Mixed Choice in Asynchronous Multiparty Session Types

LAURA BOCCHI, University of Kent, UK
RAYMOND HU, Queen Mary University of London, UK
ADRIANA LAURA VOINEA, University of Glasgow, UK
SIMON THOMPSON, University of Kent, UK

We present a multiparty session type (MST) framework with *asynchronous mixed choice* (MC). We propose a core construct for MC that allows transient inconsistencies in protocol state between distributed participants, but ensures all participants can always eventually reach a mutually consistent state. We prove the correctness of our system by establishing a progress property and an operational correspondence between global types and distributed local type projections. Based on our theory, we implement a practical toolchain for specifying and validating asynchronous MST protocols featuring MC, and programming compliant `gen_statem` processes in Erlang/OTP. We test our framework by using our toolchain to specify and reimplement part of the `amqp_client` of the RabbitMQ broker for Erlang.

CCS Concepts: • **Theory of computation** → **Distributed computing models**; **Type structures**; • **Software and its engineering** → *Distributed programming languages.*

Additional Key Words and Phrases: Mixed Choice, Asynchronous Multiparty Session Types, Type Systems

ACM Reference Format:
Laura Bocchi, Raymond Hu, Adriana Laura Voinea, and Simon Thompson. 2026. Mixed Choice in Asynchronous Multiparty Session Types. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 148 (April 2026), 28 pages. https://doi.org/10.1145/3798256

1 Introduction

Multiparty session types (MST) [Honda et al. 2008] is a typing discipline for concurrent processes that interact via message passing in communication sessions. The main idea is that an MST communication protocol can be statically checked for *communication safety*, i.e., freedom from fundamental errors such as reception errors (receiving unexpected messages), deadlocks (wait-for cycles) and orphan messages (messages that the receiver will never attempt to consume).

MST is an active area of research due to its potential to offer programmatic techniques for safe specification and lightweight verification of communication protocols in concurrent and distributed systems. The key challenges being tackled include expressiveness of the types, tractability of the metatheory, and practicality of session-based programming and verification methods. These challenges are accentuated in the setting of distributed systems (DS) where communications are inherently asynchronous and failure is the norm.

This paper tackles a crucial problem that concerns all of the above challenges: the notion of *mixed choice* in *asynchronous* MST. In classical MST [Bettini et al. 2008; Coppo et al. 2015; Honda et al. 2008], the construct for choice in a protocol, called a *directed choice*, looks as follows.

$$\mathsf{p} \twoheadrightarrow \mathsf{q} : \{a_i.\mathsf{G}_i\}_{i \in I}$$

Authors' Contact Information: Laura Bocchi, L.Bocchi@kent.ac.uk, University of Kent, Canterbury, UK; Raymond Hu, r.hu@qmul.ac.uk, Queen Mary University of London, London, UK; Adriana Laura Voinea, laura.voinea@glasgow.ac.uk, University of Glasgow, Glasgow, UK; Simon Thompson, S.J.Thompson@kent.ac.uk, University of Kent, Canterbury, UK.

ACM 2475-1421/2026/4-ART148
https://doi.org/10.1145/3798256

It specifies that a participant p makes an *internal* choice to send one of the a_i messages to q and continue in protocol G_i. Participant q receives the a_i as an *external* choice and continues in G_i correspondingly. Some systems (e.g., [Lange and Yoshida 2019; Li et al. 2023]) support generalised choice constructs that allow the a_i to be sent to different q_i. However, the key point remains that the choice is directed by p.

By contrast, this paper develops the following construct for *mixed* choice.

$$\mathsf{q} \twoheadrightarrow \mathsf{p} : a_1.\mathsf{G}_1 \quad \triangleright \quad \mathsf{p} \twoheadrightarrow \mathsf{q} : a_2.\mathsf{G}_2$$

It specifies that p and q each independently face a choice between a *mix* of input and output actions: q faces a mixed choice between sending a_1 on the left and receiving a_2 on the right, and vice versa for p. In an asynchronous setting, this means that q may opt to send a_1 and continue in protocol G_1 *concurrently with* p opting to send a_2 and continue in G_2. In this way, asynchronous mixed choices inherently describe a form of race condition, which classical MST intentionally prohibits outright because directed choice syntactically partitions all choices as input or output only.

Yet such race conditions are useful and important in many real applications. Indeed, much recent research on improving the expressiveness and practicality of MST has touched on aspects of mixed choice, including work on *exceptions* [Capecchi et al. 2016; Fowler et al. 2019; Viering et al. 2018], *interrupts* [Chen et al. 2016; Demangeon et al. 2015], *timing* [Iraci et al. 2023] and *timeouts* [Hou et al. 2024; Pears et al. 2023], *failure handling* [Adameit et al. 2017; Barwell et al. 2023, 2022; Brun and Dardha 2024], and *fault-tolerance* [Peters et al. 2023; Viering et al. 2021]. These works (implicitly) involve patterns where a participant has the option on one hand to asynchronously output a message, while on the other hand it is simultaneously prepared for some input event, say, catching a concurrent channel exception, or receiving a criss-crossing interrupt or timeout message, or handling the failure of some other participant.

The fundamental problem in reasoning about mixed choice in MST is that it allows participants to diverge in their local views of the protocol during execution. The insight of this paper is that communication safety can be achieved despite *transient* inconsistencies in distributed protocol state depending on how participants react to race-y messages and interact to resolve conflicts.

Contributions and roadmap. This paper presents the following contributions.

- We introduce **mMST**, the first theory of global and local MST with an *explicit* construct for *asynchronous mixed choice*. To date, mixed choices in asynchronous sessions have only been partially expressed through specific-purpose constructs for exceptions, failure handling and so forth. By contrast, we present a general-purpose core theory of mixed choices that captures and unifies the fundamental essence of such ad hoc constructs. We propose an *asymmetric* design for mixed choice that statically ensures participants can eventually resolve the inherent race conditions through explicit interactions and agree on how the protocol should proceed.
- We establish the correctness of mMST by proving a progress property for our mixed choice types, and an operational correspondence between global types and the corresponding system of local types. Together these guarantee that every non-terminated participant in an mMST protocol can always progress and that its behaviour is always protocol-compliant. Our results lay general foundations for a more flexible and practical concept of multiparty sessions that allows participants to deal with discrepancies in their distributed views of the protocol and safely converge on a consistent outcome.
- We apply our theory by implementing a prototype toolchain for specifying and programming mMST-based protocols as `gen_statem` programs in Erlang/OTP. We test the expressiveness and practicality of mMST by using our toolchain to implement coordination protocols in the Erlang client of the RabbitMQ message broker, as well as a selection of examples from MST literature augmented with mixed choices.

```
1  // Exception, Interrupt, etc. similar
2  global protocol Timeout(
3        role A, role B, role C) {
4    mixed { // "Left-hand" side (LHS)
5      a1() from A to B;
6      a2() from A to C;
7      a3() from B to C;
8      a4() from B to A;
9      a5() from C to A;
10   } or { // "Right-hand" side (RHS)
11     TOa() from B to A;
12     TOc() from B to C;
13 } }
```

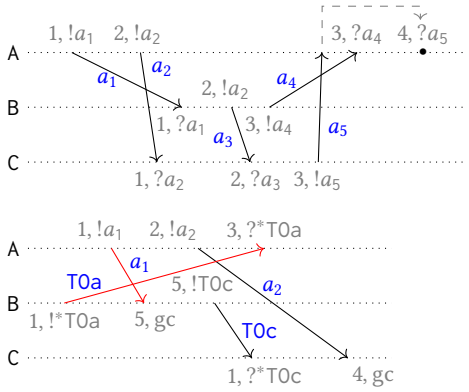


Fig. 1. A multiparty timeout pattern as a safe mMST protocol using asynchronous mixed choice.

Section 2 gives a high-level overview of our mixed choice construct. **Sections 3 and 4** present our theory and its formal properties. **Section 5** describes our toolchain for Erlang, our RabbitMQ use case and other examples. **Section 6** discusses related work, limitations and future work. Full details and proofs are available in the online long version [Bocchi et al. 2026a].

2 Overview

Our overall framework comprises two main stages.

- Specification and static validation of a source mMST protocol based on our formal theory.
- Implementation of each role in the asynchronous protocol in Erlang using correct-by-construction modules generated from the source protocol.

The following two subsections illustrate the key concepts in each stage using a practical example.

2.1 Asymmetric Mixed Choice in Asynchronous MST

Our toolchain takes mMST protocols written in our extension of the Scribble protocol language [Hu and Yoshida 2016; Yoshida et al. 2013]. Communications are asynchronous: interactions are non-blocking on the sender side, while the receiver side blocks until a message is available for reading. This means a sender moves ahead in the protocol immediately after dispatching a message without waiting for the message to be received. Messages are delivered in order of dispatch in each direction between each pair of roles; receivers read messages from the expected sender in a FIFO manner. This model reflects our target Erlang programs and wider domains such as TCP-based Internet applications and Web services.

Figure 1 (left) illustrates a small protocol called `Timeout` involving three participants, whose behaviour is abstracted in MST as *roles*. The syntax of our *mixed choice* (MC) construct is:

```
mixed { /* "Left-hand" side (LHS) */ } or { /* "Right-hand" side (RHS) */ }
```

The protocol features an MC between role `A` on the LHS and `B` on the RHS. It expresses a typical timeout pattern where `A` has the option to send message `a1` to `B` (and proceed asynchronously) on the LHS, but it must also be prepared to handle the potentially concurrent timeout message `TOa` from `B` on the RHS. Conversely, `B` has the option to wait for the `a1` message on the LHS, or give up waiting and asynchronously send `TOa` on the RHS. Depending on how `A` and `B` proceed, role `C` must be prepared to handle one or both of the `a2` from `A` and the `TOc` from `B`.

We summarise the key concepts in the design of our MC and how we ensure MST safety.

- Our MC is an *asymmetric* construct. In each MC, we designate the sender on the RHS as a special role that we refer to as the *observer* of the MC. The LHS can be considered a default or speculative branch, which can be asynchronously superceded by the RHS on the instigation of the observer.
- We identify a notion of *commitment* of roles to an MC branch (LHS or RHS). Commitment means that the inherent race condition of an MC has been resolved from the perspective of that role and it knows the protocol will henceforth proceed only in that branch.
  An MC starts with no roles committed to either branch. Regardless of (speculative) interactions between other roles in the LHS, the first action by the observer in an MC (input on the LHS or output on the RHS) commits the observer to that branch. Any subsequent action by another role `r`, where the action causally depends on the observer, commits `r` to the same branch.
- Protocol validation due to our formal theory ensures that however a protocol (speculatively) proceeds, all eventual commitments are monotonic and always consistent with the observer.

To illustrate, Figure 1 (right) depicts two possible executions of the protocol. For now, the reader can focus on the arrows and the blue labels; the grey annotations will be explained in Sec. 2.2.

- The upper chart is a run where observer `B` opts to receive `a1` and follow `A` on the LHS, i.e., it does not raise the timeout. While `A` and `C` may proceed asynchronously on the LHS, the observer `B` is the first role to commit when it consumes the `a1`. In turn `C` commits to the LHS when it consumes the `a3` from `B`, and `A` commits when it consumes the `a4`. Although the `a5` can arrive on the LHS at `A` before the `a4`, `A` must consume the `a4` (and commit to the LHS) first following the protocol.
- The lower chart is a run where observer `B` opts to overrule `A` and raise the timeout by sending `TOa`, which commits `B` to the RHS. Although `A` and `C` may be concurrently engaged in interactions on the LHS, role `A` eventually receives the `TOa`, causing it to switch from the LHS and commit to the RHS. Similarly, `C` switches and commits to the RHS when it receives the `TOc`.

Stale message purging. The latter of the above cases demonstrates that supporting mixed choices safely in asynchronous MST requires one further key concept. Due to asynchrony, actions performed by one role may concurrently render other messages that are buffered or being sent obsolete. For instance, in the lower chart, the `a1` message 'criss-crosses' with `TOa` and arrives at `B` after `B` has already committed to the RHS. From `B`'s perspective, sending the `TOa` renders the (already in-transit) `a1` obsolete; in such cases, we refer to messages like `a1` as *stale* messages. Similarly, the receipt of `TOc` by `C` renders the `a2` stale.

Our theory formalises a runtime mechanism that allows stale messages to be automatically purged *transparently* to the user program; e.g., in the case mentioned above, the runtime at `B` (resp. at `C`) will transparently purge the `a1` that arrives after sending `TOa` (resp. the `a2` after receiving `TOc`). Crucially, stale message purging can be safely performed by the *local* runtime of each distributed participant using only knowledge available to that participant. The notion of stale messages and purging never arises in classical MST. Our transparent purging mechanism could be considered a message passing analogy to garbage collection of stale objects in programming languages with automated memory management (such as Erlang).

MC protocol validation. For reference, the `Timeout` protocol written in our formal notation for *global types* (Section 3) is as follows. The ▷ separates the LHS and RHS of the MC. The additional blue colouring indicates the points at which the roles become committed.

$$\mathtt{A} \twoheadrightarrow \mathtt{B} : a_1 \,.\, \mathtt{A} \twoheadrightarrow \mathtt{C} : a_2 \,.\, \mathtt{B} \twoheadrightarrow \mathtt{C} : a_3 \,.\, \mathtt{B} \twoheadrightarrow \mathtt{A} : a_4 \,.\, \mathtt{C} \twoheadrightarrow \mathtt{A} : a_5 \,.\, \mathtt{end} \;\triangleright\; \mathtt{B} \twoheadrightarrow \mathtt{A} : \mathtt{TOa} \,.\, \mathtt{B} \twoheadrightarrow \mathtt{C} : \mathtt{TOc} \,.\, \mathtt{end}$$ ✓

Our static protocol validation ensures that role commitments are always consistent with the observer and that role termination is safe (as illustrated in the earlier example runs). By contrast, the below (renaming `TOc`) is rejected because commitment is ambiguous for `C`.

$\mathtt{A} \twoheadrightarrow \mathtt{B} : a_1 \,.\, \mathtt{A} \twoheadrightarrow \mathtt{C} : a_2 \,.\, \mathtt{B} \twoheadrightarrow \mathtt{C} : a_3 \,.\, \mathtt{B} \twoheadrightarrow \mathtt{A} : a_4 \,.\, \mathtt{C} \twoheadrightarrow \mathtt{A} : a_5 \,.\, \mathtt{end} \;\triangleright\; \mathtt{B} \twoheadrightarrow \mathtt{A} : \mathtt{TOa} \,.\, \mathtt{B} \twoheadrightarrow \mathtt{C} : a_3 \,.\, \mathtt{end}$ ✗

The below (dropping the a_3) is also rejected because C reaches end without committing on the LHS.

$$\mathtt{A} \twoheadrightarrow \mathtt{B} : a_1 \,.\, \mathtt{A} \twoheadrightarrow \mathtt{C} : a_2 \,.\, \mathtt{B} \twoheadrightarrow \mathtt{A} : a_4 \,.\, \mathtt{C} \twoheadrightarrow \mathtt{A} : a_5 \,.\, \mathtt{end} \;\triangleright\; \mathtt{B} \twoheadrightarrow \mathtt{A} : \mathtt{TOa} \,.\, \mathtt{B} \twoheadrightarrow \mathtt{C} : \mathtt{TOc} \,.\, \mathtt{end}$$ ✗

However, the below (dropping the a_4) is accepted; A can safely commit on the LHS because it has a transitive causal dependency with observer B via C.

$$\mathtt{A} \twoheadrightarrow \mathtt{B} : a_1 \,.\, \mathtt{A} \twoheadrightarrow \mathtt{C} : a_2 \,.\, \mathtt{B} \twoheadrightarrow \mathtt{C} : a_3 \,.\, \mathtt{C} \twoheadrightarrow \mathtt{A} : a_5 \,.\, \mathtt{end} \;\triangleright\; \mathtt{B} \twoheadrightarrow \mathtt{A} : \mathtt{TOa} \,.\, \mathtt{B} \twoheadrightarrow \mathtt{C} : \mathtt{TOc} \,.\, \mathtt{end}$$ ✓

Altogether, the design of our asynchronous MC, the concepts of observer and commitment and the protocol validation guarantee that every role that has not safely terminated will continue progressing through the protocol as expected. As an advance pointer, the main conditions checked by the protocol validation (which we refer to as *awareness*) and the *progress* property for global types are formally defined and established in Section 3.3.

On the expressiveness of mMST, consider again the criss-crossing pattern between A and B in the lower chart of Figure 1 (right). By contrast, the conservative syntactic structure of classical MST [Coppo et al. 2016; Honda et al. 2016] implicitly precludes all such patterns where asynchronous messages criss-cross between a pair of roles in opposing directions. The complications introduced by these patterns – such as the inherent race conditions, stale messages, and the mechanisms required to resolve these – are why reasoning about safety of mixed choices in asynchronous MST (and DS in general!) is a difficult challenge, and has remained an open problem for MST.

We have used the small Timeout example to introduce our MC and the key concepts. Nevertheless, it demonstrates how mMST supports the fundamental communication pattern underlying various important DS constructs, including exceptions, interrupts and failure handling, as found in many real-world applications. Our MC provides a *core* building block for expressing these constructs in MST. Sections 3 and 5.2 include and summarise a range of further such examples that involve combining MC with the standard directed choice of MST, recursion and nested MCs.

2.2 Mixed Choice MST Protocols in Erlang

Erlang is a concurrent, dynamically typed functional programming language designed for implementing applications out of message passing processes. It is used in major applications and platforms such as WhatsApp and RabbitMQ. Its emphasis on building fault-tolerant distributed systems makes it a good target for applying and testing mMST.

Erlang/OTP has built-in support for a set of core design patterns, known as *behaviours*, for implementing processes. In this paper, we target the gen_statem behaviour that provides a generic framework for implementing state machines. A process is formed by combining the provided generic state machine behaviour module with a user-written module of *callback* functions for handling events and state transitions according to the required application-specific logic.

Based on our theory, we have implemented a toolchain for specifying and implementing mMST protocols as gen_statem processes. Our toolchain involves these steps.

- The user writes the source *global* protocol (e.g., Timeout in Figure 1) using our mMST extension of Scribble and uses the tool to validate it based on our formal conditions.
- The tool internally *projects* valid global protocols to a *local* protocol for each role. The tool represents a local protocol as an *Event-Driven Finite State Machine* (EFSM) that matches the programming abstractions of gen_statem (see below).
- From each EFSM, the tool generates (i) a correct-by-construction *protocol- and role-specific* gen_statem behaviour module, and (ii) a corresponding template callback module for the programmer to use and adapt as required to complete the process definition.

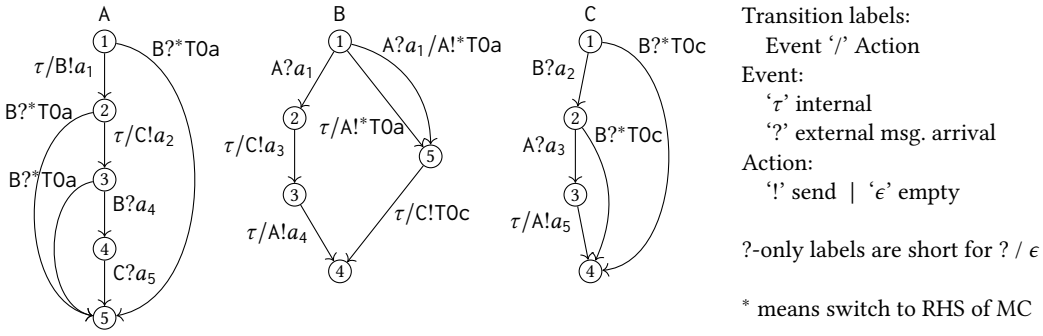


Fig. 2. Local mMST protocols for each role of `Timeout` as EFSMs in Erlang `gen_statem`.

Global-to-local projection. A local protocol is the view of the protocol from a specific role. Projecting a global protocol to a set of local protocols, one for each role, yields a distributed model. In our formal notation, the projected *local types* for `Timeout` are:

$$\texttt{Timeout} \upharpoonright \texttt{A} = \texttt{B} \oplus a_1 \,.\, \texttt{C} \oplus a_2 \,.\, \texttt{B} \,\&\, a_4 \,.\, \texttt{C} \,\&\, a_5 \,.\, \texttt{end} \;\triangleright\; \texttt{B} \,\&\, \texttt{TOa} \,.\, \texttt{end}$$
$$\texttt{Timeout} \upharpoonright \texttt{B} = \texttt{A} \,\&\, a_1 \,.\, \texttt{C} \oplus a_3 \,.\, \texttt{A} \oplus a_4 \,.\, \texttt{end} \;\triangleright\; \texttt{A} \oplus \texttt{TOa} \,.\, \texttt{C} \oplus \texttt{TOc} \,.\, \texttt{end}$$
$$\texttt{Timeout} \upharpoonright \texttt{C} = \texttt{A} \,\&\, a_2 \,.\, \texttt{B} \,\&\, a_3 \,.\, \texttt{A} \oplus a_5 \,.\, \texttt{end} \;\triangleright\; \texttt{B} \,\&\, \texttt{TOc} \,.\, \texttt{end}$$

The $\triangleright$ separates the LHS and RHS of the projected mixed choices. The $\oplus$ denotes an output-only choice (internal select), and & denotes the dual input-only choice (external branch); in this simple example, the input/output-only choices are all unary. Section 4.3 proves that the behaviour of a projected local system corresponds to that of the global protocol. Projection thus entails that a protocol is realisable as a distributed system.

EFSM representation. The core abstraction of `gen_statem` is an *Event-Driven Finite State Machine* (EFSM). Transitions are described by[1]

$$\text{State (S)} \times \text{Event (E)} \longrightarrow \text{Action (A)} \times \text{State (S')}$$

which can be read: if the *current state* is S and *event* E occurs, then perform *action* A and transition to *successor state* S' – i.e., the transition is triggered by E and A is a consequent effect. Events may have external sources, such as the arrival of a message (?), or internal (τ), such as a local computation. Actions include sending (!) messages and the empty action (ϵ). In our mMST setting, a transition can also have the effect of switching from the LHS of an MC to commit to the RHS.

Figure 2 depicts the EFSM generated by our tool for each role of `Timeout`. The notation e/α on the transitions means event e triggers the transition and action α is performed; an e on its own is shorthand for e/ϵ. For example, the $\tau/\texttt{B}!a_1$ from state 1 of A means that A can internally decide to send a_1 to B (and transition to 2). In B, the two transitions from state 1 with event $\texttt{A}?a_1$ mean that if message a_1 arrives as an external event, B can either do $\texttt{A}!^*\texttt{TOa}$ and transition to 4, or transition to 2 (with the empty action). Note, the * annotation (as also occurs in the text below) indicates the transitions that switch from the LHS to the RHS of the MC.

Classical MST permits only three kinds of local states: input-only (branch), output-only (select), and terminal; e.g., state 4 in A is a branch, and 3 in C is a select (again, both unary in this simple example). By contrast, the more expressive local behaviours enabled by our asynchronous MC can also be seen in the above EFSMs. As mentioned, observer B faces an *internal MC* in state 1 indicated by the *mix* of $\tau/!^*$ and $?/\epsilon$ transitions induced by its RHS and LHS, respectively, yielding the timeout behaviour described at the start of Section 2.1. On the other hand, A faces an *external*

[1] https://www.erlang.org/doc/system/statem.html

```
% Extract from role A
s1(internal, a1, Data) ->
  gen_a:send_s1_a1(BPid, Data), {next_state, s2,
    Data, [{next_event, internal, a2}]};
s1(cast, {BPid, 'TOa'}, Data) -> {stop, normal, Data}.

s2(internal, a2, Data) ->
  gen_a:send_s2_a2(CPid, Data), {next_state, s3, Data};
s2(cast, {BPid, 'TOa'}, Data) -> {stop, normal, Data}.

s3(cast, {BPid, a4}, Data) -> {next_state, s4, Data};
s3(cast, {BPid, 'TOa'}, Data) -> {stop, normal, Data}.

s4(cast, {CPid, a5}, Data) -> {stop, normal, Data}.


% Extract from role B
s1(internal,'TOa', Data) ->
case make_choice_TOa(Data) of
  1 -> {keep_state, Data};
  2 -> gen_b:send_s1_TOa(APid, Data), {next_state, s3,
         Data, [{next_event, internal,'TOc'}]}
end; % Continued on the right
```

```
% Continued from the left
s1(cast,{APid, a1},Data) ->
  case make_choice_a1(Data) of
    1 -> {next_state, s2, Data,
          [{next_event, internal, a3}]};
    2 -> gen_b:send_s1_TOa(APid, Data),
          {next_state, s5, Data,
            [{next_event, internal,'TOc'}]}
  end.

s5(internal, 'TOc', Data) ->
  gen_b:send_s5_TOc(CPid, Data),
    {stop, normal, Data}.

s2(internal, a3, Data) ->
  gen_b:send_s2_a3(CPid, Data), {next_state, s3,
    Data, [{next_event, internal, a4}]}.

s3(internal, a4, Data) ->
  gen_b:send_s3_a4(APid, Data),
    {stop, normal, Data}.
```

Fig. 3. Extracts from the user-facing callback modules generated for A and B in the Timeout protocol.

MC in state 1 as indicated by its mix of $?^*/\epsilon$ and $\tau/!$ transitions: A can either (wait to) receive the TOa (and switch to the RHS), or internally decide (by some internal event τ) to send a_1 and proceed to 2. Our formal semantics allows both options, but an implementation may (e.g.) give priority to the former case if TOa has already arrived. Note how A is forced to eventually commit to either the LHS or RHS (in accordance with B) by state 3 at the latest by receiving a_4 or TOa.

We give a couple of further comments on the richer behaviours expressed by our MC. First, note that external MCs may take various forms: e.g., both states 1 in A and 1 in C are external MCs featuring $?^*$, but the former is mixed with a τ whereas the latter is mixed with an ?-event. Second, note that the top-level roles of the MC are A and B, but their localised (projected) behaviours in the MC are not direct duals; e.g., state 3 in A is an external MC between a_4 and TOa that has no *direct* counterpart in B (i.e., there is no internal MC featuring a_4 in B). Additionally, $?^*$TOa events are spread over states 1, 2 and 3 in A; whereas in B, the $!^*$TOa occurs only in 1, although in two different transitions since B can choose to raise the timeout either before (τ) or after (?) the a_1 has arrived. These behaviours are considerably more exotic than those supported in classical MST and speak to the challenge of reasoning about asynchronous MC theoretically and practically.

Programming mMST-based processes in Erlang. Our toolchain uses the EFSMs to generate two Erlang modules per role for the programmer(s) to work with. The *role module* (RM) provides a customised gen_statem behaviour that is specialised to the source protocol and role. The programmer can consider it as part of the runtime and does not need to use it directly. The other, called the *callback module* (CM), is a template module of callback functions that the RM delegates to for handling event occurrences and performing state transition actions. The CM is generated with minimal placeholder code that the programmer can modify and extend with the required application-specific logic. The RM and CM together form a protocol- and role-specific mMST process. A set of processes comprising one for each role forms a complete application.

Figure 3 shows code from the CMs as generated for roles A and B. As per their respective EFSMs (Figure 2), they feature callback *state functions* (e.g., s1, s2, etc.) for handling state transitions

according to event occurrences. The callbacks are fired (from the RM) based on the current state and by pattern matching against the event, which may come from an internal (locally triggered transitions, such as timeout decisions) or external (incoming messages) source.

For instance, in `A`, state function `s1` represents an external MC corresponding to state 1 (for `A` in Figure 2): `A` may either send `a1` to `B` (the `internal` clause), or receive `'TOa'` from `B` (the `cast` clause), corresponding to the τ/B!a_1 and B?*TOa transitions, respectively. In `B`, the function `s1` represents an internal MC. In the first clause (`internal`), `B` handles the internal event from the generated template function `make_choice_TOa` by which it decides whether to remain in the current state (`keep_state`) and wait to receive `a1`, or transition to `s2` by sending `'TOa'` to `A` (cf. τ/A!*To). Generally, the programmer will modify/replace such template decision functions according to the required application logic. In the second clause (`cast`), `B` receives the message `a1` from `A`, and based on template function `make_choice_a1(Data)`, either transitions (cf. A?a_1/A!*TOa) to `s3` and schedules an internal event `a2`, or sends `'TOa'` to `A` and transitions (cf. A?a_1) to `s2`.

Section 5 demonstrates the internals of the RM that fires the callbacks in the above CM. In short, our tool generates the RM by instantiating the default `gen_statem` behaviour with the required EFSM structure (states, events, actions, and transitions); this is correct-by-construction and the programmer should not modify the RM. Following our formal semantics, the RM is also generated to encapsulate the (again, correct-by-construction) runtime mechanisms for handling MC commitment, LHS-to-RHS switching, and stale message purging specifically for the source protocol and role. The programmer can assume these mechanisms are provided and correct when working on the CM.

Recall the example executions in Figure 1 (right). The grey annotations denote the current state and relevant event/action at each role according to their EFSMs in Figure 2; e.g., $1, !a_1$ means in state 1, send message a_1. Stale message purging, denoted by gc in Figure 1, is handled internally by the RM; we exclude purge actions from our depictions of EFSMs as they are transparent to the programmer. User processes do not consume stale messages, so they must be purged from the input buffer to prevent interference with future receives (given the FIFO nature of inputs).

2.3 Properties of mMST

We end this overview by summarising the properties of our framework with advance pointers (a roadmap) to the relevant parts of our formal theory.

Section 3 formalises the syntax and metatheoretical LTS semantics of our *global types* with mixed choice and the notion of *committing* messages in MCs. It defines the conditions checked by protocol validation on protocol structure (*well-formedness* of committing messages, *balance* of roles across choice branches) and on inter-role dependencies in MCs (*awareness*). Section 3.3 proves that valid protocols enjoy a per-role *progress* property and that their roles are always consistent in their commitments (*coherence*).

Section 4 formalises the syntax and LTS semantics of distributed *local types* and asynchronous message queues, the *projection* from global to local types, and the runtime mechanism for *stale message purging*. Section 4.3 proves an operational correspondence in both directions between valid projectable global types and local systems that *preserves projection*. The operational correspondence (i.e., preservation of projection) and progress properties together safely entail that local roles never get stuck in a deadlock nor (noting the FIFO nature of communications in the local type LTS) due to receiving an unexpected message. Section 4.4 further proves orphan message freedom.

As discussed, our practical toolchain is implemented to perform protocol validation (conservatively, see Section 5.1) and projection following our formal theory. It is implemented to perform a correct-by-construction translation from local projections to EFSMs and generation of `gen_statem` modules to transfer the above correctness properties to Erlang processes. The usage contract is that the programmer should not modify the generated RM and must use the CM according to

the generated structures. The RM is generated with some internal runtime checks against the programmer supplying an invalid CM.

3 Global Types

The syntax of global types $\mathtt{G}$ is defined by the grammar below:

$$\begin{array}{lll} \mathtt{G} & ::= & \mathtt{p} \twoheadrightarrow \mathtt{q} : S \mid \mathtt{p} \rightsquigarrow \mathtt{q} : k\, S \mid \mu \mathtt{t}.\mathtt{G} \mid \mathtt{t} \mid \mathsf{end} \mid \mathtt{q} \twoheadrightarrow \mathtt{p} : S_1 \triangleright^{c} \mathtt{p} \twoheadrightarrow \mathtt{q} : S_2 \mid \mathtt{G}_1 \blacktriangleright^{c}_{\mathcal{L},\mathcal{R}} \mathtt{G}_2 \\ S & ::= & \{a_i.\mathtt{G}_i\}_{i \in I} \end{array}$$

The first five terms are standard [Deniélou and Yoshida 2013]. We recall that $\mathtt{p} \twoheadrightarrow \mathtt{q} : S$ is an interaction where $\mathtt{p}$ is the sender and $\mathtt{q}$ is the receiver. Term S specifies a set of choices: $\mathtt{p}$ can send $\mathtt{q}$ one of the labels a_i (for $i \in I$) and the protocol continues as $\mathtt{G}_i$. The message in-transit type $\mathtt{p} \rightsquigarrow \mathtt{q} : k\, S$ stands for a state where $\mathtt{p}$ has sent label a_k but $\mathtt{q}$ has not yet received it (asynchronous communication). Term $\mu \mathtt{t}.\mathtt{G}$ is a recursive definition[2] and $\mathtt{t}$ a recursive variable.

The last two terms are new and model *mixed choices* (MC). The term $\mathtt{q} \twoheadrightarrow \mathtt{p} : S_1 \triangleright^{c} \mathtt{p} \twoheadrightarrow \mathtt{q} : S_2$ is an *MC definition*: we call $\mathtt{q} \twoheadrightarrow \mathtt{p} : S_1$ the left-hand side block (LHS) of the MC and $\mathtt{p} \twoheadrightarrow \mathtt{q} : S_2$ the right-hand side block (RHS). Role $\mathtt{p}$ is set to receive a message from $\mathtt{q}$ in the LHS, but may decide to execute the RHS by sending a message to $\mathtt{q}$ instead. We say that $\mathtt{p}$ is the *observer* of the MC. We annotate MC definitions with a unique name c, which we may omit when it is not important.

The term $\mathtt{G}_1 \blacktriangleright^{c}_{\mathcal{L},\mathcal{R}} \mathtt{G}_2$ is an *active MC*. To ensure that roles eventually agree on which block (LHS or RHS) to execute, we use a notion of *commitment*. Some actions are designated as committing (defined in Section 3.1). When a role executes a committing action within a block, it commits to that block, meaning that it can no longer perform any action (send or receive) in the other block. The semantics of global types use sets $\mathcal{L}$ to keep track of the roles that are committed to the LHS, and $\mathcal{R}$ to keep track those committed to the RHS. For convenience we may use the notation $\mathtt{G}_1 \triangleright^{c:\mathtt{p}} \mathtt{G}_2$ and $\mathtt{G}_1 \blacktriangleright^{c:\mathtt{p}}_{\mathcal{L},\mathcal{R}} \mathtt{G}_2$ assuming that $\mathtt{G}_1$ and $\mathtt{G}_2$ are of the form given by the grammar and that $\mathtt{p}$ is the observer. For readability, we omit annotations c, $\mathcal{L}$ or $\mathcal{R}$ when not needed. A global type is *initial* if it has no messages in transit and no active MC.

The set of roles of $\mathtt{G}$, denoted $\mathtt{R}(\mathtt{G})$, is defined as usual except for the two new cases for MC:

$$\mathtt{R}(\mathtt{G}_1 \triangleright \mathtt{G}_2) = \mathtt{R}(\mathtt{G}_1) \cup \mathtt{R}(\mathtt{G}_2) \qquad \mathtt{R}(\mathtt{G}_1 \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathtt{G}_2) = \mathtt{R}(\mathtt{G}_1) \setminus \mathcal{R} \cup \mathtt{R}(\mathtt{G}_2) \setminus \mathcal{L}$$

The roles of a MC definition are the roles in either of its sides. The case of active MC excludes the roles that have committed to the opposite side. This will be critical when defining progress, to characterise the roles that should continue on each side. We define a context environment $\mathbf{C}$:

$$\mathbf{C} ::= [\,\_\,] \mid \mathtt{p} \twoheadrightarrow \mathtt{q} : S \cup \{a.\mathbf{C}\} \mid \mathtt{p} \rightsquigarrow \mathtt{q} : k\, S \cup \{a_k.\mathbf{C}\} \mid \mathbf{C} \blacktriangleright \mathtt{G} \mid \mathtt{G} \blacktriangleright \mathbf{C} \mid \mathbf{C} \triangleright \mathtt{G} \mid \mathtt{G} \triangleright \mathbf{C} \mid \mu \mathtt{t}.\mathbf{C}$$

We say that $\mathtt{G}'$ is a *subterm* of $\mathtt{G}$ (or *is in* $\mathtt{G}$) if there exists $\mathbf{C}$ such that $\mathtt{G} = \mathbf{C}[\mathtt{G}']$. We say that two subterms of $\mathtt{G}_1$ and $\mathtt{G}_2$ of $\mathtt{G}$ are *distinguished* if $\mathtt{G} = \mathbf{C}_1[\mathtt{G}_1] = \mathbf{C}_2[\mathtt{G}_2]$ implies $\mathbf{C}_1 \neq \mathbf{C}_2$. We say that $\mathtt{G}$ *has an active MC* $\mathtt{G}_1 \blacktriangleright \mathtt{G}_2$ if $\mathtt{G}_1 \blacktriangleright \mathtt{G}_2$ is a subterm of $\mathtt{G}$. Similarly for MC definitions.

3.1 Committing Set and Well-Formedness

In this section we formally introduce the notion of commitment, by defining the *committing set* of a MC named c, that is the set of actions by which some role commits to either the LHS or to the RHS of c. The committing set is defined on initial global types. Intuitively, the committing actions of a MC are: (1) on the LHS, the first receive action by the observer, and all receive actions of a message from a committed sender, (2) on the RHS, all the first actions of a role on that side. For simplicity, we identify committing actions (e.g., $\mathtt{pq}?a$) using only their labels (e.g., a). The committing set is therefore, a set of communication labels.

[2]We adopt iso-recursive types for a lower level view of nested MC instantiation and straightforward implementation.

In the following we denote with $\mathtt{U}(\mathtt{G})$ the standard *unfold-all-once* operation that unfolds once all recursive types in $\mathtt{G}$. We say that $\mathtt{G}_1 \triangleright^c \mathtt{G}_2$ is the *outermost occurrence* of c in $\mathtt{U}(G)$ if it is not a subterm of other MC definitions c in $\mathtt{U}(G)$. Given a global type $\mathtt{q} \twoheadrightarrow \mathtt{p} : S$ with $S = \{a_i.\mathtt{G}_i\}_{i\in I}$, we use the notation $\mathtt{labels}(S) = \{a_i\}_{i\in I}$ and $\mathtt{types}(S) = \{\mathtt{G}_i\}_{i\in I}$.

Definition 3.1 (Committing set). Let $\mathtt{G}$ be an initial global type. Let c be an MC definition in $\mathtt{G}$, and $\mathtt{q} \twoheadrightarrow \mathtt{p} : S_1 \triangleright^c \mathtt{p} \twoheadrightarrow \mathtt{q} : S_2$ be the outermost occurrence of c in $\mathtt{U}(\mathtt{G})$. The committing set of c in $\mathtt{G}$, denoted $\Downarrow^c(\mathtt{G})$, is defined as follows:

$$\Downarrow^c(\mathtt{G}) \coloneqq \mathtt{labels}(S_1 \cup S_2) \;\cup\; \bigcup_{G\in\mathtt{types}(S_1)} \Downarrow^c(G, \{\mathtt{p}\}) \;\cup\; \bigcup_{G\in\mathtt{types}(S_2)} \Downarrow^c(G, \{\mathtt{p},\mathtt{q}\})$$

where $\Downarrow^c(\mathtt{G}, C)$ is an auxiliary function parameterized by the set C of committed roles, used to track dependency with other committing actions. In brief, the case for interactions is:

$$\Downarrow^c(\mathtt{p} \twoheadrightarrow \mathtt{q} : S, C) = \begin{cases} \mathtt{labels}(S) \cup \bigcup_{\mathtt{G}\in\mathtt{types}(S)} \Downarrow^c(\mathtt{G}, C \cup \{\mathtt{q}\}) & (\mathtt{p} \in C \;\wedge\; \mathtt{q} \notin C) \\ \bigcup_{\mathtt{G}\in\mathtt{types}(S)} \Downarrow^c(\mathtt{G}, C) & \text{otherwise} \end{cases}$$

The case $\Downarrow^c(\mathtt{G}_1 \triangleright^{c'} \mathtt{G}_2, C)$ returns $\Downarrow^c(\mathtt{G}_1, C) \cup \Downarrow^c(\mathtt{G}_2, C)$ if $c' \neq c$, and the empty set otherwise. Finally, $\Downarrow^c(\mu\mathtt{t}.\mathtt{G}, C) = \Downarrow^c(\mathtt{G}, C)$ and $\Downarrow^c(\mathtt{t}, C) = \Downarrow^c(\mathtt{end}, C) = \emptyset$.

Intuitively, $\Downarrow^c(\mathtt{G})$ identifies the labels of $S_1 \cup S_2$ as the set of initial committing labels in c: the labels in S_1 are committing for $\mathtt{p}$ and labels in S_2 are committing for both $\mathtt{q}$ and $\mathtt{p}$. To these labels, we add those that depend on actions by $\mathtt{p}$ on the RHS of c and $\{\mathtt{p},\mathtt{q}\}$ on the LHS, by using the auxiliary function. The function $\Downarrow^c(\mathtt{G}, C)$ traverses the syntactic structure of $\mathtt{G}$ until it reaches $\mathtt{t}$, $\mathtt{end}$, or a nested occurrence of the same c (introduced by unfolding). In case $\Downarrow^c(\mathtt{p} \twoheadrightarrow \mathtt{q} : S, C)$ all labels that are sent from a committed role $\mathtt{p} \in C$ to a non-committed role $\mathtt{q} \notin C$ are regarded as committing. Unfolding $\mathtt{U}(\mathtt{G})$ is used to account for committing labels that are captured into a MC block only after recursive unfolding, as we illustrate in Example 3.2.

Example 3.2 (Label capture). Consider the recursive type $\mathtt{G}_c = \mu\mathtt{t}.\,\mathtt{p} \twoheadrightarrow \mathtt{r} : a.(\mathtt{q} \twoheadrightarrow \mathtt{p} : b.\,\mathtt{t} \triangleright \mathtt{G}_2)$ where label $a \notin \mathtt{G}_2$ and hence does not appear directly in the MC. If we apply the auxiliary function in Definition 3.1 to $\mathtt{G}_c$ itself (rather than its unfolding $\mathtt{U}(\mathtt{G}_c)$) then a would not be included in the committing set of $\mathtt{G}_c$. However, after some steps that involve recursive unfolding, shown in (1), an occurrence of a is introduced on the LHS of the outer MC. This second occurrence of a *is* committing (it makes $\mathtt{r}$ commit to the LHS of the outer MC). By using unfolding, Definition 3.1 correctly identifies a as committing.

$$\mathtt{G}_c \;\longrightarrow^*\; \mathtt{p} \twoheadrightarrow \mathtt{r} : a.(\mathtt{q} \twoheadrightarrow \mathtt{p} : b.(\mu\mathtt{t}.\,\mathtt{p} \twoheadrightarrow \mathtt{r} : a.(\mathtt{q} \twoheadrightarrow \mathtt{p} : b.\,\mathtt{t} \triangleright \mathtt{G}_2)) \blacktriangleright \mathtt{G}_2) \tag{1}$$

Well-formedness. Global types where a label occurs multiple times, and where some occurrences are committing and some others are not, are problematic. A simple example is (2), where b appears twice, and is non-committing on the LHS of c while it is committing on the RHS.

$$\mathtt{G}_{\boldsymbol{\chi}} = \mathtt{q} \twoheadrightarrow \mathtt{p} : a.\,\mathtt{q} \twoheadrightarrow \mathtt{r} : b.\,\mathtt{G}_1 \triangleright^c \mathtt{p} \twoheadrightarrow \mathtt{q} : c.\,\mathtt{q} \twoheadrightarrow \mathtt{r} : b.\,\mathtt{G}_2 \tag{2}$$

It is critical that we avoid ambiguities such as those in (2), as they may lead to incorrect semantics for mixed choices. To address this, we introduce a well-formedness condition on initial global types, requiring that all occurrences of a label are exclusively either committing or non-committing.

Well-formedness relies on the notion of committing set (Definition 3.1) and on the dual notion of *non-committing set* (Definition 3.3 below). Well-formedness is formally defined in Definition 3.4 by requiring the committing and non-committing sets to be disjoint.

Definition 3.3 (Non-committing set). Let $\mathtt{G}$ be an initial global type, and let c be a MC definition in $\mathtt{G}$, and $\mathtt{q} \twoheadrightarrow \mathtt{p} : S_1 \triangleright^c \mathtt{p} \twoheadrightarrow \mathtt{q} : S_2$ be the outermost occurrence of c in $\mathtt{U}(\mathtt{G})$. The non-committing set

of c in $\mathtt{G}$, denoted $\Uparrow^c(\mathtt{G})$, is defined as follows, relying on auxiliary function $\Uparrow^c(\mathtt{G}, C)$:

$$\Uparrow^c(\mathtt{G}) \;\coloneqq\; \bigcup_{G \in \mathtt{types}(S_1)} \Uparrow^c(G, \{\mathtt{p}\}) \;\cup\; \bigcup_{G \in \mathtt{types}(S_2)} \Uparrow^c(G, \{\mathtt{p}, \mathtt{q}\})$$

In contrast to $\Downarrow^c(\mathtt{G})$ in Definition 3.1, $\Uparrow^c(\mathtt{G})$ does not include $\mathtt{labels}(S_1) \cup \mathtt{labels}(S_2)$ in the non-committing set of c.

Function $\Uparrow^c(G, C)$ is defined like $\Downarrow^c(G, C)$ from Definition 3.1 except the case for interaction types that is dual: specifically, $\Uparrow^c(\mathtt{p} \twoheadrightarrow \mathtt{q} : S, C)$ returns $\Uparrow^c(\mathtt{G}, C \cup \{\mathtt{q}\})$ if $(\mathtt{p} \in C \;\wedge\; \mathtt{q} \notin C)$, and returns $\mathtt{labels}(S) \;\cup\; \bigcup_{\mathtt{G} \in \mathtt{types}(S)} \Uparrow^c(\mathtt{G}, C)$ otherwise.

Definition 3.4 (Well-formedness). Initial $\mathtt{G}$ is *well-formed* if for all c in $\mathtt{G}$, $\Uparrow^c(\mathtt{G}) \cap \Downarrow^c(\mathtt{G}) = \emptyset$.

Since the syntactic structure of $\mathtt{U}(\mathtt{G})$ is finite, well-formedness yields a decidable algorithm based on the definitions of committing and non-committing sets. Hereafter we assume all initial global types to be well-formed.

Example 3.5 (Well-formedness). One can verify that the global type $\mathtt{G}_{\boldsymbol{\times}}$ in (2) is not well-formed by observing that $b \in \Uparrow^c(\mathtt{G}_{\boldsymbol{\times}})$ and $b \in \Downarrow^c(\mathtt{G}_{\boldsymbol{\times}})$. One can verify that $\mathtt{G}_{\boldsymbol{\checkmark}}$ below is well-formed by observing that $\Downarrow^c(\mathtt{G}_{\boldsymbol{\checkmark}}) = \{a, d, e\}$, $\Uparrow^c(\mathtt{G}_{\boldsymbol{\checkmark}}) = \{b\}$, and $\{a, d, e\} \cap \{b\} = \emptyset$.

$$\begin{array}{lcl} \mathtt{G}_{\boldsymbol{\checkmark}} & = & \mu \mathtt{t}.\, (\mathtt{q} \twoheadrightarrow \mathtt{p} : a.\, \mathtt{q} \twoheadrightarrow \mathtt{r} : b.\, \mathtt{t} \;\triangleright^c\; \mathtt{p} \twoheadrightarrow \mathtt{q} : d.\mathtt{p} \twoheadrightarrow \mathtt{r} : e.\, \mathtt{t}) \\ \mathtt{U}(\mathtt{G}_{\boldsymbol{\checkmark}}) & = & \mathtt{q} \twoheadrightarrow \mathtt{p} : a.\; \mathtt{q} \twoheadrightarrow \mathtt{r} : b.\, \mathtt{G}_{\boldsymbol{\checkmark}} \;\triangleright^c\; \mathtt{p} \twoheadrightarrow \mathtt{q} : d.\, \mathtt{p} \twoheadrightarrow \mathtt{r} : e.\, \mu \mathtt{t}.\, (\mathtt{q} \twoheadrightarrow \mathtt{p} : a.\, \mathtt{q} \twoheadrightarrow \mathtt{r} : b.\, \mathtt{t} \;\triangleright^c\; \ldots) \end{array}$$

This example is noteworthy: label b occurs multiple times in $\mathtt{U}(\mathtt{G}_{\boldsymbol{\checkmark}})$ but all occurrences are consistent as they are all non-committing. The occurrence of b on the RHS of the outer c is non-committing because the receiver $\mathtt{r}$ is already committed to that block, having previously received committing label e.

3.2 Semantics of Global Types

The semantics of global types is defined as a Labelled Transition System over terms $\mathtt{G}$ with labels

$$\ell \coloneqq \mathtt{pq}!a \mid \mathtt{pq}?a \mid \mathit{vc}$$

$\mathtt{pq}!a$ (resp. $\mathtt{pq}?a$) denotes the sending (resp. receiving) action of message a from $\mathtt{p}$ to $\mathtt{q}$. Label vc is for MC instantiation. We define the subject of a label as the singleton set containing the role performing the action described by that label: $\mathtt{sbj}(\mathtt{pq}!a) = \mathtt{sbj}(\mathtt{qp}?a) = \{\mathtt{p}\}$. In what follows, we may write $\mathtt{p}$ as a shorthand for the singleton $\{\mathtt{p}\}$. The subject vc is defined to be the empty set. In the LTS we assume knowledge of the original initial (or *base*) global type from which a given state is reached. We denote it with $\underline{\mathtt{G}}$. This is akin to assuming knowledge of the original static protocol specification of an ongoing session.

The rules for the LTS are given in Figure 4. The first set of rules for interactions is given in Figure 4 (top), and is a minor adaptation of the semantics in [Deniélou and Yoshida 2013].

The remaining rules are new for MC. [Inst] instantiates a MC. [Ctx1] and [Ctx2] handle nested instantiations. The conditions on the roles ensure that once a side is resolved (i.e., no role can act in it anymore), it cannot perform degenerate vc transitions that would artificially break our correspondence results (Section 4.3). In [Ctx1], $\mathtt{G}_l$ is resolved when all roles are committed to the right. In [Ctx2], $\mathtt{G}_r$ is resolved when some roles commit to the left. The asymmetry in the conditions of [Ctx1] and [Ctx2] reflects the fact that commitment to the RHS does not immediately preclude actions on the LHS, until all participants have committed. Maintaining the MC contexts, including the resolved sides in the global types (which will be incorporated into the projected local types), is important for correctly characterising stale messages in the projected systems, and thus for ensuring our correspondence results. [LSnd] allows a send action by the LHS if the subject $\mathtt{p}$ is not committed to the RHS. [RSnd] is symmetric, except $\mathtt{p}$ is added to $\mathcal{R}$ (any action on the RHS is

$$\mathsf{p} \twoheadrightarrow \mathsf{q} : \{a_i.\mathsf{G}_i\}_{i\in I} \xrightarrow{\mathsf{pq}!a_k} \mathsf{p} \rightsquigarrow \mathsf{q} : k\{a_i.\mathsf{G}_i\}_{i\in I} \quad (k \in I) \quad [\mathtt{Snd}]$$

$$\mathsf{p} \rightsquigarrow \mathsf{q} : k\{a_i.\mathsf{G}_i\}_{i\in I} \xrightarrow{\mathsf{pq}?a_k} \mathsf{G}_k \quad [\mathtt{Rcv}] \qquad \frac{\forall i \in I \qquad \mathsf{G}_i \xrightarrow{\ell} \mathsf{G}_i' \qquad \mathsf{p},\mathsf{q} \notin \mathsf{sbj}(\ell)}{\mathsf{p} \twoheadrightarrow \mathsf{q} : \{a_i.\mathsf{G}_i\}_{i\in I} \xrightarrow{\ell} \mathsf{p} \twoheadrightarrow \mathsf{q} : \{a_i.\mathsf{G}_i'\}_{i\in I}} \quad [\mathtt{Cont1}]$$

$$\frac{\mathsf{G}[\mu t.\mathsf{G}/t] \xrightarrow{\ell} \mathsf{G}'}{\mu t.\mathsf{G} \xrightarrow{\ell} \mathsf{G}'} \quad [\mathtt{Rec}] \qquad \frac{\mathsf{G}_k \xrightarrow{\ell} \mathsf{G}_k' \qquad \mathsf{q} \notin \mathsf{sbj}(\ell) \qquad \forall i \in I \setminus k.\, \mathsf{G}_i = \mathsf{G}_i'}{\mathsf{p} \rightsquigarrow \mathsf{q} : k\{a_i.\mathsf{G}_i\}_{i\in I} \xrightarrow{\ell} \mathsf{p} \rightsquigarrow \mathsf{q} : k\{a_i.\mathsf{G}_i'\}_{i\in I}} \quad [\mathtt{Cont2}]$$

$$\mathsf{G}_1 \triangleright^{c} \mathsf{G}_2 \xrightarrow{vc} \mathsf{G}_1 \blacktriangleright^{c}_{\emptyset,\emptyset} \mathsf{G}_2 \quad [\mathtt{Inst}] \qquad \frac{\mathsf{G}_l \xrightarrow{vc} \mathsf{G}_l' \qquad \mathsf{R}(\underline{\mathsf{G}}) \neq \mathcal{R}}{\mathsf{G} = \mathsf{G}_l \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_r \xrightarrow{vc} \mathsf{G}_l' \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_r} \quad [\mathtt{Ctx1}]$$

$$\frac{\mathsf{G}_r \xrightarrow{vc} \mathsf{G}_r' \qquad \mathcal{L} = \emptyset}{\mathsf{G} = \mathsf{G}_l \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_r \xrightarrow{vc} \mathsf{G}_l \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_r'} \quad [\mathtt{Ctx2}] \qquad \frac{\mathsf{G}_l \xrightarrow{\mathsf{pq}!a} \mathsf{G}_l' \qquad \mathsf{p} \notin \mathcal{R}}{\mathsf{G}_l \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_r \xrightarrow{\mathsf{pq}!a} \mathsf{G}_l' \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_r} \quad [\mathtt{LSnd}]$$

$$\frac{\mathsf{G}_l \xrightarrow{\mathsf{pq}?a} \mathsf{G}_l' \qquad \mathsf{q} \notin \mathcal{R} \qquad a \in \Downarrow^{c}(\underline{\mathsf{G}})}{\mathsf{G}_l \blacktriangleright^{c}_{\mathcal{L},\mathcal{R}} \mathsf{G}_r \xrightarrow{\mathsf{pq}?a} \mathsf{G}_l' \blacktriangleright^{c}_{\mathcal{L}\cup\{\mathsf{q}\},\mathcal{R}} \mathsf{G}_r} \quad [\mathtt{LRcv1}] \qquad \frac{\mathsf{G}_l \xrightarrow{\mathsf{pq}?a} \mathsf{G}_l' \qquad \mathsf{q} \notin \mathcal{R} \qquad a \notin \Downarrow^{c}(\underline{\mathsf{G}})}{\mathsf{G}_l \blacktriangleright^{c}_{\mathcal{L},\mathcal{R}} \mathsf{G}_r \xrightarrow{\mathsf{pq}?a} \mathsf{G}_l' \blacktriangleright^{c}_{\mathcal{L},\mathcal{R}} \mathsf{G}_r} \quad [\mathtt{LRcv2}]$$

$$\frac{\mathsf{G}_r \xrightarrow{\mathsf{pq}!a} \mathsf{G}_r' \qquad \mathsf{p} \notin \mathcal{L}}{\mathsf{G}_l \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_r \xrightarrow{\mathsf{pq}!a} \mathsf{G}_l \blacktriangleright_{\mathcal{L},\mathcal{R}\cup\{\mathsf{p}\}} \mathsf{G}_r'} \quad [\mathtt{RSnd}] \qquad \frac{\mathsf{G}_r \xrightarrow{\mathsf{pq}?a} \mathsf{G}_r' \qquad \mathsf{q} \notin \mathcal{L}}{\mathsf{G}_l \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_r \xrightarrow{\mathsf{pq}?a} \mathsf{G}_l \blacktriangleright_{\mathcal{L},\mathcal{R}\cup\{\mathsf{q}\}} \mathsf{G}_r'} \quad [\mathtt{RRcv}]$$

Fig. 4. Global semantics: standard rules (top) and new rules for MC (bottom)

committing). Rules [LRcv1] and [LRcv2] are for committing and non-committing receive actions on the LHS, respectively. In both cases, the subject q must not be committed on the RHS. [RRcv] is similar for receive actions in the RHS.

Let $\vec{\ell} = \ell_0, \ldots, \ell_n$ be a (possibly empty) vector. We write $\mathsf{G}_0 \xrightarrow{\vec{\ell}} \mathsf{G}_{n+1}$ if $\forall i \in 0, \ldots, n.\, \mathsf{G}_i \xrightarrow{\ell_{i+1}} \mathsf{G}_{i+1}$, (or simply $\mathsf{G} \longrightarrow^{*} \mathsf{G}'$ when labels are irrelevant). We say G' is reachable from G if $\mathsf{G} \longrightarrow^{*} \mathsf{G}'$. We just say G' *is reachable* if it is reachable from an initial global type. Labels or reached states may be omitted.

The LTS for global types has a monotonicity property: the sets $\mathcal{L}$ and $\mathcal{R}$ of a MC are monotonically non-decreasing with respect to transition.

Example 3.6 (Global MC steps). One way a well-formed MC may proceed is shown below. In the second line, assume $\mathsf{G}_1 \longrightarrow^{*} \mathsf{G}_1'$ on the LHS where observer p is not the subject in any of those steps.

$$\begin{aligned} \mathsf{q} \twoheadrightarrow \mathsf{p} : a_1.\mathsf{G}_1 \triangleright^{c} \mathsf{p} \twoheadrightarrow \mathsf{q} : a_2.\mathsf{G}_2 &\xrightarrow{vc} \mathsf{q} \twoheadrightarrow \mathsf{p} : a_1.\mathsf{G}_1 \blacktriangleright^{c}_{\emptyset,\emptyset} \mathsf{p} \twoheadrightarrow \mathsf{q} : a_2.\mathsf{G}_2 \\ &\longrightarrow^{*} \mathsf{q} \rightsquigarrow \mathsf{p} : a_1\{a_1.\mathsf{G}_1'\} \blacktriangleright^{c}_{\emptyset,\emptyset} \mathsf{p} \twoheadrightarrow \mathsf{q} : a_2.\mathsf{G}_2 \quad (*) \end{aligned}$$

The marked state may then proceed one of two ways. In the upper of the paths below, observer p also commits to the LHS, followed by q and other roles after some more steps.

$$(*) \begin{array}{ll} \xrightarrow{\mathsf{qp}?a_1} & \mathsf{G}_1' \blacktriangleright^{c}_{\{\mathsf{p}\},\emptyset} \mathsf{p} \twoheadrightarrow \mathsf{q} : a_2.\mathsf{G}_2 \longrightarrow^{*} \mathsf{G}_1'' \blacktriangleright^{c}_{\{\mathsf{p},\mathsf{q},\ldots\},\emptyset} \mathsf{p} \twoheadrightarrow \mathsf{q} : a_2.\mathsf{G}_2 \\ \xrightarrow{\mathsf{pq}!a_2} & \mathsf{q} \rightsquigarrow \mathsf{p} : a_1\{a_1.\mathsf{G}_1'\} \blacktriangleright^{c}_{\emptyset,\{\mathsf{p}\}} \mathsf{p} \rightsquigarrow \mathsf{q} : a_2\{a_2.\mathsf{G}_2'\} \longrightarrow^{*} \mathsf{q} \rightsquigarrow \mathsf{p} : a_1\{a_1.\mathsf{G}_1'''\} \blacktriangleright^{c}_{\emptyset,\{\mathsf{p},\mathsf{q},\ldots\}} \mathsf{G}_2'' \end{array}$$

The lower path corresponds to the race condition situation where q instead commits on the RHS, leading to a (transient) period with active interactions (e.g., $\rightsquigarrow$) by roles on both sides of the MC. However, assuming that all roles have appropriate inputs on the RHS by which they can learn of q's decision (such as the a_1 input by p), they will eventually follow the observer into committing on the RHS and no further actions will occur on the LHS. The conditions for ensuring safe eventual commitment by all roles are formalised in the next subsection.

3.3 Progress of Global Types

Progress ensures that any role $\mathsf{p} \in \mathsf{R}(\mathsf{G})$ either (1) can make a move (immediately or after actions by other roles) or (2) is in a final state. Intuitively, this means that no role is permanently stuck. The notion of *final state* is captured syntactically by $\mathsf{R}(\mathsf{G})$: if $\mathsf{r} \in \mathsf{R}(\mathsf{G})$ then r is *not* in a final state (i.e., it is still active in G). Recall that $\mathsf{R}(\mathsf{G}_1 \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_2) = \mathsf{R}(\mathsf{G}_1) \setminus \mathcal{R} \cup \mathsf{R}(\mathsf{G}_2) \setminus \mathcal{L}$ and hence, once a role is committed to one block, it is not active in the other block. For example, both p and q are in a final state in $\mathsf{p} \rightsquigarrow \mathsf{q} : \{a.\mathsf{end}\} \blacktriangleright_{\emptyset,\{\mathsf{p},\mathsf{q}\}} \mathsf{end}$.

Definition 3.7 (Progress). G enjoys *progress* if for all G' reachable from G the following holds:

$$\mathsf{r} \in \mathsf{R}(\mathsf{G}') \quad \Rightarrow \quad \mathsf{G}' \rightarrow^* \xrightarrow{\ell} \text{ with } \mathtt{sbj}(\ell) = \mathsf{r}$$

As standard in MST [Coppo et al. 2016; Honda et al. 2008], not all global types enjoy progress. We give two sufficient[3] conditions for progress: *awareness* and *balance*.

Awareness guarantees that each role eventually commits when the RHS is taken, and that each role with a terminating execution commits when the LHS is taken. Awareness builds on two relations over roles (Definition 3.8).

Definition 3.8 (Role dependencies $<_{\mathsf{G}}$ and $\ll_{\mathsf{G}}$). Let $\mathsf{p}, \mathsf{q} \in \mathsf{R}(\mathsf{G})$. We define two kinds of dependencies between roles: *strict dependence*, written $\mathsf{p} <_{\mathsf{G}} \mathsf{q}$, and *eventual dependence*, written $\mathsf{p} \ll_{\mathsf{G}} \mathsf{q}$.

- $\mathsf{p} <_{\mathsf{G}} \mathsf{q}$, if $\mathsf{G} \xrightarrow{\vec{\ell}\,\ell_{\mathsf{q}}} \ \wedge\ \mathtt{sbj}(\ell_{\mathsf{q}}) = \mathsf{q}$ implies $\exists \ell \in \vec{\ell}.\ \mathtt{sbj}(\ell) = \mathsf{p}$
- $\mathsf{p} \ll_{\mathsf{G}} \mathsf{q}$, if $\mathsf{G} \xrightarrow{\vec{\ell}} \mathsf{G}'$ implies $\exists a.\ \mathsf{pq}!a \in \vec{\ell} \ \vee\ \mathsf{G}' \rightarrow^* \xrightarrow{\mathsf{pq}!a}$

Namely: $\mathsf{p} <_{\mathsf{G}} \mathsf{q}$ if q will only take action after p does, and $\mathsf{p} \ll_{\mathsf{G}} \mathsf{q}$ if it is always possible for p to send a message to q. We say that p *diverges* in G if $\mathsf{G} \rightarrow^* \mathsf{G}'$ implies $\mathsf{G}' \rightarrow^* \xrightarrow{\ell}$ with $\mathtt{sbj}(\ell) = \mathsf{p}$.

Definition 3.9 (Awareness). An active MC $\mathsf{G} = \mathsf{G}_1 \blacktriangleright^{\mathsf{p}}_{\mathcal{L},\mathcal{R}} \mathsf{G}_2$ is *aware* if for all $\mathsf{r} \in \mathsf{R}(\mathsf{G}) \setminus \mathsf{p}$:
(1) $\mathcal{R} = \emptyset \implies \mathsf{p} <_{\mathsf{G}_2} \mathsf{r}$ (single-decision),
(2) $\mathcal{L} = \emptyset \implies \mathsf{p} \ll_{\mathsf{G}_1} \mathsf{r}$ or r diverges in G_1 (clear-termination).
Similarly, an MC definition $\mathsf{G} = \mathsf{G}_1 \triangleright^{\mathsf{p}} \mathsf{G}_2$ is aware if for all $\mathsf{r} \in \mathsf{R}(\mathsf{G})$, $\mathsf{p} <_{\mathsf{G}_2} \mathsf{r}$ and $\mathsf{p} \ll_{\mathsf{G}_1} \mathsf{r}$. A global type G is aware if all MC definitions and active MC in $\mathsf{U}(\mathsf{G})$ are aware.

Single-decision requires that, on the RHS, all roles depend on the observer until committed. Clear termination allows the roles to start communicating on the LHS without waiting for a committing message. However, if their execution terminates, they must receive a committing message; this is key to progress, as illustrated below in Example 3.10. The use of $\mathsf{U}(\mathsf{G})$ is necessary to deal with label capture (as shown in Example 3.2).

Example 3.10 (Clear termination). In the type below (left): if observer p receives a and terminates, then q is unable to locally determine whether to terminate or continue waiting for a b that will never arrive (i.e., termination is not *clear*); one fix is to add $\mathsf{p} \twoheadrightarrow \mathsf{q} : c$ on the LHS. As a generalisation, however we can safely lift eventual dependency for infinite executions, as shown in a *stream*

[3]Complete decidable conditions are, unfortunately, not possible [Gouda et al. 1984].

exception pattern below (right) where q establishes a communication (label c) and immediately starts the stream, and potentially handle later a connection failure message (b by p):

$$\mathsf{q} \twoheadrightarrow \mathsf{p} : a.\ \mathsf{end} \triangleright \mathsf{p} \twoheadrightarrow \mathsf{q} : b.\ \mathsf{end}\ \text{✗} \qquad \mathsf{q} \twoheadrightarrow \mathsf{p} : c.\ \mu \mathtt{t}.(\mathsf{q} \twoheadrightarrow \mathsf{p} : a.\ \mathtt{t}) \triangleright \mathsf{p} \twoheadrightarrow \mathsf{q} : b.\ \mathsf{end}\ \text{✓}$$

Consider now a *third-party exception* below, where a 'third-party' observer r may decide, after some interactions between p and q, to either commit on the LHS or raise the RHS exception:

$$\mathsf{q} \twoheadrightarrow \mathsf{r} : c.\ \mu \mathtt{t}.(\mathsf{q} \twoheadrightarrow \mathsf{p} : a.\ \mathtt{t}) \triangleright \mathsf{r} \twoheadrightarrow \mathsf{q} : b.\mathsf{r} \twoheadrightarrow \mathsf{p} : b.\ \mathsf{end}\ \text{✓}$$

Clear-termination holds because: (i) r is able to commit to the LHS by receiving c, (ii) all other roles (p, q) do not need to commit because they diverge within the LHS.

Example 3.11 (Interrupt pattern). Interrupts can be modelled with a minor extension to the theory (implemented in Section 5.2), if we allow the observer to interact on the LHS before committing on that side. To this aim, allocate a set D of choice labels used by observers to *commit on the LHS*. The *interrupt pattern* can then be expressed as the global type below, letting $d \in D$ and $a, b, c \notin D$:

$$\mathsf{q} \twoheadrightarrow \mathsf{p} : c.\ \mu \mathtt{t}.(\mathsf{q} \twoheadrightarrow \mathsf{p} : \{a.\ \mathtt{t},\ d.\ \mathsf{p} \twoheadrightarrow \mathsf{q} : e.\ \mathsf{end}\}) \triangleright \mathsf{p} \twoheadrightarrow \mathsf{q} : b.\ \mathsf{end}$$

Unlike in the stream exception (Example 3.10), now p can repeatedly *receive* messages a from q before, possibly, throwing the interrupt b (or committing/terminating on the LHS). Our theory can be easily adapted to support interrupt patterns using D by: (1) updating the definition of committing set so that the committing chains on the LHS start with actions in D , (2) adjusting the existential quantifier in Definition 3.8 (eventual dependence $\ll_{\mathsf{G}}$) by requiring label a to be in D (i.e., $\exists a \in D$).

Awareness ensures convergence of the roles in a block by requiring dependencies with the observer. However, if a role does not appear in *some* execution branches, it may not be able to converge with the observer's decision. This is a known problem also with branching choices:

$$\mathsf{p} \twoheadrightarrow \mathsf{q} : \{a.\ \mathsf{q} \twoheadrightarrow \mathsf{r} : \{c.\mathsf{end}\},\ b.\ \mathsf{q} \twoheadrightarrow \mathsf{p} : \{c.\mathsf{end}\}\} \tag{3}$$

In (3), role r does not know whether to terminate or wait for a message, and thus does not terminate. *Balance* (Definition 3.12) generalizes this idea to MC, operating on the unfolding $\mathtt{U}(\mathsf{G})$ to account for label capture. Definition 3.12 also uses a truncation operator $\mathtt{Trunc}(\mathsf{G})$ that replaces each recursive subterm in G with end. Assuming G has no free recursion variables: $\mathtt{Trunc}(\mu \mathtt{t}.\mathsf{G}') = \mathtt{Trunc}(\mathsf{end}) = \mathsf{end}$. All other cases are defined inductively. Truncation prevents expressiveness loss: Definition 3.12 universally quantifies over subterms of $\mathtt{U}(\mathsf{G})$ which would consider recursive subterms out of their intended context. For example, without truncation any instantiation of example (1) in Section 3.1 would (unnecessarily) be excluded as unbalanced.

Definition 3.12 (Balance). A global type G is *balanced* if for all subterms G' of $\mathtt{Trunc}(\mathtt{U}(\mathsf{G}))$:

(1) $\mathsf{G}' = \mathsf{p} \twoheadrightarrow \mathsf{q} : S \implies \forall \mathsf{G}_1, \mathsf{G}_2 \in \mathtt{types}(S),\ \mathtt{R}(\mathsf{G}_1) \setminus \{\mathsf{p}, \mathsf{q}\} = \mathtt{R}(\mathsf{G}_2) \setminus \{\mathsf{p}, \mathsf{q}\}$
(2) $\mathsf{G}' = \mathsf{G}_1 \triangleright \mathsf{G}_2 \implies \mathtt{R}(\mathsf{G}_1) = \mathtt{R}(\mathsf{G}_2)$ (3) $\mathsf{G}' = \mathsf{G}_1 \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_2 \implies \mathtt{R}(\mathsf{G}_1) \cup \mathcal{L} = \mathtt{R}(\mathsf{G}_2) \cup \mathcal{R}$

Case (3) allows committed roles to 'disappear' from a block. Case (1) is normally entailed by projection. We give it here for a simpler presentation and separation of concerns.

Theorem 3.13 (Progress). *If* G *is initial, aware, and balanced then it enjoys* ***progress****.*

The proof relies on a *coherence* property: all the committed roles are committed to the same side. Formally, if $\mathsf{G} = \mathbf{C}[\,\mathsf{G}_1 \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_2\,]$ then $\mathcal{L} = \emptyset \vee \mathcal{R} = \emptyset$. Essentially, we prove that (1) awareness, balance, and coherence are preserved by transition, and (2) balanced, aware, and coherent global types enjoy progress. Theorem 3.13 follows since initial global types are always coherent.

4 Local Types

The syntax of *local types* is given by the following grammar:

$$\mathsf{L} ::= \mathsf{p}\,\&_{i\in I}\, a_i.\mathsf{L}_i \mid \mathsf{p}\oplus_{i\in I} a_i.\mathsf{L}_i \mid \mu \mathsf{t}.\mathsf{L} \mid \mathsf{t} \mid \mathsf{end} \mid \mathsf{L}_1 \triangleright^c \mathsf{L}_2 \mid \mathsf{L}_1 \blacktriangleright^c \mathsf{L}_2 \mid \mathsf{L} \blacktriangleright^c \bullet \mid \bullet \blacktriangleright^c \mathsf{L}$$

The first five terms are standard; the notation for roles and labels are as in global types. We recall $\mathsf{p}\,\&_{i\in I}\, a_i.\mathsf{L}_i$ is a branching type, waiting for one of the labels $\{a_i\}_{i\in I}$ and continuing as the corresponding L_i. Term $\mathsf{p}\oplus_{i\in I} a_i.\mathsf{L}_i$ is the corresponding send/selection type. We often omit end.

Terms $\mathsf{L}_1 \triangleright^c \mathsf{L}_2$ and $\mathsf{L}_1 \blacktriangleright^c \mathsf{L}_2$ are for *MC definition* and *active MC*, respectively. The terms $\mathsf{L} \blacktriangleright^c \bullet$ and $\bullet \blacktriangleright^c \mathsf{L}$ model run-time states in which the role has *committed* on the LHS or the RHS, respectively. We omit the annotation c when it is clear from context or not relevant. In-transit/buffered (i.e., yet to be consumed) messages are modelled using FIFO queues. The localised view of a global type for a given role is thus a *configuration* $(\mathsf{p}, \mathsf{L}, \sigma)$ where p is the role, L is the local type specifying the behaviour of p, and σ is a local FIFO queue of messages that have been sent to p (and possibly arrived) but not consumed. A global type therefore corresponds overall to a collection of local configurations. We call this collection a *system*, ranged over by Y, Y'.

$$Y ::= (\mathsf{p}_i, \mathsf{L}_i, \sigma_i)_{i\in I} \qquad \sigma : \mathsf{q} \mapsto \vec{m} \qquad m ::= (a,\ \pi) \qquad \pi ::= \epsilon \mid \mathtt{l}.\pi \mid \mathtt{r}.\pi$$

We assume the configurations of a system have pairwise-distinct roles.

Messages carry a *path* π, which fully qualifies the active MC context under which the message was sent. A path specifies left-to-right the top-most MC to the inner MC of the sending local type. We often omit the trailing ϵ. As an example, $(\mathsf{L}_1 \blacktriangleright \mathsf{p} \oplus a) \blacktriangleright \mathsf{p} \oplus b$ may send two messages: $(a, \mathtt{l}.\mathtt{r})$ and $(b, \mathtt{r})$. In global types, the equivalent path information is implicit in the global type context (nested active MCs) in which the corresponding in-transit message term ($\rightsquigarrow$) occurs.

4.1 Operational Semantics

The local operational semantics uses the labels below and two main judgments:

$$\ell ::= \mathsf{pq}!a \mid \mathsf{pq}?a \mid \nu c \mid \rho \qquad Y \xrightarrow{\ell} Y' \qquad \pi : Y \xrightarrow{\ell} Y'$$

Labels ℓ now include ρ for stale message purging from local queues. The first judgment $Y \xrightarrow{\ell} Y'$ is for top-level concurrent execution of systems. It has just three rules.

$$\frac{Y \xrightarrow{\ell} Y''}{Y, Y' \xrightarrow{\ell} Y'', Y'}\ [\textsc{Par}] \qquad \frac{\mathsf{gc}(\mathsf{L}, \sigma) = \sigma'}{(\mathsf{p}, \mathsf{L}, \sigma) \xrightarrow{\rho} (\mathsf{p}, \mathsf{L}, \sigma')}\ [\textsc{Discard}] \qquad \frac{\epsilon : Y \xrightarrow{\ell} Y'}{Y \xrightarrow{\ell} Y'}\ [\textsc{Low}]$$

Rule [Par] is a structural rule that models concurrent execution. Rule [Low] uses lower-level judgments of the form $\pi : Y \xrightarrow{\ell} Y'$ (see below). Rule [Discard] performs stale message purging on the local queue σ of a configuration. This is necessary to maintain correspondence between the executions of global and local types. Consider the example in (4) where the right-hand side term is reached after the following sequence of actions: (1) q sends c to p and commits to the RHS, (2) p sends a to q (on the LHS), (3) p receives c and commits to the RHS, (3) p sends d to q on the RHS.

$$Y, (\mathsf{q},\ \mathsf{p}\&a.\mathsf{p} \oplus b.\mathsf{end} \blacktriangleright \mathsf{p} \oplus c.\mathsf{p}\&d.\mathsf{end}, \sigma) \rightarrow^* Y', (\mathsf{q},\ \bullet \blacktriangleright \mathsf{p}\&d.\mathsf{end}, \sigma : \mathsf{p} \mapsto (a, \mathtt{l}), (d, \mathtt{r})) \quad (4)$$

In the reached term – on the right-hand side of (4) – the enqueued message $(a, \mathtt{l})$ pertains to the LHS of the MC and is essentially garbage. The presence of $(a, \mathtt{l})$ in q's queue would naively break correspondence (Section 4.3) with the projection of the corresponding metatheoretical global type $\mathsf{p} \rightsquigarrow \mathsf{q} : a.\mathsf{q} \twoheadrightarrow \mathsf{p} : b.\mathsf{end} \blacktriangleright_{\emptyset,\{\mathsf{p},\mathsf{q}\}} \mathsf{p} \rightsquigarrow \mathsf{q} : d.\mathsf{end}$.

Fortunately, local types allow us to define a *purge* function gc that operates solely on information that is *local* to the configuration. In the general case of nested MC, we identify stale messages by traversing the MC structures of the receiver's local type L using the sender's path π included in

the message. This check is defined inductively as a predicate $\mathtt{stale}(\pi, \mathtt{L})$, which returns true if following π in $\mathtt{L}$ hits a stale ($\bullet$) side. Below, the otherwise cases may include non-initialized MC which are never stale as still uncommitted, and $\mathtt{stale}(\epsilon, \mathtt{L}) = \mathtt{false}$ for all $\mathtt{L}$.

$$\mathtt{stale}(\mathtt{l}.\pi, \mathtt{L}) = \begin{cases} \mathtt{true} & \mathtt{L} = \bullet \blacktriangleright \mathtt{L}_2 \\ \mathtt{stale}(\pi, \mathtt{L}_1) & \mathtt{L} = \mathtt{L}_1 \blacktriangleright \mathtt{L}_2 \textit{ or} \\ & \mathtt{L} = \mathtt{L}_1 \blacktriangleright \bullet \\ \mathtt{false} & \textit{otherwise} \end{cases} \qquad \mathtt{stale}(\mathtt{r}.\pi, \mathtt{L}) = \begin{cases} \mathtt{true} & \mathtt{L} = \mathtt{L}_1 \blacktriangleright \bullet \\ \mathtt{stale}(\pi, \mathtt{L}_2) & \mathtt{L} = \mathtt{L}_1 \blacktriangleright \mathtt{L}_2 \textit{ or} \\ & \mathtt{L} = \bullet \blacktriangleright \mathtt{L}_2 \\ \mathtt{false} & \textit{otherwise} \end{cases}$$

Purge (a.k.a. garbage collection) $\mathrm{gc}(\mathtt{L}, \sigma)$ looks into σ (for all $\mathtt{p}$ in its domain) and removes all messages (a, π) for which $\mathtt{stale}(\pi, \mathtt{L})$ is true.

Lower-level judgements $\pi \vdash Y \xrightarrow{\ell} Y'$ model the main behaviour of configurations. The environment π records the MC context for messages being sent. The first three rules in Figure 5 are standard for communications and recursion. Rule [Snd] appends an a_k message to the receiver's queue and continues as $\mathtt{L}_k$. The message is annotated with π from the context. Dually, rule [Rcv] consumes the first such message a_k annotated by π and continues as $\mathtt{L}_k$. [Rec] is standard.

The remaining rules are new and define the semantics for MC. [New] instantiates a MC definition. [LSnd] executes an output action in the LHS of an active but not committed MC. The context π is updated to $\pi.\mathtt{l}$ to indicate the action is happening on the LHS of the current MC. Analogously to global types, sending on the LHS is never committing. For LHS receive actions we have two rules: [LRcv1] (committing) and [LRcv2] (non-committing). As in the global semantics, we infer committing labels from the base $\underline{\mathtt{G}}$.[4] [RSnd] is for sending on the RHS which is always committing; likewise [RRcv] for receiving on the RHS (symmetric to [LRcv1]). Due to space constraints, we omit structural rule [RCtxt] (which is symmetric to [LCtxt]), and structural rules [NLCtxt] and [NRCtxt] for vc actions on the LHS and RHS of $\mathtt{L}_1 \blacktriangleright \mathtt{L}_2$, respectively.

4.2 Projection

The distributed counterpart of a global type is a system of configurations derived by *projection*. The projection of a global type $\mathtt{G}$ onto a role $\mathtt{r} \in \mathtt{R}(\mathtt{G})$, written $\mathtt{G} \upharpoonright \mathtt{r}$, returns a pair $(\mathtt{L}, \sigma)$ of a local type and its queue. Recall, the queue σ is a mapping from roles $\mathtt{r}' \in \mathtt{R}(\mathtt{G})$ to a vector of messages received, but not consumed, by $\mathtt{r}$ from $\mathtt{r}'$.

The main rules for projection, given in Figure 6, take a context path π, which is needed to define messages m being enqueued. Top-level projection $\mathtt{G} \upharpoonright \mathtt{r}$ is bootstrapped as $\epsilon \vdash \mathtt{G} \upharpoonright \mathtt{r}$.

The first two rules for interaction and in-transit messages are standard, except we also project the queues. Notation σ_0 stands for an 'empty queue' that maps each $\mathtt{p}$ in its domain to ϵ. The projection of a message in transit $\mathtt{p} \rightsquigarrow \mathtt{q} : k\, a_i.\mathtt{G}_i$ uses the context π to construct the queued message. The case of projection for an active MC, of the form $\mathtt{G}_1 \blacktriangleright \mathtt{G}_2$, uses the operation $\sigma_1 \circ \sigma_2$ to concatenate queues: if $\mathrm{dom}(\sigma_1) = \mathrm{dom}(\sigma_2)$ then $\sigma_1 \circ \sigma_2 = \{\mathtt{r} \mapsto \sigma_1(\mathtt{r}) \cdot \sigma_2(\mathtt{r}) \mid \mathtt{r} \in \mathrm{dom}(\sigma_1)\}$. The behaviour of an MC can be considered as starting from the LHS before possibly switching to the RHS, hence the projection accordingly concatenates the LHS and RHS queues in this order.

The notation $\sqcap_{i \in I} \mathtt{L}_i$ is the standard notion of *merge* [Deniélou and Yoshida 2012] for projecting onto third-party roles. For example, the projection of $\mathtt{p} \twoheadrightarrow \mathtt{q} : \{a_1.\,\mathtt{q} \twoheadrightarrow \mathtt{r} : \{a_1.\mathsf{end}\},\ a_2.\,\mathtt{q} \twoheadrightarrow \mathtt{r} : \{a_2.\mathsf{end}\}\}$ onto $\mathtt{r}$ is defined as $\mathtt{p}\&a_1.\mathsf{end} \sqcap \mathtt{p}\&a_2.\mathsf{end}$, yielding $\mathtt{p}\&_{i \in \{1,2\}} a_i.\mathsf{end}$. The definition (omitted) implicitly handles MC in the same way as other non-branching constructs, i.e., the projection of each case must be the same.

[4] Observe that this is purely static information and can be derived (pre-processed) from the source global type. Equivalently, we could embed this information into local types by adding annnotations.

$$\frac{k \in I \qquad m = (a_k,\ \pi)}{\pi : (\mathsf{p}, \mathsf{q} \oplus_{i\in I} a_i.\mathsf{L}_i, \sigma), (\mathsf{q}, \mathsf{L}, \sigma'[\mathsf{p} \mapsto \vec{m}]) \xrightarrow{\mathsf{pq}!a_k} (\mathsf{p}, \mathsf{L}_k, \sigma), (\mathsf{q}, \mathsf{L}, \sigma'[\mathsf{p} \mapsto \vec{m} \cdot m])} \quad [\textsc{Snd}]$$

$$\frac{k \in I \qquad \vec{m} = \vec{m}_1 \cdot (a_k,\ \pi) \cdot \vec{m}_2 \qquad (a, \pi) \notin \vec{m}_1}{\pi : (\mathsf{q}, \mathsf{p} \&_{i\in I} a_i.\mathsf{L}_i, \sigma[\mathsf{p} \mapsto \vec{m}]) \xrightarrow{\mathsf{pq}?a_k} (\mathsf{q}, \mathsf{L}_k, \sigma[\mathsf{p} \mapsto \vec{m}_1 \cdot \vec{m}_2])} \quad [\textsc{Rcv}]$$

$$\frac{\pi : (\mathsf{p}, \mathsf{L}[\mu \mathsf{t}.\mathsf{L}/\mathsf{t}], \sigma), Y \xrightarrow{\ell} Y'}{\pi : (\mathsf{p}, \mu \mathsf{t}.\mathsf{L}, \sigma), Y \xrightarrow{\ell} Y'} \qquad \pi : (\mathsf{p}, \mathsf{L} \triangleright^c \mathsf{L}', \sigma) \xrightarrow{vc} (\mathsf{p}, \mathsf{L} \blacktriangleright^c \mathsf{L}', \sigma) \quad [\textsc{Rec/New}]$$

$$\frac{\pi.1 : (\mathsf{p}, \mathsf{L}_1, \sigma), Y \xrightarrow{\mathsf{pq}!a} (\mathsf{p}, \mathsf{L}_1', \sigma), Y'}{\pi : (\mathsf{p}, \mathsf{L}_1 \blacktriangleright \mathsf{L}_2, \sigma), Y \xrightarrow{\mathsf{pq}!a} (\mathsf{p}, \mathsf{L}_1' \blacktriangleright \mathsf{L}_2, \sigma), Y'} \qquad \frac{\pi.\mathsf{r} : (\mathsf{p}, \mathsf{L}_2, \sigma), Y \xrightarrow{\mathsf{pq}!a} (\mathsf{p}, \mathsf{L}_2', \sigma), Y'}{\pi : (\mathsf{p}, \mathsf{L}_1 \blacktriangleright \mathsf{L}_2, \sigma), Y \xrightarrow{\mathsf{pq}!a} (\mathsf{p}, \bullet \blacktriangleright \mathsf{L}_2', \sigma), Y'} \quad [\textsc{LSnd/RSnd}]$$

$$\frac{\pi.1 : (\mathsf{p}, \mathsf{L}_1, \sigma) \xrightarrow{\mathsf{pq}?a} (\mathsf{p}, \mathsf{L}_1', \sigma') \quad a \in \Downarrow^c(\mathsf{G})}{\pi : (\mathsf{p}, \mathsf{L}_1 \blacktriangleright \mathsf{L}_2, \sigma) \xrightarrow{\mathsf{pq}?a} (\mathsf{p}, \mathsf{L}_1' \blacktriangleright \bullet, \sigma')} \qquad \frac{\pi.1 : (\mathsf{p}, \mathsf{L}_1, \sigma) \xrightarrow{\mathsf{pq}?a} (\mathsf{p}, \mathsf{L}_1', \sigma') \quad a \notin \Downarrow^c(\mathsf{G})}{\pi : (\mathsf{p}, \mathsf{L}_1 \blacktriangleright \mathsf{L}_2, \sigma) \xrightarrow{\mathsf{pq}?a} (\mathsf{p}, \mathsf{L}_1' \blacktriangleright \mathsf{L}_2, \sigma')} \quad [\textsc{LRcv1/2}]$$

$$\frac{\pi.\mathsf{r} : (\mathsf{q}, \mathsf{L}_2, \sigma) \xrightarrow{\mathsf{pq}?a} (\mathsf{q}, \mathsf{L}_2', \sigma')}{\pi : (\mathsf{q}, \mathsf{L}_1 \blacktriangleright \mathsf{L}_2, \sigma) \xrightarrow{\mathsf{pq}?a} (\mathsf{q}, \bullet \blacktriangleright \mathsf{L}_2', \sigma')} \qquad \frac{\pi.1 : (\mathsf{p}, \mathsf{L}, \sigma), Y \xrightarrow{\ell} (\mathsf{p}, \mathsf{L}', \sigma'), Y'}{\pi : (\mathsf{p}, \mathsf{L} \blacktriangleright \bullet, \sigma), Y \xrightarrow{\ell} (\mathsf{p}, \mathsf{L}' \blacktriangleright \bullet, \sigma'), Y'} \quad [\textsc{RRcv/LCtxt}]$$

Fig. 5. Local semantics, selected rules.

$$\pi \vdash \mathsf{p} \twoheadrightarrow \mathsf{q} : \{a_i : \mathsf{G}_i\}_{i\in I} \upharpoonright \mathsf{r} = \begin{cases} \mathsf{q} \oplus_{i\in I} a_i.\mathsf{L}_i, \sigma & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p} \&_{i\in I} a_i.\mathsf{L}_i, \sigma & \text{if } \mathsf{r} = \mathsf{q} \\ \sqcap_{i\in I} \mathsf{L}_i, \sigma & \text{if } \mathsf{r} \notin \{\mathsf{p}, \mathsf{q}\} \end{cases} \quad \text{where in all cases } \forall i \in I,\ \mathsf{L}_i, \sigma = \pi \vdash \mathsf{G}_i \upharpoonright \mathsf{r}$$

$$\pi \vdash \mathsf{p} \rightsquigarrow \mathsf{q} : k\,\{a_i.\mathsf{G}_i\}_{i\in I} \upharpoonright \mathsf{r} = \begin{cases} \mathsf{L}_k, \sigma_k & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p} \&_{i\in I} a_i.\mathsf{L}_i,\ \sigma_k[\mathsf{p} \mapsto (a_k,\ \pi) \cdot \sigma_k(\mathsf{p})] & \text{if } \mathsf{r} = \mathsf{q} \\ \mathsf{L}_k, \sigma_k & \text{if } \mathsf{r} \notin \{\mathsf{p}, \mathsf{q}\} \end{cases}$$
$$\text{where in all cases } \forall i \in I,\ \mathsf{L}_i, \sigma_i = \pi \vdash \mathsf{G}_i \upharpoonright \mathsf{r} \text{ and } \forall i \in I \setminus \{k\},\ \sigma_i = \sigma_0$$

$$\pi \vdash \mathsf{G}_1 \triangleright \mathsf{G}_2 \upharpoonright \mathsf{r} \ = \ \mathsf{L}_1 \triangleright \mathsf{L}_2, \sigma_0 \qquad \text{where } \forall i \in \{1, 2\},\ \mathsf{L}_i, \sigma_0 = \pi \vdash \mathsf{G}_i \upharpoonright \mathsf{r}$$

$$\pi \vdash \mathsf{G}_1 \blacktriangleright_{\mathcal{L},\mathcal{R}} \mathsf{G}_2 \upharpoonright \mathsf{r} = \begin{cases} \mathsf{L}_1 \blacktriangleright \bullet, \sigma_1 & \text{if } \mathsf{r} \in \mathcal{L} \text{ and } \mathsf{L}_1, \sigma_1 = \pi.1 \vdash \mathsf{G}_1 \upharpoonright \mathsf{r} \\ \bullet \blacktriangleright \mathsf{L}_2, \sigma_2 & \text{if } \mathsf{r} \in \mathcal{R} \text{ and } \mathsf{L}_2, \sigma_2 = \pi.\mathsf{r} \vdash \mathsf{G}_2 \upharpoonright \mathsf{r} \\ \mathsf{L}_1 \blacktriangleright \mathsf{L}_2, \sigma_1 \circ \sigma_2 & \text{if } \mathsf{r} \notin \mathcal{L} \cup \mathcal{R} \text{ and } \forall i \in \{1, 2\},\ \mathsf{L}_i, \sigma_i = \pi_i \vdash \mathsf{G}_i \upharpoonright \mathsf{r} \\ & \quad \text{with } \pi_1 = \pi.1 \text{ and } \pi_2 = \pi.\mathsf{r} \end{cases}$$

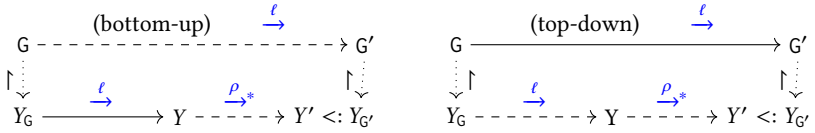
$$\pi \vdash \mu \mathsf{t}.\mathsf{G} \upharpoonright \mathsf{r} = \begin{cases} \mathsf{end}, \sigma_0 & \text{if } \mathsf{G} = \mathsf{t} \\ \mathsf{t}', \sigma_0 & \text{if } \mathsf{G} = \mathsf{t}' \wedge \mathsf{t}' \neq \mathsf{t} \\ \mu \mathsf{t}.\mathsf{L}, \sigma & \text{otherwise, with } \pi \vdash \mathsf{G} \upharpoonright \mathsf{r} = \mathsf{L}, \sigma \end{cases} \qquad \begin{array}{l} \pi \vdash \mathsf{t} \upharpoonright \mathsf{r} = \mathsf{t}, \sigma_0 \\ \pi \vdash \mathsf{end} \upharpoonright \mathsf{r} = \mathsf{end}, \sigma_0 \end{array}$$

Fig. 6. Projection of global types to local types and input FIFOs

We extend projection to systems and define the *system derived* from G as $\mathsf{G}\upharpoonright = \{(\mathsf{r}, \mathsf{L}_\mathsf{r}, \sigma_\mathsf{r})\}_{\mathsf{r}\in\mathsf{R}(\mathsf{G})}$ where $\forall \mathsf{r} \in \mathsf{R}(\mathsf{G})\ .\ \mathsf{L}_\mathsf{r}, \sigma_\mathsf{r} = \mathsf{G} \upharpoonright \mathsf{r}$.

4.3 Operational Correspondence

We establish an operational correspondence, called *fidelity*, between global types and the systems obtained by projection. Fidelity is defined as a weak correspondence relation over pairs of global types and systems, treating *vc* and ρ actions as the silent action τ. This correspondence is precise, comprising a top-down and a bottom-up property, as depicted below.



Bottom-up fidelity states that each action of a system can be matched with the same action by the corresponding global type, preserving the correspondence given by projection modulo garbage collection and a preorder '<:' over local types (discussed below). Top-down fidelity, conversely, states that each step of G can be matched by a step of Y.

The asynchronous nature of mixed choices and the decentralised semantics of systems raise two main challenges: (1) upon commitment of a configuration to a MC block, some messages in its queue may become stale; and (2) localised MC instantiations and recursive unfoldings can cause (a minor form of) misalignment of the configurations. So far, our definitions have paved the way for tackling (1): we use local states $\mathsf{L} \blacktriangleright \bullet$ and $\bullet \blacktriangleright \mathsf{L}$ to keep track of local commitments, and use function $\mathsf{gc}()$ and action ρ to (locally) identify and remove stale messages. Both top-down and bottom-up fidelity may require some number of ρ steps to preserve correspondence.

We address (2) using a preorder '<:' on pairs of systems, as we now discuss. In a global type, a MC is instantiated with a single "centralised" action *vc*, whereas the local configurations in the derived system independently perform separate decentralised *vc* actions. Decentralised MC instantiation can cause administrative issues for the correspondence. Consider the global types below where $\mathsf{G}_1 = \mathsf{q} \twoheadrightarrow \mathsf{p} : a_1.\mathsf{end}$ and c is instantiated before q's first send action:

$$\mathsf{G}_P = \mathsf{q} \twoheadrightarrow \mathsf{r} : b.(\mathsf{G}_1 \triangleright^c \mathsf{p} \twoheadrightarrow \mathsf{q} : a_2.\mathsf{end}) \xrightarrow{vc} \xrightarrow{\mathsf{pq}!a_2} \mathsf{q} \twoheadrightarrow \mathsf{r} : b.\,(\mathsf{G}_1 \blacktriangleright^c_{\emptyset,\{\mathsf{p}\}} \mathsf{p} \rightsquigarrow \mathsf{q} : 2\,a_2.\mathsf{end}\,) = \mathsf{G}'_P$$

Consider now configuration Y_q obtained by projecting G_P on q:

$$Y_\mathsf{q} = (\mathsf{q},\ \mathsf{r} \oplus b.\,(\mathsf{p} \oplus a_1.\mathsf{end} \triangleright^c \mathsf{q}\,\&\,a_2.\mathsf{end}),\ \sigma_0) \qquad Y'_\mathsf{q} = (\mathsf{q},\ \mathsf{r} \oplus b.\,(\mathsf{p} \oplus a_1.\mathsf{end} \blacktriangleright^c \mathsf{q}\,\&\,a_2.\mathsf{end}),\ \sigma_0)$$

In Y_q, q cannot instantiate the MC before it sends b to r (i.e., Y_q cannot reduce to Y'_q). Namely, the system derived from G_P cannot reach the state derived from G'_P where all configurations have instantiated c (yet q has not sent b). The preorder '<:' is introduced to regulate this deferral of instantiations which may need to be interleaved with other actions. In the example above $Y_\mathsf{q} <: Y'_\mathsf{q}$. The key rule for MC is given below (left). L is the local type used to bootstrap the derivation at the configuration level (below right). Essentially, the other cases (except for axioms $\mathsf{end} <: \mathsf{end}$ and $\mathsf{t} <: \mathsf{t}$) are defined inductively, and the preorder is applied to systems pointwise.

$$\frac{\mathsf{L}'_1 <: \mathsf{L}_1 \qquad \mathsf{L}'_2 <: \mathsf{L}_2}{\mathsf{L}'_1 \triangleright^c \mathsf{L}'_2 <: \mathsf{L}_1 \blacktriangleright^c \mathsf{L}_2} \qquad\qquad \frac{\mathsf{L}' <: \mathsf{L}}{(\mathsf{p}, \mathsf{L}', \sigma) <: (\mathsf{p}, \mathsf{L}, \sigma)}$$

Lemma 4.1 (Bottom-up fidelity). *Let* G *be a global type reachable from an initial, aware, and balanced global type, and let* $Y_\mathsf{G} = \mathsf{G}\!\upharpoonright$. *Then the following holds:*

(1) $Y_\mathsf{G} \xrightarrow{\ell} Y \ \wedge\ \ell \neq \rho \implies \mathsf{G} \xrightarrow{\ell} \mathsf{G}' \ \wedge\ \exists Y'.\ Y \xrightarrow{\rho}{}^{*}\, Y' \ \wedge\ Y' <: \mathsf{G}'\!\upharpoonright$

(2) $Y_\mathsf{G} \xrightarrow{\rho} Y \implies Y_\mathsf{G} = Y = \mathsf{G}\!\upharpoonright$

In case (1), if the action by G is committing (say for p), then it may cause some of the messages in p's queue in Y to become stale, hence a ρ action may be needed to restore correspondence. Case (2) highlights a property of projection: derived systems have no stale messages.

Lemma 4.2 (Top-down fidelity). *Let* $\mathtt{G}$ *be a global type reachable from an initial, aware, and balanced global type, and let* $Y_{\mathtt{G}} = \mathtt{G}\upharpoonright$. *Then the following holds:*

(1) $\mathtt{G} \xrightarrow{\mathtt{pq}\square a} \mathtt{G}' \implies \exists Y'.\ Y_{\mathtt{G}} \xrightarrow{\mathtt{pq}\square a} \xrightarrow{\rho}{}^{*} Y' \ \wedge\ Y' <: \mathtt{G}'\upharpoonright \qquad (\square \in \{!, ?\})$

(2) $\mathtt{G} \xrightarrow{\nu c} \mathtt{G}' \implies \exists Y'.\ Y_{\mathtt{G}} \xrightarrow{\vec{\nu c}} Y' \ \wedge\ Y' <: \mathtt{G}'\upharpoonright$

Case (1) is symmetric to case (1) of bottom-up fidelity. In case (2) action ν can be immediately matched with several actions ν by Y (others may be deferred by '<:').

The operational correspondence is formulated in terms of a weak relation over pairs of global types and systems, where ν and ρ are renamed as τ. We write $\longrightarrow^{*}$ for a possibly empty sequence of τ actions and define $\overset{\ell}{\Longrightarrow} = \longrightarrow^{*} \xrightarrow{\ell} \longrightarrow^{*}$ if $\ell \neq \tau$ and $\overset{\ell}{\Longrightarrow} = \longrightarrow^{*}$ otherwise.

Definition 4.3 (Weak Correspondence). A relation $\mathcal{R}$ over $(\mathtt{G}, Y)$ is a *weak correspondence* if whenever $(\mathtt{G}, Y) \in \mathcal{R}$ then: (1) If $\mathtt{G} \xrightarrow{\ell} \mathtt{G}'$ then $Y \overset{\ell}{\Longrightarrow} Y'$ and $(\mathtt{G}', Y') \in \mathcal{R}$; (2) If $Y \xrightarrow{\ell} Y'$ then $\mathtt{G} \overset{\ell}{\Longrightarrow} \mathtt{G}'$ and $Y' \Longrightarrow Y''$ with $(\mathtt{G}', Y'') \in \mathcal{R}$. Two states $\mathtt{G}$ and Y are weakly correspondent, written $\mathtt{G} \approx Y$, if and only if there exists a weak correspondence $\mathcal{R}$ such that $(\mathtt{G}, Y) \in \mathcal{R}$.

Theorem 4.4. *Let* $\mathtt{G}$ *be reachable from an initial, aware and balanced global type, then* $\mathtt{G} \approx \mathtt{G}\upharpoonright$.

(Theorem 4.4) follows from top-down and bottom-up fidelity, since $Y <: \mathtt{G}\upharpoonright$ implies $\mathtt{G} \approx Y$, which is mechanical by induction.

4.4 Further Properties of Systems

Local progress ensures that each configuration in a system can make further actions, unless it is in a final state (or it is final for short). Configuration $(\mathtt{p}, L, \sigma)$ is *final* if $L \equiv \mathtt{end}$, where $\equiv$ is the structural equivalence defined by the following rules: $\mathtt{end} \blacktriangleright \bullet \equiv \bullet \blacktriangleright \mathtt{end} \equiv \mathtt{end} \blacktriangleright \mathtt{end} \equiv \mathtt{end}$. Observe that $L \equiv \mathtt{end}$ implies that $\mathtt{p}$ has no further actions in L and is committed in all MC in L.

Definition 4.5 (Local Progress). Y enjoys progress if for all Y' reachable from Y:

$$(\mathtt{p}, \mathtt{L}, \sigma) \ \text{ in } \ Y' \ \text{ is not final} \quad \Rightarrow \quad Y' \longrightarrow^{*} \xrightarrow{\ell} \ \wedge\ \mathtt{sbj}(\ell) = \mathtt{p}$$

The Corollary 4.6 of Theorem 4.4 lifts global progress (Theorem 3.13) to systems.

Corollary 4.6 (Local Progress). *If* $\mathtt{G}$ *is initial, aware and balanced then* $\mathtt{G}\upharpoonright$ *enjoys progress.*

From local progress we further establish *orphan message freedom* (OMF), which states that every message in a queue can eventually be received.

Theorem 4.7 (OMF). *Let* $Y, (\mathtt{q}, \mathtt{L}, \sigma)$ *be a system reachable from the projection of an initial, aware, and balanced global type. If* $\sigma[\mathtt{p}] = \vec{m}_1 \cdot (a_k, \pi) \cdot \vec{m}_2$ *with* $(a_k, \pi) \notin \vec{m}_1$, *then there exists* $Y', (\mathtt{q}, \mathtt{L}', \sigma')$ *reachable from* $Y, (\mathtt{q}, \mathtt{L}, \sigma)$ *such that either (1)* $\mathtt{stale}(\pi, \mathtt{L}')$, *or (2)* $Y', (\mathtt{q}, \mathtt{L}', \sigma') \xrightarrow{\mathtt{pq}?(a_k, \pi)}$.

OMF in MST was first formulated [Deniélou and Yoshida 2012, 2013] to state that *final* (terminated) states have empty queues. Our Theorem 4.7 is more general because it also covers non-terminating systems. In our setting, we can state the weaker property of Deniélou and Yoshida modulo stale message purging as: if Y is *final* and derived from the projection of an initial, aware and balanced $\mathtt{G}$, then for all $(\mathtt{p}, \mathtt{L}, \sigma)$ in Y, $\mathtt{gc}(\mathtt{L}, \sigma) = \emptyset$. This follows immediately from Theorem 4.7: if a non-stale message were present in Y, by Theorem 4.7 there would exist Y' reachable from Y where that message is consumed or stale, contradicting the finality of Y.

A stronger version of OMF [Chen et al. 2017] has been studied in the setting of (binary) session types with fairness conditions by Padovani and Zavattaro [2025]. Due to MC, our Theorem 4.7

first differs by considering garbage collection (case (1) in Theorem 4.7). Secondly, their *fair termination* condition enforces that every message is *necessarily* received in every possible execution, whereas our Theorem 4.7 ensures that every enqueued message may always be *possibly* received or garbage collected (e.g., given $\mu \mathtt{t}.\, \mathtt{p} \twoheadrightarrow \mathtt{q} : a.\, \mathtt{t}$, it is always possible for q to receive but the unfair execution where p sends forever while q never receives is also possible). We leave to future work an investigation of fairness conditions à la Ciccone et al. [2024]; Padovani and Zavattaro [2025] for asynchronous MC in MST. Section 6 gives further comparisons with Ciccone et al. [2024].

5 Implementation and Further Examples

5.1 Toolchain Implementation

Global protocol validation. We have implemented mMST as an extension to the Scribble protocol language [Hu and Yoshida 2016; Yoshida et al. 2013]. The practical syntax for our MC is as demonstrated in Section 2. Following our formal theory, our implementation allows MC to be combined with the standard MST constructs for regular directed choice (non-mixed branch/select) and recursion to express a range of patterns and DS constructs found in practical applications.

Our toolchain internally translates Scribble specifications to a representation based on our formal definitions and *syntactically* checks the source protocol for (a) the well-formedness of committing message labels (Definition 3.4), (b) awareness (Definition 3.9) and balance (Definition 3.12), and (c) projectability (Section 4.2) onto all roles. In our current implementation, checking (b) syntactically means inferring role occurrences and dependencies between roles as written in the source protocol without (semantically) unfolding recursive types. This is sound: such syntactically inferred dependencies conservatively imply our formal conditions.

Code generation. The toolchain implements projection of global types to local types following the formal theory. Each local type is then translated to an event-driven finite state machine (EFSM) representation. As described in Section 2.2, from the EFSM we generate two Erlang modules for each role: a *role module* (RM) and a *callback module* (CM). In Figure 3, we showed the CM for role `B`. In Figure 7, we show the corresponding RM for `B`. The RM builds on `gen_statem`, providing an EFSM structure to manage state transitions and message passing in accordance with the protocol. Specifically, it provides state function definitions (Figure 7, lines 27-40), and callback specifications (Figure 7, lines 2-8) that must be implemented by the CM to handle incoming messages and internal events. It also provides functions for sending messages (Figure 7, lines 17-23). Moreover, it leverages `gen_statem`'s asynchronous selective receive to defer out-of-order events by returning a postpone action (e.g. `keep_state, Data, [postpone]`) (lines 27-28), requeuing them until the state machine transitions into a state that can properly handle them. Following Erlang convention, the RM excludes any application-specific logic, which instead is left to callbacks in the CM.

In mixed choices, participants may receive messages from non-selected branches until all participants become committed. To prevent protocol violations stemming from these stale messages, the toolchain implements event queue management and state data tracking within the `gen_statem` framework. The implementation follows the theory, and derives the stale messages to be purged from the global type, as defined in Section 4.1. `gen_statem` maintains an event queue where incoming events are enqueued and dispatched sequentially based on arrival order and their priority, with internal events being prioritised over external events. Each incoming MC message carries a path `Pi` recording the sides taken through active MCs. The RM keeps a local commitment map, and as illustrated in Figure 7, lines 22-26, when an event is received the RM module purges the event if the path `Pi` is stale, otherwise it forwards the event to the CM module.

Runtime requirements. Our implementation uses Erlang's core features for process spawning and asynchronous message passing, and the built-in `gen_statem` library for executing event-driven

```
% -------- Callback specifications --------
-callback s5(term(), {atom()}, state_data()) -> {stop, normal, state_data()}.
-callback s1(term(), {atom()} | {pid(), {term()}}, state_data()) ->
  {next_state, s5, state_data(), [{next_event, internal, {'TOc'}}]} | {keep_state, state_data()} |
  {keep_state, state_data(), [postpone]} | {next_state, s2, state_data(), [{next_event, internal, {
       a3}}]}.
-callback s2(term(), {atom()}, state_data()) ->
    {next_state, s3, state_data(), [{next_event, internal, {a4}}]} | {keep_state, state_data()}.
-callback s3(term(), {atom()}, state_data()) -> {stop, normal, state_data()}.
%% -------- Record and type definitions for maintaining state --------
-record(data, { a_pid :: pid(), c_pid :: pid() }). %% Process identifiers for roles A and C.
%% -------- Send helpers --------
send_s1_TOa(APid, Data) -> gen_statem:cast(APid, {self(), {'TOa'}, Path}).
send_s5_TOc(CPid, Data) -> gen_statem:cast(CPid, {self(), {'TOc'}, Path}).
send_s3_a4(APid, Data) -> gen_statem:cast(APid, {self(), {a4}, Path}).
send_s2_a3(CPid, Data) -> gen_statem:cast(CPid, {self(), {a3}, Path}).
%% ---------- Callback function definitions delegate the processing to the CM ----------
s1(EventType, {APid, {a1}, Pi}, Data) ->
  case stale(Pi, left) of
      true -> {keep_state, Data};
      false -> Side = CallbackModule:s1(EventType, {APid, {a1}}, Data)
      case Side of {next_state, s6, _} -> commit_entry(?MC1, left), Side;
                   {next_state, s3, _} -> commit_entry(?MC1, right), Side
      end;
  end;
s1(EventType, {'TOa'}, Data) -> Side = CallbackModule:s1(EventType, {'TOa'}, Data)
      case Side of {next_state, s6, _} -> commit_entry(?MC1, left), Side;
                   {next_state, s3, _} -> commit_entry(?MC1, right), Side
      end.
s5(EventType, 'TOc', Data) -> CallbackModule:s5(EventType, 'TOc', Data).
s2(EventType, a3, Data) -> CallbackModule:s2(EventType, a3, Data).
s3(EventType, a4, Data) -> CallbackModule:s3(EventType, a4, Data).
```

Fig. 7. Extract from the RM generated for B in Timeout, implementing its protocol- and role-specific gen_statem behaviour and declaring the callback functions to be implented by the CM.

finite state machines (EFSMs). However, a runtime for our theory can be implemented in any setting that supports event-driven concurrency and asynchronous messaging, over which a framework for session-based EFMSs can be readily developed [Viering et al. 2021] if not directly supported as in Erlang. Our protocol validation and projection is independent of Erlang.

5.2 Expressiveness by Examples

Table 1 summarises a range of examples from MST literature that we have extended with mixed choice (MC) to support features such as timeouts and exceptions. Our MC allows use cases like SMTP to be expressed more fully than in prior MST systems due to supporting timeouts (prior works simply disregarded that aspect). Our examples test the static elements of protocol validation, projection, EFSM translation and Erlang module generation. We also implemented minimal but functional skeleton programs for each role of the examples to test the runtime I/O and event handling dynamics, and embedded MC mechanisms such as stale message purging. However, our minimal implementations generally do not perform the full application logic of the examples.

Distributed system constructs. A key motivation for mMST is to provide a core construct that can express a range of important constructs – building blocks of many practical distributed systems – that were previously only supported by bespoke and disparate MST extensions. These include exceptions [Carbone et al. 2008], interrupts [Demangeon et al. 2015], timeouts [Hou et al. 2024;

Table 1. MST examples extended with asynchronous mixed choice (MC).

M=Multiparty B/S=Branch/Select Rec=Recursion nMC=nested MC ndMC=non-directed MC GC=purging

Example	M	B/S	Rec	MC	nMC	ndMC	GC	Source
Calculator	✓	✓		✓			✓	[Hu and Yoshida, 2016]
CircuitBreaker	✓	✓	✓	✓		✓	✓	[Lagaillardie, Neykova and Yoshida 2022]
DistributedLogging		✓	✓	✓			✓	[Lagaillardie, Neykova and Yoshida 2022]
Fibonacci		✓	✓	✓			✓	[Hu and Yoshida, 2016]
SMTP		✓	✓	✓	✓		✓	[Hu and Yoshida, 2016]
TwoBuyer	✓	✓		✓	✓	✓	✓	[Honda, Yoshida, Carbone, 2008]
TravelAgency	✓	✓	✓	✓			✓	[Hu, Yoshida and Honda 2008]
OnlineWallet	✓	✓	✓	✓		✓	✓	[Neykova, Yoshida and Hu, 2013]

```
1  global protocol FailH(role M, role W, role FD) {
2    init(Data) from M to W;
3    rec X {
4      mixed { // LHS
5        HB() from W to FD;
6        OK() from FD to M;
7        result(Data) from W to M;
8        more(Data) from M to W;
9        continue X;
10     } or { // RHS
11       Timeout() from FD to W; @'failed W'
12       Crash() from FD to M;
13 } } }
```

```
1  @'explicit-observer-left-commits'
2  global protocol Interr(role P, role Q) {
3    mixed { // LHS
4      Start() from Q to P;
5      rec X {
6        choice at Q {
7          More() from Q to P; continue X;
8        } or {
9          Stop() from Q to P*;  // P commits
10         Ack() from P to Q;
11     } }
12   } or { // RHS
13     Interrupt() from P to Q; } }
```

Fig. 8. Heartbeat-based failure detection (left), and an asynchronous interrupt (right).

Pears et al. 2023], and failure handling [Viering et al. 2018]. A basic timeout pattern was illustrated in Section 2; exceptions are similar. We illustrate interrupts and failure handling below.

Failure handling. We give an example of MC drawn from the major topic of failure handling and fault-tolerance in MST [Barwell et al. 2023; Brun and Dardha 2024; Peters et al. 2023; Viering et al. 2018, 2021]. The existing work typically models application protocols assuming that failure detection (FD) is implicitly provided by the runtime infrastructure. With MC as a core construct, we can now explicitly model such mechanisms and how role behaviours may depend on them. Figure 8 (left) gives a small example, where Worker `W` is a failure-prone role and the observer `FD` represents the FD service, based on the use cases of Viering et al. [2018, 2021] featuring heartbeat-based FD. We specify it as a recursive MC where in each iteration (i.e., unfolding of the recursive type) `FD` waits to receive a heartbeat (`HB`) from `W` with the option to send a `Crash` notification to `M` (e.g., upon a timeout, connection error, or other failure condition). The annotation on L11 is given by a simple variant of our core theory where protocol validation prohibits further usage of a role considered failed (i.e., `W` cannot occur again in the continuation of the protocol after L11). The `Timeout` message is useful if the connection is actually still live, allowing `W` to handle its own (reported) demise, but can be considered redundant otherwise. Following our theory, protocol validation allows the awareness of `W` in the LHS to be transitively relayed from `FD` to `M` via `OK`, then to `W` via `more`.

Interrupts. Recall the interrupt patterns using MC in Example 3.11. The interrupt pattern can be expressed in our Scribble extension as in Figure 8 (right).

We make an option for the user to explicitly indicate the committing interactions for observers in the LHS of an MC (as opposed to the observer implicitly committing on the first such message

received as in the core theory). The explicit committing interaction is marked by a ⋆ on the observer role occurrence, e.g., `P⋆` on Line 9.

5.3 Case Study: RabbitMQ

We apply our toolchain to a non-trivial, real-world case study, RabbitMQ. The toolchain can generate code for each participant in the chosen subset of the AMQP protocol, while also allowing for interoperability: the code generated for one participant (e.g., Consumer) can interoperate with pre-existing RabbitMQ implementations.

The Advanced Message Queuing Protocol[5] (AMQP) is an open-standard protocol designed for message-oriented middleware. RabbitMQ, a widely-used open-source message broker adhering to the AMQP standard, leverages the Erlang client library `amqp_client`[6] to facilitate interaction between Erlang and Elixir applications and RabbitMQ nodes. Within this ecosystem, the `amqp_selective_consumer` module, implemented alongside its behaviour `amqp_gen_consumer`, plays a crucial role in managing message consumption with precise control over delivery and cancellation. Notably, `amqp_selective_consumer` is already implemented as a state machine using Erlang's `gen_server` behaviour; however, this implementation is relatively ad-hoc. We replace the `amqp_selective_consumer`, and its implemented behaviour `amqp_gen_consumer` from the RabbitMQ Erlang client library, `amqp_client` with a behaviour and a callback module generated from a session type representation of the protocol.

We model the relevant part of the AMQP protocol in Scribble, Figure 9, capturing the interactions between the consumer `C`, channel `H`, and server `S` roles for selective message delivery. The consumer registers with the channel and initiates a `basic_consume` request, which is forwarded to the server. The server responds with `basic_consume_ok`, confirming the consumer's subscription. The `SelectiveMessageDelivery` recursive block models ongoing message exchanges, using an MC construct to represent message processing on one hand and cancellation on the other. MC is essential for accurately modelling the concurrency and asynchronous behaviour in AMQP. It allows the protocol to express that either the server may deliver a new message – `basic_deliver`, or the consumer may choose to cancel the subscription – `basic_cancel`.

Our toolchain validates this global type, projects it to a local type for each role, and constructs an EFSM representation of each local type. For the consumer role, `C`, the toolchain generates a correct-by-construction behaviour and callback modules. The generated modules replace `amqp_selective_consumer` and `amqp_gen_consumer`, implementing the same interfaces expected by other components. All features of `amqp_selective_consumer` are preserved, including message consumption, and cancellation. We use RabbitMQ's existing test suite to validate the new `amqp_selective_consumer` implementation.

6 Related Work, Limitations and Future Work

This paper presents the first *asynchronous multiparty* session type system with a core construct for *mixed* choice. There are two main lines of related work in the literature: session types with synchronous mixed choices, and MST extensions for bespoke exception-like constructs.

The literature includes several works on *synchronous* mixed choices for binary [Casal et al. 2020, 2022; Vasconcelos et al. 2020] and multiparty [Castagna et al. 2012; Jongmans and Ferreira 2023; Jongmans and Yoshida 2020; Peters and Yoshida 2024] session types. In synchronous settings, mixed choices behave very similarly to regular (*non*-mixed) choice: both can be modelled by an *atomic* reduction step (e.g., $G_1 + G_2 \to G_i'$ for $i \in \{1, 2\}$ and $G_i \to G_i'$). Reasoning about safety of *asynchronous*

[5] https://www.amqp.org/
[6] https://github.com/rabbitmq/rabbitmq-server/tree/main/deps/amqp_client

```
// roles: C = Consumer, H = Channel, S = Server
global protocol AMQP(role C, role H, role S) {
  register_default_consumer() from C to H;
  basic_consume(consumer_tag, nowait) from C to H;
  basic_consume(consumer_tag, nowait) from H to S;
  basic_consume_ok(consumer_tag) from S to H;
  basic_consume_ok(consumer_tag) from H to C;
  rec SelectiveMessageDelivery {
    mixed {
      basic_deliver(consumer_tag, delivery_tag,
          exchange, routing_key) from S to H;
      // Continued on right column...
    process_message() from H to C;
    basic_deliver(consumer_tag, delivery_tag,
        exchange, routing_key) from H to C;
    processing_complete(delivery_tag) from C to H;
    update_delivery_state(delivery_tag) from H to S;
    continue SelectiveMessageDelivery;
  } or {
    basic_cancel(consumer_tag, nowait) from C to H;
    basic_cancel(consumer_tag, nowait) from H to S;
    basic_cancel_ok(consumer_tag) from S to H;
    basic_cancel_ok(consumer_tag) from H to C;
  } } }
```

Fig. 9. Subset of AMQP using a recursive MC in our extended Scribble.

mixed choices (MC) and disciplining the inherent race conditions is a substantially different endeavour, as shown by our developments through the design of our MC, static validation, runtime mechanisms and metatheory. We note the work of Pears et al. [2023] on asynchronous *binary* session types with mixed choice, where each choice has a *timing* constraint. Their system allows choices with a mix of input and output actions, but they statically enforce that actions in different directions are never viable at the same point in *time*. Similar approaches based on timed session types [Hou et al. 2024] share this limitation. By contrast, we deal with bona fide asynchronous MC where both communication directions w.r.t. all pairs of participants are concurrently viable.

This paper aims to distill the *essence* of mixed choice in asynchronous (M)ST, various aspects of which were studied in several areas; e.g., interaction exceptions [Carbone et al. 2008] (binary), interaction handlers [Capecchi et al. 2016] (with synchronous triggers), interruptible blocks [Demangeon et al. 2015] (where default/interrupted blocks share common continuations), and internal exceptions [Fowler et al. 2019] (local failure control, no type construct). We have shown how MC can express multiparty asynchronous timeouts and interrupts; exceptions are similar.

A crucial area for real-world applicability of session types is support for failure handling, where aspects of mixed choice arise intrinsically. Viering et al. [2018, 2021] developed an MST system with try-handle constructs for handling *partial* failures of a protocol due to (suspected) participant crashes. In these works a process may be faced with a choice between outputs in the normal protocol flow mixed with potential inputs related to failure detection/notification. The (supposed) failure of a participant rules out all interactions with that participant thereafter. By contrast, our MC has a finer-grained notion of commitment that is *per instance* of an MC (not per protocol). Section 5.2 demonstrated such a failure handling pattern using our MC. Barwell et al. [2023] present an approach to failure handling in MST that avoids syntactic extensions (cf. [Bettini et al. 2008]). Their system designates a special message label crash to denote failure of the "sending" participant, which otherwise behaves as a regular label in choice constructs. E.g., $\mathsf{p} \twoheadrightarrow \mathsf{q} : \{a.\,\mathsf{G}_1, \mathsf{crash}.\mathsf{G}_2\}$ specifies that q awaits either message a from p or notification of p's crash. As in the related work above, crash events may occur concurrently with the I/O actions of the main protocol flow and their system incorporates some related machinery (e.g., queue cleaning); however, neither their user-level types nor processes have explicit constructs for mixed choice.

Communicating systems featuring mixed choice have been studied outside of session types. Communicating automata [Brand and Zafiropulo 1983] for instance do not rule out mixed choices but progress is, in general, undecidable. Lange et al. [2018] present a tool that infers *behavioural types* [Hüttel et al. 2016] for channel-based, shared-memory concurrency programs in Go. Their

system permits mixed choice for the `select` construct and assumes finitely buffered channels, enabling decidable model checking (e.g., deadlock-freedom) of the inferred types. Our work instead supports asynchronous communication over unbounded channels.

Recently, Li et al. [2023] presented the first sound *and complete* projection method for global types generalised with *sender-driven* choice ($\mathsf{p} \to \{\mathsf{q}_i : m_i.G_i\}_I$). Their automata-theoretic approach separates local machine synthesis from implementability checking, but, like Majumdar et al. [2021], do not support mixed choice. By contrast, our work builds on classical MST with regular (non-mixed) choices restricted to directed choice and syntactic projection. This is because our asynchronous MC is influenced by the other works discussed above on, e.g., interrupts and failure handling.

Ciccone et al. [2024] presented an MST system for *fair termination* in a synchronous session π-calculus without mixed choice. Their system guarantees processes (with multiple sessions) will fairly terminate by combining: a validation that session types always potentially terminate, a restricted fusing of session initiation and process spawning [Caires et al. 2016; Wadler 2014], a notion of ranking that limits processes to finite behaviours, a liveness-preserving subtyping relation, and a fairness assumption on executions (potential termination leads to actual termination). There may be connections between our notion of clear termination, that is per MC, with their notion of (whole) protocol termination; at present, our system differs in that we allow a session to be unbounded provided each MC individually satisfies awareness. Padovani and Zavattaro [2025] recently developed fair termination for asynchronous binary sessions (as mentioned in Sec. 4.4).

Limitations and future work. Our system builds on classical MST with regular choices restricted to directed choice, and syntactic projection and merge [Scalas and Yoshida 2019; Stutz 2023]. This yields a projection that is sound but not complete in the base MST constructs (directed choice, recursion), let alone with MC. In future work, we plan to investigate (conservative forms of) mixed choice in the generalised automata-theoretic setting of Li et al. [2023]. The challenges include reconciling their language-based approach with our mechanisms (e.g., stale message purging) and metatheory (e.g., operational correspondence and preservation of projection). Another issue is that their projection yields state machines that are more general than local types: accepting (terminating) states may have outgoing transitions, and choice branches may be unbalanced across roles. Such generality must be reconciled with the objective of our system (and classical MST) that all projected behaviours, including MCs, be realisable as fully distributed processes; e.g., we typically aim to rule out protocols where termination could be non-deterministic (cf. Def. 3.9). Building on the above, Stutz and D'Osualdo [2025] recently extended projection to a larger class of automata-based global specifications, but show that the projectability of *mixed* choice in their more general setting is undecidable. Their negative result motivates pragmatic approaches to supporting mixed choice such as in this paper. The automata-based system of Lange and Yoshida [2019] may also provide avenues for lifting directed choice and generalising our protocol validation.

One of our present limitations is that our system does not incorporate *delegation* [Bettini et al. 2008; Honda et al. 2008]. Considering our MC concepts (e.g., stale message purging and path identifiers) in a setting with delegation is a topic for future work.

An interesting question is how fair multiparty termination [Ciccone et al. 2024] may be extended to asynchronous mixed choices. One direction could be to investigate fair termination for our notion of MC global/local types and formalise a process-level language. Unlike the 'multithreaded' π-calculi in the mentioned works, however, the model of concurrency in our practical Erlang setting is event-driven [Hu et al. 2010; Viering et al. 2021] and does not have a specific primitive for fusing session initiation and process spawning. Adapting their notion of static typing of process ranks may be a challenge in languages such as Erlang.

Data-Availability Statement

The source code of our toolchain [Bocchi et al. 2026b], examples and RabbitMQ case study is available online.[7]

Acknowledgments

This work was partially funded by EPSRC project EP/T014512/1, EP/T014628/1 (STARDUST).

References

Manuel Adameit, Kirstin Peters, and Uwe Nestmann. 2017. Session Types for Link Failures. In *FORTE (Lecture Notes in Computer Science, Vol. 10321)*. Springer, 1–16.

Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. 2023. Designing Asynchronous Multiparty Protocols with Crash-Stop Failures. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:30. doi:10.4230/LIPICS.ECOOP.2023.1

Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. 2022. Generalised Multiparty Session Types with Crash-Stop Failures. In *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland (LIPIcs, Vol. 243)*, Bartek Klin, Slawomir Lasota, and Anca Muscholl (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 35:1–35:25. doi:10.4230/LIPICS.CONCUR.2022.35

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR (LNCS, Vol. 5201)*. 418–433.

Laura Bocchi, Raymond Hu, Adriana Laura Voinea, and Simon Thompson. 2026a. Mixed Choice in Asynchronous Multiparty Session Types. *arXiv preprint arXiv:2602.23927* (2026). doi:10.48550/arXiv.2602.23927

Laura Bocchi, Raymond Hu, Adriana Laura Voinea, and Simon Thompson. 2026b. *Mixed Choice in Asynchronous Multiparty Session Types Artifact*. doi:10.5281/zenodo.18808387

Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (apr 1983), 323?342. doi:10.1145/322374.322380

Matthew Alan Le Brun and Ornela Dardha. 2024. MAGπ!: The Role of Replication in Typing Failure-Prone Communication. In *Formal Techniques for Distributed Objects, Components, and Systems - 44th IFIP WG 6.1 International Conference, FORTE 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14678)*, Valentina Castiglioni and Adrian Francalanza (Eds.). Springer, 99–117. doi:10.1007/978-3-031-62645-6_6

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear logic propositions as session types. *Math. Struct. Comput. Sci.* 26, 3 (2016), 367–423. doi:10.1017/S0960129514000218

Sara Capecchi, Elena Giachino, and Nobuko Yoshida. 2016. Global escape in multiparty sessions. *Math. Struct. Comput. Sci.* 26, 2 (2016), 156–205. doi:10.1017/S0960129514000164

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2008. Structured Interactional Exceptions in Session Types. In *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5201)*, Franck van Breugel and Marsha Chechik (Eds.). Springer, 402–417. doi:10.1007/978-3-540-85361-9_32

Filipe Casal, Andreia Mordido, and Vasco T. Vasconcelos. 2020. Mixed Sessions: the Other Side of the Tape. In *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020 (EPTCS, Vol. 314)*, Stephanie Balzer and Luca Padovani (Eds.). 46–60. doi:10.4204/EPTCS.314.5

Filipe Casal, Andreia Mordido, and Vasco T. Vasconcelos. 2022. Mixed sessions. *Theor. Comput. Sci.* 897 (2022), 23–48. doi:10.1016/J.TCS.2021.08.005

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2012. On Global Types and Multi-Party Session. *Log. Methods Comput. Sci.* 8, 1 (2012). doi:10.2168/LMCS-8(1:24)2012

Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. 2017. On the Preciseness of Subtyping in Session Types. *Log. Methods Comput. Sci.* 13, 2 (2017). doi:10.23638/LMCS-13(2:12)2017

Tzu-Chun Chen, Malte Viering, Andi Bejleri, Lukasz Ziarek, and Patrick Eugster. 2016. A Type Theory for Robust Failure Handling in Distributed Systems. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9688)*, Elvira Albert and Ivan Lanese (Eds.). Springer, 96–113. doi:10.1007/978-3-319-39570-8_7

[7] https://github.com/rhu1/scribble-gt-scala/tree/artifact

Luca Ciccone, Francesco Dagnino, and Luca Padovani. 2024. Fair termination of multiparty sessions. *J. Log. Algebraic Methods Program.* 139 (2024), 100964. doi:10.1016/J.JLAMP.2024.100964

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. A Gentle Introduction to Multiparty Asynchronous Session Types. In *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures (Lecture Notes in Computer Science, Vol. 9104)*, Marco Bernardo and Einar Broch Johnsen (Eds.). Springer, 146–178. doi:10.1007/978-3-319-18941-3_4

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. Comput. Sci.* 26, 2 (2016), 238–302.

Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods Syst. Des.* 46, 3 (2015), 197–225. doi:10.1007/S10703-014-0218-8

Pierre-Malo Deniélou and Nobuko Yoshida. 2012. Multiparty Session Types Meet Communicating Automata. In *ESOP (LNCS, Vol. 7211)*. 194–213.

Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *ICALP (LNCS, Vol. 7966)*. 174–186.

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.* 3, POPL (2019), 28:1–28:29. doi:10.1145/3290341

Mohamed G. Gouda, Eric G. Manning, and Yao-Tin Yu. 1984. On the Progress of Communications between Two Finite State Machines. *Inf. Control.* 63, 3 (1984), 200–216. doi:10.1016/S0019-9958(84)80014-5

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 273–284. doi:10.1145/1328438.1328472

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. doi:10.1145/2827695

Ping Hou, Nicolas Lagaillardie, and Nobuko Yoshida. 2024. Fearless Asynchronous Communications with Timed Multiparty Session Protocols. In *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria (LIPIcs, Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:30. doi:10.4230/LIPICS.ECOOP.2024.19

Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. 2010. Type-Safe Eventful Sessions in Java. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 329–353. doi:10.1007/978-3-642-14107-2_16

Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9633)*, Perdita Stevens and Andrzej Wasowski (Eds.). Springer, 401–418. doi:10.1007/978-3-662-49665-7_24

Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016), 3:1–3:36. doi:10.1145/2873052

Grant Iraci, Cheng-En Chuang, Raymond Hu, and Lukasz Ziarek. 2023. Validating IoT Devices with Rate-Based Session Types. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1589–1617. doi:10.1145/3622854

Sung-Shik Jongmans and Francisco Ferreira. 2023. Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types (Pearl/Brave New Idea). In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 42:1–42:30. doi:10.4230/LIPICS.ECOOP.2023.42

Sung-Shik Jongmans and Nobuko Yoshida. 2020. Exploring Type-Level Bisimilarity towards More Expressive Multiparty Session Types. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 251–279. doi:10.1007/978-3-030-44914-8_10

Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A static verification framework for message passing in Go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1137–1148. doi:10.1145/3180155.3180157

Julien Lange and Nobuko Yoshida. 2019. Verifying Asynchronous Interactions via Communicating Session Automata. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 97–117. doi:10.1007/978-3-030-25540-4_6

Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. 2023. Complete Multiparty Session Type Projection with Automata. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 13966)*, Constantin Enea and Akash Lal (Eds.). Springer, 350–373. doi:10.1007/978-3-031-37709-9_17

Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. 2021. Generalising Projection in Asynchronous Multiparty Session Types. In *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference (LIPIcs, Vol. 203)*, Serge Haddad and Daniele Varacca (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 35:1–35:24. doi:10.4230/LIPICS.CONCUR.2021.35

Luca Padovani and Gianluigi Zavattaro. 2025. Fair Termination of Asynchronous Binary Sessions. In *39th European Conference on Object-Oriented Programming, ECOOP 2025, June 30 to July 2, 2025, Bergen, Norway (LIPIcs, Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:29. doi:10.4230/LIPICS.ECOOP.2025.24

Jonah Pears, Laura Bocchi, and Andy King. 2023. Safe Asynchronous Mixed-Choice for Timed Interactions. In *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13908)*, Sung-Shik Jongmans and Antónia Lopes (Eds.). Springer, 214–231. doi:10.1007/978-3-031-35361-1_12

Kirstin Peters, Uwe Nestmann, and Christoph Wagner. 2023. FTMPST: Fault-Tolerant Multiparty Session Types. *Log. Methods Comput. Sci.* 19, 4 (2023). doi:10.46298/LMCS-19(4:14)2023

Kirstin Peters and Nobuko Yoshida. 2024. Separation and Encodability in Mixed Choice Multiparty Sessions (Technical Report). *CoRR* abs/2405.08104 (2024). arXiv:2405.08104 doi:10.48550/ARXIV.2405.08104 [To appear at LICS '24.].

Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* 3, POPL (2019), 30:1–30:29. doi:10.1145/3290343

Felix Stutz. 2023. Asynchronous Multiparty Session Type Implementability is Decidable - Lessons Learned from Message Sequence Charts. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:31. doi:10.4230/LIPICS.ECOOP.2023.32

Felix Stutz and Emanuele D'Osualdo. 2025. An Automata-theoretic Basis for Specification and Type Checking of Multiparty Protocols. *CoRR* abs/2501.16977 (2025). arXiv:2501.16977 doi:10.48550/ARXIV.2501.16977

Vasco T. Vasconcelos, Filipe Casal, Bernardo Almeida, and Andreia Mordido. 2020. Mixed Sessions. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 715–742. doi:10.1007/978-3-030-44914-8_26

Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu, and Lukasz Ziarek. 2018. A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 799–826. doi:10.1007/978-3-319-89884-1_28

Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. 2021. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. doi:10.1145/3485501

Philip Wadler. 2014. Propositions as sessions. *J. Funct. Program.* 24, 2-3 (2014), 384–418. doi:10.1017/S095679681400001X

Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2013. The Scribble Protocol Language. In *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8358)*, Martín Abadi and Alberto Lluch-Lafuente (Eds.). Springer, 22–41. doi:10.1007/978-3-319-05119-2_3