Conditions for Compatibility of Components

The case of masters and slaves

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

[1] ISTI–CNR, Pisa, Italy
[2] Universitat Politècnica de Catalunya, Barcelona, Spain
[3] LIACS, Leiden University, The Netherlands

Abstract. We consider systems composed of reactive components that collaborate through synchronised execution of common actions. These multi-component systems are formally represented as team automata, a model that allows a wide spectrum of synchronisation policies to combine components into higher-level systems. We investigate the correct-by-construction engineering of such systems of systems from the point of view of correct communications between the components (no message loss or deadlocks due to indefinite waiting). This leads to a proposal for a generic definition of compatibility of components relative to the adopted synchronisation policy. This definition appears to be particularly appropriate for so-called master-slave synchronisations by which input actions (for 'slaves') are driven by output actions (from 'masters').

1 Introduction

In an increasingly connected world in which digital communication outnumbers all other forms of communication, it is important to understand the complex underlying interconnections in the numerous systems of systems governing our daily life. In fact, modern systems are often no longer monolithic, but large-scale concurrent and distributed embedded systems whose components are again complex systems and which as a whole offer more functionality and performance than the sum of their component systems [35]. This requires a deep understanding of various communication and interaction policies (e.g. client-server, peer-to-peer, and master-slave) used in such multi-component systems and the risk of failures they entail (e.g. message loss and deadlocks can have severe repercussions on reliability, safety and security). One way to approach this challenge is to lift successful design methodologies and analysis tools from single systems engineering to systems of systems engineering. In a component-based bottom-up manner, this can be addressed through correctness by construction, where correctness is concerned with not only formal verification but also issues like reliability, resilience, safety, security and even sustainability.

Correctness by construction sees the development of (software) systems (of systems) as a true form of Engineering, with a capital 'E'. It advocates a step-wise refinement process from requirements to specification to code, ideally by design tools that automatically generate error-free (software) implementations from rigorous and unambiguous specifications of requirements [22,27,28,36,41]. To establish that components

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-47166-2_55

within a system or a system and its environment always may interact correctly, a concept of compatibility can be useful. Compatibility represents an aspect of successful behaviour and as such forms a necessary ingredient for the correctness of a distributed, modular system design [25]. Compatibility failures detected in a 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 techniques supporting the design, analysis and verification of systems of systems.

We are interested in studying fundamental notions for the component-based development of correct-by-construction multi-component systems. We represent multi-component systems by team automata [3–6]. Team automata are useful to specify intended behaviour of reactive systems. Their basic building blocks are component automata that can interact with each other via shared (external) actions; internal actions are never shared. External actions are input or output to the components they belong to. Components can be added in different phases of construction, allowing for hierarchically composed systems (of systems). Team automata share the distinction of output (active), input (passive) and internal (private) actions with I/O automata [38, 39], Interface automata [1, 19–21] and Component-Interaction Automata [10], but an important difference is that team automata impose less a priori restrictions on the role of the actions and the permitted type of interactions between the components. This particularly suits systems of systems that in practice are often composed of different models of computation that interact according to a variety of synchronisation policies.

In [16], the binary notion of I/O compatibility from [11, 12] was lifted to team automata consisting of multiple reactive component automata. The aim of the ideas developed in [12] was to provide a formal framework for the synthesis of asynchronous circuits and embedded systems. The approach was restricted to two components and a closed environment, i.e. all input (output) actions of one component are output (input) actions of the other component. A characterisation was given for compatibility of two components that should engage in a dialogue free from message loss and deadlocks. Message loss occurs when one component sends a message which cannot be received by the other component as input, whereas deadlock occurs when a component is kept waiting indefinitely for a message that never arrives. Team automata proved to form a suitable formal framework for lifting the concept of compatibility to a multi-component setting in [16], in which communication and interaction may take place between more than two components at the same time (e.g. broadcasting).

In [16], emphasis was on interactions based on mandatory synchronised execution of common actions (leading to what is a.k.a. the synchronous product of the component automata). In this paper, we present an initial exploration into lifting the conditions for compatibility defined in [16] to team automata that adhere to other synchronisation strategies. We first propose a general notion of compatibility defined with respect to a given synchronisation policy. Subsequently, we focus on how to handle team automata that interact according to master-slave cooperations. In such cooperations, input (for 'slaves') is driven by output (from 'masters') under different assumptions ranging from slaves that cannot proceed on their own to masters that should always be followed by slaves. This models a well-known method of communication in which specific, more authoritative partners unidirectionally control or trigger other partners to

synchronise with them. Examples include peripherals connected to a bus in a computer, master databases from which data is replicated to (synchronised) slave databases and master (precision) clocks that provide timing signals to synchronise slave clocks. The producer-consumer design pattern known from concurrency theory and programming (e.g. threading) can be seen as a simplified case of master-slave communication, where a buffer is usually used to avoid message loss.

The main contribution of this paper is thus a proposal: a generalisation of the conditions for compatibility of components defined in [16] to the context of arbitrary sets of synchronisations. After delineating some of the difficulties involved with the proposed definition, we instantiate compatibility for master-slave policies of synchronisation and illustrate how this allows to guarantee absence of deadlocks and message loss for master-slave types of team automata to which the results from [16] cannot be applied. In the future, we plan to address follow-up questions concerning these types of systems, like "how is compatibility affected when slaves are added?" and "in what way does compatibility depend on (the type of) cooperation among slaves?". Furthermore, it remains to investigate the applicability of our proposed definition to team automata composed according to still other synchronisation policies.

Outline After introducing the team automata modelling framework in Sect. 2, we discuss and illustrate in Sect. 3 two specific synchronisation policies. Section 4 contains our main contribution: we propose a generalisation of the notion of compatibility in a multi-component environment as defined in [16] from synchronous product to arbitrary synchronisation policies. After an application in the context of master-slave synchronisations, we provide some initial observations for a restricted class of so-called master-slave systems in Sect. 5. We conclude with a list of possible applications of our approach in Sect. 6, followed by a discussion of related and future work.

2 Component and Team Automata

Notation We use $\prod_{i=1}^{n} V_i$ to denote the Cartesian product of sets $V_1, \ldots, V_n$. If $v = (v_1, \ldots, v_n) \in \prod_{i=1}^{n} V_i$ and $i \in \{1, \ldots, n\}$, then the i-th entry of v is obtained by applying the projection function $proj_i : \prod_{i=1}^{n} V_i \to V_i$ defined by $proj_i(v_1, \ldots, v_n) = v_i$.

Component automata Team automata are systems composed of reactive component automata that can interact through synchronised executions of shared actions. Each such component automaton is a *labelled transition system* (LTS) in which input, output and internal actions are explicitly distinguished.

Definition 1. *A* (reactive) component automaton *is an LTS* $\mathcal{A} = (P, \Gamma, \gamma, J)$*, with set* P *of* states*; set* Γ *of* actions*, such that* $P \cap \Gamma = \emptyset$ *and* Γ *is the union of three pairwise disjoint sets* Γ_{inp}*,* Γ_{out} *and* Γ_{int} *of* input*,* output*, and* internal *actions, respectively;* $\gamma \subseteq P \times \Gamma \times P$ *is its set of (labelled)* transitions*; and* $J \subseteq P$ *its set of* initial states. □

A component automaton (P, Γ, γ, J), with input actions Γ_{inp}, output actions Γ_{out} and internal actions Γ_{int} can also be specified as $(P, (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int}), \gamma, J)$. The actions $\Gamma \setminus \Gamma_{int} = \Gamma_{out} \cup \Gamma_{inp}$ are *external*. For an action $a \in \Gamma$, we define the set of *a-transitions* as $\gamma_a = \gamma \cap (P \times \{a\} \times P)$. Especially in figures, we may append input and output actions with ? and !, respectively, to indicate their roles (cf. Fig. 1).

The (dynamic) behaviour of a component automaton is determined by the execution of actions enabled at the current state. We say that a is *enabled* in $\mathcal{A}$ at state $p \in P$, denoted by $a\ en_{\mathcal{A}}\ p$, if there exists $p' \in P$ such that $(p, a, p') \in \gamma$. The sequential *computations* of $\mathcal{A}$, denoted by $C_{\mathcal{A}}$, are now defined as those finite sequences $p_0 a_1 p_1 a_2 \cdots p_k$ and infinite sequences $p_0 a_1 p_1 a_2 \cdots$ such that $p_0 \in J$ and $(p_{i-1}, a_i, p_i) \in \gamma$ for all $i \in \{1, \ldots, k\}$ and all $i \geq 1$, respectively. A state $p \in P$ is said to be *reachable* if there exists a finite computation $p_0 a_1 p_1 a_2 \cdots p_j \in C_{\mathcal{A}}$ for some $j \geq 0$ such that $p = p_j$.

Team automata The components forming a team automaton interact by synchronising on common actions. Their internal actions however are not meant to be externally observable and are thus unavailable for synchronisation and cannot be shared. This leads to the concept of composability.

Let $\mathcal{S} = \{\mathcal{A}_i \mid 1 \leq i \leq n\}$ be a set of component automata specified, for each $i \in \{1, \ldots, n\}$, as $\mathcal{A}_i = (Q_i, (\Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}), \delta_i, I_i)$ with $\Sigma_i = \Sigma_{i,inp} \cup \Sigma_{i,out} \cup \Sigma_{i,int}$. Then $\mathcal{S}$ is a *composable system* if $\Sigma_{i,int} \cap \bigcup_{j=1, j\neq i}^{n} \Sigma_j = \emptyset$ for all $i \in \{1, \ldots, n\}$. Note that every subset of a composable system is again a composable system.

For the remainder of this paper, we let $\mathcal{S}$ as just specified, be an arbitrary but fixed, composable system. We refer to $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$ as its set of actions and to $Q = \prod_{i=1}^{n} Q_i$ as its state space. The team automata we consider are defined over a composable system $\mathcal{S}$ as above and have set of actions Σ and set of states Q. Their transitions are *synchronisations* involving transitions of the automata from $\mathcal{S}$.

Synchronisations in a composable system are global transitions that combine one or more (local) transitions of different component automata; these local transitions are all labelled with the same action name. Intuitively, each component automaton that participates through a local transition in such a synchronisation changes its state accordingly. The local states of automata not taking part in the synchronisation are not affected.

Definition 2. *A transition $(q, a, q') \in Q \times \Sigma \times Q$ is a* synchronisation *on a (in $\mathcal{S}$) if $(proj_i(q), a, proj_i(q')) \in \delta_i$, for some $i \in \{1, \ldots, n\}$; moreover for all $i \in \{1, \ldots, n\}$, 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 synchronisations on a in $\mathcal{S}$, while $\Delta(\mathcal{S}) = \bigcup_{a\in\Sigma} \Delta_a(\mathcal{S})$ is the set of all synchronisations in $\mathcal{S}$. □

Note that the composability of $\mathcal{S}$ implies that in synchronisations on internal actions always exactly one component automaton is involved.

Team automata over a composable system are determined by their (global) transitions, i.e. a choice from the set of all synchronisations in that system under the additional condition that *all* transitions labelled with internal actions are included (combining automata into teams does not affect their ability to execute internal actions). The actions of a team automaton comprise the actions of its components. Again we distinguish between input, output, and internal actions. This division is inherited from the original roles of the actions in the component automata and is the same for all team automata over a given composable system. The internal actions of any team automaton over a composable system are the internal actions of the individual components. For the external actions, the idea is that component automata have control over their output actions whereas input actions are passive (driven by the environment). As a consequence, actions that appear as an output action in one or more of the components will be output

of the team (even when they are input to some other components). Input actions that do not appear as output are input actions of the team.

Recall that $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$ is the set of actions of $\mathcal{S}$. Then $\Sigma_{int} = \bigcup_{i=1}^{n} \Sigma_{i,int}$ is its set of internal actions, $\Sigma_{out} = \bigcup_{i=1}^{n} \Sigma_{i,out}$ its set of output actions and $\Sigma_{inp} = (\bigcup_{i=1}^{n} \Sigma_{i,inp}) \setminus \Sigma_{out}$ its set of input actions. All team automata over $\mathcal{S}$ will have Σ as their set of actions, with Σ_{int} as internal, Σ_{out} as output and Σ_{inp} as input actions. Moreover, $I = \prod_{i=1}^{n} I_i$ is the set of initial states of every team automaton over $\mathcal{S}$. Consequently, it is the choice of a *synchronisation policy* over $\mathcal{S}$ (i.e. a subset $\delta \subseteq \Delta(\mathcal{S})$ with $\delta_a = \Delta_a(\mathcal{S})$ for all $a \in \Sigma_{int}$) that defines a specific team automaton.

Definition 3. *The* team automaton *over $\mathcal{S}$ defined by the synchronisation policy δ over $\mathcal{S}$ is the reactive component automaton $\mathcal{T} = (Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, I)$.* □

Since every team automaton is a reactive component automaton, they can be used to construct hierarchical systems of systems.

Subteams Given a team automaton over a composable system, one can distinguish subteams determined by a selection of component automata from the system.

Let $\mathcal{T}$ be a team automaton over $\mathcal{S}$ and let $\delta \subseteq \Delta(\mathcal{S})$ be its set of transitions. Let $J \subseteq \{1, \ldots, n\}$ be such that $J \neq \emptyset$. The *subteam $SUB_J(\mathcal{S}, \delta)$ of $\mathcal{T}$ determined by J* is the automaton specified as $(\prod_{j\in J} Q_j, (\Sigma_{J,inp}, \Sigma_{J,out}, \Sigma_{J,int}), \delta_J, \prod_{j\in J} I_j)$. Here $\Sigma_{J,int} = \bigcup_{j\in J} \Sigma_{j,int}$, $\Sigma_{J,out} = \bigcup_{j\in J} \Sigma_{j,out}$ and $\Sigma_{J,inp} = (\bigcup_{j\in J} \Sigma_{j,inp}) \setminus \Sigma_{J,out}$. It may happen that an output action of $\mathcal{T}$ is an input action in a subteam, namely when it does not have an output role in any of the component automata forming the subteam. Finally, $\delta_J = \{(q, a, q') \in \delta \mid (proj_J(q), a, proj_J(q')) \in \Delta(\{\mathcal{A}_j \mid j \in J\})\}$. Hence the transitions of the subteam are restrictions of those transitions of $\mathcal{T}$ in which at least one of the components from $\{\mathcal{A}_j \mid j \in J\}$ is actively involved. It follows that $SUB_J(\mathcal{S}, \delta)$ is the team automaton over $\{\mathcal{A}_j \mid j \in J\}$ defined by the synchronisation policy δ_J.

Input and output domains The domain of an action appearing in a composable system is determined by the components in which it appears as an action. If it is an external action it may be an input action for some components and an output action for others. Thus all external actions of a composable system have a non-empty input domain or a non-empty output domain or both. A synchronisation on an action that involves components in which that action is an input action *and* components in which it is an output action, models a *communication* between the input and output subteams associated with that action.

Let $a \in \Sigma$ be an action of $\mathcal{S}$. Then $\mathrm{dom}_a(\mathcal{S}) = \{i \mid a \in \Sigma_i\}$ is the *domain* of a in $\mathcal{S}$; $\mathrm{dom}_{a,inp}(\mathcal{S}) = \{i \mid a \in \Sigma_{i,inp}\}$ is its *input domain*; and $\mathrm{dom}_{a,out}(\mathcal{S}) = \{i \mid a \in \Sigma_{i,out}\}$ is its *output domain*. Action a is *communicating* (in $\mathcal{S}$) if both its output domain and its input domain are not empty. Hence we define the communicating actions of $\mathcal{S}$ as $\Sigma_{com} = \bigcup_{i=1}^{n} \Sigma_{i,inp} \cap \bigcup_{i=1}^{n} \Sigma_{i,out}$.

For each external action $a \in \Sigma$, we write $\mathcal{S}_{a,inp} = \{\mathcal{A}_i \mid i \in \mathrm{dom}_{a,inp}(\mathcal{S})\}$ and $\mathcal{S}_{a,out} = \{\mathcal{A}_i \mid i \in \mathrm{dom}_{a,out}(\mathcal{S})\}$ to denote the composable subsystems of $\mathcal{S}$ comprising the *input components* and *output components of a in $\mathcal{S}$*, respectively.

Let $\delta \subseteq \Delta(\mathcal{S})$ be the set of transitions of a team automaton $\mathcal{T}$ over $\mathcal{S}$. Let a be an external action of $\mathcal{S}$. If the output domain $\mathrm{dom}_{a,out}(\mathcal{S})$ of a is not empty, then $\mathrm{SUB}_{\mathrm{dom}_{a,out}(\mathcal{S})}(\mathcal{S}, \delta)$ is the *output subteam of a in $\mathcal{T}$*; it is the subteam of $\mathcal{T}$ determined

by the output domain of a and thus a team automaton over the output components of a; it will be usually be denoted by $\mathrm{SUB}_{a,out}(\mathcal{S},\delta)$. Similarly, if $a \in \Sigma$ has a non-empty input domain $\mathrm{dom}_{a,inp}(\mathcal{S})$, then the *input subteam* $SUB_{dom_{a,inp}(\mathcal{S})}(\mathcal{S},\delta)$ *of* a *in* $\mathcal{T}$ is denoted by $\mathrm{SUB}_{a,inp}(\mathcal{S},\delta)$; it is a team automaton over the input components of a. If no confusion arises, we may omit referencing $\mathcal{S}$ and δ, and write $SUB_{a,out}$ and $SUB_{a,inp}$, respectively.

3 Specific Synchronisation Policies

Team automata are defined through their synchronisation policies. For all (global, product) states and each (external) action enabled at a corresponding local state of at least one of the components, it has to be decided which synchronisations involving that action are to be included as a transition of the team. It will however seldom be the case that this decision is made explicitly for every candidate synchronisation separately. Rather, the designer of the system has a certain idea about the interaction between components when combining them into one system. We will discuss here two such globally defined synchronisation policies, after which we will introduce the notion of *state-sharing*. This is a relevant notion here, since as demonstrated in [16], the concept of compatibility can be transferred from synchronous products to arbitrary team automata provided that they are *not* state-sharing.

Synchronous product A natural and frequently used method for combining components into a team automaton, or composing automata in general, is to always and only include transitions modelling the execution of an action in which *all* components participate that have that action in common.

Let $a \in \Sigma$ be an action of $\mathcal{S}$. Then we define

$$\chi_a^{\mathcal{S}} = \{\, (q,a,q') \in \Delta(\mathcal{S}) \mid \forall\, i \in \{1,\ldots,n\} :\ a \in \Sigma_i \Rightarrow (proj_i(q), a, proj_i(q')) \in \delta_i \,\}$$

as the set of all *product synchronisations* on a (in $\mathcal{S}$). We let $\chi^{\mathcal{S}} = \bigcup_{a\in\Sigma} \chi_a^{\mathcal{S}}$.
Note that $\chi^{\mathcal{S}}$ is a proper synchronisation policy. In particular, $\chi_a^{\mathcal{S}} = \Delta_a(\mathcal{S})$ for every internal action a. When $\mathcal{S}$ is understood, we may omit the superscript to write χ and χ_a.

Definition 4. *The* synchronous product (automaton) $\mathcal{X}(\mathcal{S})$ *over* $\mathcal{S}$ *is the team automaton over* $\mathcal{S}$ *with* $\chi^{\mathcal{S}}$ *as its set of transitions.* □

Master-slave synchronisations Another natural policy, relevant to automata models that make a distinction between input and output actions, was introduced in [3, 4]. It focusses on communication and thus relates input and output domains of an external action. First, we formulate an approach based on relations between actions rather than between full components. In Section 5, this will be restricted to a simpler set-up at the level of the components by assuming that they are either output components (without input actions) or input components (without output actions).

When input actions are seen as passive (under control of the environment) and output actions as active (under the local control of the component), the designer could opt for a master-slave paradigm underlying the team's definition. Intuitively, master-slave cooperation requires that input actions ('slaves') are driven by output actions ('masters'). This means that in a master-slave synchronisation on an external action a, always

an output component of a participates. In other words, input actions ('slaves') never proceed on their own. This does not exclude the possibility that a is executed by one or more of its output components without simultaneous execution of a by an input component. Thus the policy could be modified by the additional requirement that, if a is communicating, then there is always an input component of a that also participates (a master is always accompanied by one or more slaves), or—in a weaker form—masters are accompanied by slaves whenever possible.

Definition 5. *Let $a \in \Sigma_{com}$, $J = dom_{a,out}(\mathcal{S})$ and $K = dom_{a,inp}(\mathcal{S})$.*

1. *The set of all* master-slave synchronisations *on a in $\mathcal{S}$ is defined as $\mathcal{MS}_a^{\mathcal{S}} = \{(q,a,q') \in \Delta(\mathcal{S}) \mid (proj_J(q), a, proj_J(q')) \in \Delta(\mathcal{S}_{a,out})\}$;*
2. *The set of all* strong master-slave synchronisations *on a in $\mathcal{S}$ is defined as $s\mathcal{MS}_a^{\mathcal{S}} = \mathcal{MS}_a^{\mathcal{S}} \cap \{(q,a,q') \in \Delta(\mathcal{S}) \mid (proj_K(q), a, proj_K(q')) \in \Delta(\mathcal{S}_{a,inp})\}$;*
3. *The set of all* weak master-slave synchronisations *on a in $\mathcal{S}$ is defined as $w\mathcal{MS}_a^{\mathcal{S}} = \mathcal{MS}_a^{\mathcal{S}} \cap \{(q,a,q') \in \Delta(\mathcal{S}) \mid (proj_K(q), a, proj_K(q')) \in \Delta(\mathcal{S}_{a,inp})$ if there exists a $k \in K$ such that $a\, en_{\mathcal{A}_k}\, proj_k(q))\}$.* □

In addition, we stipulate that if an action a is not communicating, then by default all synchronisations on a are master-slave, strong master-slave and weak master-slave. Thus $\mathcal{MS}_a^{\mathcal{S}} = s\mathcal{MS}_a^{\mathcal{S}} = w\mathcal{MS}_a^{\mathcal{S}} = \Delta_a(\mathcal{S})$ for all non-communicating actions a of $\mathcal{S}$. Note that strong master-slave synchronisations are also weak master-slave synchronisations, which again by definition, are also master-slave.

Let $\mathcal{MS}^{\mathcal{S}} = \bigcup_{a\in\Sigma} \mathcal{MS}_a^{\mathcal{S}}$, $s\mathcal{MS}^{\mathcal{S}} = \bigcup_{a\in\Sigma} s\mathcal{MS}_a^{\mathcal{S}}$ and $w\mathcal{MS}^{\mathcal{S}} = \bigcup_{a\in\Sigma} w\mathcal{MS}_a^{\mathcal{S}}$. The superscript $\mathcal{S}$ may be omitted if this does not lead to confusion. Similar as for synchronous product, we can now define a unique automaton over $\mathcal{S}$ by including all and only those transitions that satisfy the requirements.

Definition 6. *The* (strong, weak) master-slave team automaton *or ($s\mathcal{MS}$*-team*, $w\mathcal{MS}$*-team*) $\mathcal{MS}$*-team automaton over $\mathcal{S}$ *is the team automaton over $\mathcal{S}$ with ($s\mathcal{MS}^{\mathcal{S}}$, $w\mathcal{MS}^{\mathcal{S}}$) $\mathcal{MS}^{\mathcal{S}}$, respectively, as its set of transitions.* □

Note that all synchronisations in to the synchronous product are strong master-slave: $\chi \subseteq s\mathcal{MS}$ always holds.

State-sharing Given a composable system, the designer chooses the team's transitions and determines which components participate in the execution of an action with which local transitions. Hence, it may happen that at some global state, an action can be executed in a certain way, while a similar synchronisation involving the same local transitions is not possible at another global state, even though it concerns the same local states for all components participating in that synchronisation. This phenomenon, by which the local states of components not actively involved in a synchronisation determine whether or not it may underlie a global transition, was coined *state-sharing* in [24] and formalised in [3].

Definition 7. *A team automaton $\mathcal{T}$, specified as in Definition 3, is* state-sharing *if there exist a transition $(p,a,p') \in \delta$, and a state $q \in Q$ such that $proj_i(q) = proj_i(p)$ for all i such that $(proj_i(p), a, proj_i(p')) \in \delta_i$, while there is no state $q' \in Q$ such that $(q,a,q') \in \delta$ with $proj_i(q') = proj_i(p')$ for all i such that $(proj_i(p), a, proj_i(p')) \in \delta_i$, and $proj_i(q') = proj_i(q)$ for all other i.* □

Thus in a state-sharing team automaton, there is a situation in which the possibility to synchronise on a common action by certain components depends also on the local state of one or more components not actually involved in the synchronisation. Hence, when a team automaton is not state-sharing (or *non-state-sharing*), then the possibility of executing a common action depends only on the local states of components that take part in the synchronisation. As already noted in [16], synchronous product automata are always non-state-sharing. Moreover, every (strong) master-slave team automaton is non-state-sharing. This follows from the fact that the (strong) master-slave requirement refers to participation of certain components and as a synchronisation policy includes all synchronisations that satisfy that requirement and thus does not exclude any synchronisation because non-participating components are not in a particular local state. As the next example shows, there exist however weak master-slave team automata that are state-sharing.

Example 1. Consider the component automata $\mathcal{A}_1$ and $\mathcal{A}_2$ depicted in Fig. 1.

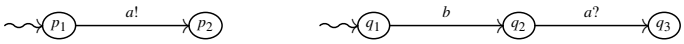


Fig. 1. Two reactive component automata: $\mathcal{A}_1$ (left) and $\mathcal{A}_2$ (right)

Figure 2 depicts the wMS-team automaton $\mathcal{T}^w$ over $\{\mathcal{A}_1, \mathcal{A}_2\}$, in which $a?!$ denotes a synchronisation of action a in its input role $a?$ and its output role $a!$.

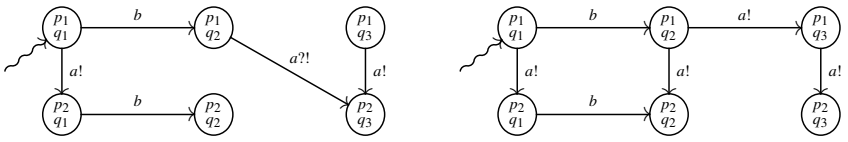


Fig. 2. Two team automata over $\{\mathcal{A}_1, \mathcal{A}_2\}$; on the left the wMS-team automaton $\mathcal{T}^w$

We see that on the one hand $(\binom{p_1}{q_1}, a, \binom{p_2}{q_1}), (\binom{p_1}{q_3}, a, \binom{p_2}{q_3}) \in wMS_a$, while on the other hand, $(\binom{p_1}{q_2}, a, \binom{p_2}{q_2}) \notin wMS_a$. This implies that $\mathcal{T}^w$ is state-sharing: component $\mathcal{A}_1$ can only execute a by itself if $\mathcal{A}_2$ is *not* in state q_2.

Note that $(\binom{p_1}{q_1}, a, \binom{p_2}{q_1})$, $(\binom{p_1}{q_2}, a, \binom{p_2}{q_2})$ and $(\binom{p_1}{q_3}, a, \binom{p_2}{q_3})$ are all master-slave, but none of them is a strong master-slave synchronisation on a in $\{\mathcal{A}_1, \mathcal{A}_2\}$. However, $(\binom{p_1}{q_2}, a, \binom{p_2}{q_3})$ is both master-slave and strong-master slave. □

As shown in [16], any team automaton $\mathcal{T}$ (over $\mathcal{S}$) can be converted into a synchronous product automaton $\chi(\mathcal{S}')$ over the new composable system $\mathcal{S}'$ derived from $\mathcal{S}$ by using synchronisations as action names. The behaviour of $\mathcal{T}$ (its sequential computations) can be obtained by a simple mapping from the behaviour of $\chi(\mathcal{S}')$. In general $\chi(\mathcal{S}')$ may exhibit too much behaviour. If, however, $\mathcal{T}$ is non-state-sharing, then each of the computations of $\chi(\mathcal{S}')$ corresponds to a computation of $\mathcal{T}$ (Theorem 13 in [16]).

4 Conditions for Compatibility

We are now ready to continue the investigation of conditions that guarantee a composable system of reactive component automata to be compatible with respect to a certain synchronisation policy, or in other words, whenever the component automata are composed according to this specific policy, they form a team automaton free from message loss and deadlocks. This approach to compatibility was originally considered in [12] for two reactive components in which all external actions are communicating and which are composed as a synchronous product. In [16], the concept was extended to systems with an arbitrary (finite) number of components and not necessarily 'complete', meaning that some of the external actions can be non-communicating. This reflects the idea that a system may be further extended by additional components (or teams). In [16], the synchronisation policy considered is the synchronous product. However, once it was demonstrated how non-state-sharing team automata policies can be encoded as synchronous products it was discussed how this might lead to a concept of compatibility for the original synchronisation policy, though no definite results could be claimed yet. Before giving the formal definition of compatibility from [16], we recall some notions.

Basic notions Let $\mathcal{A}$ be a component automaton specified as in Definition 1.

First we introduce a notion to reflect that components may have a (planned) option to halt: A state $p \in P$ is said to be *terminating* if no action is enabled at p and $\mathcal{A}$ is *terminating* if it has at least one reachable terminating state.

The next notion describes the—in general undesirable—situation that a component automaton may exhibit interminable internal behaviour, thus avoiding 'visible' behaviour and in particular communication: $\mathcal{A}$ exhibits a *livelock* if there exists an infinite computation $p_0 a_1 p_1 a_2 \cdots \in C_{\mathcal{A}}$, such that all actions $a_1, a_2, \ldots$ are internal actions. A livelock-free component will always ultimately execute an external action or terminate.

Finally, given a composable system $\mathcal{S}$ as before, we introduce a notion to describe when an external action can be executed by its (input, output) domain.

An external action $a \in \Sigma$ is *input-domain (output-domain) enabled* at a state $q \in Q$ if $a\ en_{\mathcal{A}_i}\ proj_i(q)$ for all $i \in \mathrm{dom}_{a,inp}(\mathcal{S})$ (for all $i \in \mathrm{dom}_{a,out}(\mathcal{S})$, respectively). An external action is *domain enabled* if it is both input-domain and output-domain enabled. Note that input (output) actions which are not communicating in $\mathcal{S}$ are output-domain (input-domain, respectively) enabled at all states in Q, because they have an empty output (input) domain. It should also be noted here that for an external action with non-empty input-domain, input-domain enabledness (and similarly output-domain enabledness if it has a non-empty output-domain) coincides with enabledness in its input subteam (output subteam) in the synchronous product automaton.

4.1 Compatibility and Synchronous Product

The definition of compatibility proposed in [16] applies to component automata that together form a composable system and assumes that the team will be defined by the synchronous product policy.

Definition 8. *$\mathcal{R} \subseteq \Pi_{i=1}^{n} Q_i$ is a* compatibility relation for *$\mathcal{S}$ if $\Pi_{i=1}^{n} I_i \subseteq \mathcal{R}$ and for all $p \in \mathcal{R}$ the following conditions are satisfied.*

Non-communicating progress *For all* $a \in \bigcup_{i=1}^{n} \Sigma_i \setminus \Sigma_{com}$*: if* $a\, en_{\mathcal{A}_i}\, proj_i(p)$ *for all* $i \in dom_a(\mathcal{S})$*, then* $p' \in \mathcal{R}$*, whenever* $(p, a, p') \in \chi_a^{\mathcal{S}}$.

Receptiveness *For all* $a \in \Sigma_{com}$*: if* a *is output-domain enabled at* p*, then* a *is input-domain enabled at* p*, and* $p' \in \mathcal{R}$ *whenever* $(p, a, p') \in \chi_a^{\mathcal{S}}$.

Deadlock-freeness *If some action* $a \in \Sigma_{\text{com}}$ *is input-domain enabled at* p*, then there are* $b \in \bigcup_{i=1}^{n} \Sigma_i$ *and* $p' \in \Pi_{i=1}^{n} Q_i$ *such that* $(p, b, p') \in \chi^{\mathcal{S}}$.

$\mathcal{S}$ *is said to be* compatible *if each of its component automata* $\mathcal{A}_i$ *is livelock-free and there exists a compatibility relation for* $\mathcal{S}$. □

Thus compatibility is phrased in terms of a relation over the local states (in the binary case [12] similar to a bisimulation [40]). This relation is a subset of all possible states of any team over $\mathcal{S}$ and always includes all initial states. As long as no communications take place according to the synchronisation policy, the states thus reached will all be in the relation (Non-communicating progress). Whenever the output subteam of a communicating action is enabled, its input subteam is ready for synchronisation on that action and the resulting state will still be in the relation (Receptiveness). If the input subteam of a communicating action is enabled, there is always a possibility for the system to proceed (Deadlock-freeness); as proved in [16] the new states will be again in the relation if the conditions of Non-communicating progress and Receptiveness are also fulfilled. Moreover, the absence of livelocks guarantees that the (synchronous product) team automaton proceeds visibly as long as there are pending input requests.

As said, in [16] it was shown that compatibility concepts can be transferred from synchronous product to non-state-sharing team automata. More precisely, violations of any of the three requirements for a compatibility relation in the obtained synchronous product were pre-existent in the original team automaton (Definition 6 in [16]).

To ensure deadlock-freedom, [8] uses a method based on system invariants that relate states of components to approximate global states and the method checks that the overapproximation matches an equivalent of deadlock-freeness. A difference with the set-up here is however that our composition is not formulated in terms of interactions that are added to composite systems.

4.2 Compatibility and Master-Slave Policies

Now we can formulate, as a main contribution of this paper, a proposal for a definition of compatibility between components without assumptions regarding the actual synchronisations that may take place. To do so, we lift the concept of compatibility relations to compatibility with respect to a set of team transitions.

Definition 9. *Let* $\delta \subseteq \Delta(\mathcal{S})$ *be a synchronisation policy over* $\mathcal{S}$*. Then* $\mathcal{R} \subseteq \Pi_{i=1}^{n} Q_i$ *is a* compatibility relation with respect to δ for $\mathcal{S}$ *if* $\Pi_{i=1}^{n} I_i \subseteq \mathcal{R}$ *and for all* $p \in \mathcal{R}$ *the following conditions are satisfied.*

Non-communicating progress *For all* $a \in \bigcup_{i=1}^{n} \Sigma_i \setminus \Sigma_{com}$*: if* $(p, a, p') \in \delta$*, then* $p' \in \mathcal{R}$.

Receptiveness *For all* $a \in \Sigma_{com}$*: if* $a\, en_{SUB_{a,out}}\, proj_J(p)$ *with* $J = dom_{a,out}(\mathcal{S})$*, then* $a\, en_{SUB_{a,inp}}\, proj_K(p)$ *with* $K = dom_{a,inp}(\mathcal{S})$*, and* $p' \in \mathcal{R}$ *whenever* $(p, a, p') \in \delta$.

Deadlock-freeness *For all $a \in \Sigma_{com}$: if $a\ en_{SUB_{a,inp}}\ proj_K(p)$ with $K = dom_{a,inp}(\mathcal{S})$, then there are $b \in \bigcup_{i=1}^{n} \Sigma_i$ and $p' \in \Pi_{i=1}^{n} Q_i$ such that $(p, b, p') \in \delta$.*

$\mathcal{S}$ is said to be compatible with respect to *δ if each of its component automata $\mathcal{A}_i$ is livelock-free and there exists a compatibility relation with respect to δ for $\mathcal{S}$.* □

Note that a compatibility relation with respect to an arbitrary δ relates to enabledness of actions in output and input subteams of the team automaton defined through δ, rather than enabledness of that action in simply the input or output domains as we did for χ. However, in case of the synchronous product, enabledness in all output (input) components is the same as enabledness of the subteam (at the same state). In fact, the above definition generalises the concept of compatibility defined in [16]. Thus, more precisely, $\mathcal{R}$ is a compatibility relation with respect to the synchronous product χ according to the above Definition 9, if and only if it is a compatibility relation as defined in Definition 8 (which in its turn is Definition 4 from [16]). In order to see this, one observes first that, if $\delta = \chi$, then $a\ en_{\mathcal{A}_i}\ proj_i(p)$ for all $i \in \mathrm{dom}_a(\mathcal{S})$ and $(p, a, p') \in \chi_a^{\mathcal{S}}$ if and only if $(p, a, p') \in \delta$. Secondly, output-domain (input-domain) enabledness of an action at a global state coincides with enabledness of that action in its output (input, respectively) subteam of the synchronous product automaton at the corresponding (projected) state of the subteam.

The reason for requiring enabledness in subteams rather than enabledness in individual components (forming the output or input subteam) is the generalisation to arbitrary synchronisation policies. There is in general no reason to assume that all components that share an action have to participate in all synchronisations on that action. Hence we view subteams as 'black boxes' and treat the synchronisations that take place within them as given.

Nevertheless, there is still an implicit assumption even in this generalised definition regarding the collaboration between output and input subteams. According to the requirement of Receptiveness in Definition 9, whenever the output subteam of an action is ready to execute that action, its input subteam should be ready to participate. There is however no guarantee that the team will actually synchronise on the action from the given global state (cf. Example 2 below). It may well be the case that δ has a transition that combines the transition of the output subteam with a transition from the input subteam starting from another state of the subteam (and vice versa). Hence Receptiveness gives a necessary condition to avoid message loss, but does not impose it on δ.

Example 2 (Example 1 continued). Figure 2 (right) displays a team automaton over the component automata $\mathcal{A}_1$ and $\mathcal{A}_2$ depicted in Fig. 1.

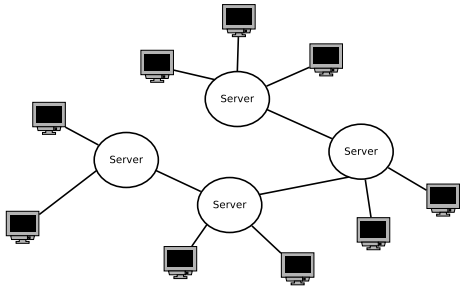
Clearly, the output and input subteams of action a in this team automaton are basically the component automata $\mathcal{A}_1$ and $\mathcal{A}_2$. We see that both are enabled in state $\binom{p_1}{q_2}$, but the synchronisation $(\binom{p_1}{q_2}, a, \binom{p_2}{q_3})$ is not part of the team. □

Therefore, we propose to investigate the case of master-slave synchronisations, because these policies express in a natural way the relation between output and input as expressed by Receptiveness, precluding message loss (reflected by masters followed by slaves) and deadlocks when the system gets blocked in a waiting state (with slaves that cannot proceed on their own).

In a preliminary exploration, we reconsider now Definition 9 with $\delta = s\mathcal{MS}^S$, $\delta = w\mathcal{MS}^S$ or $\delta = \mathcal{MS}^S$. Assume that the output subteam defined by δ is currently (global state p) enabled to execute action a and that the input subteam is also ready to execute a. We then know that the corresponding transitions of the input subteam involving a occur in δ in combination with all transitions of a in its output subteam, as desired. However, $\delta = \mathcal{MS}^S$ will also have a transition that does not involve the input subteam of a (as in the master-slave case, masters may proceed on their own). Thus even though the input subteam can oblige, messages may still get lost when its participation cannot be enforced. The weak master-slave and the strong master-slave policies however would have only transitions in which there is an (ouput-input) communication role for a. Thus, more restrictive assumptions regarding the collaboration between input and output subteams (rather than their internal workings) would support a general definition of compatibility like the one we propose here.

Before concluding this section, we recall once more that team automata composed according to non-state-sharing synchronisation policies may be encoded as synchronous product automata. Thus we can now use master-slave synchronisation policies to further our understanding of general compatibility requirements. For instance, we could apply this approach to strong master-slave team automata (as observed earlier, these are non-state-sharing) and investigate how compatibility in the encoded version relates to compatibility in the original system. Next, after encoding master-slave team automata (also these are non-state-sharing) as synchronous product automata we can investigate how compatibility in their encoded versions translates to (desirable) properties in the original system.

4.3 Applications of Compatibility



A typical example of the usefulness of a notion of compatibility in a setting of systems constructed according to synchronisation policies that differ from the policy of synchronous product may be found in the context of client-server architectures. A common solution to make such architectures more robust, i.e. resilient against server failures, is to replicate the server and thus move from a centralised architecture to a decentralised one, as depicted in the above figure. Then, when a server fails, other servers still running will continue to send messages to clients.

It might be important that these servers appear as one to the clients. This can be achieved by composing the set of servers according to so-called *(strong) output peer-to-peer synchronisations* [4]. For now, think of a scenario in which the set of servers as a whole uses so-called multicast communication to send messages to the clients (compared to broadcast communication, only clients that are within reach of the server, i.e. enabled, can receive its messages). This can be achieved by composing the set of servers (as a whole) with their clients according to weak master-slave synchronisations, upon which the definition/results of Sect. 4.2 become applicable.

We continue by describing another example, adapted from [16], inspired by the Esterel program of a ring of stations sharing a bus that was presented in [9].

Consider a system of three identical stations hooked to form a ring (cf. Fig. 3). A station's user can perform requests for accessing a common bus. User requests are granted depending on whether or not the corresponding station has the right to grant access, which is implemented by means of tokens flowing along the ring. While a station has the token, it has the right to grant access. To ensure fairness, a user is granted access for just one clock tick, after which the token is passed on. This implies we assume the presence of a global clock (not shown in the figure) whose only behaviour is producing ticks, thus synchronising all components.

The behaviour of the ith station is defined by the component automaton $\mathcal{T}_i$ in Fig. 3. If it has the token, then it checks whether the ith user has requested access within the current tick. If so, it grants the user bus access, after which the token is passed on upon the next tick and it returns to the initial state. If not, upon the next tick, the token is passed on and it returns to the initial state. The (simplified) behaviour of the ith user is defined by the component automaton $\mathcal{U}_i$ in Fig. 3. At any time, it can request access to the bus, upon which access is granted, unless a tick is received first, after which it returns to the initial state.

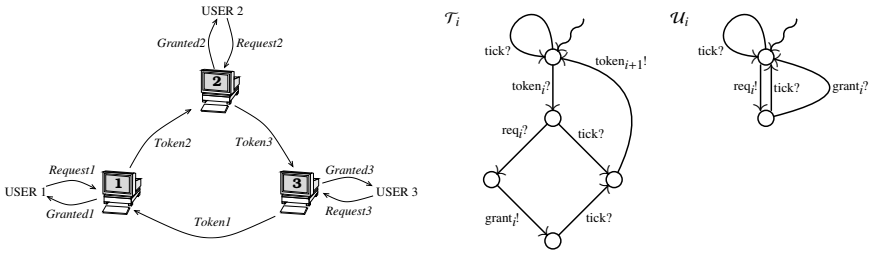


Fig. 3. A ring of stations sharing a bus, the ith station $\mathcal{T}_i$ and the ith user $\mathcal{U}_i$

Now consider the (weak) master-slave team automata over $\mathcal{T}_i$ and $\mathcal{U}_i$ depicted in Fig. 4. Note that all occurrences of tick? actually denote synchronisations of input actions tick? as (peer-to-peer) collaborations between the station and the user.

The master-slave team automaton $\mathcal{MS}$ is the one without the dotted red req$_i$? transition.[4] In this case, message req$_i$! can obviously be lost. From the initial state, the sequence req$_i$! tick? leads back to the initial state with a non-granted access request, meaning that the user made a request, but the clock tick occurred before the station reacted. However, $\mathcal{MS}$ is non-state-sharing, which means we can apply Theorem 13 and Definition 6 from [16]: there exists a synchronous product automaton $\chi(\{\mathcal{T}_i', \mathcal{U}_i'\})$ whose every computation corresponds to a computation of $\mathcal{MS}$ and in which no compatibility problems occur that did not exist in $\mathcal{MS}$, i.e. $\mathcal{MS}$ has a Receptiveness/Non-communicating progress/Deadlock-freeness violation at state q if $\chi(\{\mathcal{T}_i', \mathcal{U}_i'\})$ does.

[4] We have drawn this arc as an explicit example of a non master-slave synchronisation in which the station executes input action req$_i$? and does not synchronise with the req$_i$! of the user.

The weak master-slave team automaton $w\mathcal{MS}$ is $\mathcal{MS}$ without the dotted red $\text{req}_i?$ transition and it also misses the dashed green $\text{req}_i!$ and $\text{grant}_i!$ transitions. Hence, when a token has arrived in the initial state, the request action is executed synchronously by the user and the station, after which access is granted by another synchronous execution. However, also in this case message $\text{req}_i!$ can be lost, but—more importantly—this team is state-sharing, which means that we cannot apply Theorem 13 from [16]. This is where our new definition of compatibility (Definition 9) comes into play. (Cf. [16] for other team behaviour.)

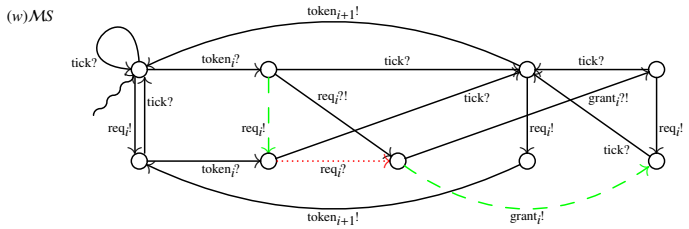


Fig. 4. The (weak) master-slave team automaton $(w)\mathcal{MS}$ over $\{\mathcal{T}_i, \mathcal{U}_i\}$.

5 Master-Slave Systems

Even without explicit reference to the actual synchronisations that take place within subteams, it is quite challenging, also in case of master-slave policies, to define compatibility in terms of correct input/output behaviour. In fact, which subteams can or should communicate varies with the evolution of the system as this depends on the current state of the system (cf. [16]).

In this section, we again consider master-slave synchronisations but now with the additional assumption that all components of the system are either masters or slaves, meaning that they can have output actions or input actions, but not both. As a consequence, every component has a fixed role (master or slave) in any communication in which it is involved. This assumption leads to a simple set-up facilitating the investigation of the communication behaviour. In particular, we expect the static dichotomy of the system in masters and slaves to support an iterative (bottom-up) approach to the construction of compatible systems. After a formal definition, we will discuss some simple cases to illustrate this point.

Definition 10. *A component automaton* $(P, (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int}), \gamma, J)$ *is a* master automaton *if* $\Gamma_{inp} = \emptyset$ *and it is a* slave automaton *if* $\Gamma_{out} = \emptyset$.

For a set $\mathcal{A}$ *of component automata,* $\mu(\mathcal{A})$ *denotes the subset of all master automata belonging to* $\mathcal{A}$ *and* $\sigma(\mathcal{A})$ *denotes the subset of all slave automata belonging to* $\mathcal{A}$.

$\mathcal{A}$ *is a* master-slave system *if* $\mathcal{A} = \mu(\mathcal{A}) \cup \sigma(\mathcal{A})$. □

Clearly, in general, $\mu(\mathcal{A}) \cup \sigma(\mathcal{A}) = \mathcal{A}$ does not hold, as $\mathcal{A}$ may contain automata with both output and input actions. Moreover, $\mu(\mathcal{A}) \cap \sigma(\mathcal{A})$ is not necessarily empty as

there may be an automaton in $\mathcal{A}$ with only internal actions. A component with external actions can be either an input component or an output component, but never both.

With every component not capable of receiving input or of providing output (or both), it is clear that master-slave systems cannot be used to describe 'protocol' behaviour, i.e. chains of 'action-response' events leading to some successful computation. Instead, master-slave systems behave as 'producer-consumer' systems. The type of chain behaviour described by master-slave systems can be observed in manufacturing systems (cf. [43] and the references therein) and it is also a typical design pattern in concurrency theory and programming (e.g. threading), where a buffer is usually used to avoid message loss.

Let $\mathcal{A}$ be a master automaton and let $\mathcal{A}'$ and $\mathcal{A}''$ be two slave automata forming a composable system. Assume that both $\mathcal{S}_1 = \{\mathcal{A}, \mathcal{A}'\}$ and $\mathcal{S}_2 = \{\mathcal{A}, \mathcal{A}''\}$ are composable master-slave systems that are strong master-slave compatible (i.e. they are compatible with respect to $s\mathcal{MS}^{\mathcal{S}_1}$ and with respect to $s\mathcal{MS}^{\mathcal{S}_2}$, respectively). Thus in each of the team automata, the master is always followed by the slave. Moreover, it is then also guaranteed that the master will always be followed by the two slaves in a single system if the two slaves are synchronised (e.g. in a synchronous product construction) and then the resulting slave (!) automaton is combined with the master. The system that is obtained in this way, is again a master-slave system and strong master-slave compatible. Formally, $\mathcal{S}_3 = \{\mathcal{A}, \mathcal{X}(\{\mathcal{A}', \mathcal{A}''\})\}$ is a composable master-slave system compatible with respect to $s\mathcal{MS}^{\mathcal{S}_3}$. A compatibility relation $\mathcal{R}$ for the new system $\mathcal{S}_3$ can be constructed from the compatibility relations $\mathcal{R}'$ for $\mathcal{S}_1$ and $\mathcal{R}''$ for $\mathcal{S}_2$ by letting $(q, q', q'') \in \mathcal{R}$ iff $(q, q') \in \mathcal{R}'$ and $(q, q'') \in \mathcal{R}''$.

As a second example, consider again a master automaton $\mathcal{A}$ and two slave automata $\mathcal{A}'$ and $\mathcal{A}''$. The component automaton $\mathcal{B}$ is constructed from $\mathcal{A}'$ by changing all (input) actions it shares with $\mathcal{A}''$ into output actions. Thus depending on whether $\mathcal{A}'$ shares none, all, or some of its input actions with $\mathcal{A}''$, the new $\mathcal{B}$ is a slave automaton, a master automaton, or neither. Here we assume that $\mathcal{B}$ is a master automaton. Furthermore, we suppose that the master-slave systems $\mathcal{S}_4 = \{\mathcal{A}, \mathcal{A}'\}$ and $\mathcal{S}_5 = \{\mathcal{B}, \mathcal{A}''\}$ are composable and strong master-slave compatible (i.e. they are compatible with respect to $s\mathcal{MS}^{\mathcal{S}_4}$ and with respect to $s\mathcal{MS}^{\mathcal{S}_5}$, respectively). Hence, $\mathcal{A}''$ is always ready to synchronise with $\mathcal{A}'$ in $\mathcal{X}(\{\mathcal{A}', \mathcal{A}''\})$. We conjecture that also the composable master-slave system $\mathcal{S}_6 = \{\mathcal{A}, \mathcal{X}(\{\mathcal{A}', \mathcal{A}''\})\}$ is compatible with respect to $s\mathcal{MS}^{\mathcal{S}_6}$ (to be proved as above by combining the compatibility relations for $\mathcal{S}_4$ and $\mathcal{S}_5$).

6 Applications of Asynchronicity

The main characteristic of the team automata framework is that it caters for component-based modelling and composition according to a wide range of synchronisation policies. The usefulness of such a flexible framework for compatibility is witnessed by examples from both hardware and software in which synchronisation deviates from the standard synchronous product. Hence the contributions of this paper may open the door to apply correct-by-construction design techniques in unprecedented areas. We provide in this section some examples in this direction.

Swarm Intelligence Recently, the notion of *swarm networks* has appeared as an alternative computation paradigm in the field of swarm intelligence [34]. In a swarm network, an agent communicates/cooperates through its sensors, actuators and connectors. Sensors and actuators allow the asynchronous communication through the receiving and sending of signals. Due to their limited capabilities, agents sometimes need to self-organise in communities through their connectors, in order to accomplish certain tasks. This ability resembles the hierarchical construction allowed by the team automata framework. Because of the huge number of agents a swarm network has, the simulation of such an environment may miss important facts about its correctness. Instead, the construction of swarm networks with certain compatibility guarantees may represent an important step towards their satisfactory application.

Hardware Design The end of Moore's Law may bring in front hardware technology architectures that provide, i.a., flexible (clock) synchronisation. Hence *asynchronous circuits*, and—in a more realistic incarnation—*globally asynchronous locally synchronous* (GALS) [18] or *elastic circuits* [13], represent viable alternatives for dealing with phenomena like the problem of *clock skew*. In the late 80s, David L. Dill proposed for the first time the theory of *conformance* between an asynchronous specification and the corresponding implementation. As happened with conformance almost thirty years ago, we believe that the notion of compatibility presented in this paper, which lifts most of the restrictions the conformance notion has, may assist the safe design of future hardware architectures.

Software Engineering Nowadays the use of the unified modeling language (UML) still dominates the field of software engineering for the design of systems. Unfortunately, due to UML's imprecise semantics, the compatibility checking of UML designs is a challenging task. Current solutions (e.g. [30]) only consider the synchronous product of UML State Charts as main composition operation, thus missing out on important constructs when modelling real systems. We believe that the work proposed in this paper can generalise previous attempts to accomplish the task of compatibility checking of UML specifications (cf. [29, 30]).

Manufacturing By focussing on particular subclasses of team automata, we have shown in Sect. 5 very interesting properties on the corresponding hierarchical construction. The systems modelled in Sect. 5 may represent a wide class of manufacturing systems, where not a protocol but (a chain of) producer-consumer behaviour is observed. In the context of manufacturing systems, incompatibility may lead to faults due to deadlocks or receptiveness violations, which may hamper the manufacturing of items.

Concurrent Asynchronous Programming The last decade, concurrent asynchronous programming languages have reached a certain maturity, demonstrated by their widespread industrial adoption. Erlang [2] is a prominent example; its asynchronous communication mode allows for a very flexible communication architecture, but on the other hand if used incorrectly may lead to invalid/suboptimal implementations of a system. To the best of our knowledge, current approaches follow the verification principle, i.e. a *post-mortem* approach to certify certain properties (e.g. liveness, safeness) of Erlang programs [17, 26]. Instead, correct-by-construction design might become possible if the theories described in this paper are used in the specification of Erlang programs.

Web Services Two services are *protocol compatible* if every joint execution of these services leads to a proper final state [23]. In [44], two main types of protocol mismatches were defined over a pair of service protocols: unspecified reception (lack of receptiveness in our setting) and mutual deadlock. In Fig. 2 of [23] we see a clear resemblance to our compatibility notion. Again, the limitation of using the synchronous product to verify compatibility, together with the extreme flexibility of team automata, might make the theory of this paper well-suited for the description of flexible compatibility relations also in the area of Web services.

7 Conclusions

In this paper, we continued the quest of [12, 15, 16] for precise conditions for the compatibility of components in systems of systems that (by construction) guarantee correct communications, free from message loss and deadlocks. We proposed a definition of compatibility for components that applies to any synchronisation policy allowed by team automata, after which we briefly discussed its application to master-slave synchronisations. While we still defined the generalised compatibility relation in terms of Non-communicating progress, Receptiveness and Deadlock-freeness, it refers to the enabledness of actions in output and input subteams rather than in their constituting components.

Related work Non-communicating progress prescribes that internal actions do not lead outside a compatibility relation. This extends the Internal progress condition of the I/O compatibility from [11, 12] reflecting the role of silent actions in bisimulations.

Receptiveness is a weak version of the input-enabledness requirement imposed on I/O automata [38, 39] by which output actions can never be blocked by components not ready to receive this communication as input because in each state, every input action has to be enabled. However, I/O automata are composed as synchronous products, meaning that one cannot distinguish different types of master-slave synchronisations, since all synchronisations on communicating actions are by definition strong master-slave (cf. Sect. 3). As the applications sketched in Sect. 6 confirm, input-enabledness is in general too strong a requirement. This was recognised also in the theory of Interface Automata [20, 21], where a form of receptiveness is achieved without imposing input-enabledness by a notion of compatibility that always guarantees at least one synchronisation that does not lead to an error state. Its extension into Sociable Interface Automata [19] moreover allows multi-way communication, while its associated tool [1] allows to check notions of composability and compatibility.

Deadlock-freeness prescribes that the system cannot terminate if an input subteam is still waiting for input, thus generalising—as in the case of Receptiveness—the notion from [12,16] in which this is required at the level of an individual component rather than a subteam. As noted in [32], interface automata compatibility does not imply deadlock-freeness.

In [31–33], an approach similar to ours considers communication-safety as a notion of compatibility in multi-component environments composed according to assembly theories to express the absence of communication errors. In this case, the modelling framework is a generalisation of both interface automata and modal I/O-transition systems [37] but the synchronous product is the only composition operator considered.

Future work Most importantly, we would like to investigate in depth our proposed definition of compatibility of components with respect to arbitrary synchronisation policies. This would range from explicitly taking other policies than master-slave synchronisation into account to studying the case of master-slave compatibility in more detail.

In general, we could extend Definition 9 with requirements relating to the actual synchronisations *within* the input and output subsystems. For instance, when prescribing a master-slave policy for an action a, we could require in addition that the output subteam of a is a synchronous product (masters operating as peers) or that the input subteam of a is a synchronous product, implying that all components with a as an input have to follow the master or masters (slaves operating as peers). These additional requirements could also be weakened by requiring such peer-to-peer collaborations only between enabled components. Also the obligation of slaves following masters may be formulated in a variety of ways, based on how the participation of input components (of an action's input subteam) is realised (e.g. requiring 'at least one', 'exactly one', 'all' or 'only those in which the action is enabled at the current state' to participate).

As anticipated in the Introduction, it would be interesting to study how (master-slave) compatibility is affected when (slave) components are added. Perhaps the combination of non-state-sharing and master-slave-compatible systems may lead to an incremental construction of compatible systems. For instance, assume that we have a non-state-sharing master-slave-compatible team automaton, and we add a new component to the team in a way that the new team is still non-state-sharing. Then what are the necessary conditions to also preserve the master-slave compatibility?

Finally, it would interesting to study possible cross-fertilisation with work on synthesis. In [14], the binary notion of I/O compatibility from [12] is applied to the synthesis of asynchronous circuits modelled as Petri nets. It would be interesting to see whether this can be extended to multi-component systems based on the compatibility of components with respect to synchronisation policies other than the synchronous product. In [7], supervisory control theory [42] is applied to product lines. This theory provides a means to synthesise a supervisory controller automaton from a set of components and requirements. If such a synthesised supervisory controller exists, then the resulting synchronous product of the components and the supervisory controller not only satisfies the requirements, but it is moreover non-blocking (the system can always reach an accepted stable state), controllable (only the components' external actions can be influenced, internal actions cannot) and maximally permissive (allowing as much behaviour of the components without violating the requirements). It would be interesting to see whether this mechanism can be extended to deal with components synchronised according to policies other than the synchronous product, possibly in combination with the product line theories presented in [37].

Acknowledgements We thank the reviewers for their suggestions and additional references to related work. M.H. ter Beek was supported by the CNR through a Short-Term Mobility grant and J. Carmona was supported by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R).

References

1. B. T. Adler, L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, V. Raman, and P. Roy. Ticc: A Tool for Interface Compatibility and Composition. In T. Ball and R. B. Jones, editors, *CAV 2006*, volume 4144 of *LNCS*, pages 59–62. Springer, 2006.
2. J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.
3. M. H. ter Beek. *Team Automata: A Formal Approach to the Modeling of Collaboration Between System Components*. PhD thesis, Leiden University, 2003.
4. 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.
5. M. H. ter Beek and J. Kleijn. Team Automata Satisfying Compositionality. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003*, volume 2805 of *LNCS*, pages 381–400. Springer, 2003.
6. M. H. ter Beek and J. Kleijn. Modularity for teams of I/O automata. *Inf. Process. Lett.*, 95(5):487–495, 2005.
7. M. H. ter Beek, M. A. Reniers, and E. P. de Vink. Supervisory Controller Synthesis for Product Lines using CIF 3. In T. Margaria and B. Steffen, editors, *ISoLA 2016*, LNCS. Springer, 2016.
8. S. Bensalem, M. Bozga, B. Boyer, and A. Legay. Incremental Generation of Linear Invariants for Component-Based Systems. In *Proceedings of the 13th International Conference on Application of Concurrency to System Design (ACSD 2013)*, pages 80–89. IEEE, 2013.
9. G. Berry. *The Esterel v5 Language Primer*. Ecole des Mines de Paris/INRIA, 2000.
10. 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.
11. J. Carmona. *Structural Methods for the Synthesis of Well-Formed Concurrent Specifications*. PhD thesis, Universitat Politècnica de Catalunya, 2004.
12. J. Carmona and J. Cortadella. Input/Output Compatibility of Reactive Systems. In M. Aagaard and J. W. O'Leary, editors, *FMCAD 2002*, volume 2517 of *LNCS*, pages 360–377. Springer, 2002.
13. J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic Circuits. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 28(10):1437–1455, 2009.
14. J. Carmona, J. Cortadella, and E. Pastor. Synthesis of Reactive Systems: Application to Asynchronous Circuit Design. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Advances in Petri Nets*, volume 2549 of *LNCS*, pages 108–151. Springer, 2002.
15. J. Carmona and J. Kleijn. Interactive Behaviour of Multi-Component Systems. In J. Cortadella and A. Yakovlev, editors, *ToBaCo 2004*, pages 27–31, 2004.
16. J. Carmona and J. Kleijn. Compatibility in a multi-component environment. *Theor. Comput. Sci.*, 484:1–15, 2013.
17. D. Castro, V. M. Gulías, C. B. Earle, L. Fredlund, and S. Rivas. A Case Study on Verifying a Supervisor Component Using McErlang. *ENTCS*, 271:23–40, 2011.
18. D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, 1984.
19. L. de Alfaro, L. D. da Silva, M. Faella, A. Legay, P. Roy, and M. Sorea. Sociable Interfaces. In B. Gramlich, editor, *FroCoS 2005*, volume 3717 of *LNCS*, pages 81–105. Springer, 2005.
20. L. de Alfaro and T. A. Henzinger. Interface Automata. In *ESEC/FSE 2001*, pages 109–120. ACM, 2001.
21. L. de Alfaro and T. A. Henzinger. Interface-Based Design. In M. Broy, J. Grünbauer, D. Harel, and T. Hoare, editors, *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series*, pages 83–104. Springer, 2005.

22. E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numer. Math.*, 8(3):174–186, 1968.
23. M. Dumas, B. Benatallah, and H. R. M. Nezhad. Web Service Protocols: Compatibility and Adaptation. *IEEE Data Eng. Bull.*, 31(3):40–44, 2008.
24. G. Engels and L. Groenewegen. Towards Team-Automata-Driven Object-Oriented Collaborative Work. In W. Brauer, H. Ehrig, J. Karhumäki, and A. Salomaa, editors, *Formal and Natural Computing*, volume 2300 of *LNCS*, pages 257–276. Springer, 2002.
25. G. Gössler and J. Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55:161–183, 2005.
26. Q. Guo, J. Derrick, C. B. Earle, and L. Fredlund. Model-Checking Erlang — A Comparison between EtomCRL2 and McErlang. In L. Bottaci and G. Fraser, editors, *TAIC PART 2010*, volume 6303 of *LNCS*, pages 23–38. Springer, 2010.
27. A. Hall. Correctness by Construction: Integrating Formality into a Commercial Development Process. In L. Eriksson and P. A. Lindsay, editors, *FME 2002*, volume 2391 of *LNCS*, pages 224–233. Springer, 2002.
28. A. Hall and R. Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Softw.*, 19(1):18–25, Jan/Feb 2002.
29. Y. Hammal. A Modular State Exploration and Compatibility Checking of UML Dynamic Diagrams. In *AICCSA 2008*, pages 793–800. IEEE, 2008.
30. Y. Hammal. Behavioral Compatibility of Active Components. In *SEFM 2008*, pages 372–376. IEEE, 2008.
31. R. Hennicker and A. Knapp. Modal Interface Theories for Communication-Safe Component Assemblies. In A. Cerone and P. Pihlajasaari, editors, *ICTAC 2011*, volume 6916 of *LNCS*, pages 135–153. Springer, 2011.
32. R. Hennicker and A. Knapp. Moving from interface theories to assembly theories. *Acta Inf.*, 52(2-3):235–268, 2015.
33. R. Hennicker, A. Knapp, and M. Wirsing. Assembly Theories for Communication-Safe Component Systems. In S. Bensalem, Y. Lakhneck, and A. Legay, editors, *FPS 2014*, volume 8415 of *LNCS*, pages 145–160. Springer, 2014.
34. T. Isokawa *et al.* Computing by Swarm Networks. In H. Umeo *et al.*, editor, *ACRI 2008*, volume 5191 of *LNCS*, pages 50–59. Springer, 2008.
35. M. Jamshidi, editor. *System of Systems Engineering: Innovations for the Twenty-First Century*. Wiley, 2008.
36. D. G. Kourie and B. W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, 2012.
37. K. G. Larsen, U. Nyman, and A. Wąsowski. Modal I/O Automata for Interface and Product Line Theories. In R. De Nicola, editor, *ESOP 2007*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.
38. N. A. Lynch and M. R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *PODC 1987*, pages 137–151. ACM, 1987.
39. N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.
40. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
41. C. C. Morgan. *Programming from Specifications*. Prentice Hall, 2nd edition, 1994.
42. P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.
43. M. Silva and R. Valette. Petri Nets and Flexible Manufacturing. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 424 of *LNCS*, pages 374–417. Springer, 1990.
44. D. M. Yellin and R. E. Strom. Protocol Specifications and Component Adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.