \mathtt{else}\ \mathbf{x}_6(i,z) \\
& \mathbf{x}_3(t,o,z) = z!\langle \mathtt{C}, \mathit{Offer}\langle o\rangle\rangle.\mathbf{x}_4(t,o+5,z) \\
& \mathbf{x}_4(t,o,z) = z?\langle \mathtt{C}, \mathit{Counter}(i)\rangle.\mathbf{x}_5(t,o,i,z) \\
& \mathbf{x}_6(i,z) = \mathbf{x}_7(i,z) \mid \mathbf{x}_8(i,z) \\
& \mathbf{x}_7(i,z) = z!\langle \mathtt{S}, \mathit{Final}\langle i\rangle\rangle.\mathbf{x}_9(z) \\
& \mathbf{x}_8(i,z) = z!\langle \mathtt{C}, \mathit{Result}\langle i\rangle\rangle.\mathbf{x}_{10}(z) \\
& \mathbf{x}_9(z) \mid \mathbf{x}_{10}(z) = \mathbf{x}_{11}(z) \\
& \mathbf{x}_{11}(z) = \mathsf{end} \qquad \mathsf{in}\ \mathbf{x}(a,42,0)
\end{array}$$

New Name Creation The following $\mathbf{P}_1$ corresponds to $\mu X.x[\mathtt{p}](y).(\nu a)y!\langle \mathtt{p}, l\langle a\rangle\rangle.X$ where $\mu X.P$ represents the standard recursion. This agent sends a fresh name whenever asked.

$$\begin{array}{rl}
\mathbf{P}_1 = \mathsf{def}\ \mathbf{x}_0(x) = x[\mathtt{p}](y).\mathbf{x}_1(x,y) \\
\mathbf{x}_1(x,y) = (\nu a)\mathbf{x}_2(x,y,a) \\
\mathbf{x}_2(x,y,z) = y!\langle \mathtt{p}, l\langle z\rangle\rangle.\mathbf{x}_0(x)\ \mathsf{in}\ \mathbf{x}_0(b)
\end{array}$$

The following $\mathbf{P}_2$ represents $(\nu a)\mu X.x[\mathtt{p}](y).y!\langle \mathtt{p}, l\langle a\rangle\rangle.X$ which sends the same name each time.

$$\begin{array}{rl}
\mathbf{P}_2 = (\nu a)\mathsf{def}\ \mathbf{x}_0(x,z) = z[\mathtt{p}](y).\mathbf{x}_1(x,z,y) \\
\mathbf{x}_1(x,z,y) = y!\langle \mathtt{p}, l\langle x\rangle\rangle.\mathbf{x}_0(x,z)\ \mathsf{in}\ \mathbf{x}_0(a,b)
\end{array}$$

D.2 Structure Congruence Rules

We define structural congruence for process states, processes and networks which are defined in figure 22 (we omit the queue as it is the same as in [3]).

D.3 Omitted Operational Semantics

We list the omitted operational semantics in figure 23.

Fig. 22. Structural Congruence for Processes and Networks

$$\mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{y}) \mid \mathbf{x}''(\tilde{z}) \vdash \mathbf{x}(\tilde{v}) \equiv \mathbf{x}'(\tilde{y}[\tilde{v}/\tilde{x}]) \mid \mathbf{x}''(\tilde{z}[\tilde{v}/\tilde{x}])$$
$$\mathbf{x}(\tilde{y}) \mid \mathbf{x}'(\tilde{z}) = \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}(\tilde{y}[\tilde{v}/\tilde{x}]) \mid \mathbf{x}'(\tilde{z}[\tilde{v}/\tilde{x}]) \equiv \mathbf{x}''(\tilde{v})$$
$$\mathbf{x}(\tilde{x}) + \mathbf{x}'(\tilde{x}) = \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}(\tilde{v}) \equiv \mathbf{x}''(\tilde{v})$$
$$\mathbf{x}(\tilde{x}) + \mathbf{x}'(\tilde{x}) = \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}'(\tilde{v}) \equiv \mathbf{x}''(\tilde{v})$$
$$\mathbf{x}(\tilde{x}) = \mathbf{0} \vdash \mathbf{x}(\tilde{v}) \equiv \mathbf{0}$$
$$\widetilde{P} \vdash \mathbf{X} \mid (\nu a)\mathbf{X}' \equiv (\nu a)(\mathbf{X} \mid \mathbf{X}') \quad (a \notin \mathbf{X})$$
$$\widetilde{P} \vdash \mathbf{X} \mid \mathbf{X}' \equiv \mathbf{X}' \mid \mathbf{X}$$
$$\frac{\widetilde{P} \vdash \mathbf{X} \equiv \mathbf{X}''}{\widetilde{P} \vdash \mathbf{X} \mid \mathbf{X}' \equiv \mathbf{X}'' \mid \mathbf{X}'} \qquad \frac{\widetilde{P}' \vdash \mathbf{X} \equiv \mathbf{X}'}{\widetilde{P}, \widetilde{P}', \widetilde{P}'' \vdash \mathbf{X} \equiv \mathbf{X}'}$$
$$\mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0} \qquad \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ (\nu a)\ \mathbf{X} \equiv (\nu a)\ \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X}$$
$$\mathbf{N} \,||\, \mathbf{0} \equiv \mathbf{N} \qquad \mathbf{N} \,||\, \mathbf{N}' \equiv \mathbf{N}' \,||\, \mathbf{N} \qquad \mathbf{N} \,||\, (\mathbf{N}' \,||\, \mathbf{N}'') \equiv (\mathbf{N} \,||\, \mathbf{N}') \,||\, \mathbf{N}''$$
$$\mathbf{N} \,||\, (\nu n)\mathbf{N}' \equiv (\nu n)(\mathbf{N} \,||\, \mathbf{N}') \quad (n \notin \mathbf{N}) \qquad (\nu n)\ \mathbf{0} \equiv \mathbf{0} \qquad (\nu s)(s : h) \equiv \mathbf{0}$$

E Appendix for Section 5

E.1 Operators in figure 13

We give the full definition of operators used in figure 13. We then give the detailed explanation for typing rules in figure 13.

Assume $\mathbf{T}_i = \mathsf{def}\ \widetilde{T_i}\ \mathsf{in}\ \mathbf{x}_i$ $(i = 1, 2)$ and $\mathbf{T} = \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}'$.

1. $\mathbf{T} \uplus \mathbf{x} = \mathbf{x}' \stackrel{\text{def}}{=} \mathsf{def}\ \mathbf{x} = \mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x}$
2. $\mathbf{T}' \uplus \mathbf{x} =\!! \langle \mathrm{p}, l\langle U \rangle \rangle.\mathbf{x}' \stackrel{\text{def}}{=} \mathsf{def}\ \mathbf{x} = \mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x}$
3. $\mathbf{T}'_i \uplus \mathbf{x} =? \langle \mathrm{p}, l\langle U \rangle \rangle.\mathbf{x}' \stackrel{\text{def}}{=} \mathsf{def}\ \mathbf{x} =? \langle \mathrm{p}, l\langle U \rangle \rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x}$
4. $(\mathbf{T}_1 \cup \mathbf{T}_2) \uplus \mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2 \stackrel{\text{def}}{=} \mathsf{def}\ \mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2, \widetilde{T_1} \cup \widetilde{T_2}\ \mathsf{in}\ \mathbf{x}$
5. $(\mathbf{T}_1 \oplus \mathbf{T}_2) \uplus \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2 \stackrel{\text{def}}{=} \mathsf{def}\ \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2, \widetilde{T_1} \cup \widetilde{T_2}\ \mathsf{in}\ \mathbf{x}$
6. $(\mathbf{T}_1 \& \mathbf{T}_2) \uplus \mathbf{x} = \mathbf{x}_1 \& \mathbf{x}_2 \stackrel{\text{def}}{=} \mathsf{def}\ \mathbf{x} = \mathbf{x}_1 \& \mathbf{x}_2, \widetilde{T_1} \cup \widetilde{T_2}\ \mathsf{in}\ \mathbf{x}$
7. $\mathbf{T}_i \uplus \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x} \stackrel{\text{def}}{=} \mathsf{def}\ \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_i$
8. $\mathbf{T}_i \uplus \mathbf{x}_1 | \mathbf{x}_2 = \mathbf{x}' \stackrel{\text{def}}{=} \mathsf{def}\ \mathbf{x}_1 | \mathbf{x}_2 = \mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x}_i$

E.2 Omitted Rules from Fig. 13

We list the omitted typing rules in figure 24.

E.3 Explanations of Typing Rules

1. The rules for expressions ([TRUE, NAMES, NOT]) are standard.
2. Rule [INIT] types the initialisation where $\tilde{y}$ is variables for sorts, while $\tilde{z}$ are variables for session types. $\tilde{y}$ should cover x and variables in $\tilde{e}$ appeared in the right hand side. State variables $\mathbf{x}$ and $\mathbf{x}'$ are assigned by corresponding type where z_i has type $\mathbf{T} \uplus \mathbf{x} = \mathbf{x}'$, which means that we record $\mathbf{x} = \mathbf{x}'$ at the head of $\mathbf{T}$. This is essentially the same as usual session type.

Fig. 23. Operational Semantics (remaining rules)

$$\frac{e[\tilde{v}/\tilde{x}]\downarrow \mathsf{false} \qquad \tilde{e}''[\tilde{v}/\tilde{x}]\downarrow \tilde{v}''}{\mathbf{x}(\tilde{x}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}'(\tilde{x})\ \mathtt{else}\ \mathbf{x}''(\tilde{x}) \vdash \mathbf{x}(\tilde{v}) \xrightarrow{\tau} \mathbf{x}''(\tilde{v}'')}\,[\textsc{If}\text{F}] \qquad \frac{\widetilde{P} \vdash \mathbf{X}_0 \xrightarrow{\alpha} \mathbf{X}_0'}{\widetilde{P} \vdash \mathbf{X}_0 \mid \mathbf{X} \xrightarrow{\alpha} \mathbf{X}_0' \mid \mathbf{X}}\,[\textsc{Par}]$$

$$\frac{\widetilde{P}' \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'}{\widetilde{P}, \widetilde{P}', \widetilde{P}'' \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'}\,[\textsc{Weak}] \qquad \frac{\widetilde{P} \vdash \mathbf{X} \equiv \mathbf{X}_0 \xrightarrow{\alpha} \mathbf{X}_0' \equiv \mathbf{X}'}{\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'}\,[\textsc{Str}] \qquad \frac{\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}' \quad a \notin \alpha}{\widetilde{P} \vdash (\nu a)\mathbf{X} \xrightarrow{\alpha} (\nu a)\mathbf{X}'}\,[\textsc{Res}]$$

$$\frac{\mathbf{N} \to \mathbf{N}'}{(\nu n)\,\mathbf{N} \to (\nu n)\,\mathbf{N}'}\,[\textsc{Res}_N] \qquad \frac{\mathbf{N} \to \mathbf{N}'}{\mathbf{N} \mid\mid \mathbf{N}'' \to \mathbf{N}' \mid\mid \mathbf{N}''}\,[\textsc{Par}_N] \qquad \frac{\mathbf{N} \equiv \mathbf{N}_0 \to \mathbf{N}_0' \equiv \mathbf{N}'}{\mathbf{N} \to \mathbf{N}'}\,[\textsc{Str}_N]$$

3. Rule [Req] is similar except we record introduced session type $\mathbf{T} = \mathbf{G} \upharpoonright \mathtt{p}$ in $\mathbf{x}'$ in the right side.
4. Rule [Send] records the send type for z_i ($T_i = \mathbf{x} = !\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}'$) and $\mathbf{x} = \mathbf{x}'$ for all other sessions.
5. Rule [Recv] is its symmetric rule. It records $\mathbf{T}_i = \mathbf{T}_i' \uplus \mathbf{x} = ?\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}'$ for z_i and $\mathbf{x} = \mathbf{x}'$ for all other sessions.
6. Rule [Deleg] types the delegation (sending a session name) where we record the type $\mathbf{T}$ of the sent session name y as the argument of $\mathbf{x}$ (as the standard rule [14]).
7. Rule [Catch] is its symmetric rule, recording $\mathbf{T}$ for y as the final argument $\mathbf{x}'$.
8. Rule [Par] introduces the parallel composition. The operation $(\mathbf{T}_1 \cup \mathbf{T}_2) \uplus \mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2$ means we record $\mathbf{x} = \mathbf{x}_1 \mid \mathbf{x}_2$ at the head of a union of $\mathbf{T}_1$ and $\mathbf{T}_1$. Note that $\tilde{y}$ covers a union of $\tilde{y}_1$ and $\tilde{y}_2$ and session names $\tilde{z}$ are common so that the substitution is always defined.
9. Rule [Join] is its symmetric rule.
10. Rule [If] introduces the internal choice. We note that variables in e and $\tilde{e}$ are covered by $\tilde{y}$.
11. Rule [Choice] types the external choice.
12. Rule [Merge] merges two types.
13. In rule [Res], we record type $\langle \mathbf{G}\rangle$ for a (the last argument of $\mathbf{x}'$).

E.4 Typing Run-Time Processes

This subsection defines the typing systems for runtime processes. We use the following abbreviations for message types.

$$\begin{array}{rcl} \mathbf{T} & ::= & \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathtt{X} \\ \mathtt{T} & ::= & !\langle \mathtt{p}_1, l_1\langle U_1\rangle\rangle; !\langle \mathtt{p}_2, l_2\langle U_2\rangle\rangle; \cdots !\langle \mathtt{p}_n, l_n\langle U_n\rangle\rangle \end{array}$$

Type $\mathtt{T}$ is called *message type* which contains single $x = \varepsilon$ (instead of $x = \mathsf{end}$) and others are sequence of outputs which represents a sequence of messages stored in a queue [3].

$$\begin{array}{l} \Delta \mid \Delta' = \Delta \backslash \mathsf{dom}(\Delta') \cup \Delta' \backslash \mathsf{dom}(\Delta) \\ \qquad\qquad \cup \{c : \Delta(c) \mid \Delta'(c) \mid c \in \mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta')\} \end{array}$$

Fig. 24. Typing System for Initial State Processes (remaining rules)

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\,[\textsc{True}] \qquad \frac{u : U \in \Gamma}{\Gamma \vdash u : U}\,[\textsc{Name}] \qquad \frac{\Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash \mathsf{not}(e) : \mathsf{bool}}\,[\textsc{Not}]$$

$$\frac{\tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}' \quad \mathbf{T}_i = \mathbf{T}'_i \uplus \mathbf{x} = !\langle \mathtt{p}, l\langle \mathbf{T}\rangle\rangle.\mathbf{x}' \quad \forall j \neq i, \mathbf{T}_j = \mathbf{T}'_j \uplus \mathbf{x} = \mathbf{x}'}{\vdash \mathbf{x}(\tilde{y}\tilde{z}y) = z_i\,!\langle \mathtt{p}, l\langle y\rangle\rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \rhd \mathbf{x} : \tilde{U}\,\widetilde{\mathbf{T}}\mathbf{T} \;[\!]\; \mathbf{x}' : \tilde{U}'\,\widetilde{\mathbf{T}}'}\,[\textsc{Deleg}]$$

$$\frac{\tilde{y} : \tilde{U}, y : U \vdash \tilde{e} : \tilde{U}' \quad \mathbf{T}_i = \mathbf{T}'_i \uplus \mathbf{x} = ?\langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}' \quad \forall j \neq i, \mathbf{T}_j = \mathbf{T}'_j \uplus \mathbf{x} = \mathbf{x}'}{\vdash \mathbf{x}(\tilde{y}\tilde{z}) = z_i\,?\langle \mathtt{p}, l(y)\rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \rhd \mathbf{x} : \tilde{U}\,\widetilde{\mathbf{T}} \;[\!]\; \mathbf{x}' : \tilde{U}'\,\widetilde{\mathbf{T}}'}\,[\textsc{Recv}]$$

$$\frac{\tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}' \quad \mathbf{T}_i = \mathbf{T}'_i \uplus \mathbf{x} = ?\langle \mathtt{p}, l\langle \mathbf{T}\rangle\rangle.\mathbf{x}' \quad \forall j \neq i, \mathbf{T}_j = \mathbf{T}'_j \uplus \mathbf{x} = \mathbf{x}'}{\vdash \mathbf{x}(\tilde{y}\tilde{z}) = z_i\,?\langle \mathtt{p}, l\langle \mathbf{T}\rangle\rangle.\mathbf{x}'(\tilde{e}\tilde{z}y) \rhd \mathbf{x} : \tilde{U}\,\widetilde{\mathbf{T}} \;[\!]\; \mathbf{x}' : \tilde{U}'\,\widetilde{\mathbf{T}}'\mathbf{T}}\,[\textsc{Catch}]$$

$$\frac{\tilde{y} : \tilde{U} \vdash e : U \quad \tilde{y} : \tilde{U} \vdash \tilde{e}_1 : \tilde{U}_1 \quad \tilde{y} : \tilde{U} \vdash \tilde{e}_2 : \tilde{U}_2 \quad \forall i, \mathbf{T}_i = (\mathbf{T}_{1i} \cup \mathbf{T}_{2i}) \uplus \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2}{\vdash \mathbf{x}(\tilde{y}\tilde{z}) = \mathtt{if}\ e\ \mathtt{then}\ \mathbf{x}_1(\tilde{e}_1\tilde{z})\ \mathtt{else}\ \mathbf{x}_2(\tilde{e}_2\tilde{z}) \rhd \mathbf{x} : \tilde{U}\,\widetilde{\mathbf{T}} \;[\!]\; \mathbf{x}_1 : \tilde{U}_1\,\widetilde{\mathbf{T}}_1, \mathbf{x}_2 : \tilde{U}_2\,\widetilde{\mathbf{T}}_2}\,[\textsc{If}]$$

$$\frac{\forall i, \mathbf{T}_i = (\mathbf{T}_{1i} \cup \mathbf{T}_{2i}) \uplus \mathbf{x} = \mathbf{x}_1 \,\&\, \mathbf{x}_2}{\vdash \mathbf{x}(\tilde{y}\tilde{z}) = \mathbf{x}_1(\tilde{y}\tilde{z}) \,\&\, \mathbf{x}_2(\tilde{y}\tilde{z}) \rhd \mathbf{x} : \tilde{U}\,\widetilde{\mathbf{T}} \;[\!]\; \mathbf{x}_1 : \tilde{U}\,\widetilde{\mathbf{T}}_1, \mathbf{x}_2 : \tilde{U}\,\widetilde{\mathbf{T}}_2}\,[\textsc{Choice}]$$

$$\frac{\forall i, \mathbf{T}_i = \mathbf{T}'_i \uplus \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}}{\vdash \mathbf{x}_1(\tilde{y}\tilde{z}) + \mathbf{x}_2(\tilde{y}\tilde{z}) = \mathbf{x}(\tilde{y}\tilde{z}) \rhd \;\mathbf{x}_1 : \tilde{U}\,\widetilde{\mathbf{T}}, \mathbf{x}_2 : \tilde{U}\,\widetilde{\mathbf{T}} \;[\!]\; \mathbf{x} : \tilde{U}\,\widetilde{\mathbf{T}}'}\,[\textsc{Merge}]$$

$$\frac{\forall i, \widetilde{\mathbf{T}}'_i = \widetilde{\mathbf{T}}_i \uplus \mathbf{x} = \mathbf{x}'}{\vdash \mathbf{x}(\tilde{y}\tilde{z}) = (\nu a)\mathbf{x}'(a\tilde{y}\tilde{z}) \rhd \mathbf{x} : \tilde{U}\,\widetilde{\mathbf{T}} \;[\!]\; \mathbf{x}' : \langle \mathbf{G}\rangle \tilde{U}\,\widetilde{\mathbf{T}}'}\,[\textsc{Res}] \qquad \frac{\forall i, \mathbf{T}_i = \mathsf{def}\ \mathbf{x} = \mathsf{end\ in}\ \mathbf{x}}{\vdash \mathbf{x}(\tilde{y}\tilde{z}) = \mathbf{0} \rhd \mathbf{x} : \tilde{U}\,\widetilde{\mathbf{T}} \;[\!]}\,[\textsc{Nil}]$$

$$\frac{\Gamma \vdash \mathbf{N}_1 \quad \Gamma \vdash \mathbf{N}_2}{\Gamma \vdash \mathbf{N}_1 \,||\, \mathbf{N}_2}\,[\textsc{NPar}] \qquad \frac{\Gamma, a : \langle \mathbf{G}\rangle \vdash \mathbf{N}}{\Gamma \vdash (\nu a)\mathbf{N}}\,[\textsc{NRes}] \qquad \frac{}{\Gamma \vdash \mathbf{0}}\,[\textsc{NNil}]$$

where

$$(\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1) \mid (\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_2) = \mathsf{def}\ \widetilde{T}\ \mathsf{in}\ (\mathbf{x}_1 \mid \mathbf{x}_2)$$
$$!\langle \mathtt{p}_1, l_1\langle U_1\rangle\rangle; \cdots !\langle \mathtt{p}_n, l_1\langle U_n\rangle\rangle \mid (\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}) = \mathsf{def}\ \widetilde{T}'\ \mathsf{in}\ \mathbf{x}$$

with $\widetilde{T}' = \mathbf{x} = !\langle \mathtt{p}_1, l_1\langle U_1\rangle\rangle.\mathbf{x}_1, \cdots \mathbf{x}_n = !\langle \mathtt{p}_2, l_2\langle U_2\rangle\rangle.\mathbf{x}_{n+1}, \widetilde{T}[\mathbf{x}_{n+1}/\mathbf{x}]$.

$\mid$ is the parallel composition of two definition types or we concatenate a message type to the top of the definition type. Other cases are undefined.

Fig. 25. Typing Rules for Run-Time State Processes

$$\frac{\vdash P_i \rhd \Sigma_i \;[\!]\; \Sigma'_i \quad comp(\{\Sigma_i; \Sigma'_i\}_i) \quad \mathbf{x} : \tilde{U}\tilde{\mathbf{T}} \in \Sigma_i \quad \Gamma \vdash \tilde{v} : \tilde{U}}{\Gamma, \widetilde{P} \vdash \mathbf{x}(\tilde{v}\tilde{c}) \rhd \tilde{c} : \tilde{\mathbf{T}}}\,[\textsc{State}]$$

$$\frac{\Gamma, \widetilde{P} \vdash \mathbf{X}_i \rhd \Delta_i \quad (i = 1,2)}{\Gamma, \widetilde{P} \vdash \mathbf{X}_1 \mid \mathbf{X}_2 \rhd \Delta_1 \mid \Delta_2}\,[\textsc{Par}] \qquad \frac{\Gamma, a : \langle \mathbf{G}\rangle, \widetilde{P} \vdash \mathbf{X} \rhd \Delta}{\Gamma, \widetilde{P} \vdash (\nu a)\mathbf{X} \rhd \Delta}\,[\textsc{Res}]$$

$$\frac{\vdash P_i \rhd \Sigma_i; \Sigma'_i \quad comp(\{\Sigma_i; \Sigma'_i\}_i)}{\Gamma, \widetilde{P} \vdash \mathbf{0} \rhd \tilde{c} : \mathsf{def}\ \mathbf{x} = \mathsf{end\ in}\ \mathbf{x}}\,[\textsc{Nil}] \qquad \frac{\Gamma, \widetilde{P} \vdash \mathbf{X} \rhd \Delta}{\Gamma \vdash \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X} \rhd \Delta}\,[\textsc{RDef}]$$

We define the generalised session types which have both message and completed types.

$$\begin{array}{rll} \text{Generalised } \mathsf{T} ::= & T & \textit{session} \\ | & \mathtt{T} & \textit{message} \\ | & \mathtt{T};T & \textit{continuation} \end{array}$$

Then ; is defined by:

$$\Delta;\{s[\mathtt{q}:r]:\mathtt{T}\} = \begin{cases} \Delta', s[\mathtt{q}:r]:\mathtt{T}';\mathtt{T} & \text{if } \Delta = \Delta', s[\mathtt{q}:r]:\mathtt{T}', \\ \Delta, s[\mathtt{q}:r]:\mathtt{T} & \text{otherwise.} \end{cases}$$

Fig. 26. Typing System for Run-Time State Processes

$$\frac{}{\Gamma \vdash_{\mathscr{S}} s : \varepsilon \triangleright \emptyset}\ [\textsc{QInit}]$$

$$\frac{\Gamma \vdash_{\mathscr{S}} s : h \triangleright \Delta \qquad \Gamma \vdash \tilde{v} : U}{\Gamma \vdash_{\mathscr{S}} s : h \cdot (\mathtt{q},\mathtt{p},l\langle v\rangle) \triangleright \Delta;\{s[\mathtt{q}] :!\langle \mathtt{p}, l\langle U\rangle\rangle\}}\ [\textsc{QVal}]$$

$$\frac{\Gamma \vdash_{\mathscr{S}} s : h \triangleright \Delta}{\Gamma \vdash s : h \cdot (\mathtt{q},\mathtt{p},l\langle s'[\mathtt{p}']\rangle) \triangleright (\Delta, s'[\mathtt{p}'] : \mathbf{T}');\{s[\mathtt{q}] :!\langle \mathtt{p}, l\langle \mathbf{T}'\rangle\rangle\}}\ [\textsc{QDeleg}]$$

Fig. 27. Typing system for queues

$$\frac{\Gamma \vdash \mathbf{P} \triangleright \Delta}{\Gamma \vdash_{\varnothing} \mathbf{P} \triangleright \Delta}\ [\textsc{Prom}] \qquad \frac{\Gamma \vdash_{\mathscr{S}} \mathbf{N} \triangleright \Delta \quad \Delta \approx \Delta'}{\Gamma \vdash_{\mathscr{S}} \mathbf{N} \triangleright \Delta'}\ [\textsc{Eq}]$$

$$\frac{\Gamma \vdash_{\mathscr{S}} \mathbf{N} \triangleright \Delta \quad \Gamma \vdash_{\mathscr{S}'} \mathbf{N}' \triangleright \Delta'}{\Gamma \vdash_{\mathscr{S} \uplus \mathscr{S}'} \mathbf{N} \,||\, \mathbf{N}' \triangleright \Delta \mid \Delta'}\ [\textsc{GPar}] \qquad \frac{\Gamma \vdash_{\mathscr{S}} \mathbf{N} \triangleright \Delta \quad co(s,\Delta)}{\Gamma \vdash_{\mathscr{S} \setminus s} (\nu\, s)\mathbf{N} \triangleright \Delta \setminus s}\ [\textsc{GSRes}]$$

$$\frac{\Gamma, a : \langle \mathbf{G}\rangle \vdash_{\mathscr{S}} \mathbf{N} \triangleright \Delta}{\Gamma \vdash_{\mathscr{S}} (\nu\, a)\mathbf{N} \triangleright \Delta}\ [\textsc{GNRes}] \qquad \frac{\Gamma \vdash a : \langle \mathbf{G}\rangle \quad \mathbf{G} \upharpoonright \mathtt{p} = \mathbf{T}}{\Gamma \vdash_{\emptyset} a[\mathtt{p}]\langle s\rangle \triangleright s[\mathtt{p}] : \mathbf{T}}\ [\textsc{GInit}]$$

Typing Systems The typing of the state is similar with the definition agent, given in Fig. 25. The typing of network processes and the typing for queues are essentially the same as the communication typing system in [3] since the network level processes can be seen as the same as the original session calculus. Hence we omit $co(s,\Delta)$ (which means Δ is a complete collection of projections from some $\mathbf{G}$ at s) and the standard type equivalence.

F Appendix for Section 6

F.1 Transition Rules between Environments

We define the transition system between environments in figure 28. We also omitted the following structure rule from the type transition system.

$$\frac{\mathbf{T}_0 \equiv \mathbf{T} \xrightarrow{\ell} \mathbf{T}' \equiv \mathbf{T}'_0}{\mathbf{T}_0 \xrightarrow{\ell} \mathbf{T}'_0} \lfloor \textsc{Str} \rfloor$$

Fig. 28. Environment Transition System

$$(\Gamma, \Delta) \xrightarrow{\tau} (\Gamma, \Delta) \quad [\textsc{Tau}]$$

$$(\Gamma, a : \langle \mathbf{G} \rangle, \Delta) \xrightarrow{a\langle \mathbf{G} \rangle} (\Gamma, a : \langle \mathbf{G} \rangle, \Delta) \quad [\textsc{Init}]$$

$$(\Gamma, a : \langle \mathbf{G} \rangle, \Delta) \xrightarrow{a\langle s \rangle[\mathtt{p}]} (\Gamma, a : \langle \mathbf{G} \rangle, \Delta, s[\mathtt{p}] : \mathbf{G} \upharpoonright \mathtt{p}) \quad [\textsc{Acc}]$$

$$\frac{\mathbf{T} \xrightarrow{!\langle \mathtt{q}, l\langle U \rangle\rangle} \mathbf{T}' \qquad \Gamma \vdash v : U}{(\Gamma, \Delta, s[\mathtt{p}] : \mathbf{T}) \xrightarrow{s[\mathtt{p},\mathtt{q}]!l\langle v \rangle} (\Gamma, \Delta, s[\mathtt{p}] : \mathbf{T}')} [\textsc{Send}] \qquad \frac{\mathbf{T} \xrightarrow{?\langle \mathtt{p}, l\langle U \rangle\rangle} \mathbf{T}' \qquad \Gamma \vdash v : U}{(\Gamma, \Delta, s[\mathtt{p}] : \mathbf{T}) \xrightarrow{s[\mathtt{q},\mathtt{p}]?l\langle v \rangle} (\Gamma, \Delta, s[\mathtt{p}] : \mathbf{T}')} [\textsc{Recv}]$$

$$\frac{\mathbf{T} \xrightarrow{!\langle \mathtt{q}, l\langle \mathbf{T}_s \rangle\rangle} \mathbf{T}'}{(\Gamma, \Delta, s[\mathtt{p}] : \mathbf{T}, s'[\mathtt{p}'] : \mathbf{T}_s) \xrightarrow{s[\mathtt{p},\mathtt{q}]!l\langle s'[\mathtt{p}'] \rangle} (\Gamma, \Delta, s[\mathtt{p}] : \mathbf{T}')} [\textsc{Deleg}]$$

$$\frac{\mathbf{T} \xrightarrow{?\langle \mathtt{p}, l\langle \mathbf{T}_r \rangle\rangle} \mathbf{T}'}{(\Gamma, \Delta, s[\mathtt{p}] : \mathbf{T}) \xrightarrow{s[\mathtt{p},\mathtt{q}]?l\langle s'[\mathtt{p}'] \rangle} (\Gamma, \Delta, s[\mathtt{p}] : \mathbf{T}', s'[\mathtt{p}'] : \mathbf{T}_r)} [\textsc{Catch}]$$

$$\frac{\mathbf{T}_1 \xrightarrow{!\langle \mathtt{q}, l\langle U \rangle\rangle} \mathbf{T}'_1 \qquad \mathbf{T}_2 \xrightarrow{?\langle \mathtt{p}, l\langle U \rangle\rangle} \mathbf{T}'_2}{(s[\mathtt{p}] : \mathbf{T}_1, s[\mathtt{q}] : \mathbf{T}_2, \Delta) \to (s[\mathtt{p}] : \mathbf{T}'_1, s[\mathtt{q}] : \mathbf{T}'_2, \Delta)} [\textsc{Com}]$$

In figure 28, rules $\lfloor$Init,Acc$\rfloor$ are used for the session initiations. Others promote transitions between local types to transitions between environments. Other rules are straightforward.

F.2 Proofs

Proof of Lemma 6.1

1. (Process P) For the axioms in figure 22, we use the type equalities in figure 9. Others are straightforward by induction.
2. (Def agents **P**) For the case of join and merge, we use the structure rule for local types defined in figure 14.
3. (Network **N**) The same as [3].

Proof of Theorem 6.1 (Process P)

Case [SEND] Suppose:

$$\mathbf{x}(\tilde{y}\tilde{z}) = x\,!\,\langle \mathtt{p}, l\langle e\rangle\rangle.\mathbf{x}'(\tilde{e}\tilde{z}) \vdash \mathbf{x}(\tilde{v}\tilde{c}) \xrightarrow{s[\mathtt{q},\mathtt{p}]!l\langle v\rangle} \mathbf{x}'(\tilde{v}'\tilde{c})$$

with $x[\tilde{c}/\tilde{z}] = s[\mathtt{q}]$, $e[\tilde{v}/\tilde{y}] \downarrow v$ and $\tilde{e}[\tilde{v}/\tilde{y}] \downarrow \tilde{w}$ by [SEND]. Then by [STATE] in figure 25,

$$\Gamma, P \vdash \mathbf{x}(\tilde{v}\tilde{c}) \rhd \tilde{c} : \widetilde{\mathbf{T}}$$

and

$$P = \mathbf{x}(\tilde{y}\tilde{z}) = x\,!\,\langle \mathtt{p}, l\langle e\rangle\rangle.\mathbf{x}'(\tilde{e}\tilde{z})$$

has type

$$\vdash P \rhd \mathbf{x} : \tilde{U}\,\widetilde{\mathbf{T}} \;\;[\!]\;\; \mathbf{x}' : \tilde{U}'\,\widetilde{\mathbf{T}}'$$

with

$$\tilde{y} : \tilde{U} \vdash e : U, \quad \tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}',$$

and

$$\mathbf{T}_i = \mathbf{T}_i' \uplus \mathbf{x} =\!!\, \langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}' \quad \forall j \neq i, \mathbf{T}_j = \mathbf{T}_j' \uplus \mathbf{x} = \mathbf{x}'$$

with $z_i = x$. By substitution lemma,

$$\tilde{w} : \tilde{U} \vdash e[\tilde{v}/\tilde{y}] : U, e_i[\tilde{v}/\tilde{y}] : U_i$$

On the other hand, by $\lfloor$SEND$_\ell\rfloor$ in figure 14, we have

$$\mathbf{T}_i = \mathbf{T}_i' \uplus \mathbf{x} =\!!\, \langle \mathtt{p}, l\langle U\rangle\rangle.\mathbf{x}' \xrightarrow{!\langle \mathtt{p}, l\langle U\rangle\rangle} \mathbf{T}_i'$$

This implies, by [SEND] in figure 28, with $\Gamma \vdash v : U$ and $x[\tilde{c}/\tilde{z}] = s[\mathtt{q}]$, we have:

$$(\Gamma, \Delta) \xrightarrow{s[\mathtt{q},\mathtt{p}]!l\langle v\rangle} (\Gamma, \Delta')$$

where $\Delta = \tilde{c} : \tilde{\mathbf{T}}$ and $\Delta' = \Delta \setminus s[\mathtt{q}] \cup \{s[\mathtt{q}] : \mathbf{T}_i'\}$ as desired.

Case [RECV] Similar with [SEND].

Case [INIT] Similar with next [ACCEPT].

Case [ACCEPT]

Suppose

$$\mathbf{x}(\tilde{y}\tilde{z}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}\tilde{z}y) \vdash \mathbf{x}(\tilde{w}\tilde{c}) \xrightarrow{a[\mathtt{p}]\langle s\rangle} \mathbf{x}'(\tilde{v}\tilde{c}s)$$

with $a = x[\tilde{w}/\tilde{y}]$, $\tilde{e}[\tilde{w}/\tilde{y}] \downarrow \tilde{v}$, By [REQ] in figure 13,

$$\vdash \mathbf{x}(\tilde{y}\tilde{z}) = x[\mathtt{p}](y).\mathbf{x}'(\tilde{e}\tilde{z}y) \rhd \mathbf{x} : \tilde{U}\,\widetilde{\mathbf{T}} \;\;[\!]\;\; \mathbf{x}' : \tilde{U}'\,\widetilde{\mathbf{T}}'\mathbf{T}$$

with $\tilde{y} : \tilde{U} \vdash \tilde{e} : \tilde{U}'$, $\tilde{y} : \tilde{U} \vdash x : \langle \mathbf{G}\rangle$, $\forall i, \mathbf{T}_i = \mathbf{T}_i' \uplus \mathbf{x} = \mathbf{x}'$ and $\mathbf{T} = \mathbf{G} \upharpoonright \mathtt{p}$. Then by [STATE] in figure 25, we have:

$$\Gamma, \widetilde{P} \vdash \mathbf{x}(\tilde{w}\tilde{c}) \rhd \tilde{c} : \tilde{\mathbf{T}}$$

and, by [Acc] in figure 28, we have

$$(\Gamma, \Delta) \xrightarrow{a[\mathtt{p}]\langle s\rangle} (\Gamma, \Delta, s[\mathtt{p}] : \mathbf{T})$$

By [State] in figure 25, we have

$$\Gamma, \widetilde{P} \vdash \mathbf{x}'(\tilde{v}\tilde{c}s) \rhd \tilde{c} : \tilde{\mathbf{T}}, s[\mathtt{p}] : \mathbf{T}$$

as desired, noting $\mathbf{T}'_i \equiv \mathbf{T}_i$.

Case [IfT,IfF] Direct by using $\lfloor$Cond$\rfloor$ in figure 14.

Case [New] Trivial by [State] and [Res] in figure 25.

Case [Ext] Direct by using $\lfloor$Choice$\rfloor$ in figure 14.

Other cases ([Par],[Weak],[Str],[Res]) are straightforward by induction.

(Def agent P)

1. The case $\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'$ is straightforward by [Def] in figure 13 and [State] in figure 25.
2. The case [Tau] is trivial.

(Network N)

Case [Put] Assume

$$\mathbf{P} \mid\mid s : h \to \mathbf{P}' \mid\mid s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v\rangle)$$

with

$$\mathbf{P} \xrightarrow{s[\mathtt{p},\mathtt{q}]!l\langle v\rangle} \mathbf{P}'$$

Since there is a typed transition, $s[\mathtt{p}]$ in $\mathbf{P}$ has a type

$$\mathbf{T}_1 \stackrel{\text{def}}{=} (\mathsf{def}\ \mathbf{x} = !\langle \mathtt{p}_n, l_1\langle U_n\rangle\rangle.\mathbf{x}_1, \widetilde{T}\ \mathsf{in}\ \mathbf{x})$$

and $s[\mathtt{p}]$ in $s : h$ has a type

$$\mathtt{T}_1 \stackrel{\text{def}}{=} !\langle \mathtt{p}_1, l_1\langle U_1\rangle\rangle; \cdots; !\langle \mathtt{p}_{n-1}, l_1\langle U_{n-1}\rangle\rangle$$

On the other hand, $s[\mathtt{p}]$ in $\mathbf{P}'$ has a type:

$$\mathbf{T}_2 \stackrel{\text{def}}{=} (\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1)$$

and $s[\mathtt{p}]$ in $s : h \cdot\cdot (\mathtt{p}, \mathtt{q}, l\langle v\rangle)$ has a type

$$\mathtt{T}_2 \stackrel{\text{def}}{=} !\langle \mathtt{p}_1, l_1\langle U_1\rangle\rangle; \cdots !\langle \mathtt{p}_{n-1}, l_1\langle U_{n-1}\rangle\rangle \cdot !\langle \mathtt{p}_n, l_1\langle U_n\rangle\rangle$$

Then by the parallel composition rule, we have $\mathbf{T}_2 \mid \mathtt{T}_2 = \mathbf{T}_1 \mid \mathtt{T}_1$, as desired.

Case [Get] Similar with [Put].

Case [INIT$_N$]

Easy using [GInit], and noting a parallel composition of $a[\mathtt{p}_i]\langle s\rangle$ forms the complete type

$co(s,\Delta)$ with $\Delta = \{s[\mathtt{p}_i] : \mathbf{T}_i\}_i$, hence we can close by name hiding.

Case [Acc_N] .

By [GInit] in figure 26, we can assume $\Gamma \vdash \mathbf{P} \,||\, a[\mathtt{p}]\langle s\rangle \triangleright \Delta, s[\mathtt{p}] : \mathbf{G} \upharpoonright \mathtt{p}$ with $\Gamma \vdash a : \langle \mathbf{G}\rangle$. Then by [Req] in figure 13, $\Gamma \vdash \mathbf{P}' \triangleright \Delta, s[\mathtt{p}] : \mathbf{G} \upharpoonright \mathtt{p}$, as required.

Other cases [Res_N,Par_N,Str_N] are standard.

This concludes all cases for Subject Transition and Reduction Theorems. □

Theorem 6.5 Since $\mathbf{P}$ and $\mathbf{X}$ does not contain P such that a primitive which can be blocked from the outside, i.e. $\widetilde{P}$ in $\mathsf{def}\ \mathbf{x}_0(x) = x[\mathtt{p}](y).\mathbf{x}_1, \widetilde{P}\ \mathsf{in}\ \mathbf{x}_0(a)$ does not contain any initiator, acceptor, name creator, delegation nor catch (sending and receiving session channels as arguments), completeness for $\mathbf{P}$ and $\mathbf{X}$ is trivial due to a tight correspondence between types and processes. Hence we only show the main case, (3).

Assume $\mathbf{N}$ is simple and $\mathbf{N} \longrightarrow^* \mathbf{N}'$. W.l.o.g, we can assume $\mathbf{N} \longrightarrow^* \mathbf{N}' \equiv (\nu s)(s : \emptyset \,||\, \mathbf{P}_1 \,||\, \cdots \,||\, \mathbf{P}_n)$ with $a : \langle \mathbf{G}\rangle \vdash \mathbf{P}_j \triangleright s[\mathtt{p}_j] : \mathbf{T}_j$. By assumption, there exists $i \neq j$ such that

$$\frac{\mathbf{T}_1 \xrightarrow{!\langle \mathtt{p}_j, l\langle U\rangle\rangle} \mathbf{T}_1' \qquad \mathbf{T}_2 \xrightarrow{?\langle \mathtt{p}_i, l\langle U\rangle\rangle} \mathbf{T}_2'}{(s[\mathtt{p}_i] : \mathbf{T}_1, s[\mathtt{p}_j] : \mathbf{T}_2, \Delta) \to (s[\mathtt{p}_i] : \mathbf{T}_1', s[\mathtt{p}_j] : \mathbf{T}_2', \Delta)}\,[\text{Com}]$$

Then by assumption,

$$\mathbf{P}_i \xrightarrow{s[\mathtt{p}_i, \mathtt{p}_j]!l\langle v\rangle} \mathbf{P}_i'$$

Hence, we have:

$$\mathbf{P}_i \,||\, s : h \to \mathbf{P}_i' \,||\, s : h \cdot (\mathtt{p}, \mathtt{q}, l\langle v\rangle)$$

by [Put]. On the other hand, we have:

$$\mathbf{P}_j \xrightarrow{s[\mathtt{p}_i, \mathtt{p}_j]?l\langle v\rangle} \mathbf{P}_j'$$

This implies

$$\mathbf{P}_j \,||\, s : (\mathtt{p}_i, \mathtt{p}_j, l\langle v\rangle) \cdot h \to \mathbf{P}_j'$$

Hence, even in the case $h = \emptyset$, there exists at least two steps reductions such that: $\mathbf{P}_i \,||\, \mathbf{P}_j \,||\, s : h \longrightarrow\longrightarrow \mathbf{P}_i' \,||\, \mathbf{P}_j' \,||\, s : h$, as required.