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 [12] (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, 14] are a type discipline that can enforce strong communication safety for distributed processes [3, 14], 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 [8, 22], protocol optimisations for distributed objects [21] and parallel algorithms [17], and have recently lead to industrial projects [19, 20].

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 [13]. 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 [12] and later by Villard [23]) 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 [13]. This subclass, compatible deterministic two-machine without mixed states [12] (see § 3 and § 6), 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 [14] 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:

$$\begin{tikzcd}
\begin{array}{c}\text{Generalised}\\\text{Global Type}\end{array} \arrow[r, "\text{Projection}"] & \begin{array}{c}\text{Local Types}\\\approx\\\text{CFSMs (MSA)}\end{array} \arrow[r, "\text{Type checking}"] & \begin{array}{c}\text{General}\\\text{Multiparty}\\\text{Processes}\end{array}
\end{tikzcd}$$

Generalised Global Types This paper proposes a new global type syntax which encompasses previous systems [3, 14] 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

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

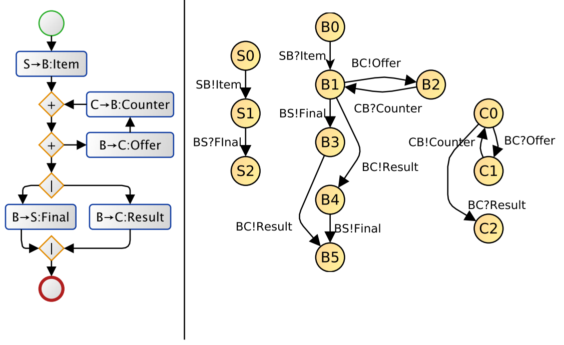


Fig. 1. Trade Example: Global Type and CFSM

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.

$\mathbf{G}_{\text{Trade}}$ 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.

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, 23] (§ 3).
– We develop a new typing system for multiparty session mobile processes generalised with choice, fork, merge and join constructs (§ 4, § 5.1), and prove that typed processes conform the safety and liveness properties defined in CFSMs (§ 5.2).
– We compare our framework with existing session type theories and CFSMs results (§ 6). Our framework (global type well-formedness checking, projection, type-checking) is notably polynomial in the size of the global type or mobile processes.

The long version [18] 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. [1]The new features are flexible fork, choice, merge and join operations for precise thread management.

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

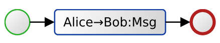
A global type $\mathbf{G} = \mathsf{def}\ \widetilde{G}\ \mathsf{in}\ \mathbf{x}_0$ describes an interaction between a fixed number of participants. The prescribed interaction starts from $\mathbf{x}_0$, which we call the *initial state*, and proceeds according to the transitions specified in $\widetilde{G}$. The *state variables* $\mathbf{x}$ in $\widetilde{G}$ represent the successive distributed states of the interaction. Transitions can be *labelled message exchanges* $\mathbf{x} = \mathsf{p} \to \mathsf{p}' : l\langle U \rangle; \mathbf{x}'$ where p and 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 p can go from $\mathbf{x}$ to the continuation $\mathbf{x}'$ by sending message l, while 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 U include shared channel types $\langle \mathbf{G} \rangle$ or base types.$\mathbf{x} = \mathbf{x}' + \mathbf{x}''$ 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.

[1] We omit the delegation for space reason. Its inclusion is straightforward, see [18].

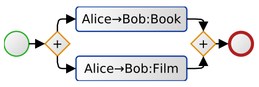
The motivation behind this choice of graph syntax is to support general graphs. A traditional global type syntax tree, with operators fork $|$ and choice $+$, even with recursion [3, 6, 10, 14], is limited to series-parallel graphs.

Example 2.1 (Generalised Global Types). We now give several example, 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.

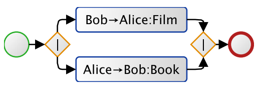
1.
$$\begin{aligned} \mathbf{G}_1 = \ & \mathsf{def}\ \mathbf{x}_0 = \mathtt{Alice} \to \mathtt{Bob} : \mathit{Msg}\langle \mathsf{nat} \rangle ; \mathbf{x}_1 \\ & \mathbf{x}_1 = \mathsf{end}\ \mathsf{in}\ \mathbf{x}_0 \end{aligned}$$



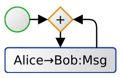
2.
$$\begin{aligned} \mathbf{G}_2 = \mathsf{def}\ \mathbf{x}_0 &= \mathbf{x}_1 + \mathbf{x}_2 \\ \mathbf{x}_1 &= \mathtt{Alice} \to \mathtt{Bob} : \mathit{Book}\langle \mathsf{string} \rangle ; \mathbf{x}_3 \\ \mathbf{x}_2 &= \mathtt{Alice} \to \mathtt{Bob} : \mathit{Film}\langle \mathsf{string} \rangle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



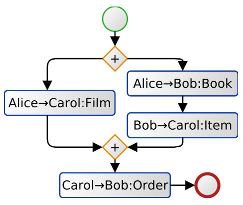
3.
$$\begin{aligned} \mathbf{G}_3 = \mathsf{def}\ \mathbf{x}_0 &= \mathbf{x}_1 \mid \mathbf{x}_2 \\ \mathbf{x}_1 &= \mathtt{Alice} \to \mathtt{Bob} : \mathit{Book}\langle \mathsf{string} \rangle ; \mathbf{x}_3 \\ \mathbf{x}_2 &= \mathtt{Bob} \to \mathtt{Alice} : \mathit{Film}\langle \mathsf{string} \rangle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



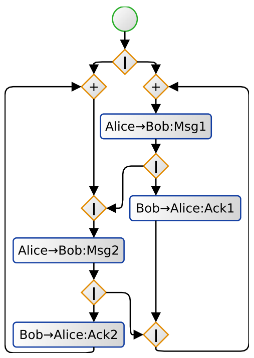
4.
$$\begin{aligned} \mathbf{G}_4 = \mathsf{def}\ \mathbf{x}_0 + \mathbf{x}_2 &= \mathbf{x}_1 \\ \mathbf{x}_1 &= \mathtt{Alice} \to \mathtt{Bob} : \mathit{Msg}\langle \mathsf{string} \rangle ; \mathbf{x}_2\ \mathsf{in}\ \mathbf{x}_0 \end{aligned}$$



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



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



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. $\mathbf{G}_{AB}$ 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 § 6). 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.

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

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 [18] for the polynomial verification algorithm). In $\mathbf{G}_{\neg\mathrm{thr}}$ (written above), 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`.

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 $Rcv(\widetilde{G})(\mathbf{x})$ below, which computes the set of all the participants that will be expecting at least one message starting from state $\mathbf{x}$. Additionally, $Rcv(\widetilde{G})(\mathbf{x})$ returns the label l of the received message and the merging points $\tilde{\mathbf{x}}$ encountered. We say that the equality $Rcv(\widetilde{G})(\mathbf{x}_1) = Rcv(\widetilde{G})(\mathbf{x}_2)$ holds if $\forall(\mathtt{p} : l_1 : \tilde{\mathbf{x}}_1) \in Rcv(\widetilde{G})(\mathbf{x}_1), \forall(\mathtt{p} : l_2 : \tilde{\mathbf{x}}_2) \in Rcv(\widetilde{G})(\mathbf{x}_2), l_1 \neq l_2 \vee \tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2$ 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`).

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

$$
\begin{array}{l}
\mathbf{G}_{\neg\mathrm{loc}} = \mathsf{def}\ \mathbf{x}_0 = \mathbf{x}_1 + \mathbf{x}_2 \\
\quad \mathbf{x}_1 = \mathtt{Alice} \to \mathtt{Bob} : \mathit{Book}\langle \mathsf{string} \rangle; \mathbf{x}_3 \\
\quad \mathbf{x}_2 = \mathtt{Bob} \to \mathtt{Alice} : \mathit{Film}\langle \mathsf{string} \rangle; \mathbf{x}_4 \\
\mathbf{x}_3 + \mathbf{x}_4 = \mathbf{x}_5 \\
\quad \mathbf{x}_5 = \mathsf{end}\ \mathsf{in}\ \mathbf{x}_0
\end{array}
$$

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 $ASend(\widetilde{G})(\mathbf{x}) = \mathtt{p}$ 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 [18]). As an example, we give above an illegal global type $\mathbf{G}_{\neg\mathrm{loc}}$ 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**) $Rcv(\widetilde{G})(\mathbf{x}') = Rcv(\widetilde{G})(\mathbf{x}'')$; and **(2)** (**Unique sender**) $\exists \mathtt{p}, ASend(\widetilde{G})(\mathbf{x}) = \mathtt{p}$.

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 $Lin(\widetilde{G})(\mathbf{x}_1)$ and $Lin(\widetilde{G})(\mathbf{x}_2)$ 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 *Msg*1 and *Msg*2 from both being renamed *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* $Lin(\widetilde{G})(\mathbf{x}') = Lin(\widetilde{G})(\mathbf{x}'')$.

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 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 $\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x}_0$.

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

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 $\mathbf{G} \upharpoonright \mathtt{p}$) below. The projection is straightforward: $\mathbf{x} = \mathtt{p} \to \mathtt{q} : l\langle U\rangle; \mathbf{x}'$ is an output from p's viewpoint and an input from 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. See [18].

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

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 $|$).

$$\begin{aligned}
\mathbf{T}_{\text{TradeS}} = \;& \mathsf{def}\ \mathbf{x}_0 = !\,\langle \texttt{SB}, \mathit{Item}\langle \mathsf{string}\rangle\rangle.\mathbf{x}_1\\
& \mathbf{x}_1 = ?\,\langle \texttt{BS}, \mathit{Final}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_{10}\\
& \mathbf{x}_{10} = \mathsf{end} \quad \mathsf{in}\ \mathbf{x}_0
\end{aligned}$$

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

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

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: $\mathcal{P} \subseteq \{\texttt{Alice}, \texttt{Bob}, \texttt{Carol}, \ldots, \texttt{A}, \texttt{B}, \texttt{C}, \ldots, \texttt{S}, \ldots\}$.

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) $C = \{\texttt{pq} \in \mathcal{P}^2 \mid \texttt{p} \neq \texttt{q}\}$ 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, $\texttt{pq}!a$ denotes the *sending* action of a from process $\texttt{p}$ to process $\texttt{q}$, and $\texttt{pq}?a$ denotes the *receiving* action of a from $\texttt{p}$ by $\texttt{q}$. $\pi, \pi', \ldots$ 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 $q' = q''$.[2]

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

[2] "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.

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 pq, while (2-b) gets the content a from a channel 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 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 $ASend(t_1\cdots t_m) = \{\mathtt{p} \mid \exists i \leq m, t_i = (s_i, \mathtt{pq}!a, s_{i+1}) \wedge \forall k < i.\ t_k \neq (s_k, \mathtt{p}'\mathtt{p}?b, s_{k+1})\}$ and represent the participants who could immediately send from state s_1. These definitions match the global 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*[3] if for all $s \in RS(S)$, there exists $s \longrightarrow^* s'$ such that s' is final.

[3] The terminology follows [6].

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:

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

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

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

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

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

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:

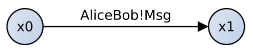
- Q is defined as the set of states $\mathtt{X}$ built from the recursion variables $\{\mathbf{x}_i\}$ of $\mathbf{T}$. Q is defined up to the equivalence relation $\equiv_{\widetilde{T}}$.
- $C = \{\mathtt{pq} \mid \mathtt{p}, \mathtt{q} \in \mathbf{G}\}$; $q_0 = \mathbf{x}_0$; and $\mathbb{A}$ is the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

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.

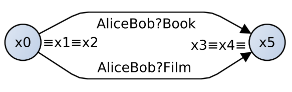
The generation of an MSA from a global type $\mathbf{G}$ is exponential in the size of $\mathbf{G}$. It is however polynomial in the absence of parallel composition. Note that neither well-formedness nor type-checking requires the explicit generation of MSAs.

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.

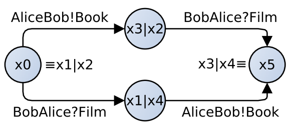
1. $\begin{array}{rl} \mathbf{G}_1 \upharpoonright \mathtt{Alice} = \mathsf{def}\ \mathbf{x}_0 &= !\,\langle \mathtt{Bob}, \mathit{Msg}\langle \mathsf{nat}\rangle\rangle.\mathbf{x}_1 \\ \mathbf{x}_1 &= \mathsf{end}\ \mathsf{in}\ \mathbf{x}_0 \end{array}$



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



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



1. The MSA of the projection of $\mathbf{G}_1$ to `Alice` has two states and one transition.
2. Since `Bob` is receiving `Alice`'s messages, the projection of $\mathbf{G}_2$ to `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 `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 $\mathtt{pq}!a$ is immediately followed by $\mathtt{pq}?a$) 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 [9, § 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) [9, 14],[4] 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* . 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, 14].

[4] It is formally defined in [9, 14] and [18].

Syntax The syntax of processes is defined below.

$$
\begin{array}{rcll}
v & ::= & a \mid \texttt{true} \mid \texttt{false} \mid \ldots & \text{values} \\
\mathbf{P} & ::= & \mathsf{def}\ \widetilde{P}\ \mathsf{in}\ \mathbf{X} & \text{definition} \\
P & ::= & & \text{process transition} \\
 & \mid & \mathbf{x}(\tilde{x}) = x\langle \mathbf{G} \rangle.\mathbf{x}'(\tilde{e}) & \text{init} \\
 & \mid & \mathbf{x}(\tilde{x}) = x[\mathrm{p}](y).\mathbf{x}'(\tilde{e}) & \text{request} \\
 & \mid & \mathbf{x}(\tilde{x}) = x\,!\,\langle \mathrm{p}, l\langle e \rangle \rangle.\mathbf{x}'(\tilde{e}) & \text{send} \\
 & \mid & \mathbf{x}(\tilde{x}) = x\,?\,\langle \mathrm{p}, l(y) \rangle.\mathbf{x}'(\tilde{e}) & \text{receive} \\
 & \mid & \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{y}) \mid \mathbf{x}''(\tilde{z}) & \text{parallel} \\
 & \mid & \mathbf{x}(\tilde{x}) = \texttt{if}\ e\ \texttt{then}\ \mathbf{x}'(\tilde{e}')\ \texttt{else}\ \mathbf{x}''(\tilde{e}'') & \text{conditional} \\
 & \mid & \mathbf{x}(\tilde{x}) = \mathbf{x}'(\tilde{x})\ \&\ \mathbf{x}''(\tilde{x}) & \text{external choice} \\
 & \mid & \mathbf{x}(\tilde{y}) \mid \mathbf{x}'(\tilde{z}) = \mathbf{x}''(\tilde{x}) & \text{join} \\
 & \mid & \mathbf{x}(\tilde{x}) + \mathbf{x}'(\tilde{x}) = \mathbf{x}''(\tilde{x}) & \text{merge} \\
 & \mid & \mathbf{x}(\tilde{x}) = (\nu a)\ \mathbf{x}'(a\tilde{x}) & \text{new name} \\
 & \mid & \mathbf{x}(\tilde{x}) = \mathbf{0} & \text{null}
\end{array}
$$

$$
\begin{array}{rcll}
e & ::= & v \mid x \mid e \wedge e \mid \ldots & \text{expression} \\
h & ::= & \emptyset \mid h \cdot (\mathrm{p}, \mathrm{q}, l\langle v \rangle) & \text{messages} \\
\mathbf{X} & ::= & & \text{state} \\
 & \mid & \mathbf{x}(\tilde{v}) & \text{thread} \\
 & \mid & \mathbf{X} \mid \mathbf{X} & \text{parallel} \\
 & \mid & (\nu a)\mathbf{X} & \text{restriction} \\
 & \mid & \mathbf{0} & \text{null} \\
\mathbf{N} & ::= & & \text{network} \\
 & \mid & \mathbf{P} & \text{def} \\
 & \mid & \mathbf{N} \parallel \mathbf{N} & \text{parallel} \\
 & \mid & (\nu a)\mathbf{N} & \text{new name} \\
 & \mid & \mathbf{0} & \text{null} \\
 & \mid & (\nu s)\mathbf{N} & \text{new session} \\
 & \mid & s : h & \text{queue} \\
 & \mid & a\langle s \rangle[\mathrm{p}] & \text{invitation}
\end{array}
$$

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 $\mathbf{x}(x,z) = z\,?\,\langle \mathrm{p}, l(y) \rangle.\mathbf{x}'(x,y,z)$, 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 $\mathbf{x}(\tilde{x}) = x[\mathrm{p}](y).\mathbf{x}'(\tilde{e})$ (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 $x!\langle \mathrm{p}, l\langle e \rangle \rangle$ allows in session x to send to p a value e labelled by a constant l. The reception $x?\langle \mathrm{p}, l(y) \rangle.\mathbf{x}'(\tilde{e})$ expects from 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 ($\texttt{if}\ e\ \texttt{then}\ \mathbf{x}'(\tilde{e}')\ \texttt{else}\ \mathbf{x}''(\tilde{e}'')$) 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 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 below. 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.

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

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

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

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

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

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

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

Rule [Send] emits a message from p to 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 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.

Example 4.1 (Trade Example). 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.

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

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

$\mathbf{P}_{\mathtt{S}}$ and $\mathbf{P}_{\mathtt{C}}$, respectively correspond to the seller `S` and client `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 `S`, `B` and `C`. $\mathbf{P}_{\mathtt{S}}$ then joins the session as the seller `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 “`HGG`” and z the session name. The execution of $\mathbf{x}_0(y,z)$ sends a message *Item* with payload “`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 `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 Properties of Generalised Multiparty Session Processes

5.1 Typing Generalised Multiparty Session Processes

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:

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

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.

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
$\Gamma \vdash \mathbf{P} \rhd \Delta$	Process $\mathbf{P}$ has type Δ under Γ
$\Gamma, \widetilde{P} \vdash \mathbf{X} \rhd \Delta$	State variable $\mathbf{X}$ has type Δ under Γ and $\widetilde{P}$
$\Gamma \vdash \mathbf{N} \rhd \Delta$	Network $\mathbf{N}$ has type Δ under Γ

Typing Rules We only list two typing rules. 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.

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

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

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 $\mathbf{T} = \mathbf{G} \upharpoonright \mathrm{p}$.

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.*

5.2 Properties of Typed Multiparty Session Processes

This subsection 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.

Let ℓ range over transition labels for types: $\ell ::= \tau \mid !\langle \mathrm{p}, l\langle U\rangle\rangle \mid ?\langle \mathrm{p}, l\langle U\rangle\rangle$. We define below a labelled transition relation between types $\mathbf{T} \xrightarrow{\ell} \mathbf{T}'$, defined modulo structure rules (for join and merge) and type equality.

$$\begin{array}{rcll}
\mathsf{def}\ \widetilde{T}\ \mathsf{in}\ \mathbf{x} & \equiv & \mathsf{def}\ \widetilde{T}'\ \mathsf{in}\ \mathbf{x} \qquad (\widetilde{T} = \widetilde{T}') & \lfloor\textsc{Eq}\rfloor \\
\mathsf{def}\ \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_i & \equiv & \mathsf{def}\ \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x} \quad (i = 1 \text{ or } i = 2) & \lfloor\textsc{Merge}\rfloor \\
\mathsf{def}\ \mathbf{x}_1 \mid \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_1 \mid \mathbf{x}_2 & \equiv & \mathsf{def}\ \mathbf{x}_1 \mid \mathbf{x}_2 = \mathbf{x}, \widetilde{T}\ \mathsf{in}\ \mathbf{x} & \lfloor\textsc{Join}\rfloor \\
\mathsf{def}\ \mathbf{x} = !\langle \mathrm{p}, l\langle U\rangle\rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x} & \xrightarrow{!\langle \mathrm{p}, l\langle U\rangle\rangle} & \mathsf{def}\ \mathbf{x} = !\langle \mathrm{p}, l\langle U\rangle\rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x}' & \lfloor\textsc{Send}_\ell\rfloor \\
\mathsf{def}\ \mathbf{x} = ?\langle \mathrm{p}, l\langle U\rangle\rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x} & \xrightarrow{?\langle \mathrm{p}, l\langle U\rangle\rangle} & \mathsf{def}\ \mathbf{x} = ?\langle \mathrm{p}, l\langle U\rangle\rangle.\mathbf{x}', \widetilde{T}\ \mathsf{in}\ \mathbf{x}' & \lfloor\textsc{Recv}_\ell\rfloor \\
\mathsf{def}\ \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2, \widetilde{T}\ \mathsf{in}\ \mathbf{x} & \xrightarrow{\tau} & \mathsf{def}\ \mathbf{x} = \mathbf{x}_1 \oplus \mathbf{x}_2, \widetilde{T}\ \mathsf{in}\ \mathbf{x}_i \quad (i = 1 \text{ or } i = 2) & \lfloor\textsc{Cond}\rfloor
\end{array}$$

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

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

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 $\lfloor\textsc{Com}\rfloor$ which represents the reduction between a message queue and a process at the network level. Other omitted rules are straightforward.

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 5.1 (Subject Congruence, Transition and Reduction).

1. *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* $\mathbf{N}$.
2. $\Gamma, \widetilde{P} \vdash \mathbf{X} \triangleright \Delta$ *and* $\widetilde{P} \vdash \mathbf{X} \xrightarrow{\alpha} \mathbf{X}'$ *imply* $\Gamma', \widetilde{P} \vdash \mathbf{X}' \triangleright \Delta'$ *with* $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma', \Delta')$.
3. $\Gamma \vdash \mathbf{P} \triangleright \Delta$ *and* $\mathbf{P} \xrightarrow{\alpha} \mathbf{P}'$ *imply* $\Gamma' \vdash \mathbf{P}' \triangleright \Delta'$ *with* $(\Gamma, \Delta) \xrightarrow{\alpha} (\Gamma', \Delta')$.
4. $\Gamma \vdash \mathbf{N} \triangleright \Delta$ *and* $\mathbf{N} \longrightarrow \mathbf{N}'$ *imply* $\Gamma \vdash \mathbf{N}' \triangleright \Delta'$ *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 $\overset{\ell}{\Longrightarrow}$ for $(\xrightarrow{\tau})^* \xrightarrow{\ell} (\xrightarrow{\tau})^*$ and similarly for $\overset{\tilde{\ell}}{\Longrightarrow}$. The proof is straightforward by the definition in § 3.3.

Theorem 5.2 (CFSMs and Local Types). $(\mathbf{G} \upharpoonright \mathtt{p}) \overset{\tilde{\ell}}{\Longrightarrow}$ *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). The following theorem is derived by Theorems 5.1 and 5.2.

Theorem 5.3 (Type Safety). *Suppose* $\Gamma \vdash \mathbf{N}$. *For any* $\mathbf{N}'$ *such that* $\mathbf{N} \longrightarrow^* \mathbf{N}'$, $\mathbf{N}'$ *has no type error.*

Using Theorem 3.2, boundedness is derived as Theorem 5.4.

Theorem 5.4 (Boundedness). *Suppose for all* $\mathbf{G}$ *in* Γ, $\mathcal{A}(\{\mathbf{G} \upharpoonright \mathtt{p}_i\}_{1\le i\le n})$ *with* $\mathtt{p}_1,...,\mathtt{p}_n \in \mathbf{G}$ *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 [12] by applying the global buffer analysis on $\langle \mathbf{G} \rangle$ from [9].

5.3 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* [14, 24] 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 $\mathsf{def}\ \mathbf{x}_0(\tilde{x}) = x\langle \mathbf{G} \rangle.\mathbf{x}_1, \mathbf{x}_1 = \mathbf{0}\ \mathsf{in}\ \mathbf{x}_0(a)$ 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 or name creator. 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 5.1).

Theorem 5.5 (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 5.1.*

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 $1 \le j \le n$, 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). The following theorem can be proved by the deadlock-freedom of MSA (Theorem 3.1) and Completeness (Theorem 5.5) with Theorem 5.2.

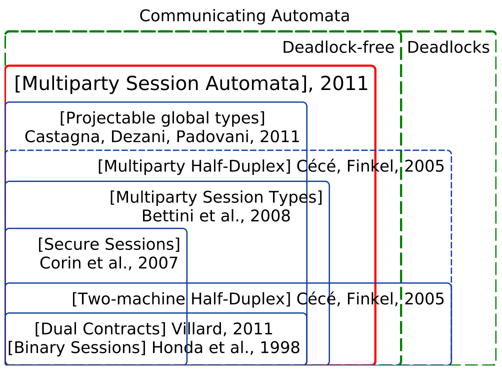
Theorem 5.6 (Deadlock Freedom). *Suppose* $\Gamma \vdash \mathbf{N}$ *is simple. Then there is no reduction such that* $\mathbf{N} \longrightarrow^* \mathbf{N}'$ *and* $\mathbf{N}'$ *is a deadlock.*

Below (1), is by Theorem 5.5 and (2) is by Theorems 5.2 and 5.5 with (1).

Theorem 5.7. (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* $\mathcal{A}(\{\mathbf{G} \upharpoonright \mathtt{p}_i\}_{1\le i\le n})$ *satisfies liveness with* $\mathtt{p}_1,...,\mathtt{p}_n \in \mathbf{G}$. *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* $\mathbf{N} \longrightarrow^* \mathbf{0}$.

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 $\mathbf{G}$.

6 Related Work



Communicating Automata
Deadlock-free
Deadlocks
[Multiparty Session Automata], 2011
[Projectable global types] Castagna, Dezani, Padovani, 2011
[Multiparty Half-Duplex] Cécé, Finkel, 2005
[Multiparty Session Types] Bettini et al., 2008
[Secure Sessions] Corin et al., 2007
[Two-machine Half-Duplex] Cécé, Finkel, 2005
[Dual Contracts] Villard, 2011
[Binary Sessions] Honda et al., 1998

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 [13] correspond to the set of compatible half-duplex deterministic two-machine systems without mixed states [12, 23] (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 [8], 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, 14], 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 [12] and Villard's extension [16] of [23] to unreliable systems, which proves that safety properties and boundedness are still decidable. These works [12, 16, 23] 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 [14]). 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 [23] 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 embed a single MSC. Many variants of MSGs are investigated in the literature [11] 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 [11, 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 (see [18]), 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.

7 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 [12] 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, 14], while keeping a polynomial tool chain. Our general choice is already included into Scribble 1.0 [20], 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 [15], relating MSA with models of true concurrency, including Mazurkiewicz traces, extending MSA to parameterisation [24], multiroles [10] and multiparty contracts [16, 23].

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. Corin, R., Deniélou, P.M., Fournet, C., Bhargavan, K., Leifer, J.: Secure implementations for typed session abstractions. In: CSF. pp. 170–186 (2007)
9. Deniélou, P.M., Yoshida, N.: Buffered communication analysis in distributed multiparty sessions. In: CONCUR'10. LNCS, vol. 6269, pp. 343–357. Springer (2010)
10. 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`
11. Genest, B., Muscholl, A., Peled, D.: Message sequence charts. In: Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 537–558 (2004)
12. Gouda, M., Manning, E., Yu, Y.: On the progress of communication between two finite state machines. Information and Control. 63, 200–216 (1984)
13. 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)
14. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL'08. pp. 273–284. ACM (2008)
15. Kouzapas, D., Yoshida, N., Honda, K.: On asynchronous session semantics. In: FMOODS/-FORTE. LNCS, vol. 6722, pp. 228–243 (2011)
16. Lozes, E., Villard, J.: Reliable contracts for unreliable half-duplex communications. In: WS-FM. Springer (2011), to appear
17. 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)
18. Online Appendix, `http://www.doc.ic.ac.uk/~malo/msa/`
19. Savara JBoss Project, `http://www.jboss.org/savara`
20. Scribble JBoss Project, `http://www.jboss.org/scribble`
21. Sivaramakrishnan, K.C., Nagaraj, K., Ziarek, L., Eugster, P.: Efficient session type guided distributed interaction. In: COORDINATION. LNCS, vol. 6116, pp. 152–167 (2010)
22. 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)
23. Villard, J.: Heaps and Hops. Ph.D. thesis, ENS Cachan (2011)
24. Yoshida, N., Deniélou, P.M., Bejleri, A., Hu, R.: Parameterised multiparty session types. In: FoSSaCs. LNCS, vol. 6014, pp. 128–145 (2010)