Asynchronous Team Automata

Davide Basile[1], Maurice H. ter Beek[1], and José Proença(✉)[2]

[1] Formal Methods and Tools lab, CNR–ISTI, Pisa, Italy
`{davide.basile,maurice.terbeek}@isti.cnr.it`
[2] CISTER & University of Porto, Porto, Portugal
`jose.proenca@fc.up.pt`

Abstract. Team automata were introduced as a flexible extension of I/O automata to model collaborative behaviour in component-based and distributed systems. Their distinctive features include multi-party communication and a liberal synchronisation mechanism: components may jointly execute shared actions according to synchronisation policies that specify which subsets of components participate as senders or receivers. While this makes team automata well suited for modelling coordination, existing communication is synchronous and therefore insufficient for capturing certain behavioural aspects (e.g., due to message reordering) of modern networks and distributed systems, in which communication is typically asynchronous and message delays are unpredictable.
In this paper, we introduce asynchronous team automata (ATeams), which extend team automata with buffers to model asynchronous communication, in addition to conventional synchronous interaction. ATeams support individual interactions involving multiple senders and receivers, unlike well-known asynchronous models such as communicating finite-state machines and multi-party session types. We formalise the syntax and operational semantics of ATeams, study well-formedness and well-behavedness conditions, and present the prototypical A-Team tool that supports specification, animation and automated checks. This proposes ATeams as a unifying semantic foundation for modelling and analysis of heterogeneous synchronous–asynchronous multi-party interactions.

1 Introduction

Team automata have originally been proposed for modelling components of groupware systems and their interconnections [38]. They were inspired by Input/Output (I/O) automata [53] and in particular inherit their distinction between internal (private, i.e., not observable by other automata) actions and external (i.e., input and output) actions used for communication with the environment (i.e., other automata). The underlying philosophy was that automata cooperate and collaborate by jointly executing (synchronising) transitions with the same action label (but possibly of different nature, i.e., input or output) as agreed upfront. Team automata were formally defined in terms of component automata that synchronise on certain executions of (common) actions according to specific synchronisation policies [19]. Technically, team automata are an extension of I/O automata [24], since a number of the restrictions of I/O automata (in particular input-enabledness and disjointness of output actions) were dropped for more flexible modelling of several kinds of interactions in groupware systems. Unlike I/O automata, team automata impose hardly any restrictions on the role of actions in components and their composition is not limited to the synchronous product. Composing a team automaton requires defining its transitions by providing the actions and synchronisations that can take place from the combined states of the components. Each team automaton is thus a composite automaton, defined over component automata, the transitions of which are exactly those combinations of component transitions that represent a synchronisation on a common action by all the components that share that action (possible involving multiple sending and receiving actions). The effect of a synchronously executed action on the state of the composed automaton is described in terms of the local state changes of the automata that take part in the synchronisation. The distinguishing feature of team automata is this very loose nature of synchronisation determined by synchronisation policies that define how many component automata can participate in the synchronised execution of a shared external action, either as a sender (i.e., via an output action) or as a receiver (i.e., via an input action). This flexibility makes team automata capable of capturing in a precise manner a variety of notions related to coordination in distributed systems (of systems).

Motivation Most distributed systems and real networks (e.g., the Internet) are asynchronous, with unpredictable message delays. Modelling this explicitly thus allows the verification of subtle properties (e.g., deadlocks, causal consistency) that depend on message reordering and delay. Synchronous models, on the other hand, can silently hide certain behaviour (e.g., long message delays, message reordering) that does occur in practice and may lead to bugs in actual deployment. This has been acknowledged in actor-based and object-based programming models and languages such as Erlang, Rebeca, ABS, and Akka [4,52,47,46,41,1,44].

Contribution We investigate *asynchronous* team automata (ATeams), in which component automata use either FIFO channels or multisets to act as buffers for asynchronous communication (like in asynchronous multi-party session types [43] or communicating finite state machines (CFSMs) [29,34]), in addition to synchronous communication based on simultaneous execution of shared actions (as common in related models like contract automata [12,11] or choreography automata [7,8]). However, CFSMs and related asynchronous models are based on peer-to-peer communication. Therefore, as sketched in Fig. 1, the challenge is how to generalise their approach by taking into account multi-party communication specified by synchronisation types. Moreover, multi-party session types and other choreography models typically enrich local actions with the names of the communication partners (e.g., a binary interaction $n \to m : \mathtt{a}$ leads to a local output action $n\, m ! \mathtt{a}$ for n and a local input action $n\, m ? \mathtt{a}$ for m). In team automata and other component-based development approaches supporting interface-based design [53,2,19,51], on the other hand, transitions describing local behaviour are

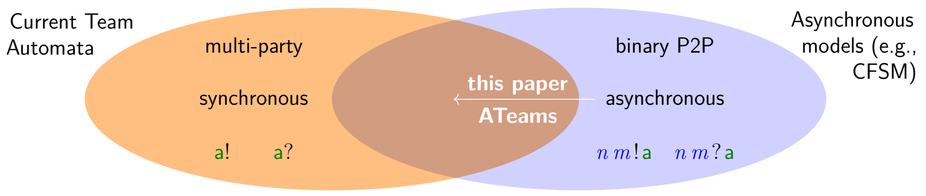


Fig. 1. This paper's challenge: equip team automata with asynchronous communication

often labelled just by message names accompanied by information on whether the message is output or input of a component (e.g., a binary interaction $n \rightarrow m : \mathsf{a}$ leads to a local output action $\mathsf{a}!$ for n and a local input action $\mathsf{a}?$ for m). In [22], we coined the former "rich local actions" and the latter "poor local actions". From a software designer's point of view, the poor localisation style better supports the principle of loose coupling, whereas the rich style better avoids undesired synchronisations.

To address the challenge outlined above and sketched in Fig. 1, we formalise the syntax and operational semantics of ATeams, provide well-formedness and well-behavedness conditions that guarantee desirable properties regarding the structure and behaviour of ATeams, and present the prototypical A-Team tool that supports the specification, animation and automated checks of ATeams.

Outline We start by motivating ATeams via elementary illustrative examples (in Section 2.1), followed by defining its syntax (in Section 2.2) and semantics (in Section 2.3). We then present desirable properties of ATeams with respect to its structure (well-formedness, in Section 2.4) and its behaviour (well-behavedness, in Section 2.5). and describe the A-Team tool to animate and analyse systems in ATeams, which supports fast experimentation (in Section 3). We discuss related work in Section 4 and conclude in Section 5.

2 Asynchronous Team Automata

This section introduces *asynchronous team automata* (ATeams), starting with a motivating example before presenting the syntax, operational semantics, and some desirable properties. We refer to a concrete instance (automaton) of ATeams as a *system*, consisting of a set of communicating concurrent components (automata) enriched with synchronisation types restricting their communication.

We have developed in Scala and JavaScript a prototypical companion tool, called the A-Team, to analyse ATeams. This open-source tool can be used online or downloaded at https://fm-dcc.github.io/a-team and is described in Section 3. All examples in this paper are predefined in the tool, like the synchronous Race, including a hyperlink to open the tool with the specific example.

```
// Synchronous Race
acts
 start:  1->2, sync;
 finish: 1->1, sync;
proc
 Ctr = start!.finish?.finish?.Ctr
 R = start?.finish!.R
init
 Ctr || R || R
```

```
// Async. Race (sender)
acts
 start:  1->2, fifo@snd;
 finish: 2->1, fifo@snd;
proc
 Ctr = start!.finish?r1,r2.Ctr
 R = start?c.finish!.R
init
 c:Ctr || r1:R || r2:R
```

```
// Async. Race (global)
acts
 start: 1->2, fifo@global;
 finish:2->1, fifo@global;
proc
 Ctr = start!.finish?.Ctr
 R = start?.run!.finish!.R
init
 Ctr || R || R
```

Fig. 2. Variations of the Race example in ATeams

2.1 Motivating Race Example

We illustrate ATeams using different variations of the Race example, specified in Fig. 2 and borrowed from recent work on Team Automata [18,23]. The leftmost version is the synchronous version of Team Automata, in which a controller `Ctr` simultaneously asks two runners `R` to start a race, who then reply one at a time once they finish. The initial **`acts`** block declares that the action start can be used to communicate between one sender and two receivers (`1->2`), and uses synchronous communication, while the finish uses 1-to-1 communication. The second **`proc`** block defines a set of recursive processes using a variation of CCS [54], and the third **`init`** block specifies the agents being executed.

The other Race examples use FIFO buffers to communicate asynchronously. The middle Race example specifies that the start actions must be buffered at the sender side (fifo@snd). Consequently, the runners must specify who is the sender, writing start ? c to state that the buffer can be found in agent c. Similarly for finish. The rightmost Race example instead uses a single global FIFO queue, shared by all messages, hence no references to senders or receivers are needed by the components. This freedom to intertwine different interaction patterns, and even different buffer types, facilitate experimenting with the impact of modifying the communication channels. One can specify, e.g., the notion of mailbox in Rebeca [1], and the notions of multicast and broadcast in computer networks.

The semantics of these three Race systems are not the same. The synchronous semantics can more compactly describe, in a single transition, the execution of actions from multiple agents, while the asynchronous semantics requires more actions for each sending and receiving. More importantly, these versions can have undesirable behaviour. For instance, the middle Race example can deadlock if the same runner reads the start message before the other runner. The rightmost Race example does not deadlock, but allows a runner to start twice before the other runner starts, which may not be the intention. All such undesired behaviour can be detected using our A-Team implementation online (cf. Section 3).

2.2 Syntax

We consider a global set of agent names $n \in \mathcal{N}$, action names $\mathsf{a} \in \mathcal{A}$, process names $K \in \mathcal{K}$. We also write $P \in \mathcal{P}$ to range over process definitions and $st \in \mathcal{S}$ to range over synchronisation types, defined in Fig. 3.

$$
\begin{array}{rll}
P & := K \mid \mathbf{0} \mid \alpha.P \mid P+P & \text{(process)}\\
\alpha & := \mathsf{a}! \mid \mathsf{a}? \mid \mathsf{a}!\overline{n} \mid \mathsf{a}?\overline{n} & \text{(action)}\\
st & := (\mathsf{sync}, intrv, intrv) \mid (\mathsf{async}, intrv, intrv, buft, loct) & \text{(synchronisation type)}\\
intrv & := (i,i) \mid (i,\infty) & \text{(interval)}\\
buft & := \mathsf{fifo} \mid \mathsf{bag} \mid \ldots & \text{(buffer type)}\\
loct & := \mathsf{@snd} \mid \mathsf{@rcv} \mid \mathsf{@global} \mid \mathsf{@snd\text{-}rcv} & \text{(location type)}
\end{array}
$$

Fig. 3. Syntax of processes and synchronisation types, where $i \in \mathbb{N}$ and $\overline{n}$ denotes a non-empty sequence of agent names in $\mathcal{N}$

A *process* P is a variation of a CCS process [54] where (1) send and receive actions can explicitly state who are the target processes, and (2) each interaction can involve multiple senders and receivers (as in team automata). Internal actions, which are actions not used in the interactions [18], are not considered here for simplicity. A *synchronisation type* describes the *synchronisation mode* of a given send/receive action, i.e., whether communication is (1) synchronous (requiring interaction between all senders and receivers to occur atomically), or (2) asynchronous (using a buffer for interaction between senders and receivers). In the asynchronous case, actions are sent without blocking and are later consumed by their corresponding receivers. Depending on the *location type*, a new buffer is instantiated for each sender (@snd), for each receiver (@rcv), globally (@global), or for each pair of sender and receiver (@snd-rcv).

Finally, a *system* is defined as a triple $sys = \langle \gamma; \sigma; \rho \rangle$, in which $\gamma : \mathcal{K} \to \mathcal{P}$ is a function mapping process names to their (recursive) definitions, $\sigma : \mathcal{A} \to \mathcal{S}$ maps action names to their synchronisation types, and $\rho : \mathcal{N} \to \mathcal{P}$ maps agent names to their process definitions.

2.3 Operational Semantics

We define the semantics of a system using a small-step (operational) semantics over a configuration, which intuitively captures the current state of the processes and of the buffers of the asynchronous channels. We consider global sets of *locations* $\mathcal{L} = (\mathcal{N} \cup \{\bot\}) \times (\mathcal{N} \cup \{\bot\})$ and *buffers* $\mathcal{B}$, writing ℓ and B to range over $\mathcal{L}$ and $\mathcal{B}$, respectively.

Intuitively, each location consists of an optional sender $snd \in \mathcal{N}$ and an optional receiver $rcv \in \mathcal{N}$. A location type denotes whether the sender, the receiver, or both are specified. For example, we write $\ell = (n, m)$ to denote the location where n is the sender and m is the receiver (i.e., ℓ has location type @snd-rcv). In this case, both the sender and the receiver know each other. In locations of type @snd (e.g., $\ell = (n, \bot)$) the receiver knows who deposited the message in the buffer, i.e., the sender n, but the sender does not know who will consume this message, i.e., it could be any receiver. This is analogous for locations of type @rcv (e.g., $\ell = (\bot, m)$). In locations of type @global, where $\ell = (\bot, \bot)$, the sender is unknown to the receiver and the receiver is unknown to the sender.

$$\text{CONST}\ \frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'}\ K{:}P \in \gamma \qquad \text{PREFIX}\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{SUM-L}\ \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \text{SUM-R}\ \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

$$\text{SYNC}\ \frac{\begin{array}{c} \mathsf{a}{:}(\mathsf{sync}, \mathit{from}, \mathit{to}) \in \sigma \qquad \# \mathit{snds} \in \mathit{from} \qquad \# \mathit{rcvs} \in \mathit{to} \\ \mathit{snds} \subseteq \{n{:}s' \mid n{:}s \in \rho,\ s \xrightarrow{\mathsf{a}!} s'\} \cup \{n{:}s' \mid n{:}s \in \rho,\ s \xrightarrow{\mathsf{a}!\overline{m}} s', \overline{m} \subseteq \mathsf{dom}(\mathit{rcvs})\} \\ \mathit{rcvs} \subseteq \{n{:}r' \mid n{:}r \in \rho,\ r \xrightarrow{\mathsf{a}?} r'\} \cup \{n{:}r' \mid n{:}r \in \rho,\ r \xrightarrow{\mathsf{a}?\overline{m}} r', \overline{m} \subseteq \mathsf{dom}(\mathit{snds})\} \end{array}}{\langle \rho, \beta \rangle \xrightarrow{\mathsf{dom}(\mathit{snds}) \to \mathsf{dom}(\mathit{rcvs}) : a} \langle \rho \uplus \mathit{snds} \uplus \mathit{rcvs}, \beta \rangle}$$

Fig. 4. Operational rules for processes and synchronous communication

Each buffer B is an algebraic structure such that $B' = B \oplus \mathsf{a}$ adds an element a to the buffer, and $B' = B \ominus \mathsf{a}$ consumes a from the buffer. Our semantics considers that $\oplus$ always succeeds[3] (non-blocking additions) and $\ominus$ only succeeds if the given value is available. Our prototypical implementation (cf. Section 3) currently supports unbounded FIFO queues (describing order-preserving communication) and multisets (describing unsorted message passing, called bags).

Finally, a *configuration* in our semantics is a pair $\langle \rho, \beta \rangle$, with ρ as before and $\beta : \mathcal{L} \to \mathcal{B}$ maps locations to buffers. A configuration evolves by the transition $\langle \rho, \beta \rangle \xrightarrow{\lambda}_{\gamma,\sigma} \langle \rho', \beta' \rangle$, where λ is a global label for ATeams that can either be an (asynchronous) action α or a (synchronous) interaction $\overline{n} \to \overline{m} : \mathsf{a}$, and γ, σ provide the context with a fixed set of definitions of processes and synchronisation types. The operational rules are presented in Fig. 4 (for synchronous communication) and Figs. 5 and 6 (for asynchronous communication), using the notation introduced below. Given a system $\langle \gamma, \sigma, \rho \rangle$, the initial configuration is $\langle \rho, \beta_0 \rangle$, where β_0 initialises each buffer to an empty buffer, based on the location types and on the set of existing agent names, and the context is given by γ and σ.

Notation Given a function M, we write $M \uplus M'$ to denote the function M updated with the function M', i.e., $(M \uplus M')(k) = v$ iff $M'(k) = v$ or $M(k) = v$ whenever $M'(k)$ is undefined. Moreover, we write $k{:}v \in M$ whenever $M(k) = v$ and we write $\mathsf{dom}(M)$ to denote the domain of M. Finally, we write $\langle \rho, \beta \rangle \xrightarrow{\lambda} \langle \rho', \beta' \rangle$ instead of $\langle \rho, \beta \rangle \xrightarrow{\lambda}_{\gamma,\sigma} \langle \rho', \beta' \rangle$, omitting the context for simplicity.

Example Our Race example with fifo@snd communication is formalised by the system $\langle \gamma, \sigma, \rho \rangle$, with $\gamma = \{\mathsf{start}{:}(\mathsf{async}, (1,1), (2,2), \mathsf{fifo}, \mathsf{@snd}), \mathsf{finish}{:}(\mathsf{async}, (2,2), (1,1), \mathsf{fifo}, \mathsf{@snd})\}$, $\sigma = \{\mathtt{Ctr}{:}\mathsf{start}!.\mathsf{finish}?\mathit{r1},\mathit{r2}.\mathtt{Ctr}, \mathtt{R}{:}\mathsf{start}?.\mathsf{finish}!\mathit{c}.\mathtt{R}\}$, and $\rho = \{c{:}\mathtt{Ctr}, \mathit{r1}{:}\mathtt{R}, \mathit{r2}{:}\mathtt{R}\}$. The initial configuration is formalised by the pair $\langle \rho, \beta_0 \rangle$, with $\beta_0 = \{\, (\bot, n){:}\mathsf{EmptyFifo} \mid n \in \{c, \mathit{r1}, \mathit{r2}\}\,\}$. Initially, only rule SEND@SND can be applied, evolving by $\langle \rho, \beta_0 \rangle \xrightarrow{c:\mathsf{start}!} \langle \rho', \beta' \rangle$, where ρ' maps c to $\mathsf{finish}?\mathit{r1},\mathit{r2}.\mathtt{Ctr}$ and β' maps $(c, \bot)$ to a FIFO buffer with two occurrences of start.

[3] Exceptionally, $\oplus$ fails when trying to add an action associated with a buffer type that differs from the existing buffer. Well-formedness conditions (in Section 2.4) guarantee that this will never happen.

$$\textsc{send@rcv}\ \dfrac{\begin{array}{cccc}\mathsf{a}{:}(\mathsf{async}, \_\_, \mathit{to}, \_\_, \texttt{@rcv}) \in \sigma & n{:}s \in \rho & s \xrightarrow{\mathsf{a}\,!\,\overline{m}} s' & \#\overline{m} \in \mathit{to} \\ \text{all} \oplus \text{succeed} & & \overline{m} \subseteq \gamma & \end{array}}{\langle \rho, \beta \rangle \xrightarrow{\mathsf{a}\,!\,\overline{m}} \langle \rho \uplus \{n{:}s'\}, \beta \uplus \{(\bot, m){:}(\beta(\bot, m) \oplus \mathsf{a}) \mid m \in \overline{m}\}\rangle}$$

$$\textsc{send@snd}\ \dfrac{\mathsf{a}{:}(\mathsf{async}, \_\_, (t,t), \_\_, \texttt{@snd}) \in \sigma \quad n{:}s \in \rho \quad s \xrightarrow{\mathsf{a}!} s' \quad \text{all} \oplus \text{succeed}}{\langle \rho, \beta \rangle \xrightarrow{\mathsf{a}!} \langle \rho \uplus \{n{:}s'\}, \beta \uplus \{(n, \bot){:}(\beta(n, \bot) \oplus \underbrace{\mathsf{a} \oplus \cdots \oplus \mathsf{a}}_{t \text{ times}})\}\rangle}$$

$$\textsc{send@glob}\ \dfrac{\mathsf{a}{:}(\mathsf{async}, \_\_, (t,t), \_\_, \texttt{@global}) \in \sigma \quad n{:}s \in \rho \quad s \xrightarrow{\mathsf{a}!} s' \quad \text{all} \oplus \text{succeed}}{\langle \rho, \beta \rangle \xrightarrow{\mathsf{a}!} \langle \rho \uplus \{n{:}s'\}, \beta \uplus \{(\bot, \bot){:}(\beta(\bot, \bot) \oplus \underbrace{\mathsf{a} \oplus \cdots \oplus \mathsf{a}}_{t \text{ times}})\}\rangle}$$

$$\textsc{send@snd-rcv}\ \dfrac{\begin{array}{cccc}\mathsf{a}{:}(\mathsf{async}, \_\_, \mathit{to}, \_\_, \texttt{@snd-rcv}) \in \sigma & n{:}s \in \rho & s \xrightarrow{\mathsf{a}\,!\,\overline{m}} s' & \#\overline{m} \in \mathit{to} \\ \text{all} \oplus \text{succeed} & & \overline{m} \subseteq \gamma & \end{array}}{\langle \rho, \beta \rangle \xrightarrow{\mathsf{a}\,!\,\overline{m}} \langle \rho \uplus \{n{:}s'\}, \beta \uplus \{(n, m){:}(\beta(n, m) \oplus \mathsf{a}) \mid m \in \overline{m}\}\rangle}$$

Fig. 5. Operational rules for asynchronous communication (sending)

$$\textsc{recv@snd}\ \dfrac{\begin{array}{cccc}\mathsf{a}{:}(\mathsf{async}, \mathit{fr}, \_\_, \_\_, \texttt{@snd}) \in \sigma & n{:}s \in \rho & s \xrightarrow{\mathsf{a}\,?\,\overline{m}} s' & \#\overline{m} \in \mathit{fr} \\ \text{all} \ominus \text{succeed} & & \overline{m} \subseteq \gamma & \end{array}}{\langle \rho, \beta \rangle \xrightarrow{\mathsf{a}\,?\,\overline{m}} \langle \rho \uplus \{n{:}s'\}, \beta \uplus \{(m, \bot){:}(\beta(m, \bot) \ominus \mathsf{a}) \mid m \in \overline{m}\}\rangle}$$

$$\textsc{recv@rcv}\ \dfrac{\mathsf{a}{:}(\mathsf{async}, (f,f), \_\_, \_\_, \texttt{@rcv}) \in \sigma \quad n{:}s \in \rho \quad s \xrightarrow{\mathsf{a}?} s' \quad \text{all} \ominus \text{succeed}}{\langle \rho, \beta \rangle \xrightarrow{\mathsf{a}?} \langle \rho \uplus \{n{:}s'\}, \beta \uplus \{(\bot, n){:}(\beta(\bot, n) \ominus \underbrace{\mathsf{a} \ominus \cdots \ominus \mathsf{a}}_{f \text{ times}}\}\rangle}$$

$$\textsc{recv@glob}\ \dfrac{\mathsf{a}{:}(\mathsf{async}, (f,f), \_\_, \_\_, \texttt{@global}) \in \sigma \quad n{:}s \in \rho \quad s \xrightarrow{\mathsf{a}?} s' \quad \text{all} \ominus \text{succeed}}{\langle \rho, \beta \rangle \xrightarrow{\mathsf{a}?} \langle \rho \uplus \{n{:}s'\}, \beta\{(\bot, \bot){:}(\beta(\bot, \bot) \ominus \underbrace{\mathsf{a} \ominus \cdots \ominus \mathsf{a}}_{f \text{ times}})\}\rangle}$$

$$\textsc{recv@snd-rcv}\ \dfrac{\begin{array}{cccc}\mathsf{a}{:}(\mathsf{async}, \mathit{fr}, \_\_, \_\_, \texttt{@snd-rcv}) \in \sigma & n{:}s \in \rho & s \xrightarrow{\mathsf{a}\,?\,\overline{m}} s' & \#\overline{m} \in \mathit{fr} \\ \text{all} \ominus \text{succeed} & & \overline{m} \subseteq \gamma & \end{array}}{\langle \rho, \beta \rangle \xrightarrow{\mathsf{a}\,?\,\overline{m}} \langle \rho \uplus \{n{:}s'\}, \beta \uplus \{(m, n){:}(\beta(m, n) \ominus \mathsf{a}) \mid m \in \overline{m}\}\rangle}$$

Fig. 6. Operational rules for asynchronous communication (receiving)

Synchronous rules Rules CONST, PREFIX, SUM-L, and SUM-R in Fig. 4 are standard in CCS [54]. The empty process **0** is a special process that does not evolve. Rule SYNC models the synchronisation over an action named a sent by the senders in $\mathsf{dom}(\mathit{snds})$ and to the receivers in $\mathsf{dom}(\mathit{rcvs})$. The synchronisation type of a, given by $\sigma(\mathsf{a}) = (\mathsf{sync}, \mathit{from}, \mathit{to})$, captures that a is synchronous and the number of senders (receivers, resp.) must be in the interval *from* (*to*, resp.). Synchronously sending to or receiving from multiple agents does not guarantee the maximal number of participants in a single action;[4] instead, any number permitted by the synchronisation type and the available participants is allowed. The sets of senders *snds* (receivers *rcvs*, resp.) are obtained by updating the state of the corresponding process to the target state of the sending (receiving, resp.) transitions. Additionally, the send or receive transition may identify a set of recipients $\overline{m}$. In this case, it is required that these recipients be involved in performing the synchronisation (e.g., $\overline{m} \subseteq \mathsf{dom}(snds)$).

Asynchronous rules The asynchronous rules for sending are presented in Fig. 5, and those for receiving are presented in Fig. 6. These are dual to each other and are guided by a few guiding principles, explained below, describing desirable properties of successful executions.

Principle 1: *Asynchronous sending messages should not block.*

This is mainly a consequence from requiring buffers to only block when being consumed, and to always succeed when adding new elements. In rule SEND@RCV, a sender n sends a to a set of receivers $\overline{m}$ with cardinality within the interval *to* defined by the synchronisation type $\sigma(a)$. Since the location type is $\mathsf{@rcv}$, for each receiver $m \in \overline{m}$, the message is added to its buffer (i.e., $\beta(\bot, m) \oplus \mathsf{a}$). Rule SEND@SND-RCV is similar, but the message is added to the buffer $\beta(n, m)$, identifying both sender and receiver. Rule RCV@SND is dual to SEND@RCV. In this case, instead of the receivers, the interval of senders *fr* is considered, and for each sender m the buffer $\beta(m, \bot)$ is updated by removing a message. Similarly, rule RCV@SND-RCV is the dual of SEND@SND-RCV. In this case, messages are removed from the buffer $\beta(n, m)$. This behaviour follows our next principle.

Principle 2: *The sender or receiver of an asynchronous action must not be explicitly mentioned when this is disregarded by the buffer location.*

Rule SEND@SND deposits the message in the buffer $\beta(n, \bot)$ of the sender, and does not need to know the receiver. However, our semantics requires the number of receivers to be known, leading to the next principle.

Principle 3: *The number of asynchronous messages sent or received must be deterministic.*

In other words, the number of senders (receivers, resp.) must be known by explicitly specifying the senders (receivers, resp.), or by using a singleton interval in the corresponding synchronisation type. Hence, no set of receivers is identified in rule SEND@SND and it is required that the interval of receivers in synchronisation type $\sigma(a)$ is a singleton (t, t). The buffer $\beta(n, \bot)$ is updated by adding t copies of the message a. Rule SEND@GLOB is similar, but global buffer $\beta(\bot, \bot)$, which does not identify any sender or receiver, is updated. The receiving-side

[4] This differs from other approaches in the literature such as, e.g., broadcast communication in UPPAAL [25].

$$\begin{array}{lll}
P = \alpha.P' & \Rightarrow & P' \text{ is well-formed} \qquad (1) \\
P = \mathsf{a}\,!\,\overline{n}.P' & \Rightarrow & \mathsf{a}{:}(\_, \_, \mathit{to}, \ldots) \in \sigma \wedge \#\overline{n} \in \mathit{to} \wedge \overline{n} \subseteq \mathsf{dom}(\rho) \qquad (2) \\
P = \mathsf{a}\,?\,\overline{n}.P' & \Rightarrow & \mathsf{a}{:}(\_, \mathit{from}, \ldots) \in \sigma \wedge \#\overline{n} \in \mathit{from} \wedge \overline{n} \subseteq \mathsf{dom}(\rho) \qquad (3) \\
P = \mathsf{a}\,!\,\overline{n}.P' & \Rightarrow & (\mathsf{a}{:}(\mathsf{sync}, \ldots) \in \sigma \vee \mathsf{a}{:}(\mathsf{async}, \_, \_, \_, \mathsf{@rcv}) \in \sigma \\
 & & \quad \vee\, \mathsf{a}{:}(\mathsf{async}, \_, \_, \_, \mathsf{@snd\text{-}rcv}) \in \sigma) \wedge \overline{n} \subseteq \mathsf{dom}(\rho) \qquad (4) \\
P = \mathsf{a}\,?\,\overline{n}.P' & \Rightarrow & (\mathsf{a}{:}(\mathsf{sync}, \ldots) \in \sigma \vee \mathsf{a}{:}(\mathsf{async}, \_, \_, \_, \mathsf{@snd}) \in \sigma \\
 & & \quad \vee\, \mathsf{a}{:}(\mathsf{async}, \_, \_, \_, \mathsf{@snd\text{-}rcv}) \in \sigma) \wedge \overline{n} \subseteq \mathsf{dom}(\rho) \qquad (5) \\
P = \mathsf{a}!.P' & \Rightarrow & \mathsf{a}{:}(\mathsf{sync}, \ldots) \in \sigma \vee \mathsf{a}{:}(\mathsf{async}, \_, (t, t), \_, \mathsf{@snd}) \in \sigma \\
 & & \quad \vee\, \mathsf{a}{:}(\mathsf{async}, \_, (t, t), \_, \mathsf{@glob}) \in \sigma \qquad (6) \\
P = \mathsf{a}?.P' & \Rightarrow & \mathsf{a}{:}(\mathsf{sync}, \ldots) \in \sigma \vee \mathsf{a}{:}(\mathsf{async}, (f, f)\_, \_, \mathsf{@rcv}) \in \sigma \\
 & & \quad \vee\, \mathsf{a}{:}(\mathsf{async}, (f, f)\_, \_, \mathsf{@glob}) \in \sigma \qquad (7) \\
P = P_1 + P_2 & \Rightarrow & P_1 \text{ and } P_2 \text{ are well-formed} \qquad (8) \\
P = K & \Rightarrow & K{:}\_ \in \gamma \qquad (9)
\end{array}$$

Fig. 7. Conditions for well-formedness

dual of SEND@SND is RECV@RCV. Similarly, RCV@GLOB is dual to SEND@GLOB. In both cases, the interval of senders is a singleton (f, f), and f copies of the message are removed from the buffer of the receiver n in the first case (i.e., $\beta(\perp, n)$), and from the global buffer $\beta(\perp, \perp)$ in the second case. Note that requiring a singleton interval is only necessary to achieve a deterministic number of asynchronous messages. This constraint can be lifted at the cost of introducing non-determinism in the number of possible messages exchanged in the above cases, thereby increasing the state space.

2.4 Well-formedness

The operational semantics presented in Figs. 4 to 6 perform some well-formedness checks at runtime. These include type checking the buffers (i.e., operations $\oplus$ and $\ominus$ must succeed, cf. Section 2.3) and enforcing Principles 2 and 3, i.e., they constrain the synchronisation types of actions to singleton intervals *from* or *to*. Indeed, problems that arise from violating the expected synchronisation types, or by using unknown names of actions or agents, can lead to unexpected scenarios in which no rule can be applied (triggering runtime errors in the A-Team tool, cf. Section 3). To avoid these problems, we formalise below a set of well-formedness conditions that can statically detect them, all of them implemented in A-Team.

Given a system $\langle\gamma; \sigma; \rho\rangle$, a process P occurring in γ or in ρ is said to be well-formed if the 9 conditions in Fig. 7 and the 10 conditions in Fig. 8 hold. We first discuss Fig. 7. Condition (1), (8), and (9) are straightforward. Condition (2) checks that an agent never tries to send a message to more senders than specified in its synchronisation type; Condition (3) is the dual for the receiving side. Conditions (4) to (7) avoid violating Principle 2, while Conditions (6) and (7) also avoid violating Principle 3 (cf. Section 2.3).

Fig. 8 displays the conditions for checking the well-formedness of buffer types. The A-Team tool statically checks these conditions. They ensure that the buffer types declared by the actions are compliant with their usage within processes: in particular, a specific buffer at a given location should have a unique type. This in turn ensures that, in the semantic rules in Section 2.3, operation $\oplus$ is always enabled (guaranteeing Principle 1), whilst operation $\ominus$ is never disabled by mismatching buffer types. Here, $act(P)$ denotes the actions occurring in process P, structurally defined as $act(\alpha.P) = \{\alpha\} \cup act(P)$, $act(P_1 + P_2) = act(P_1) \cup act(P_2)$, $act(\mathbf{0}) = \emptyset$, and $act(P) = act(K)$ when $P = K$.

Condition (1) ensures that all asynchronous actions written into (or read from) the global buffer must declare the same (global) buffer type. Conditions (2) to (4) deal with (asynchronous) actions with the same location type snd but different buffer types. Condition (2) guarantees that every process n does not place in its buffer $\beta(n, \bot)$ actions declaring different buffer type, whilst Condition (3) checks that whenever a process n writes into its buffer $\beta(n, \bot)$ an action with a specific buffer type, no other process may read from $\beta(n, \bot)$ an action with a different buffer type. Condition (4) ensures that all actions read from a sender buffer $\beta(s, \bot)$ by processes declare the same buffer type.

Conditions (5) to (7) are used for (asynchronous) actions with the same location type rcv but different buffer types. Condition (5) guarantees that all actions inserted into a receiver buffer $\beta(\bot, r)$ by processes declare the same buffer type. Similarly, Condition (6) requires that all actions read from buffer $\beta(\bot, r)$ by process r must have a declared buffer type consistent with that of the actions previously written into $\beta(\bot, r)$, whilst Condition (7) guarantees that every process n does not read from its buffer $\beta(\bot, n)$ actions declaring different buffer types.

Finally, Conditions (8) to (10) are used for (asynchronous) actions with the same location type snd-rcv but different buffer types. Condition (8) stipulates that a process n cannot place in a buffer $\beta(n, r)$ shared with a receiver r actions declaring different buffer types. Likewise, Condition (9) demands that all actions read from buffer $\beta(n, r)$ by process r must have a declared buffer type consistent with that of the actions written into $\beta(n, r)$ by process n, whilst Condition (10) guarantees that a process n cannot read from a buffer $\beta(r, n)$ shared with a sender r actions declaring different buffer types.

Note that some process may violate these conditions even if at runtime it will never raise buffer type errors. For instance, the process a! + b! such that both a and b have location type @glob but different buffer types, violates Condition (1). Indeed, the global buffer has an inconsistent type declaration by actions a and b. However, at runtime this error will never occur.

2.5 Well-behavedness

This section presents conditions that guarantee well-behavedness, to complement the previous well-formedness conditions. While well-formedness conditions can be checked syntactically, well-behavedness conditions require traversing all reachable states. These conditions are organised below into five subsections and are inspired on existing asynchronous models like CFSMs [29], i.e., with only binary peer-to-peer asynchronous communication and with rich local actions.

$$\forall \mathsf{a}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{ba}, \mathsf{@glob}), \mathsf{b}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{bb}, \mathsf{@glob}) \in \sigma \Rightarrow \mathsf{ba} = \mathsf{bb} \tag{1}$$

$$\forall n{:}P \in \gamma, \forall \mathsf{a}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{ba}, \mathsf{@snd}), \mathsf{b}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{bb}, \mathsf{@snd}) \in \sigma, \mathsf{ba} \neq \mathsf{bb}.\\ \mathsf{a}! \in act(P) \Rightarrow \mathsf{b}! \notin act(P) \tag{2}$$

$$\forall n{:}P, r{:}Q \in \gamma, \forall \mathsf{a}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{ba}, \mathsf{@snd}), \mathsf{b}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{bb}, \mathsf{@snd}) \in \sigma, \mathsf{ba} \neq \mathsf{bb}.\\ \mathsf{a}! \in act(P) \wedge \mathsf{b}?\overline{m} \in act(Q) \Rightarrow n \notin \overline{m} \tag{3}$$

$$\forall n{:}P, r{:}Q \in \gamma, \forall \mathsf{a}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{ba}, \mathsf{@snd}), \mathsf{b}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{bb}, \mathsf{@snd}) \in \sigma, \mathsf{ba} \neq \mathsf{bb}.\\ \mathsf{a}?\overline{m_1} \in act(P) \wedge \mathsf{b}?\overline{m_2} \in act(Q) \Rightarrow \overline{m_1} \cap \overline{m_2} = \emptyset \tag{4}$$

$$\forall n{:}P, r{:}Q \in \gamma, \forall \mathsf{a}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{ba}, \mathsf{@rcv}), \mathsf{b}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{bb}, \mathsf{@rcv}) \in \sigma, \mathsf{ba} \neq \mathsf{bb}.\\ \mathsf{a}!\overline{m_1} \in act(P) \wedge \mathsf{b}!\overline{m_2} \in act(Q) \Rightarrow \overline{m_1} \cap \overline{m_2} = \emptyset \tag{5}$$

$$\forall n{:}P, r{:}Q \in \gamma, \forall \mathsf{a}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{ba}, \mathsf{@rcv}), \mathsf{b}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{bb}, \mathsf{@rcv}) \in \sigma, \mathsf{ba} \neq \mathsf{bb}.\\ \mathsf{a}!\overline{m_1} \in act(P) \wedge \mathsf{b}? \in act(Q) \Rightarrow r \notin \overline{m} \tag{6}$$

$$\forall n{:}P \in \gamma, \forall \mathsf{a}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{ba}, \mathsf{@rcv}), \mathsf{b}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{bb}, \mathsf{@rcv}) \in \sigma, \mathsf{ba} \neq \mathsf{bb}.\\ \mathsf{a}? \in act(P) \Rightarrow \mathsf{b}? \notin act(P) \tag{7}$$

$$\forall n{:}P \in \gamma, \forall \mathsf{a}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{ba}, \mathsf{@snd\text{-}rcv}), \mathsf{b}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{bb}, \mathsf{@snd\text{-}rcv}) \in \sigma, \mathsf{ba} \neq \mathsf{bb}.\\ \mathsf{a}!\overline{m_1}, \mathsf{b}!\overline{m_2} \in act(P) \Rightarrow \overline{m_1} \cap \overline{m_2} = \emptyset \tag{8}$$

$$\forall n{:}P, r{:}Q \in \gamma, \forall \mathsf{a}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{ba}, \mathsf{@snd\text{-}rcv}), \mathsf{b}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{bb}, \mathsf{@snd\text{-}rcv}) \in \sigma, \mathsf{ba} \neq \mathsf{bb}.\\ \mathsf{a}!\overline{m_1} \in act(P) \wedge \mathsf{b}?\overline{m_2} \in act(Q) \Rightarrow n \notin \overline{m_2} \tag{9}$$

$$\forall n{:}P \in \gamma, \forall \mathsf{a}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{ba}, \mathsf{@snd\text{-}rcv}), \mathsf{b}{:}(\mathsf{async}, \_\_, \_\_, \mathsf{bb}, \mathsf{@snd\text{-}rcv}) \in \sigma, \mathsf{ba} \neq \mathsf{bb}.\\ \mathsf{a}?\overline{m_1}, \mathsf{b}?\overline{m_2} \in act(P) \Rightarrow \overline{m_1} \cap \overline{m_2} = \emptyset \tag{10}$$

Fig. 8. Conditions for well-formedness of buffer types

We include several literature references to motivate and justify these conditions, anticipating the related work in Section 4.

Compliance and compatibility of components have been recognised as fundamental properties for component-based and distributed systems for guaranteeing safe (successful) communication [2,31,28,33,14,36,32,9,16,15,10,21]. We extensively investigated requirements for safe communication in the context of (synchronous) team automata [15,21,20]. We defined a generic procedure to derive requirements for *receptiveness* and *responsiveness* that guarantee absence of message loss (output actions of one component not accepted as input by some other component) and indefinite waiting (for input to be received in the form of an appropriate output action provided by another component), respectively. A team automaton is defined to be *compliant* with a given set of communication requirements (in the form of receptiveness and responsiveness) if in each of its reachable states, the desired communications can occur immediately.[5] We

[5] A team automaton is defined to be *weakly compliant* if the communication can eventually occur, possibly after some other actions have been performed (cf., e.g., [23]).

implemented both notions in the A-Team tool, which is described in Section 3. However, in asynchronous communications the messages spend time in buffers, hence we focus on those in our quest for well-behavedness conditions for ATeams.

Receptiveness and Responsiveness According to guiding Principle 1, the sending action in an asynchronous communication in ATeams is never blocked, meaning that receptiveness properties (ensuring that a message is eventually sent) are always satisfied. Receiving actions, however, block until they are available in the buffers, giving room to responsiveness problems (when none of the receiving options is available at a given point). For CFSMs, in [29], it was studied how to avoid *unspecified reception* configurations, in which there is a receiving state unable to receive a message from its buffers because in each channel from which it could consume there is a message which it cannot receive in that state.

The A-Team tool detects both (synchronous) receptiveness and (synchronous and asynchronous) responsiveness problems. For example, in our synchronous Race example, the tool produces the warning that *"1 can receive start, but the system cannot (yet) send it: {0: finish?.Ctr, 1: R, 2: finish!.R}."*, where *1* is the name given to the first runner. The asynchronous versions raise similar warnings, e.g., the version with global buffers gives six responsiveness warnings, including that *"r1 can receive start, but the system cannot (yet) send it"*. No receptiveness warnings are detected in these examples. The weaker versions of these communication properties[5] are not implemented, i.e., to detect that a sending or receiving action can eventually be performed after the rest of the system evolves, which would require a more complex analysis (cf. [18]). In our experience, most asynchronous examples are not (strongly) responsive, i.e., it is likely that a process has to wait until a given receiving action can be performed, suggesting that ensuring this property may not be useful in most scenarios. Hence we expect the weaker form to be more useful.

Absence of orphan messages For multi-party session types, it was studied in [35] how to avoid *orphan message configurations* (with each component in a final state, but still at least one non-empty buffer). For I/O transition systems, in [42] a notion of *asynchronous compatibility* was defined, which guarantees that in any reachable configuration, if there is a channel (buffer) with a message on its top position, then some component can consume (receive) it. However, as for CFSMs, in both cases only binary peer-to-peer communication is considered.

The A-Team tool detects scenarios with orphan messages when there are non-empty buffers with messages sent to agents that successfully terminated. We added an example of a coffee machine that starts with an extra coin and can terminate after receiving coffee. In this example, a coffee can be left in the buffers, leading to the warning *"Agent u terminated and has incoming messages"*.

Absence of infinite buffers The semantics of asynchronous communication in ATeams uses unbounded buffers, assuming that adding new messages never fails. Consequently, it is possible to model a scenario in which a buffer keeps on growing indefinitely. In previous work, we addressed this problem in the context of asynchronous choreographies, using a pomset structure for the semantics [37] and borrowing the notion of *dependent guardedness* to restrict recursion from Rensink and Wehrheim [58]. The actor-based Rebeca model [1] also includes analyses to detect if the buffers storing messages between agents are bounded.

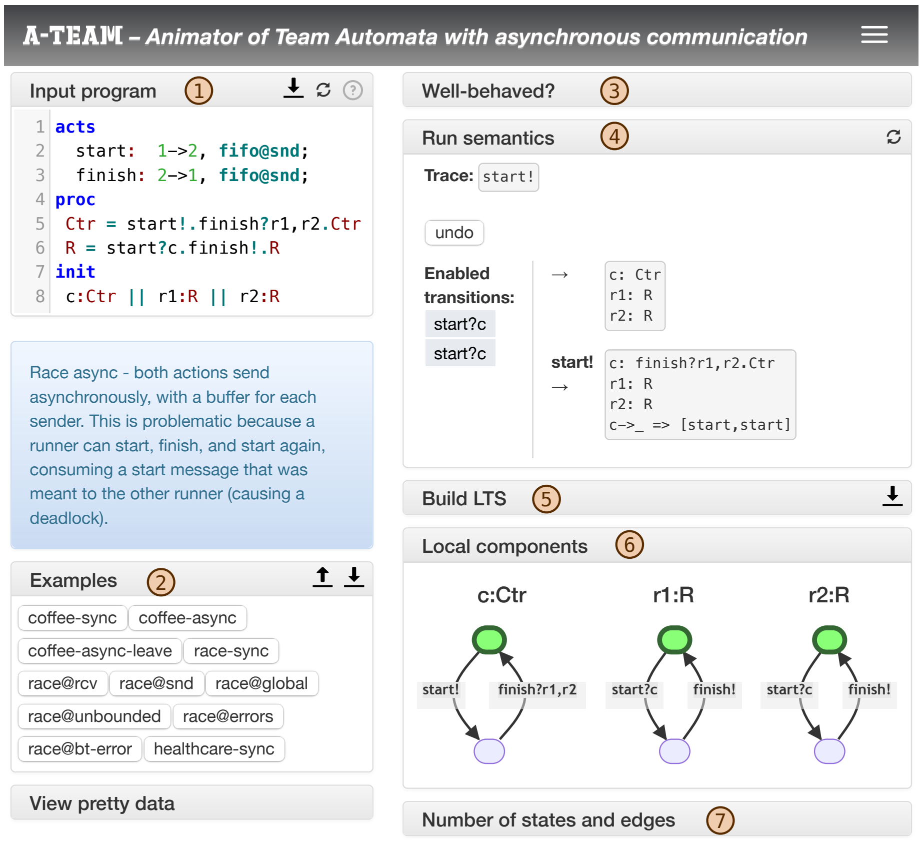
Inspired by these approaches, the A-Team tool can detect simple scenarios with infinite buffers, namely when a sender sends a message asynchronously but its state remains unchanged. This can happen, e.g., in a variation of our asynchronous race with an unbounded buffer that can grow indefinitely, whereas the controller can ask the runners to rest without waiting for an acknowledgement. This leads to the warning: *"can send rest!r1,r2 forever: c: Ctr, r1: R, r2: R"*. Our tool can also detect a similar loop in a larger case study by Pal et al. in the context of Italian Healthcare [56]. This healthcare example specifies the local behaviour of each agent, using 1-to-1 interactions, used by Pal et al. to push the limits of their approach to reason about realisability based on pomsets. The authors compare the performance using both a synchronous and an asynchronous semantics, without comparing behavioural differences, and unfolding the loops a small number of times. In our example, these interactions are specified as being synchronous, but changing them instead to being asynchronous triggers a warning that the sending of proc!*gDH* can be executed indefinitely.

Absence of deadlocks The relationship between *deadlock-freedom*, used in many different disguises in the literature, and communication properties for team automata is subtle, mainly because the distinction between input and output actions is not relevant for the more traditional notions of deadlock and because typically *global* and *local* deadlocks are distinguished [5]. In case of some specific assumptions on the synchronisation types (cf. [21]), weak versions of receptiveness and responsiveness can be used to guarantee (global) deadlock-freedom, in the sense of BIP [40], choreography automata [7], or *stuck-freeness* in multi-party session types [59]. However, global deadlock-freedom does not imply receptiveness, as we discussed in previous work [23]. For CFSMs, finally, it was recently studied in [6] how to guarantee *lock-freedom* (for any reachable configuration, there is no component in a receiving state that never receives a message in all possible transition sequences from that configuration).

The A-Team tool detects some global deadlocks by searching for states where no action can be performed, but some individual agent can still evolve; e.g., the version of the Race example that uses fifo@snd has a deadlock, resulting in the message *"Deadlock found: c: finish?r1,r2.Ctr, r1: R, r2: R, r2->_=>[finish,finish]"* (and a symmetric one in which *r1* rather than *r2* is started twice in a row).

3 Tool Support

We implemented a prototypical tool, called the A-Team, to specify ATeams, to animate its semantics, and to analyse well-formedness and well-behavedness



```
acts
  start:  1->2, fifo@snd;
  finish: 2->1, fifo@snd;
proc
 Ctr = start!.finish?r1,r2.Ctr
 R = start?c.finish!.R
init
 c:Ctr || r1:R || r2:R
```

Race async - both actions send asynchronously, with a buffer for each sender. This is problematic because a runner can start, finish, and start again, consuming a start message that was meant to the other runner (causing a deadlock).

Trace: `start!`

Enabled transitions:

start?c →

```
c: Ctr
r1: R
r2: R
```

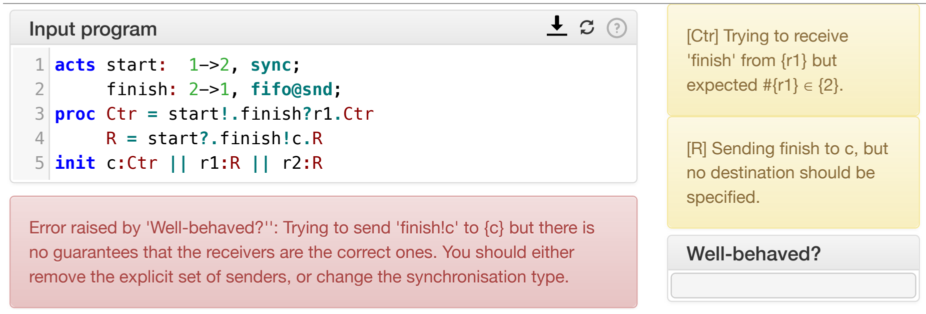
start?c start! →

```
c: finish?r1,r2.Ctr
r1: R
r2: R
c->_ => [start,start]
```

Fig. 9. Screenshot of the prototypical A-Team online tool

conditions. A-Team is open-source and can be used online by loading the static website https://fm-dcc.github.io/a-team. Figure 9 illustrates its main interface, organised into several widgets: ① is a text editor to specify a system, using a dedicated domain-specific language; ② loads examples of ATeams, including the ones presented in this paper; ③ lists all the well-behavedness errors found (up to a bounded number of transitions); ④ allows the guided step-by-step execution of the operational rules from Figs. 4 to 6; ⑤ unfolds and draws all the operational steps in a graph (until a fixed bound); ⑥ draws the local automata from the **init** block; and ⑦ counts the number of edges and states from widget ⑤ (useful for larger systems). A hidden widget verifies all well-formedness conditions (cf. Section 2.4) and throws warnings when these do not hold. Some of these widgets are collapsed (e.g., ③, ⑤, and ⑦) and the others are expanded (e.g., ① and ②); clicking the header alternates between these states. Only expanded widgets are evaluated, and expanding a collapsed widget triggers its (re-)evaluation.

A-Team is written in Scala and uses the CAOS libraries [57] to generate an interactive website without any server, whereas all functionality is compiled to



```
acts start:  1->2, sync;
     finish: 2->1, fifo@snd;
proc Ctr = start!.finish?r1.Ctr
     R = start?.finish!c.R
init c:Ctr || r1:R || r2:R
```

Error raised by 'Well-behaved?'': Trying to send 'finish!c' to {c} but there is no guarantees that the receivers are the correct ones. You should either remove the explicit set of senders, or change the synchronisation type.

[Ctr] Trying to receive 'finish' from {r1} but expected #{r1} ∊ {2}.

[R] Sending finish to c, but no destination should be specified.

Fig. 10. A-Team providing feedback regarding well-formedness and well-behavedness

JavaScript using `Scala.js`. A-Team is meant to provide quick insights on the expressivity of our semantics, experiment with different communication types, and have a small code-base easy to adapt and maintain. For example, the full semantics is encoded in file https://github.com/FM-DCC/a-team/blob/main/src/main/scala/ateams/backend/Semantics.scala. Here the function `nextState(st:St)` receives a current `st` with the action and process declarations, and returns a set of possible steps, each consisting of an action and its associated new state (including the updated buffers). The behavioural analyses are currently not meant to be particularly efficient or scalable, but to showcase how to traverse the state space to search for desirable properties, and to provide early insights on some of these.

Example with errors A screenshot from A-Team with an asynchronous variation of the synchronous Race example with well-formedness and well-behavedness problems is depicted in Fig. 10. In this variation, start is synchronous and finish uses a FIFO buffer; however, finish is used incorrectly in both process definitions, as described in the warning messages on the top-right: *Ctr* tries to receive from a single runner (instead of two) and *R* tries to send to an explicit agent, which should be omitted. The well-behavedness widget does not succeed because the tool raises an error (cf. bottom left in Fig. 10) while traversing the state space, due to a well-formedness violation (cf. top right in Fig. 10). Errors from any of the widgets are displayed below the input program, which can be seen at the bottom left in Fig. 10.

4 Related Work

In this section, we discuss and provide pointers to related work on synchronous automata-based or coordination models and on asynchronous communication models, and we briefly recall related work on properties that guarantee safe communication in component-based and distributed systems.

In the introduction, we related team automata with I/O automata [53]. Further closely related I/O automata-based models include I/O systems [49], interface automata [2], reactive transition systems [31], interacting state machines [55], and component-interaction (CI) automata [30]. All these models distinguish input, output, and internal actions. However, I/O automata are *input-enabled* (in each state of the automaton, every input action of the automaton is enabled). Most of these models define composition as the synchronous product of automata. However, I/O automata only allow the compositon of I/O automata whose output actions are disjoint. Moreover, interface automata restrict product states to compatible states. Finally, CI automata share the distinguishing feature of multi-party composition à la team automata, but communication is restricted to binary synchronisation between a pair of input and output actions.

In [23], team automata are compared in detail to the coordination models Reo/port automata [3,50], BIP [13,27], contract automata [12,11], choreography automata [7,8], and (synchronous) multi-party session types [59,60], including for each model the definition of composition, an overview of existing tool support, and a specification of the Race example in the specific model. Reo's semantic model of port automata abstracts from data constraints to focus on (action) synchronisation and (automata) composition. Synchronisation types in team automata restrict the number of inputs and outputs of ports (actions) with shared names, while synchronisation in port automata forces how this is done. BIP (without priorities) has a synchronisation mechanism that ignores input and output but, similar to synchronisation types in team automata, it has a formalisation parameterised on the number of components. Ports (actions) can have multiple instances and bounds on the number of allowed components they can synchronise with and on the number of interactions they can be involved in. Both contract and choreography automata support only binary synchronisations in which each interaction has a single sender (the component that offers) and a single receiver (the component that requests). Synchronous multi-party session types, finally, support a more limited set of communication patterns than team automata. They typically support only binary synchronisation, yet variants exist that support multiple receivers [26] or multiple senders [45].

In the introduction, we mentioned actor-based and object-based programming models and languages like Erlang [4], Rebeca [1,44], Creol/ABS [47,46], and Akka [41], as well as asynchronous communication models like asynchronous multi-party session types [43] and CFSMs [29], as part of our inspiration for addressing asynchronous communication in team automata in this paper. Erlang/Akka, Rebeca, and Creol/ABS are actor-based models with no shared memory that instead model systems as actors (components) communicating by asynchronous message-passing, making them particularly suitable for specifying concurrent and distributed systems. The Basic Actor-based Language (Babel) [39] is a core actor-based language with a semantics with variation points for communication and coordination that can be instantiated to obtain the concrete semantics of actor-based languages like Rebeca, ABS, and Erlang/Akka. In particular, the operations of sending and receiving messages are distinguished by variation points based on when a message can be added to the buffer (i.e., based on the buffer's type and capacity) and when it can be consumed from the buffer (e.g., FIFO, EDF (Earliest Deadline First), based on priority or pattern matching, or nondeterministically). In the future, we could consider adding some of these policies to our buffer types. CFSMs and multi-party session types, on the other hand, are meant for specifying and reasoning (through formal verification and type checking) about safe communication in message-passing concurrent and distributed systems rather than actor-based models. However, communication in CFSMs and most related asynchronous models is limited to peer-to-peer communication. This is part of the challenge of addressing asynchronous communication in team automata, as sketched in Fig. 1.

In Section 2.5, we discussed in detail related studies on guaranteeing safe communication in terms of properties like receptiveness, responsiveness, absence of orphan messages, infinite buffers, and deadlocks in particular in models such as CFSMs, BIP, choreography automata, and multi-party session types.

5 Conclusion and Future Work

Building on the extensive body of work on team automata [23], including work presented at FM [17,18], we introduced ATeams, extending team automata with asynchronous communication. Unlike established asynchronous models such as CFSMs, ATeams naturally support multi-party communication with multiple senders and multiple receivers. We formalised their syntax and operational semantics, identified conditions for well-formed and well-behaved specifications, and provided tool support for modelling, animation, and automated checks for well-formedness and well-behavedness. Together, these contributions position ATeams as a unifying semantic foundation for the modelling and analysis of heterogeneous synchronous–asynchronous multi-party systems.

Future work In particular the well-behavedness conditions for ATeams need to be fine-tuned, expanded and implemented in A-Team to offer further analyses guaranteeing a rich set of communication properties, such as deadlock detection (considering the decidability issues mentioned below). These analyses could be further improved by developing back-ends that rely on efficient tools, such as mCRL2 for team automata [18]. The proposed heterogeneous communication channels with multi-party interactions could also be exploited when describing systems at a *global* rather than local level. These specifications could be analysed to reason about realisability, similar to our previous work on realising team automata from global specifications [22], or about multi-party session types for multilingual programming [48].

We conjecture that CFSMs can be encoded into ATeams, by only allowing actions whose synchronisation type has buffer type FIFO and location type SND-RCV. It would then follow that, in general, deadlock detection is undecidable. As future work, we plan to investigate whether decidability may be recovered under suitable restrictions, such as when FIFO ordering is ruled out (i.e., only unordered buffers are permitted), or when buffers are finite (although checking for buffer finiteness may itself be undecidable).

Acknowledgments. Partially funded by MUR PRIN 2020TL3X8X project T-LADIES (Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems), by CNR project "Formal Methods in Software Engineering 2.0", CUP B53C24000720005, by the CISTER Research Unit (UIDP/UIDB/04234/2020), financed by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), and by National Funds through the FCT within the project POISE, with reference 2024.15783. PEX (https://doi.org/10.54499/2024.15783.PEX).

The authors would like to thank Rolf Hennicker for inspiring discussions on adding asynchronous communication to team automata, and the anonymous reviewers for their useful comments and suggestions.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. In: Mousavi, M.R., Ravara, A. (eds.) Proceedings of the 10th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2011). EPTCS, vol. 58, pp. 1–19 (2011). https://doi.org/10.4204/EPTCS.58.1
2. de Alfaro, L., Henzinger, T.A.: Interface Automata. In: Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2001). pp. 109–120. ACM (2001). https://doi.org/10.1145/503209.503226
3. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004). https://doi.org/10.1017/S0960129504004153
4. Armstrong, J., Virding, R., Williams, M.: Concurrent Programming in Erlang. Prentice Hall, 2 edn. (1996)
5. Attie, P.C., Bensalem, S., Bozga, M., Jaber, M., Sifakis, J., Zaraket, F.A.: Global and Local Deadlock Freedom in BIP. ACM Trans. Softw. Eng. Methodol. **26**(3), 9:1–9:48 (2018). https://doi.org/10.1145/3152910
6. Barbanera, F., Hennicker, R.: Safe Composition of Systems of Communicating Finite State Machines. In: Aubert, C., Di Giusto, C., Fowler, S., Ka I Pun, V. (eds.) Proceedings of the 17th Interaction and Concurrency Experience (ICE 2024). EPTCS, vol. 414, pp. 39–57 (2024). https://doi.org/10.4204/EPTCS.414.3
7. Barbanera, F., Lanese, I., Tuosto, E.: Choreography Automata. In: Bliudze, S., Bocchi, L. (eds.) Proceedings of the 22nd IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION 2020). LNCS, vol. 12134, pp. 86–106. Springer (2020). https://doi.org/10.1007/978-3-030-50029-0_6
8. Barbanera, F., Lanese, I., Tuosto, E.: A Theory of Formal Choreographic Languages. Log. Meth. Comp. Sci. **19**(3), 9:1–9:36 (2023). https://doi.org/10.46298/LMCS-19(3:9)2023
9. Bartoletti, M., Cimoli, T., Zunino, R.: Compliance in Behavioural Contracts: A Brief Survey. In: Bodei, C., Ferrari, G.L., Priami, C. (eds.) Programming Languages with Applications to Biology and Security. LNCS, vol. 9465, pp. 103–121. Springer (2015). https://doi.org/10.1007/978-3-319-25527-9_9

10. Basile, D., ter Beek, M.H., Degano, P., Legay, A., Ferrari, G.L., Gnesi, S., Di Giandomenico, F.: Controller synthesis of service contracts with variability. Sci. Comput. Program. **187** (2020). https://doi.org/10.1016/j.scico.2019.102344
11. Basile, D., Degano, P., Ferrari, G.L.: Automata for Specifying and Orchestrating Service Contracts. Log. Meth. Comp. Sci. **12**(4:6), 1–51 (2016). https://doi.org/10.2168/LMCS-12(4:6)2016
12. Basile, D., Degano, P., Ferrari, G., Tuosto, E.: From Orchestration to Choreography through Contract Automata. In: Lanese, I., Lluch Lafuente, A., Sokolova, A., Torres Vieira, H. (eds.) Proceedings of the 7th Interaction and Concurrency Experience (ICE 2014). EPTCS, vol. 166, pp. 67–85 (2014). https://doi.org/10.4204/EPTCS.166.8
13. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Components in BIP. In: Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006). pp. 3–12. IEEE (2006). https://doi.org/10.1109/SEFM.2006.27
14. Bauer, S.S., Mayer, P., Schroeder, A., Hennicker, R.: On Weak Modal Compatibility, Refinement, and the MIO Workbench. In: Esparza, J., Majumdar, R. (eds.) Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010). LNCS, vol. 6015, pp. 175–189. Springer (2010). https://doi.org/10.1007/978-3-642-12002-2_15
15. ter Beek, M.H., Carmona, J., Hennicker, R., Kleijn, J.: Communication Requirements for Team Automata. In: Jacquet, J.M., Massink, M. (eds.) Proceedings of the 19th IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION 2017). LNCS, vol. 10319, pp. 256–277. Springer (2017). https://doi.org/10.1007/978-3-319-59746-1_14
16. ter Beek, M.H., Carmona, J., Kleijn, J.: Conditions for Compatibility of Components: The Case of Masters and Slaves. In: Margaria, T., Steffen, B. (eds.) Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA 2016). LNCS, vol. 9952, pp. 784–805. Springer (2016). https://doi.org/10.1007/978-3-319-47166-2_55
17. ter Beek, M.H., Cledou, G., Hennicker, R., Proença, J.: Featured Team Automata. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) Proceedings of the 24th International Symposium on Formal Methods (FM 2021). LNCS, vol. 13047, pp. 483–502. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_26
18. ter Beek, M.H., Cledou, G., Hennicker, R., Proença, J.: Can we Communicate? Using Dynamic Logic to Verify Team Automata. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) Proceedings of the 25th International Symposium on Formal Methods (FM 2023). LNCS, vol. 14000, pp. 122–141. Springer (2023). https://doi.org/10.1007/978-3-031-27481-7_9
19. ter Beek, M.H., Ellis, C.A., Kleijn, J., Rozenberg, G.: Synchronizations in Team Automata for Groupware Systems. Comput. Sup. Coop. Work **12**(1), 21–69 (2003). https://doi.org/10.1023/A:1022407907596
20. ter Beek, M.H., Hennicker, R., Kleijn, J.: Compositionality of Safe Communication in Systems of Team Automata. In: Ka I Pun, V., Stolz, V., Simão, A. (eds.) Proceedings of the 17th International Colloquium on Theoretical Aspects of Computing (ICTAC 2020). LNCS, vol. 12545, pp. 200–220. Springer (2020). https://doi.org/10.1007/978-3-030-64276-1_11
21. ter Beek, M.H., Hennicker, R., Kleijn, J.: Team Automata@Work: On Safe Communication. In: Bliudze, S., Bocchi, L. (eds.) Proceedings of the 22nd IFIP WG

6.1 International Conference on Coordination Models and Languages (COORDINATION 2020). LNCS, vol. 12134, pp. 77–85. Springer (2020). https://doi.org/10.1007/978-3-030-50029-0_5
22. ter Beek, M.H., Hennicker, R., Proença, J.: Realisability of Global Models of Interaction. In: Ábrahám, E., Dubslaff, C., Tapia Tarifa, S.L. (eds.) Proceedings of the 20th International Colloquium on Theoretical Aspects of Computing (ICTAC 2023). LNCS, vol. 14446, pp. 236–255. Springer (2023). https://doi.org/10.1007/978-3-031-47963-2_15
23. ter Beek, M.H., Hennicker, R., Proença, J.: Team Automata: Overview and Roadmap. In: Castellani, I., Tiezzi, F. (eds.) Proceedings of the 26th IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION 2024). LNCS, vol. 14676, pp. 161–198. Springer (2024). https://doi.org/10.1007/978-3-031-62697-5_10
24. ter Beek, M.H., Kleijn, J.: Modularity for Teams of I/O Automata. Inf. Process. Lett. **95**(5), 487–495 (2005). https://doi.org/10.1016/j.ipl.2005.05.012
25. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) Revised Lectures of the 4th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Formal Methods for the Design of Real-Time Systems (SFM-RT 2004). LNCS, vol. 3185, pp. 200–236. Springer (2004). https://doi.org/10.1007/978-3-540-30080-9_7
26. Bejleri, A., Yoshida, N.: Synchronous Multiparty Session Types. Electr. Notes Theor. Comput. Sci. **241**, 3–33 (2008). https://doi.org/10.1016/j.entcs.2009.06.002
27. Bliudze, S., Sifakis, J.: The Algebra of Connectors: Structuring Interaction in BIP. IEEE Trans. Comput. **57**(10), 1315–1330 (2008). https://doi.org/10.1109/TC.2008.26
28. Bordeaux, L., Salaün, G., Berardi, D., Mecella, M.: When are Two Web Services Compatible? In: Shan, M.C., Dayal, U., Hsu, M. (eds.) Proceedings of the 5th International Workshop on Technologies for E-Services (TES 2004). LNCS, vol. 3324, pp. 15–28. Springer (2005). https://doi.org/10.1007/978-3-540-31811-8_2
29. Brand, D., Zafiropulo, P.: On Communicating Finite-State Machines. J. ACM **30**(2), 323–342 (1983). https://doi.org/10.1145/322374.322380
30. Brim, L., Černá, I., Vařeková, P., Zimmerová, B.: Component-Interaction Automata as a Verification-Oriented Component-Based System Specification. ACM Softw. Eng. Notes **31**(2) (2006). https://doi.org/10.1145/1118537.1123063
31. Carmona, J., Cortadella, J.: Input/Output Compatibility of Reactive Systems. In: Aagaard, M., O'Leary, J.W. (eds.) Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2002). LNCS, vol. 2517, pp. 360–377. Springer (2002). https://doi.org/10.1007/3-540-36126-X_22
32. Carmona, J., Kleijn, J.: Compatibility in a multi-component environment. Theor. Comput. Sci. **484**, 1–15 (2013). https://doi.org/10.1016/j.tcs.2013.03.006
33. Castagna, G., Gesbert, N., Padovani, L.: A Theory of Contracts for Web Services. ACM Trans. Program. Lang. Syst. **31**(5), 19:1–19:61 (2009). https://doi.org/10.1145/1538917.1538920
34. Clemente, L., Herbreteau, F., Sutre, G.: Decidable Topologies for Communicating Automata with FIFO and Bag Channels. In: Baldan, P., Gorla, D. (eds.) Proceedings of the 25th International Conference on Concurrency Theory (CONCUR 2014). LNCS, vol. 8704, pp. 281–296. Springer (2014). https://doi.org/10.1007/978-3-662-44584-6_20

35. Deniélou, P., Yoshida, N.: Multiparty Session Types Meet Communicating Automata. In: Seidl, H. (ed.) Proceedings of the 21st European Symposium on Programming (ESOP 2012). LNCS, vol. 7211, pp. 194–213. Springer (2012). https://doi.org/10.1007/978-3-642-28869-2_10
36. Durán, F., Ouederni, M., Salaün, G.: A generic framework for n-protocol compatibility checking. Sci. Comput. Program. **77**(7-8), 870–886 (2012). https://doi.org/10.1016/j.scico.2011.03.009
37. Edixhoven, L., Jongmans, S.S., Proença, J., Castellani, I.: Branching pomsets: design, expressiveness and applications to choreographies. J. Log. Algebr. Methods Program. **136** (2024). https://doi.org/10.1016/j.jlamp.2023.100919
38. Ellis, C.S.: Team Automata for Groupware Systems. In: Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge (GROUP 1997). pp. 415–424. ACM (1997). https://doi.org/10.1145/266838.267363
39. Ghassemi, F., Sirjani, M., Khamespanah, E., Mirani, M., Hojjat, H.: Transparent Actor Model. In: Proceedings of the 11th International Conference on Formal Methods in Software Engineering (FormaliSE 2023). pp. 97–107. IEEE (2023). https://doi.org/10.1109/FORMALISE58978.2023.00018
40. Gössler, G., Sifakis, J.: Composition for component-based modeling. Sci. Comput. Program. **55**, 161–183 (2005). https://doi.org/10.1016/j.scico.2004.05.014
41. Haller, P.: On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective. In: Agha, G.A., Bordini, R.H., Marron, A., Ricci, A. (eds.) Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions (AGERE! 2012). pp. 1–6. ACM (2012). https://doi.org/10.1145/2414639.2414641
42. Hennicker, R., Bidoit, M.: Compatibility Properties of Synchronously and Asynchronously Communicating Components. Log. Meth. Comp. Sci. **14**(1), 1–31 (2018). https://doi.org/10.23638/LMCS-14(1:1)2018
43. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008). pp. 273–284. ACM (2008). https://doi.org/10.1145/1328438.1328472
44. Jahandideh, I., Ghassemi, F., Sirjani, M.: An actor-based framework for asynchronous event-based cyber-physical systems. Softw. Syst. Model. **20**(3), 641–665 (2021). https://doi.org/10.1007/S10270-021-00877-Y
45. Ji, Z., Wang, S., Xu, X.: Session Types with Multiple Senders Single Receiver. In: Hermanns, H., Sun, J., Bu, L. (eds.) Proceedings of the 9th International Symposium on Dependable Software Engineering. Theories, Tools, and Applications (SETTA 2023). LNCS, vol. 14464, pp. 112–131. Springer (2023). https://doi.org/10.1007/978-981-99-8664-4_7
46. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Revised Papers of the 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). LNCS, vol. 6957, pp. 142–164. Springer (2010). https://doi.org/10.1007/978-3-642-25271-6_8
47. Johnsen, E.B., Owe, O.: An Asynchronous Communication Model for Distributed Concurrent Objects. Softw. Syst. Model. **6**(1), 39–58 (2007). https://doi.org/10.1007/S10270-006-0011-2
48. Jongmans, S., Proença, J.: ST4MP: A Blueprint of Multiparty Session Typing for Multilingual Programming. In: Margaria, T., Steffen, B. (eds.) Proceedings of

the 11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles (ISoLA 2022). LNCS, vol. 13701, pp. 460–478. Springer (2022). https://doi.org/10.1007/978-3-031-19849-6_26
49. Jonsson, B.: Compositional Verification of Distributed Systems. Ph.D. thesis, Uppsala University (1987)
50. Koehler, C., Clarke, D.: Decomposing Port Automata. In: Shin, S.Y., Ossowski, S. (eds.) Proceedings of the 24th ACM Symposium on Applied Computing (SAC 2009). pp. 1369–1373. ACM (2009). https://doi.org/10.1145/1529282.1529587
51. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O Automata for Interface and Product Line Theories. In: De Nicola, R. (ed.) Proceedings of the 16th European Symposium on Programming (ESOP 2007). LNCS, vol. 4421, pp. 64–79. Springer (2007). https://doi.org/10.1007/978-3-540-71316-6_6
52. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
53. Lynch, N.A., Tuttle, M.R.: An Introduction to Input/Output Automata. CWI Q. **2**(3), 219–246 (1989), https://ir.cwi.nl/pub/18164, reprinted in ACM Form. Asp. Comput. (2026), https://doi.org/10.1145/3788687
54. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
55. von Oheimb, D.: Interacting State Machines: A Stateful Approach to Proving Security. In: Abdallah, A.E., Ryan, P.Y.A., Schneider, S.A. (eds.) Revised Papers of the 1st International Conference on Formal Aspects of Security (FASec 2002). LNCS, vol. 2629, pp. 15–32. Springer (2002). https://doi.org/10.1007/978-3-540-40981-6_4
56. Pal, S., Guanciale, R., Lanese, I., Tuosto, E., Clo, M.: Pomsets for Process Management: A Healthcare Case Study. In: Liu, Z., Saoud, A., Wehrheim, H. (eds.) Proceedings of the 22nd International Colloquium on Theoretical Aspects of Computing (ICTAC 2025). LNCS, vol. 16237, pp. 378–395. Springer (2025). https://doi.org/10.1007/978-3-032-11176-0_22
57. Proença, J., Edixhoven, L.: Caos: A Reusable Scala Web Animator of Operational Semantics. In: Jongmans, S.S., Lopes, A. (eds.) Proceedings of the 25th IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION 2023). LNCS, vol. 13908, pp. 163–171. Springer (2023). https://doi.org/10.1007/978-3-031-35361-1_9
58. Rensink, A., Wehrheim, H.: Process algebra with action dependencies. Acta Inform. **38**(3), 155–234 (2001). https://doi.org/10.1007/s002360100070
59. Scalas, A., Yoshida, N.: Less Is More: Multiparty Session Types Revisited. Proc. ACM Program. Lang. **3**, 30:1–30:29 (2019). https://doi.org/10.1145/3290343
60. Severi, P., Dezani-Ciancaglini, M.: Observational Equivalence for Multiparty Sessions. Fundam. Inform. **170**(1-3), 267–305 (2019). https://doi.org/10.3233/FI-2019-1863