Communication Requirements for Team Automata

Maurice H. ter Beek[1], Josep Carmona[2], Rolf Hennicker[3], and Jetty Kleijn[4]

[1] ISTI–CNR, Pisa, Italy
[2] Universitat Politècnica de Catalunya, Barcelona, Spain
[3] Ludwig-Maximilians-Universität, München, Germany
[4] LIACS, Leiden University, The Netherlands

Abstract. Compatibility of components is an important issue in the quest for systems of systems that guarantee successful communications, free from message loss and indefinite waiting for inputs. In this paper, we investigate compatibility in the context of systems consisting of reactive components which may communicate through the synchronised execution of common actions. We model such systems in the team automata framework, which does not impose any a priori restrictions on the synchronisation policy followed to combine the components. We identify a family of representative synchronisation types based on the number of sending and receiving components participating in synchronisations. Then, we provide a generic procedure to derive, for each synchronisation type, requirements for receptiveness and for responsiveness of team automata that prevent that outputs are not accepted and inputs are not provided, respectively. Due to the genericity of our approach w.r.t. synchronisation policies, we can capture compatibility notions for various multi-component system models known from the literature.

1 Introduction

Modern systems are often large-scale concurrent and distributed systems of interconnected, reactive components which collaborate through message exchange. For their correct functioning it is not only important that each component satisfies application-specific properties, but it is also essential that no communication failures, like message loss or indefinite waiting for input, occur during system execution. This requires a deep understanding of the typical communication and interaction policies used in such multi-component systems. To establish that components within a system interact correctly, a concept known as compatibility is useful. In [1], a characterisation was given for compatibility of two components that should engage in a dialogue free from message loss and indefinite waiting. In [2], this binary notion of compatibility was lifted to multi-component systems, in which communication may take place between more than two components at the same time (e.g. broadcasting). Compatibility failures detected in a distributed, modular system model may reveal important problems in the design of one or more of its components, to be repaired before implementation.

Compatibility checks considering various communication and interaction policies thus significantly aid the development of correct component-based systems.

I/O-transition systems are frequently used as a model for reactive components on which to formally define and analyse compatibility. To express reactivity, I/O-transition systems rely on distinguished output (active), input (passive) and internal (privately active) actions. They come in several flavours, like I/O automata [3,4], team automata [5,6], interface automata [7,8], component-interaction automata [9] or modal I/O automata [10]. Several compatibility notions studied in the literature are influenced by the interface automata approach, which uses synchronous point-to-point communication. Two interface automata are said to be compatible if no illegal state can be reached autonomously in the synchronous product of the two. A state is illegal if "one of the automata may produce an output action that is an input action of the other automaton, but not accepted" [7]. The notion was weakened in [11] by allowing a component to still perform some internal actions before accepting the input. Outputs which are not accepted as input are considered as message loss or as unspecified receptions [12, 13]. If any (autonomously chosen) output is accepted, we call this receptiveness [14]. An orthogonal issue concerns the viewpoint of a component waiting to receive an input. It expects an appropriate output to be provided. But in this case the environment can choose which input to serve. Here we refer to this kind of communication requirement (which was already considered as part of a notion of I/O-compatibility in [1]) as responsiveness.

Conditions for receptiveness and responsiveness have been considered in [13] for services and in [2] for team automata. Both approaches support compatibility in multi-component environments for synchronous products, which are known for their appealing compositionality and modularity properties [4,15–18]. A first exploration on how compatibility notions could be generalised to arbitrary synchronisation policies was performed in [14] in the framework of team automata. However, due to the very loose nature of synchronisation policies in team automata, a systematic methodology on how to formalise compatibility conditions in such general settings is still missing. It is the motivation for this work.

The present paper uses as a foundation again the team automata framework, but we additionally define a representative set of communication patterns, called synchronisation types, which help to classify the synchronisation policies that can be realised in team automata. A synchronisation type (snd, rcv) can specify ranges for the number of senders and receivers which can take part in a communication inside the system (possibly based on side conditions). Any synchronisation type uniquely determines a synchronisation policy if the underlying system of components is closed. Otherwise, synchronisation policies with the same type may vary concerning options for interaction with the environment of the system. In any global state of a system $\mathcal{S}$, one of its components or—more generally—a group of components in $\mathcal{S}$ may require certain communications with other components in the system depending on the currently enabled actions. If (common) outputs are enabled in a group of components this leads to requirements for reception. Conversely, enabled inputs lead to requirements for

providing appropriate output, i.e. responsiveness requirements. This allows us to define a notion of compatibility for team automata in terms of their compliance with communication requirements. A team automaton is said to be compliant with communication requirements if the desired communications can immediately occur in the team; it is said to be weakly compliant if the communication can eventually occur after some internal actions have been performed.

In this paper, we propose a general procedure to systematically derive receptiveness and responsiveness requirements from any synchronisation type. Then we can check for any team automaton of synchronisation type (snd, rcv) whether it is compliant with the receptiveness and/or responsiveness requirements derived from (snd, rcv). Thus we get a family of compatibility notions indexed by synchronisation types. Our methodology is illustrated with several examples. We show that our notions can be instantiated with well-known compatibility notions from the literature where particular synchronisation types are considered. In particular, our approach can express two different paradigms for compatibility in open systems, often called the optimistic and pessimistic approaches (cf. [19]).

The paper is organised as follows. In Sect. 2, we introduce team automata followed by the notion of synchronisation types in Sect. 3. In Sect. 4, we define communication requirements for receptiveness and responsiveness and the compliance of team automata with such requirements. In Sect. 5, we show how to derive these requirements from synchronisation types and how known compatibility notions from the literature can be captured. We conclude with Sect. 6.

2 Component Automata and Team Automata

Component automata and team automata are defined as (reactive) automata without final states which distinguish input, output and internal actions and which can be combined by synchronisations on common actions according to synchronisation policies. First we fix some notation.

Given a finite index set $\mathcal{I} = \{1, \ldots, n\}$, we denote the Cartesian product of sets $V_1, \ldots, V_n$ as $\prod_{i \in \mathcal{I}} V_i$. If $v = (v_1, \ldots, v_n) \in \prod_{i \in \mathcal{I}} V_i$ and $i \in \mathcal{I}$, then the i-th entry of v is obtained by applying the projection function $proj_i : \prod_{i \in \mathcal{I}} V_i \to V_i$ defined by $proj_i(v_1, \ldots, v_n) = v_i$.

Definition 1 (Component automaton). *A* component automaton *is a tuple* $\mathcal{A} = (Q, \Sigma, \delta, I)$*, with set* Q *of* states*; set* Σ *of* actions*, such that* $Q \cap \Sigma = \varnothing$*, and* Σ *is the union of three pairwise disjoint sets* Σ_{inp}*,* Σ_{out} *and* Σ_{int} *of* input*,* output *and* internal *actions, respectively;* $\delta \subseteq Q \times \Sigma \times Q$ *is its set of (labelled)* transitions*; and* $\emptyset \neq I \subseteq Q$ *its set of* initial states. □

A (component) automaton (Q, Σ, δ, I) with input, output and internal actions Σ_{inp}, Σ_{out} and Σ_{int}, respectively, may be specified as $(Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, I)$. By Σ_{ext} we denote the set $\Sigma_{inp} \cup \Sigma_{out}$ of external actions. Especially in figures, we may emphasise the role of external actions by appending input actions with ? and output actions with !. For an action $a \in \Sigma$, we define the set of *a-transitions* as $\delta_a = \delta \cap (Q \times \{a\} \times Q)$. We may write $p \xrightarrow{a}_{\mathcal{A}} p'$ instead of $(p, a, p') \in \delta$.

The behaviour of an automaton $\mathcal{A}$ is determined by the execution of actions enabled at its current state. We say that a is *enabled* in $\mathcal{A}$ at state $p \in Q$, denoted by $a\, en_{\mathcal{A}}\, p$, if there exists $p' \in Q$ such that $p \xrightarrow{a}_{\mathcal{A}} p'$. The (finite, sequential) *computations* of $\mathcal{A}$, denoted by $\mathcal{C}(\mathcal{A})$, are those sequences $p_0 a_1 p_1 \cdots p_{k-1} a_k p_k$ such that $k \geq 0$, $p_0 \in I$ and $p_{i-1} \xrightarrow{a_i}_{\mathcal{A}} p_i$ for all $i \in \{1, \ldots, k\}$. For $X \subseteq \Sigma$, we write $p \xrightarrow{X}{}^*_{\mathcal{A}} p'$ if there exists $p_0 \xrightarrow{a_1}_{\mathcal{A}} p_1, \ldots, p_{j-1} \xrightarrow{a_j}_{\mathcal{A}} p_j$ for some $j \geq 0$, with $p_0, \ldots p_j \in Q$, $a_1, \ldots, a_j \in X$, $p = p_0$, and $p' = p_j$. A state $p \in Q$ is *reachable* if $p_0 \xrightarrow{\Sigma}{}^*_{\mathcal{A}} p$ (with $p_0 \in I$) and the set of reachable states of $\mathcal{A}$ is denoted by $\mathcal{R}(\mathcal{A})$.

As usual, we may omit subscripts referring to $\mathcal{A}$ if no confusion can arise.

Team automata consist of component automata that collaborate through synchronised executions of shared actions. When and which actions are executed and by how many components depends on the chosen synchronisation policy.

Let $\mathcal{I} = \{1, \ldots, n\}$ be a finite index set. Let $\mathcal{S} = \{\, \mathcal{A}_i \mid i \in \mathcal{I} \,\}$ be a set of component automata defined, for each $i \in \mathcal{I}$, as $\mathcal{A}_i = (Q_i, (\Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}), \delta_i, I_i)$ with $\Sigma_i = \Sigma_{i,inp} \cup \Sigma_{i,out} \cup \Sigma_{i,int}$. $\mathcal{S}$ is *composable* if $\Sigma_{i,int} \cap \bigcup_{j=1, j\neq i}^{n} \Sigma_j = \varnothing$ for all $i \in \mathcal{I}$. Thus in a composable system, internal actions are not shared. Note that every subset of a composable set of component automata is again composable.

$\Sigma = \bigcup_{i\in\mathcal{I}} \Sigma_i$ is the set of actions of $\mathcal{S}$, $\Sigma_{int} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,int}$ its set of internal actions and $\Sigma_{ext} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,ext}$ its set of external actions. Moreover, $\Sigma_{com} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,inp} \cap \bigcup_{i\in\mathcal{I}} \Sigma_{i,out}$ is the set of *communicating* actions of $\mathcal{S}$. Hence, an action (of $\mathcal{S}$) is communicating if it occurs in Σ_{ext} both as an input action of one of the automata and as an output action of an automaton.

For an action $a \in \Sigma$, its domain in $\mathcal{S}$, denoted by $\mathrm{dom}_a(\mathcal{S})$, consists of the indices of all automata from $\mathcal{S}$ in which it appears as an action. So, $\mathrm{dom}_a(\mathcal{S}) = \{\, i \mid a \in \Sigma_i \,\}$. Hence in a composable system, the domain of an internal action is always a singleton set. For $a \in \Sigma_{ext}$, we let $\mathrm{dom}_{a,inp}(\mathcal{S}) = \{\, i \mid a \in \Sigma_{i,inp} \,\}$ be its *input domain* (in $\mathcal{S}$) and $\mathrm{dom}_{a,out}(\mathcal{S}) = \{\, i \mid a \in \Sigma_{i,out} \,\}$ its *output domain* (in $\mathcal{S}$). Hence an action is a communicating action of $\mathcal{S}$ if both its output and its input domain in $\mathcal{S}$ are not empty.

Finally, we say that $\mathcal{S}$ is *open* if it has external actions that are not communicating (they appear only as an input or only as an output action). If $\mathcal{S}$ is not open, it may be referred to as *closed*; in this case all its external actions are communicating (all have at least one communication partner).

Notation. *For the remainder of this paper, we fix $\mathcal{I}$ and $\mathcal{S}$ as above. Moreover, $\mathcal{S}$ is composable. We refer to $Q = \prod_{i\in\mathcal{I}} Q_i$ as the state space of $\mathcal{S}$ and to Σ, Σ_{int}, Σ_{ext} and Σ_{com} as its set of actions, internal actions, external actions and communicating actions, respectively.*

Definition 2 (System transition). *A tuple $(q, a, q') \in Q \times \Sigma \times Q$ is a* transition on a *(in $\mathcal{S}$) if there exists an $i \in \mathcal{I}$ such that $(proj_i(q), a, proj_i(q')) \in \delta_i$, and if for all $i \in \mathcal{I}$, either $(proj_i(q), a, proj_i(q')) \in \delta_i$ or $proj_i(q) = proj_i(q')$.*

For $a \in \Sigma$, $\Delta_a(\mathcal{S})$ is the set of all transitions on a *in $\mathcal{S}$, while $\Delta(\mathcal{S}) = \bigcup_{a\in\Sigma} \Delta_a(\mathcal{S})$ is the set of all* transitions *in $\mathcal{S}$.* □

If $(q, a, q') \in \Delta(\mathcal{S})$, then any component $\mathcal{A}_i$ for which $(proj_i(q), a, proj_i(q')) \in \delta_i$ is said to be *involved in* (q, a, q'). By definition, in all transitions in $\mathcal{S}$, at least one

component is involved through a 'local' transition. Moreover, all transitions in $\Delta_a(\mathcal{S})$ are combinations of existing a-transitions from the component automata in $\mathcal{S}$ and all possible combinations occur in $\Delta_a(\mathcal{S})$. As in earlier papers, we will often refer to the elements of $\Delta_a(\mathcal{S})$ as *synchronisations on* a also when no more than one component is actively involved. In particular, when a is an internal action of a component automaton, then all transitions on a are executed by that component alone. Moreover, for each transition on an external action in one of the automata, $\Delta(\mathcal{S})$ will also contain all synchronisations that involve only that component through that particular local transition. When a synchronisation on an external action a involves both a component in which a is an input action and one in which it is an output action, it is called a *communication*.

All team automata over $\mathcal{S}$ will have Σ as their set of actions, consisting of the external actions Σ_{ext} of the components and the internal actions Σ_{int} comprising all internal actions of the components. In addition, we need to define the sets of *input* and *output* actions. We follow the idea from [6] that components have control over their output actions whereas input actions are passive, i.e. driven by the environment. As a consequence, actions that appear as an output action in one or more of the components are considered to be under the control of the team and hence will be output actions of the team (even if they are input to some other components). Input actions that do not appear as output, are input actions of the team. Formally, $\Sigma_{out} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,out}$ and $\Sigma_{inp} = (\bigcup_{i\in\mathcal{I}} \Sigma_{i,inp})\backslash\Sigma_{out}$. Furthermore, $Q = \prod_{i\in\mathcal{I}} Q_i$ will be the set of states of every team automaton over $\mathcal{S}$ and $I = \prod_{i\in\mathcal{I}} I_i$ its set of initial states.

Finally, it is the choice of synchronisations, thus the choice of a subset δ of $\Delta(\mathcal{S})$, that defines a specific team automaton. As internal actions are assumed to be under the control of the component automata, all transitions on internal actions will always be included as transitions of any team automaton over $\mathcal{S}$. Subsets δ of $\Delta(\mathcal{S})$, such that $\delta_a = \Delta_a(\mathcal{S})$ for all $a \in \Sigma_{int}$, are referred to as *synchronisation policies* (over $\mathcal{S}$).

Definition 3 (Team automaton). *The* team automaton *over* $\mathcal{S}$ *with synchronisations* δ *is the component automaton* $\mathcal{T} = (Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, I)$. □

Each team automaton determines a synchronisation policy over $\mathcal{S}$ and vice versa. Since every team automaton is a component automaton, team automata can be used in hierarchical constructions (systems of systems).

3 Synchronisation Types

We have seen that team automata over a composable system are defined by synchronisation policies. For all states of the system and for each external action enabled at the corresponding local state of at least one of its components, it has to be decided which synchronisations on that action to include as a (team) transition. In practice, this will seldom be decided individually for every candidate synchronisation. The system designer will most likely have a certain synchronisation pattern in mind. In this section, we introduce so-called synchronisation types which allow us to define in a compact way specific synchronisation policies.

Synchronisation types specify lower and upper bounds on the number of components involved in a synchronisation or they indicate that the synchronisation is of an *action-indispensible* or *state-indispensible* type. These notions were originally introduced in [6]. There an action-indispensible synchronisation policy requires for every team transition on a given action the involvement of all components to which that action belongs; a policy is state-indispensible if in every team transition on a given action all components that could be involved (because that action is enabled at the current local state) are involved. Here, we apply this idea to communicating actions and distinguish between their input and output roles. We use ai and si to indicate the number of input or output components that could maximally be involved in a synchronisation on a communicating action (having that action as input or output, respectively, and for si the action is moreover enabled at the current local state).

The next definition introduces synchronisation types as pairs that can be used to specify for a synchronisation on a communicating action, possible numbers of components involved as sending components (for which the action executed is an output action) and as receiving components (for which the action is an input).

Definition 4 (Synchronisation type). *A* synchronisation type *is a pair* (snd, rcv) *such that for* $x = snd$ *and for* $x = rcv$ *either* x *is an interval* $[k, m]$ *with* $0 \leq k$ *and* $(k \leq m$ *or* $m = *)$ *or* $x \in \{ai, si\}$. *We call* snd *and* rcv *the* sending *and* receiving multiplicity*, respectively, of the synchronisation type.* □

Next, we turn to synchronisations. For $(p, a, p') \in \Delta(\mathcal{S})$, the number of automata involved as output or input component in (p, a, p') is denoted as follows:

$$out_a(p, a, p') = \#\{\, i \in \mathcal{I} \mid (proj_i(p), a, proj_i(p')) \in \delta_i \text{ and } a \in \Sigma_{i,out} \,\}$$
$$inp_a(p, a, p') = \#\{\, i \in \mathcal{I} \mid (proj_i(p), a, proj_i(p')) \in \delta_i \text{ and } a \in \Sigma_{i,inp} \,\}$$

To be able to deal with si, we denote the number of automata, for which an output or input action $a \in \Sigma_{com}$ is locally enabled at state $p \in Q$, as follows:

$$out_{si}(p, a) = \#\{\, i \in \mathcal{I} \mid a \; en_{\mathcal{A}_i} \; proj_i(p) \text{ and } a \in \Sigma_{i,out} \,\}$$
$$inp_{si}(p, a) = \#\{\, i \in \mathcal{I} \mid a \; en_{\mathcal{A}_i} \; proj_i(p) \text{ and } a \in \Sigma_{i,inp} \,\}$$

In what follows, $\ell \in \mathbb{N}$ is said to *satisfy an interval* $[k, m]$ with $0 \leq k \leq m$ whenever $k \leq \ell \leq m$; and ℓ satisfies $[k, *]$ if $k \leq \ell$.

Definition 5 (Typed synchronisation policy). *Let* $a \in \Sigma_{com}$, $p \in Q$ *and* $(p, a, p') \in \Delta(\mathcal{S})$. *Then*

$$(p, a, p') \textit{ is of type } (snd, rcv) \textit{ if } \begin{cases} snd = [o_1, o_2] & \textit{and } out_a(p, a, p') \textit{ satisfies } [o_1, o_2] \\ snd = ai & \textit{and } out_a(p, a, p') = \#dom_{\mathrm{a,out}}(\mathcal{S}) \\ snd = si & \textit{and } out_a(p, a, p') = out_{si}(p, a) \\ rcv = [i_1, i_2] & \textit{and } inp_a(p, a, p') \textit{ satisfies } [i_1, i_2] \\ rcv = ai & \textit{and } inp_a(p, a, p') = \#dom_{\mathrm{a,inp}}(\mathcal{S}) \\ rcv = si & \textit{and } inp_a(p, a, p') = inp_{si}(p, a) \end{cases}$$

We say that a synchronisation policy $\delta \subseteq \Delta(\mathcal{S})$ is of type (snd, rcv) *if* δ *contains, for all* $a \in \Sigma_{com}$*, all transitions on* a *of type* (snd, rcv) *and no other transitions on* a*. A team automaton* $\mathcal{T}$ *over* $\mathcal{S}$ *with synchronisation policy* δ is of type (snd, rcv) *if* δ *is of type* (snd, rcv)*.* □

From Definition 5 it follows that for closed systems where all external actions are communicating, a synchronisation type (snd, rcv) determines a unique synchronisation policy δ and hence a team automaton. Synchronisation types do not apply to non-communicating external actions and so, if the system is open, a synchronisation policy of a certain type may contain any subset of transitions $(p, a, p') \in \Delta(\mathcal{S})$ with actions $a \in \Sigma_{ext} \setminus \Sigma_{com}$. If all of them are selected, then the synchronisation policy is called *maximal*.

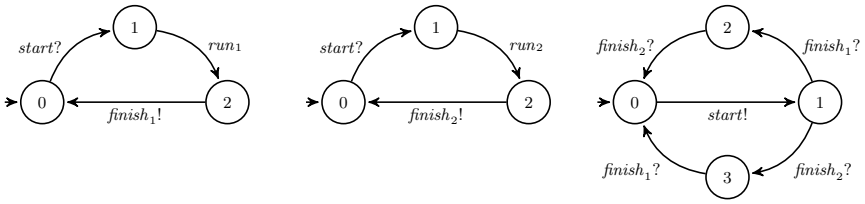
Note that a transition in $\mathcal{S}$ may be of several, different types. Furthermore, a team automaton may have a synchronisation policy that includes communications that do not have a common synchronisation type.

Let us now consider some familiar synchronisation types which occur in the literature and in concrete systems.

- $([1,1],[1,1])$: binary communication, meaning that a communicating action can be executed only as a synchronisation involving exactly one component for which it is an output action and exactly one for which it is an input action.
- $([1,1],[0,1])$: as directly above, but now over a lossy channel, meaning that a communicating action can be lost (i.e. involving exactly one component for which it is an output action and at most one for which it is an input action).
- $([1,1],[0,*])$: multicast communication, meaning that a communicating action can be executed only as a synchronisation involving exactly one component for which it is an output action and any number of the components in which it is an input action. This is called weak synchronisation in BIP [20].
- $([1,1], si)$: broadcast communication, meaning that whenever a communicating action is executed it occurs exactly once in its output role in that transition with as many as possible (all currently enabled) input components involved.
- $([1,1], ai)$: strong broadcast communication, as directly above, but now with all input components involved. This is called strong synchronisation in BIP.
- (ai, ai): transitions on communicating actions are always 'full' synchronisations, meaning that all components that share a communicating action are involved in all transitions on that action. When all external actions are communicating ($\mathcal{S}$ is a closed system), this means that we are dealing with the classical synchronous product of automata (cf., e.g., [2, 14, 21]).
- $([1,*],[0,*])$: transitions on communicating actions always involve at least one component where that action is an output action. This is the idea of 'master-slave' communication (cf. [6]), according to which a master (output) can always be executed and slaves (input) never proceed on their own.
- $([1,*],[1,*])$: as directly above, but now at least one slave has to 'obey' (the master). This is called 'strong master-slave' communication (cf. [6]), by which a master (output) can always be executed and slaves (input) must be involved.
- $([0,1],[0,1])$: not obligatory binary communication (communicating actions may also be executed as stand alone) like in CCS [22]. □

These synchronisation types define team automata based on one type of synchronisation only, but for future work combinations could be imagined as well.

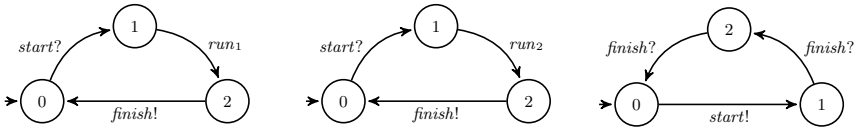
Example 1. We consider the system $Sys_1 = \{\mathcal{R}unner_1, \mathcal{R}unner_2, \mathcal{C}ontroller\}$ depicted in Fig. 1. Here and in all subsequent examples components have exactly one initial state denoted by 0. All actions apart from the internal actions run_1 and run_2 are communicating. We want to combine these components in a team in a way that the controller component starts both runner components at the same time, but each runner can separately signal to the controller when it has reached the finish line. To this aim, the synchronisation type (ai, ai) with all transitions on communicating actions being full synchronisations is appropriate. Thus we obtain the team automaton $\mathcal{T}_1$ of type (ai, ai) over Sys_1. (Since the system is closed, this team is unique.) □



(a) $\mathcal{R}unner_1$ (b) $\mathcal{R}unner_2$ (c) $\mathcal{C}ontroller$

Fig. 1: Automata $\mathcal{R}unner_i$, with $i \in \{1, 2\}$, and $\mathcal{C}ontroller$ of $\mathcal{S}ys_1$

Example 2. Now we consider the system $Sys_2 = \{\mathcal{R}unner'_1, \mathcal{R}unner'_2, \mathcal{C}ontroller'\}$ depicted in Fig. 2. The idea is similar to Example 1. As before, the controller should start the runners at the same time and each runner should separately send its finish signal to the controller. The difference with Sys_1 is that both runners use the same finish signal to communicate with the controller. Therefore we cannot use the synchronisation type (ai, ai) but choose the type $([1, 1], ai)$ instead. The sending multiplicity $[1, 1]$ enforces that communication in the system will always involve exactly one sender, which precludes the two runners sending their finish signal together. The receiving multiplicity is ai since the two runners must receive the start signal together. This leads to the team automaton $\mathcal{T}_2$ of type $([1, 1], ai)$ over the system Sys_2. □



(a) $\mathcal{R}unner'_1$ (b) $\mathcal{R}unner'_2$ (c) $\mathcal{C}ontroller'$

Fig. 2: Automata $\mathcal{R}unner'_i$, with $i \in \{1, 2\}$, and $\mathcal{C}ontroller'$ of $\mathcal{S}ys_2$

4 Communication Requirements

In this paper, we are interested in the communications between components in a team built over the system $\mathcal{S}$. In any state p of $\mathcal{S}$, one of its components or, more generally, a group of components in $\mathcal{S}$ may require certain communications with other components in the system. This is formally expressed by communication requirements. In the following, we represent a group of components in $\mathcal{S}$ by their indices, i.e. by a non-empty subset $\mathcal{J} \subseteq \mathcal{I}$. By abuse of terminology, we will often identify $\mathcal{J}$ with the group of components represented by $\mathcal{J}$.

For a communicating action $a \in \Sigma_{com}$, a group $\mathcal{J} \subseteq \mathrm{dom}_{a,out}(\mathcal{S})$ in the output domain of a may have a communication requirement $(\mathcal{J}, a)$ at some state p, if (output) action a is enabled in the local states $proj_j(p)$ of all components $\mathcal{A}_j$ with $j \in \mathcal{J}$. This requirement expresses that at least one component in the input domain of a should communicate with group $\mathcal{J}$ and receive a in the current state. Thus (synchronised groups of) sending components can have demands w.r.t. the reception of an output action and therefore $(\mathcal{J}, a)$ will be called a *receptiveness requirement*. According to Defs. 7 and 8 below, it will depend on the synchronisation policy of a team whether receptiveness requirements are fulfilled.

Similarly, we consider groups $\mathcal{J} \subseteq \mathrm{dom}_{a,inp}(\mathcal{S})$ in the input domain of a. Then a communication requirement $(\mathcal{J}, a)$ can be given for a state p, if (input) action a is enabled in the local states $proj_j(p)$ of all components $\mathcal{A}_j$ with $j \in \mathcal{J}$. According to this requirement at least one component in the output domain of a should communicate with the group and send a in the current state. Thus (synchronised groups of) receiving components may require output from other components and then $(\mathcal{J}, a)$ will also be called a *responsiveness requirement* (although it is not necessarily a response to a former call). Again it will depend on the synchronisation policy of a team whether responsiveness requirements are satisfied (cf. Defs. 7 and 8).

Communication requirements can be combined by conjunction and disjunction. As we shall see in Sect. 5, the former will be in particular useful for combining receptiveness requirements and the latter for responsiveness requirements.

Definition 6 (Communication requirement).

(i) *A* receptiveness requirement at $p \in Q$ *is a pair* $(\mathcal{J}, a)$ *with* $a \in \Sigma_{com}$ *and* $\varnothing \neq \mathcal{J} \subseteq dom_{a,out}(\mathcal{S})$ *such that* $a\, en_{\mathcal{A}_j}\, proj_j(p)$ *for all* $j \in \mathcal{J}$.

(ii) *A* responsiveness requirement at $p \in Q$ *is a pair* $(\mathcal{J}, a)$ *with* $a \in \Sigma_{com}$ *and* $\varnothing \neq \mathcal{J} \subseteq dom_{a,inp}(\mathcal{S})$ *such that* $a\, en_{\mathcal{A}_j}\, proj_j(p)$ *for all* $j \in \mathcal{J}$.

(iii) *An* atomic communication requirement at $p \in Q$ *is either the trivial requirement true or a receptiveness requirement at* p *or a responsiveness requirement at* p.

(iv) *A* communication requirement at $p \in Q$ *is either an atomic communication requirement or a conjunction* $\psi_1 \wedge \psi_2$ *or a disjunction* $\psi_1 \vee \psi_2$ *of communication requirements* ψ_1 *and* ψ_2 *at* p □

When all non-trivial atomic requirements occurring in a communication requirement φ are receptiveness (responsiveness) requirements, we also refer to φ as a receptiveness (responsiveness) requirement, respectively.

Definition 7 (Compliance). *A team automaton $\mathcal{T}$ over $\mathcal{S}$ with synchronisation policy δ is* compliant with *a communication requirement φ at $p \in Q$ if either $p \notin \mathcal{R}(\mathcal{T})$ or $\varphi = true$, or*

(a) $\varphi = (J, a)$ is a receptiveness requirement at p and there exist $i \in dom_{a,inp}(\mathcal{S})$ and a transition $p \xrightarrow{a}_{\mathcal{T}} p'$ such that $(proj_k(p), a, proj_k(p')) \in \delta_k$ for all $k \in \mathcal{J} \cup \{i\}$;
(b) $\varphi = (J, a)$ is a responsiveness requirement at p and there exist $i \in dom_{a,out}(\mathcal{S})$ and a transition $p \xrightarrow{a}_{\mathcal{T}} p'$ such that $(proj_k(p), a, proj_k(p')) \in \delta_k$ for all $k \in \mathcal{J} \cup \{i\}$;
(c) $\varphi = \psi_1 \wedge \psi_2$ and $\mathcal{T}$ is compliant with ψ_1 at p and with ψ_2 at p;
(d) $\varphi = \psi_1 \vee \psi_2$ and $\mathcal{T}$ is compliant with ψ_1 at p or with ψ_2 at p. □

Note that when $\mathcal{T}$ is compliant with an atomic receptiveness requirement at a state p, then according to (a) above, the output a from the components defined by $\mathcal{J}$ can be received by a component $\mathcal{A}_i$, but this may be realised through a synchronisation at p involving more components from the output and input domains of a. A similar remark holds for compliance with responsiveness requirements.

Communication requirements can be used to express various properties that may emerge during the computations of a team automaton, such as progress properties. As an example, when a team automaton $\mathcal{T}$ is compliant with a non-trivial communication requirement at state p, then *communication progress* is possible at p, i.e. $a\, en_{\mathcal{T}}\, p$ for some $a \in \Sigma_{com}$.

In general, the definition of compliance as we have it now, may be too strong in the sense that it could be natural to allow the team to execute some intermediate, internal ('silent') actions before it would be ready for the required communication. This leads to the notion of *weak compliance* following the ideas of weak compatibility introduced in [11].

Definition 8 (Weak compliance). *A team automaton $\mathcal{T}$ over $\mathcal{S}$ with synchronisation policy δ is* weakly compliant with *a communication requirement φ at $p \in Q$ if either $p \notin \mathcal{R}(\mathcal{T})$ or $\varphi = true$, or*

(a) $\varphi = (J, a)$ is a receptiveness requirement at p and there exist $q \in Q$ with $proj_j(p) = proj_j(q)$, for all $j \in \mathcal{J}$, and $i \in dom_{a,inp}(\mathcal{S})$ such that $p \xrightarrow{\Sigma_{int}}{}^*_{\mathcal{T}} q \xrightarrow{a}_{\mathcal{T}} p'$ holds and $(proj_k(q), a, proj_k(p')) \in \delta_k$ for all $k \in \mathcal{J} \cup \{i\}$;
(b) $\varphi = (J, a)$ is a responsiveness requirement at p and there exist $q \in Q$ with $proj_j(p) = proj_j(q)$, for all $j \in \mathcal{J}$, and $i \in dom_{a,out}(\mathcal{S})$ such that $p \xrightarrow{\Sigma_{int}}{}^*_{\mathcal{T}} q \xrightarrow{a}_{\mathcal{T}} p'$ and $(proj_k(q), a, proj_k(p')) \in \delta_k$ for all $k \in \mathcal{J} \cup \{i\}$;
(c) $\varphi = \psi_1 \wedge \psi_2$ and $\mathcal{T}$ is weakly compliant with ψ_1 at p and with ψ_2 at p;
(d) $\varphi = \psi_1 \vee \psi_2$ and $\mathcal{T}$ is weakly compliant with ψ_1 at p or with ψ_2 at p. □

Obviously, compliance implies weak compliance. Observe furthermore that we require that the components defined by $\mathcal{J}$ do not change their local state as a result of the execution of the silent actions. This implies that these components do not (have to) execute internal actions to reach the global state where the required

communication would be possible. Moreover, the definition given here makes it possible that also components not involved in the eventual communication, take part in the silent computation needed to reach a team state where that communication could take place. This phenomenon is known as 'state-sharing' (cf. [6,23]) and allows components to influence potential synchronisations through their local states without being involved in the actual transition.

Example 3. We consider the team automaton $\mathcal{T}_1$, introduced in Example 1, with synchronisation type (ai, ai). We denote the states of $\mathcal{T}_1$ by triples (q_1, q_2, q_3) where q_1 is a state of $\mathcal{C}ontroller$, q_2 a state of $\mathcal{R}unner_1$, and q_3 a state of $\mathcal{R}unner_2$. Examples for receptiveness requirements are:

$(\{\mathcal{C}ontroller\}, start)$ at $(0,0,0)$
$(\{\mathcal{R}unner_1\}, finish_1) \wedge (\{\mathcal{R}unner_2\}, finish_2)$ at $(1,2,2)$

The first one expresses that in the initial state the start signal of the controller should be received (by at least one runner); the second one that in state $(1,2,2)$ each runner wants its finish signal to be received (by the controller). Obviously, the team automaton $\mathcal{T}_1$ is compliant with both receptiveness requirements.

Examples for responsiveness requirements are:

$(\{\mathcal{R}unner_1, \mathcal{R}unner_2\}, start)$ at $(0,0,0)$
$(\{\mathcal{C}ontroller\}, finish_1) \vee (\{\mathcal{C}ontroller\}, finish_2)$ at $(1,1,1)$

The first requirement concerns the group consisting of the two runners which together request to be started. The second one expresses that in state $(1,1,1)$ the controller expects a finish signal either from $\mathcal{R}unner_1$ or from $\mathcal{R}unner_2$. This illustrates the use of disjunctions to reflect external choice of inputs. Note that $\mathcal{T}_1$ is *not* compliant but only *weakly* compliant with this requirement at $(1,1,1)$. In this state, neither $finish_1$ nor $finish_2$ can be sent immediately to the controller, but either one can be sent when the respective component has done its running (an internal action). □

Theorem 1. *Let $\mathcal{T}$ be a team automaton over $\mathcal{S}$ and, for each $p \in \mathcal{R}(\mathcal{T})$, let ϕ_p be a non-trivial*[5] *communication requirement such that $\mathcal{T}$ is weakly compliant with each ϕ_p at p. Then at all reachable states of $\mathcal{T}$, at least one action is enabled.*[6]

Proof. For every reachable state p of $\mathcal{T}$ there is at least one atomic communication requirement with which $\mathcal{T}$ is compliant. Hence, in state p, the requested action a can eventually be executed in the team. □

5 Deriving Communication Requirements

In the previous section, we have introduced the concepts of communication requirement and compliance of team automata. However, we provided no methodological guidelines outlining when which requirements would be meaningful. Consider, for instance, the team automaton $\mathcal{T}_2$ of Example 2 with synchronisation

[5] i.e. it cannot be logically reduced to *true*.
[6] i.e. $\mathcal{R}(\mathcal{T})$ contains no deadlock states.

type $([1,1], ai)$. The global state $(1,2,2)$ is a reachable state of $\mathcal{T}_2$ at which for both runner components, the output action *finish* is locally enabled. Hence, $(\{\mathcal{R}unner_1, \mathcal{R}unner_2\}, finish)$ is formally a receptiveness requirement at $(1,2,2)$. This requirement does not make much sense, though, because of the sending multiplicity $[1,1]$ by which always exactly one sender participates in the execution of a communicating action. Therefore, the choice of suitable communication requirements should take the synchronisation type of the team into account.

In this section, we propose a general procedure to derive communication requirements from an arbitrary synchronisation type. The approach was inspired by initial ideas for a generic definition of compatibility of components relative to the adopted synchronisation policy in [14] (where no classification of synchronisation types was considered and no derivation procedure was envisioned). We will do so separately for receptiveness (Sect. 5.1) and responsiveness requirements (Sect. 5.2). Thus, we get for all synchronisation types introduced in Def. 4, a compatibility notion w.r.t. receptiveness (Def. 9) and responsiveness (Def. 10) suitable for all team automata with a synchronisation policy of that type.

5.1 Deriving Receptiveness Requirements

We first formulate receptiveness requirements for each synchronisation type (snd, rcv). We distinguish the following cases.

Case: *snd* arbitrary, $rcv = [0, i_2]$ or $rcv = si$. In this case, the synchronisation policy allows that sending components progress also when their output will not be received. Thus we have no more than the trivial receptiveness requirement `true` at all states $p \in Q$.

In the following cases, we assume that neither $rcv = [0, i_2]$ nor $rcv = si$ and proceed with a case distinction on *snd*.

Case: $snd = [o_1, o_2]$. In this case, the subsets *relevant* to our considerations are those $\mathcal{J} \subseteq \mathcal{I}$ with $o_1 \leq |\mathcal{J}| \leq o_2$. Let $p \in Q$ be a global state. For each such $\mathcal{J}$, we consider all communicating (output) actions a which are simultaneously enabled at the current local states $proj_j(p)$ of the components $\mathcal{A}_j$, i.e. $a\ en_{\mathcal{A}_j}\ proj_j(p)$ for all $j \in \mathcal{J}$. This leads to the following receptiveness requirement at p:

$$\bigwedge \{\, (\mathcal{J}, a) \mid \varnothing \neq \mathcal{J} \subseteq \mathcal{I},\ o_1 \leq |\mathcal{J}| \leq o_2, \\ \text{and, for all } j \in \mathcal{J}, a \in \Sigma_{j,out} \cap \Sigma_{com} \text{ and } a\ en_{\mathcal{A}_j}\ proj_j(p) \,\}$$

We use conjunction here to reflect that whatever output action will be executed, a corresponding input is required. If the set of all pairs $(\mathcal{J}, a)$ considered above is empty, then there is no proper receptiveness requirement at p other than the trivial requirement `true`.

To conclude, a team automaton $\mathcal{T}$ over $\mathcal{S}$ of type $([o_1, o_2], rcv)$ (such that neither $rcv = [0, i_2]$ nor $rcv = si$) is (weakly) compliant with the receptiveness requirements at p if whenever a group of components wants to perform an output a, then the team can (eventually) carry out a synchronisation such that a is sent by the group and received by some (at least one) other component.

Case: *snd* $= ai$. In this case, an action a is only executed as an output action if *all* components in its output domain are at a state at which a is enabled. Hence we formulate the following receptiveness requirements for states $p \in Q$:

$$\bigwedge\{\,(\mathcal{J},a) \mid \varnothing \neq \mathcal{J} = \mathrm{dom}_{a,out}(\mathcal{S}),\ a \in \Sigma_{com} \text{ and, for all } j \in \mathcal{J},\ a\, en_{\mathcal{A}_j}\, proj_j(p)\,\}$$

These receptiveness requirements apply, in particular, to the synchronous product where the synchronisation type is (ai, ai).

Case: *snd* $= si$. This case is similar to $snd = ai$, but taking into account that now an (output) action can be executed in any synchronisation involving all components where it is locally enabled at p.

$$\bigwedge\{\,(\mathcal{J},a) \mid \varnothing \neq \mathcal{J} = \{\,i \in \mathcal{I} \mid a\, en_{\mathcal{A}_i}\, proj_i(p) \text{ and } a \in \Sigma_{i,out}\,\},\ a \in \Sigma_{com}\,\}$$

Combining the synchronisation types from Def. 4 with the above receptiveness requirements gives rise to the following definition.

Definition 9 (Receptive team automaton). *Let $\mathcal{T}$ be a team automaton of type (snd, rcv) over $\mathcal{S}$. $\mathcal{T}$ is* (weakly) receptive *if it is (weakly) compliant at all $p \in \mathcal{R}(\mathcal{T})$ with the receptiveness requirements derived above for (snd, rcv).* □

5.2 Deriving Responsiveness Requirements

We now formulate responsiveness requirements for each synchronisation type (snd, rcv). We distinguish the following cases.

Case: *snd* $= [0, o_2]$ or *snd* $= si$, *rcv* arbitrary. In this case, the synchronisation policy allows that receiving components progress also without being triggered by output. Thus we have no more than the trivial responsiveness requirement `true` at all states $p \in Q$.

In the following cases, we assume that neither $snd = [0, o_2]$ nor $snd = si$ and proceed with a case distinction on rcv.

Case: *rcv* $= [i_1, i_2]$. In this case, the subsets *relevant* to our considerations are those $\mathcal{J} \subseteq \mathcal{I}$ with $i_1 \leq |\mathcal{J}| \leq i_2$. Let $p \in Q$ be a global state. For each such $\mathcal{J}$, we consider all communicating (input) actions a which are simultaneously enabled at the current local states $proj_j(p)$ of the components $\mathcal{A}_j$, i.e. $a\, en_{\mathcal{A}_j}\, proj_j(p)$ for all $j \in \mathcal{J}$. This leads to the following responsiveness requirement at p:

$$\bigvee\{\,(\mathcal{J},a) \mid \varnothing \neq \mathcal{J} \subseteq \mathcal{I},\ o_1 \leq |\mathcal{J}| \leq o_2,\ \text{and, for all } j \in \mathcal{J}, a \in \Sigma_{j,inp} \cap \Sigma_{com} \text{ and } a\, en_{\mathcal{A}_j}\, proj_j(p)\,\}$$

We use disjunction here to reflect that the choice of a particular input action is made by the environment, but (at least) one of the inputs must be served. If the set of all pairs $(\mathcal{J}, a)$ considered above is empty, then there is no proper responsiveness requirement at p other than the trivial requirement `true`.

To conclude, a team automaton $\mathcal{T}$ over $\mathcal{S}$ of type $(snd, [i_1, i_2])$ (such that neither $snd = [0, o_2]$ nor $snd = si$) is (weakly) compliant with the responsiveness

requirements at p if whenever a group of components can perform an input action a, then the team can (eventually) carry out a synchronisation such that a is sent by some (at least one) component of the group.

Case: $rcv = ai$. In this case, an action a is only executed as an input action if *all* components in its input domain are at a state at which a is enabled. Hence we formulate the following responsiveness requirements for states $p \in Q$:

$$\bigvee\{\,(J,a) \mid \varnothing \neq \mathcal{J} = \mathrm{dom}_{a,inp}(\mathcal{S}),\ a \in \Sigma_{com} \text{ and, for all } j \in \mathcal{J},\ a\, en_{\mathcal{A}_j}\, proj_j(p)\,\}$$

These responsiveness requirements apply, in particular, to the synchronous product where the synchronisation type is (ai, ai).

Case: $rcv = si$. This case is similar to $rcv = ai$, but taking into account that now an (input) action can be executed in any synchronisation involving all components where it is locally enabled at p.

$$\bigwedge\{\,(\mathcal{J},a) \mid \varnothing \neq \mathcal{J} = \{\,i \in \mathcal{I} \mid a\, en_{\mathcal{A}_i}\, proj_i(p) \text{ and } a \in \Sigma_{i,inp}\,\},\ a \in \Sigma_{com}\,\}$$

Combining the synchronisation types from Def. 4 with the above responsiveness requirements gives rise to the following definition.

Definition 10 (Responsive team automaton). *Let $\mathcal{T}$ be a team automaton of type (snd, rcv) over $\mathcal{S}$. $\mathcal{T}$ is* (weakly) responsive *if it is (weakly) compliant at all $p \in \mathcal{R}(\mathcal{T})$ with the responsiveness requirements derived above for (snd, rcv).* □

5.3 Examples

Example 4. We consider the team automaton $\mathcal{T}_1$ of Example 1 with synchronisation type (ai, ai). It is sufficient to consider communication requirements only for those states which are reachable. First we derive receptiveness requirements according to Sect. 5.1:

$(\{\mathcal{C}ontroller\}, start)$ at $(0,0,0)$
$true$ at $(1,1,1)$
$(\{\mathcal{R}unner_1\}, finish_1)$ at $(1,2,1)$
$(\{\mathcal{R}unner_2\}, finish_2)$ at $(1,1,2)$
$(\{\mathcal{R}unner_1\}, finish_1) \wedge (\{\mathcal{R}unner_2\}, finish_2)$ at $(1,2,2)$
$true$ at $(2,0,1)$
$(\{\mathcal{R}unner_2\}, finish_2)$ at $(2,0,2)$
$true$ at $(3,1,0)$
$(\{\mathcal{R}unner_1\}, finish_1)$ at $(3,2,0)$

All receptiveness requirements are straightforward and express our intuition that if a component can send an action then the other component(s) should be ready to receive it. Obviously, $\mathcal{T}_1$ is compliant with all receptiveness requirements.

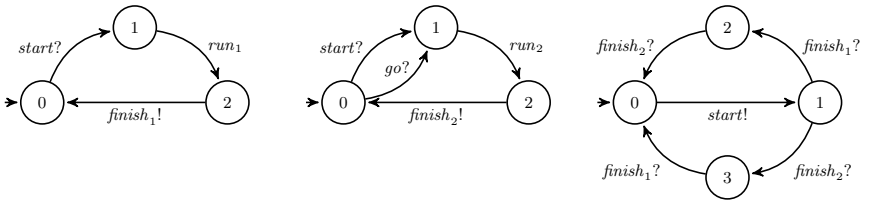
Let us now derive responsiveness requirements according to Sect. 5.2. We get:

$(\{\mathcal{R}unner_1, \mathcal{R}unner_2\}, start)$ at $(0,0,0)$
$(\{\mathcal{C}ontroller\}, finish_1) \vee (\{\mathcal{C}ontroller\}, finish_2)$
at $\{(1,1,1),(1,2,1),(1,1,2),(1,2,2)\}$
$(\{\mathcal{C}ontroller\}, finish_2)$ at $\{(2,0,1),(2,0,2)\}$
$(\{\mathcal{C}ontroller\}, finish_1)$ at $\{(3,1,0),(3,2,0)\}$

The first requirement at $(0,0,0)$ concerns the group of the two runners which both together request to be started. The responsiveness requirements at states $(1,1,1)$, $(1,2,1)$, $(1,1,2)$ and $(1,2,2)$ all express that the controller component wants to receive a finish signal either of $\mathcal{R}unner_1$ or of $\mathcal{R}unner_2$. This illustrates the use of disjunctions to reflect external choice of inputs. The responsiveness requirements at states $(2,0,1)$ and $(2,0,2)$ express that the controller wants to receive a finish signal from $\mathcal{R}unner_2$. The requirements at states $(3,1,0)$ and $(3,2,0)$ are analogous. As discussed in Example 3, $\mathcal{T}_1$ is *not* compliant with all responsiveness requirements, but it is *weakly* compliant with all of them. □

Example 5. As an open system, we consider $\mathcal{S}\widehat{ys}_1$ depicted in Fig. 3. The difference with Sys_1 is that the runner component $\widehat{\mathcal{R}unner}_2$ may decide not to wait for the start signal of the controller and start by performing an input action *go*. This is an external action but not a communicating action, which may be called by the system's environment.[7] Thus, in contrast to Sys_1, $\mathcal{S}\widehat{ys}_1$ has an open input.

We consider again the synchronisation type (ai, ai) but, since the system is open, the synchronisation policy is not uniquely determined by the synchronisation type. Let us first choose the maximal synchronisation policy over (ai, ai). Then the state $(0,0,1)$ becomes reachable with the external *go* action. For this state, our method derives receptiveness requirement $(\{\mathcal{C}ontroller\}, start)$ at $(0,0,1)$. Clearly, the team is *not* (weakly) compliant with this receptiveness requirement at $(0,0,1)$, since the second runner would also be needed to start. So we must choose a different synchronisation policy. The solution is simple. We just omit the transition with input action *go* from state $(0,0,0)$ to state $(0,0,1)$. Then $(0,0,1)$ is no longer reachable and the new team is compliant with all receptiveness requirements. Removal of the 'bad' open input transition matches the approach of interface automata in [7], where components are considered to be *compatible* if they can work properly together in a 'helpful' environment. □



(a) $\mathcal{R}unner_1$ (b) $\widehat{\mathcal{R}unner}_2$ (c) $\mathcal{C}ontroller$

Fig. 3: Automata $\mathcal{R}unner_1$, $\widehat{\mathcal{R}unner}_2$ and $\mathcal{C}ontroller$ of $\mathcal{S}\widehat{ys}_1$

Example 6. Consider team automaton $\mathcal{T}_2$, introduced in Example 2, with synchronisation type $([1,1], ai)$. From sending multiplicity $[1,1]$ we derive, for instance, the receptiveness requirement $(\{\mathcal{R}unner_1\}, finish) \wedge (\{\mathcal{R}unner_2\}, finish)$ at $(1,2,2)$. It is easy to verify that the team automaton $\mathcal{T}_2$ is compliant with this requirement and with all other derived ones not shown here.

[7] For instance, a false start signal coming from the outside.

Let us play a bit with this example to see the importance of synchronisation types. Assume we would have chosen the sending multiplicity $[1, 2]$ instead. The corresponding synchronisation policy δ' would then allow that the two runners send simultaneously the finish signal to the controller, i.e. we get an additional transition from state $(1, 2, 2)$ to state $(2, 0, 0)$ labelled with *finish*. Then we derive responsiveness requirement $(\{\mathcal{C}ontroller\}, finish) \vee (\{\mathcal{R}unner_1, \mathcal{R}unner_2\}, start)$ at the newly reachable state $(2, 0, 0)$. Clearly this requirement is not fulfilled by the team with synchronisation policy δ' (and hence it was a good idea to choose the sending multiplicity $[1, 1]$ for the system Sys_1). □

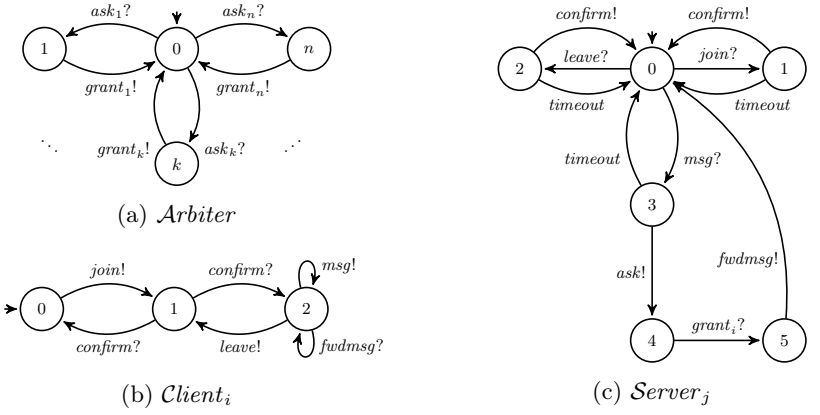


Fig. 4: Automata $\mathcal{A}rbiter$, $\mathcal{C}lient_i$ and $\mathcal{S}erver_j$, with $1 \leq i \leq m$ and $1 \leq j \leq n$

Example 7. As a more realistic example, consider a distributed chat system where buddies can interact once they register into the system. Three types of components form the distributed chat system (cf. Fig. 4): servers, clients and an arbiter. To increase the robustness of the system, not one but several servers are devoted to control both new entries into or exits from the chat, as well as to coordinate the main activity in the chat, viz. forwarding client messages to the chat. Communicating actions are partitioned into chat access actions (*join*, *leave*, *confirm*), chat messaging (*msg*, *fwdmsg*) and arbiter selection of the forwarding server (ask_i, $grant_i$). The overall messaging protocol is that clients communicate messages to the servers (action *msg*), and only one among the set of servers broadcasts the received message to the whole set of clients in the chat (through action *fwdmsg*). The selection of the server that forwards the message is done by the Arbiter. Note that in some of the states, servers contain an internal action *timeout* to allow a server to return to its initial state whenever it does not participate in the communication. Consider team automaton $\mathcal{T}_{chat}$ constructed over the aforementioned system with synchronisation type $([1, 1], [1, *])$. We will assume the system to contain n servers, m clients and one arbiter. States in this system are expressed as $n + m + 1$ tuples $(q_1, \ldots, q_n, q_{n+1}, \ldots, q_{n+m}, q_{n+m+1})$, i.e. the first n states correspond to server states, the second m states denote client states and the last state corresponds to the arbiter state.

Let us now derive two examples of receptiveness requirements (cf. Sect. 5.1):

$(\{\mathcal{C}lient_i\}, join)$ at $(q_1, \ldots, q_n, \ldots q_{n+i-1}, 0, q_{n+i+1}, \ldots, q_{n+m+1})$
$(\{\mathcal{S}erver_j\}, fwdmsg)$ at $(q_1, \ldots, q_{j-1}, 5, q_{j+1}, \ldots, q_{n+m+1})$

These requirements express receptiveness obligations fulfilled by $\mathcal{T}_{chat}$. For instance, the first requirement expresses that in the state 0 of $\mathcal{C}lient_i$, join actions should be received by at least one server. Likewise, the second requirement expresses that the message forwarded to the chat by a server will be received by the clients. In this second requirement, it is assumed that some client is still in the chat (i.e. $\exists i, n+1 \leq i \leq n+m : q_i = 2$).

An example of a responsiveness requirement is the following (cf. Sect. 5.2):

$(\{\mathcal{S}erver_j\}, join) \vee (\{\mathcal{S}erver_j\}, leave) \vee (\{\mathcal{S}erver_j\}, msg)$
at $(q_1, \ldots, q_{j-1}, 0, q_{j+1}, \ldots, q_n, \ldots q_{n+m+1})$

This responsiveness requirement at state 0 of $\mathcal{S}erver_j$ provides a choice concerning the server's functionality: it can either coordinate joining or exiting actions from a client, or messages sent in the chat. As before for the server's receptiveness requirement, it is assumed that some client is still in the chat (i.e. $\exists\, i, n+1 \leq i \leq n+m : q_i = 2$).

Another example of a responsiveness requirement is as follows (cf. Sect. 5.2):

$(\{\mathcal{A}rbiter\}, ask_1) \vee \cdots \vee (\{\mathcal{A}rbiter\}, ask_n)$ at $(q_1, \ldots, q_{n+m}, 0)$

This requirement applies when at least one server is asking for permission to forward the received message (i.e. $\exists j, 1 \leq j \leq n : q_j = 3$) and at least one client is in the chat (i.e. $\exists i, n+1 \leq i \leq n+m : q_i = 2$). It makes an obligation for some of the servers to provide some of the required inputs for the arbiter. □

5.4 Related Compatibility Notions

In the literature, compatibility notions are often considered for systems built according to a specific synchronisation type. For instance, interface automata [7] and many others, like [10, 19, 24], consider synchronous products of composable I/O-transition systems with (binary) point-to-point communication, i.e. the synchronisation type is $([1,1],[1,1])$. These papers moreover deal with the aspect of receptiveness only. We can say that a team automaton of type $([1,1],[1,1])$ over a closed system of components $\mathcal{A}_1$ and $\mathcal{A}_2$ is receptive in the sense of Def. 9 iff $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible in the sense of [7] iff they are strongly compatible in the sense of [24] iff they are receptive in the sense of [1].

For open systems, the theory of interface automata relies on the *optimistic* approach. Two components are compatible if there exists a 'helpful' environment which avoids that the system can reach a communication error. As we have seen, synchronisation types define synchronisation policies uniquely only for closed systems, while for open systems there is still a possibility to restrict the set of transitions with external, non-communicating actions. Therefore, we can find an appropriate policy to make two components receptive iff they are compatible in the sense of [7]. Weak receptiveness corresponds to weak compatibility in [24] and is also captured by unspecified receptions compatibility in [13]. We are aware

of only a few approaches that consider compatibility w.r.t. responsiveness. In [1], responsiveness is captured by deadlock-freeness and in [13] it is expressed by part of the definition of bidirectional complementarity compatibility which, however, does not support choice of inputs as we do.

5.5 Applications

The contributions of this paper enable to explore component-based modelling and composition according to a wide range of synchronisation policies, not limited to the classical synchronous product, bringing upfront the communication requirements that must be fulfilled to derive a compliant system.

We foresee many application areas where the perspective taken in this paper can play an important role to enhance the interaction and communication policies that are used. In Swarm Intelligence, for instance, agents communicate by means of sensors, actuators and connectors. Such sensors and actuators allow communication through the receiving and sending of signals. This communication often concerns a small selection of agents that changes over time, thus deviating from the synchronous product [25]. Being able to construct swarm networks that fulfil certain compatibility guarantees on alternative communication policies, like the ones considered in this paper, may represent an important step towards their satisfactory application.

Another application area is Software Engineering. In particular, the provision of compatibility theories that go beyond limited formalisms like UML statecharts composed according to the synchronous product, will rise the expressibility level, thus widening the applicability scope to cover much more real-world situations.

Also concurrent asynchronous programming languages can benefit from having a general theory of compatibility such as the one we envision in this paper. Erlang [26] is a prominent example: its asynchronous communication mode allows for a very flexible communication architecture, but if used incorrectly it may lead to invalid/suboptimal system implementations. To the best of our knowledge, current approaches follow a *post-mortem* approach to verify properties like liveness and safety of Erlang programs. Instead, correct-by-construction design might become applicable if the theories described in this paper were used in the specification of Erlang programs.

Finally, the field of Web services may also be a nice application arena for the ideas put forward in this paper. Like in some of the previous examples, we are only aware of notions of compatibility for the composition of Web services defined over the restricted synchronous product [27, 28].

6 Conclusion

We have investigated compatibility notions concerning receptiveness and responsiveness in the team automata framework. Team automata are characterised by the synchronisation policy they use to coordinate the components of a given

system. There is a huge variety of possible synchronisation policies. The synchronisation types as we introduced them here support a systematic approach to the investigation of compatibility notions related to communication. To find appropriate compatibility notions, we first analysed what kind of communication requirements can occur when components are composed. We distinguished receptiveness and responsiveness requirements and we showed how such requirements can be systematically derived depending on a synchronisation type. A team automaton is compliant with a communication requirement if (groups of) components in the team issuing requests for communication can successfully find partners to join. If this is the case for all receptiveness (responsiveness) requirements, then the team automaton is receptive (responsive, respectively).

Our approach is appropriate for both closed and open systems. A team automaton over an open system is itself a reactive component and thus gives rise to hierarchical composition. One of the next steps in our research will be to study compatibility in the context of hierarchical composition and of synchronisation policies that are not necessarily uniform but combine different synchronisation types. The latter would also concern an investigation of compatibility notions tailored to particular connectors as used, e.g., in BIP and Reo (cf. [29] for a comparison). Also the incorporation of asynchronous communication in synchronisation policies and the study of compatibility notions in this case [24, 30] is a topic for future research. Moreover, appropriate notions of equivalences and refinements for team automata and how they behave w.r.t. our receptiveness and responsiveness notions are interesting questions to consider.

Acknowledgments We thank the reviewers for their comments. J. Carmona is supported by the Spanish Ministry for Economy and Competitiveness (MINECO) and the EU (FEDER funds) under grant COMMAS (TIN2013-46181-C2-1-R).

References

1. J. Carmona and J. Cortadella. Input/Output Compatibility of Reactive Systems. In *FMCAD'02*, volume 2517 of *LNCS*, pages 360–377. Springer, 2002.
2. J. Carmona and J. Kleijn. Compatibility in a multi-component environment. *Theor. Comput. Sci.*, 484:1–15, 2013.
3. N. A. Lynch and M. R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *PODC'87*, pages 137–151. ACM, 1987.
4. N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989. `https://ir.cwi.nl/pub/18164`.
5. C. A. Ellis. Team Automata for Groupware Systems. In *GROUP'97*, pages 415–424. ACM, 1997.
6. M. H. ter Beek, C. A. Ellis, J. Kleijn, and G. Rozenberg. Synchronizations in Team Automata for Groupware Systems. *Comput. Sup. Coop. Work*, 12(1):21–69, 2003.
7. L. de Alfaro and T. A. Henzinger. Interface Automata. In *ESEC/FSE'01*, pages 109–120. ACM, 2001.
8. L. de Alfaro and T. A. Henzinger. Interface-Based Design. In *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series*, pages 83–104. Springer, 2005.

9. L. Brim, I. Cerná, P. Vareková, and B. Zimmerova. Component-Interaction Automata as a Verification-Oriented Component-Based System Specification. *ACM Softw. Eng. Notes*, 31(2), 2006.
10. K. G. Larsen, U. Nyman, and A. Wąsowski. Modal I/O Automata for Interface and Product Line Theories. In *ESOP'07*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.
11. S. S. Bauer, P. Mayer, A. Schroeder, and R. Hennicker. On Weak Modal Compatibility, Refinement, and the MIO Workbench. In *TACAS'10*, volume 6015 of *LNCS*, pages 175–189. Springer, 2010.
12. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.
13. F. Durán, M. Ouederni, and G. Salaün. A generic framework for n-protocol compatibility checking. *Sci. Comput. Program.*, 77(7-8):870–886, 2012.
14. M. H. ter Beek, J. Carmona, and J. Kleijn. Conditions for Compatibility of Components: The Case of Masters and Slaves. In *ISoLA'16*, volume 9952 of *LNCS*, pages 784–805. Springer, 2016.
15. B. Jonsson. Compositional Specification and Verification of Distributed Systems. *ACM Trans. Program. Lang. Syst.*, 16(2):259–303, 1994.
16. M. H. ter Beek and J. Kleijn. Team Automata Satisfying Compositionality. In *FME'03*, volume 2805 of *LNCS*, pages 381–400. Springer, 2003.
17. G. Gössler and J. Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55:161–183, 2005.
18. M. H. ter Beek and J. Kleijn. Modularity for teams of I/O automata. *Inf. Process. Lett.*, 95(5):487–495, 2005.
19. G. Lüttgen, W. Vogler, and S. Fendrich. Richer interface automata with optimistic and pessimistic compatibility. *Acta Inf.*, 52(4-5):305–336, 2015.
20. A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *SEFM'06*, pages 3–12. IEEE, 2006.
21. A. Arnold. *Finite Transition Systems: Semantics of Communicating Systems*. Prentice Hall, 1994.
22. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
23. G. Engels and L. Groenewegen. Towards Team-Automata-Driven Object-Oriented Collaborative Work. In *Formal and Natural Computing*, volume 2300 of *LNCS*, pages 257–276. Springer, 2002.
24. R. Hennicker, M. Bidoit, and T.-S. Dang. On Synchronous and Asynchronous Compatibility of Communicating Components. In *COORDINATION'16*, volume 9686 of *LNCS*, pages 138–156. Springer, 2016.
25. T. Isokawa, F. Peper, M. Mitsui, J.-Q. Liu, K. Morita, H. Umeo, N. Kamiura, and N. Matsui. Computing by Swarm Networks. In *ACRI'08*, volume 5191 of *LNCS*, pages 50–59. Springer, 2008.
26. J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.
27. M. H. ter Beek, A. Bucchiarone, and S. Gnesi. Web Service Composition Approaches: From Industrial Standards to Formal Methods. In *ICIW'07*. IEEE, 2007.
28. Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu. Web services composition: A decade's overview. *Inf. Sci.*, 280:218–238, 2014.
29. K. Dokter, S.-S. T. Q. Jongmans, F. Arbab, and S. Bliudze. Combine and conquer: Relating BIP and Reo. *J. Log. Algebr. Meth. Program.*, 86(1), 2017.
30. J. L. Fiadeiro and A. Lopes. An interface theory for service-oriented design. *Theor. Comput. Sci.*, 503:1–30, 2013.