
Computer Supported Cooperative Work **12:** 21–69, 2003.

Synchronizations in Team Automata for Groupware Systems

MAURICE H. TER BEEK[1], CLARENCE A. ELLIS[2], JETTY KLEIJN[1] & GRZEGORZ ROZENBERG[1,2]

[1]*Leiden Institute of Advanced Computer Science, Universiteit Leiden, P.O. Box 9512, 2300 RA, Leiden, The Netherlands (E-mails: mtbeek@liacs.nl; kleijn@liacs.nl; rozenber@liacs.nl);*
[2]*Department of Computer Science, University of Colorado, Campus Box 430, Boulder, CO 80309-0430, U.S.A. (E-mail: skip@colorado.edu)*

Abstract. Team automata have been proposed in Ellis (1997) as a formal framework for modeling both the conceptual and the architectural level of groupware systems. Here we define team automata in a mathematically precise way in terms of component automata which synchronize on certain executions of actions. At the conceptual level, our model serves as a formal framework in which basic groupware notions can be rigorously defined and studied. At the architectural level, team automata can be used as building blocks in the design of groupware systems.

Key words: CSCW, formalization, groupware systems, master-slave, peer-to-peer, synchronizations, team automata framework

1. Introduction

Computer Supported Cooperative Work (CSCW for short) is concerned with understanding how people work together, and ways in which technology can assist. By the nature of the field this technology mostly consists of multi-user software, so-called groupware. CSCW is a rapidly developing field which benefits from the evolution and formulation of basic ideas and intuitions. Our understanding is enhanced when these are accompanied by formal versions which require an exact formulation of basic features. Such formalizations make analysis possible and properties can be rigorously proved.

At a conceptual level, CSCW needs a precise and consistent terminology while at a lower, architectural level, CSCW has been searching for a rigorous framework to describe, compare and contrast groupware systems. Team automata were proposed in Ellis (1997) explicitly for the specification and analysis of CSCW phenomena and groupware systems, and were shown to be useful at the conceptual and architectural levels. The present paper is a direct continuation of Ellis (1997) – we elaborate here further the concept of a team automaton. In particular, we present team automata as a framework well suited to clarify and capture precisely notions related to coordination, cooperation and collaboration in distributed systems.

Team automata consist of an abstract specification of the components of a system and allow one to describe different interconnection mechanisms based upon the concept of 'shared action', called synchronizations here. A synchronization is the simultaneous execution of a single action by several (one or more) components. In such an execution, the remaining components are not involved. By the postulate that those components that do not participate in a synchronization, remain idle, one obtains a clocked synchronous model. Components can be combined in a loose or more tight fashion depending on which actions are to be shared, and when. Such aggregates of components can then in turn be used as components in a higher-level team. The freedom to model different types of interactions between components and subteams is the main reason why the team automata framework is well suited for the design of groupware systems. Its rigorous definitions form the basis for the analysis and verification of the design.

Team automata thus fit nicely with the needs and the philosophy of groupware systems. Moreover, thanks to the formal automata theoretic setup, results and methodologies from automata theory are applicable. While inappropriate for capturing aspects of group activity such as social aspects and informal unstructured activity, the team automata framework has proved useful in various CSCW modeling areas (see, e.g., Ellis, 1997; Sun and Ellis, 1998; ter Beek et al., 2001b). A spectrum from hardware components to protocols for interacting groups of people can be modeled by team automata.

This paper is mainly concerned with a precise description of the team automata framework and the possibilities it offers for the design of groupware systems by focusing on modeling concepts related to synchronizations and hierarchical constructions. Not much is said about behavioral aspects and analysis. It is written to serve both theoretical computer scientists interested in formal models with a clear practical motivation, and those working in CSCW well motivated to look for formalizations of core problems in their field. We hope to achieve this by accompanying our formal definitions by explanations and examples, providing the motivation for and the interpretation of these definitions.

The organization of the paper is as follows. First we fix some mathematical notation for future reference. Consequently we introduce team automata, followed by an exposition of their possibilities to define hierarchies by iteration. Next we focus on different types of synchronization and turn to the question of how to (uniquely) construct team automata satisfying certain synchronization constraints. We then switch our attention from the conceptual level to the architectural level by showing how team automata can be used as architectural building blocks, after which we present an example of how to model a groupware architecture by team automata. Finally, we compare team automata with some related formalisms and we conclude the paper with a discussion on the use of team automata in the design of groupware systems.

2. Some notation

This section lists some mathematical notation used throughout this paper and may be skipped on first reading.

$\mathbb{N}$ denotes the set of positive integers. Let $\mathcal{I} \subseteq \mathbb{N}$ be a set of indices given by $\mathcal{I} = \{i_1, i_2, \ldots\}$ with $i_j < i_l$ if $1 \leq j < l$. For sets V_i, $i \in \mathcal{I}$, we denote by $\prod_{i\in\mathcal{I}} V_i$ the cartesian product $\{(v_{i_1}, v_{i_2}, \ldots) | v_{i_j} \in V_{i_j}$, for all $j \geq 1\}$. If $v_i \in V_i$, for all $i \in \mathcal{I}$, then $\prod_{i\in\mathcal{I}} v_i$ denotes the element $(v_{i_1}, v_{i_2}, \ldots)$ of $\prod V_i$. If $\mathcal{I} = \varnothing$, then $\prod_{i\in\mathcal{I}} V_i = \varnothing$. In addition to the prefix notation $\prod_{i\in\mathcal{I}} V_i$ or $\prod_{i\in\mathcal{I}} v_i$ for a cartesian product, we also use the infix notation $V_{i_1} \times V_{i_2} \times \cdots$ or $v_{i_1} \times v_{i_2} \times \cdots$, respectively. Let $A \subseteq \prod_{i\in\mathcal{I}} V_i$. Then, for $j \in \mathcal{I}$, $\mathrm{proj}_j : A \to V_j$ is defined by $\mathrm{proj}_j(a_{i_1}, a_{i_2}, \ldots) = a_j$. For $J \subseteq \mathcal{I}$, $\mathrm{proj}_J : A \to \prod_{i\in J} V_i$ is defined by $\mathrm{proj}_J(a) = \prod_{j\in J} \mathrm{proj}_j(a)$. The set $\{V_i | i \in \mathcal{I}\}$ is said to form a partition (of $\bigcup_{i\in\mathcal{I}} V_i$) if the V_i are pairwise disjoint, nonempty sets.

Set inclusion is denoted by $\subseteq$, whereas proper inclusion is denoted by $\subset$. The set difference of sets V and W is denoted by $V \setminus W$. For a finite set V, its cardinality is denoted by $\#V$. For convenience, we sometimes denote the set $\{1, 2, \ldots, n\}$ by $[n]$. Then $[0] = \varnothing$.

An alphabet is a possibly infinite, possibly empty, set of symbols. A word or string over an alphabet Σ is a finite sequence of symbols from Σ. The empty word is denoted by λ. A language L over Σ is a set of words over Σ and Σ^* denotes the set of all words over Σ.

Let $f : A \to A'$ and let $g : B \to B'$ be (total) functions. Then $f \times g : A \times B \to A' \times B'$ is defined as $(f \times g)(a, b) = (f(a), g(b))$. We will use $f^{[2]}$ as shorthand notation for $f \times f : A \times A \to A' \times A'$. Thus $f^{[2]}(a, b) = (f(a), f(b))$. This notation should not be confused with iterated function application. In particular, we will use $\mathrm{proj}_i{}^{[2]}$ as shorthand notation for $\mathrm{proj}_i \times \mathrm{proj}_i$ and likewise $\mathrm{proj}_J{}^{[2]}$ for $\mathrm{proj}_J \times \mathrm{proj}_J$. If $C \subseteq A$, then $f(C) = \{f(a) | a \in C\}$, and if $D \subseteq A \times A$, then $f^{[2]}(D) = \{(f(d_1), f(d_2)) | (d_1, d_2) \in D\}$.

3. Component automata and team automata

A main objective of this paper is to unambiguously present the notion of a team automaton as a fundamental building block for groupware. We thus begin this presentation by mathematically defining the notions of team automata and their components. The basic concept underlying both team and component automata is a labeled transition system with initial states, which captures the idea of a system with states (configurations, possibly an infinite number of them), together with actions the executions of which lead to (non-deterministic) state changes.

Definition 1. An *initialized labeled transition system*, ilts for short, is a construct $\mathcal{A} = (Q, \Sigma, \delta, I)$, where

Q is its set of *states*, possibly infinite,

Σ is its alphabet of *actions*, such that $Q \cap \Sigma = \varnothing$,

$\delta \subseteq Q \times \Sigma \times Q$ is its set of (*labeled*) *transitions* and
$I \subseteq Q$ is its set of *initial states*.
Its set of (finite) *computations* is denoted by $\mathbf{C}_{\mathcal{A}}$ and is defined as

$$\mathbf{C}_{\mathcal{A}} = \{q_0 a_1 q_1 a_2 \cdots a_n q_n | q_0 \in I,\ n \geq 0 \text{ and } (q_i, a_{i+1}, q_{i+1}) \in \delta, \text{ for all } 0 \leq i < n\}. \quad \square$$

The *language* $\mathbf{L}_{\mathcal{A}}$ of an ilts $\mathcal{A}$ is the set of action sequences that can be executed starting from an initial state. They can be obtained by filtering out the state information from the computations.

Let $\mathcal{A} = (Q, \Sigma, \delta, I)$ be an ilts and let $a \in \Sigma$. Then we denote by δ_a its set of *a-transitions*, i.e. $\delta_a = \{(q, q') | (q, a, q') \in \delta\}$. An a-transition $(q, q) \in \delta_a$ is called a *loop* (on a). Let $q \in Q$. Then we say that a is *enabled* (*in* $\mathcal{A}$) *at* q, denoted by a en$_{\mathcal{A}}$ q, if there is a state q' of $\mathcal{A}$ such that $(q, q') \in \delta_a$.

A component automaton is an ilts together with a classification of its actions. The actions are divided into two main categories, one of which is subdivided into two more categories. *Internal* actions have strictly local visibility and can thus not be used for communication with other components, whereas *external* actions are observable by other components. These external actions can be used for communication between components and are divided into *input* actions and *output* actions. As formulated in Ellis (1997): "input actions are not under the local system's control and are caused by another non-local component, the output actions are under the system's control and are externally observable by other components, and internal actions are under the local system's control but are not externally observable". When describing a component automaton, one of the design issues that thus has to be considered is the role of the actions within that component in relation to the other components within the system under design.

Definition 2. A *component automaton* is a construct $\mathcal{C} = (Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, I)$ such that $(Q, \Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}, \delta, I)$ is an ilts, called the *underlying* ilts of $\mathcal{C}$ and denoted by $und(\mathcal{C})$, and Σ_{inp}, Σ_{out} and Σ_{int} are mutually disjoint alphabets called the *input*, *output* and *internal* alphabet of $\mathcal{C}$, respectively. $\square$

The *external* alphabet of $\mathcal{C}$, denoted by Σ_{ext}, is defined by $\Sigma_{ext} = \Sigma_{inp} \cup \Sigma_{out}$. The elements of the input, output, internal and external alphabets are called input actions, output actions, internal actions and external actions, respectively.

For a given component automaton the notions of computations and language can be defined through its underlying ilts. Thus the set of computations (language) of a component automaton is the set of computations (language) of its underlying ilts, i.e. $\mathbf{C}_{\mathcal{C}} = \mathbf{C}_{und(\mathcal{C})}$ and $\mathbf{L}_{\mathcal{C}} = \mathbf{L}_{und(\mathcal{C})}$. The language of a component automaton is thus obtained by preserving only the actions from its computations. In general however, given the different roles actions may have in a component automaton one may also opt, e.g., for information on the external behavior by preserving only the external actions from the computations.

We refer to $\mathcal{C}$ as the *trivial automaton* if $\mathcal{C} = (\varnothing, (\varnothing, \varnothing, \varnothing), \varnothing, \varnothing)$. Obviously, the trivial automaton has an empty set of computations and an empty language. Note, however, that this holds for any component automaton with an empty set of initial states.

Before we turn to the definition of a team automaton formed from component automata we fix some notation for the rest of the paper.

Notation 1. In the sequel we assume a fixed, but arbitrary and possibly infinite set $\mathcal{I} \subseteq \mathbb{N}$, which we will use to index the component automata involved. For each $i \in \mathcal{I}$, we let $\mathcal{C}_i = (Q_i, (\Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}), \delta_i, I_i)$ be a fixed component automaton and we use Σ_i to denote its set of actions $\Sigma_{i,inp} \cup \Sigma_{i,out} \cup \Sigma_{i,int}$ and $\Sigma_{i,ext}$ to denote its set of external actions $\Sigma_{i,inp} \cup \Sigma_{i,out}$. Moreover, we let $\mathcal{S} = \{\mathcal{C}_i | i \in \mathcal{I}\}$ be a set (or system) consisting of the component automata $\mathcal{C}_i$. Note that $\mathcal{I} \subseteq \mathbb{N}$ implies that $\mathcal{I}$ is ordered by the usual $\leq$ relation on $\mathbb{N}$, thus inducing an ordering on $\mathcal{S}$. Also note that the $\mathcal{C}_i$ are not necessarily different. □

When composing a team automaton, we require that the internal actions of the components involved are private, i.e. uniquely associated to one component automaton. This is formally expressed as follows.

Definition 3. $\mathcal{S}$ is a *composable system* if, for all $i \in \mathcal{I}$,
$\Sigma_{i,int} \cap \bigcup_{j \in \mathcal{I} \setminus \{i\}} \Sigma_j = \varnothing$. □

We refer to the condition above as the *composability condition*. Note that every subset of a composable system is again a composable system.

A state q of any team automaton over the composable system $\mathcal{S}$ is a product $(q_1, q_2, \ldots, q_i, \ldots)$, with q_i a state of component automaton $\mathcal{C}_i$. Thus q is a description of the states that each of the component automata is in. Recall that for each $i \in \mathcal{I}$, Q_i is the set of states of the component automaton $\mathcal{C}_i$. The state space of any team automaton $\mathcal{T}$ formed from $\mathcal{S}$ is thus the product $\prod_{i \in \mathcal{I}} Q_i$ of the state spaces of the component automata of $\mathcal{S}$, with the product $\prod_{i \in \mathcal{I}} I_i$ of their initial states forming the set of initial states of $\mathcal{T}$. The transition relation of such $\mathcal{T}$ is defined by allowing certain synchronizations and excluding others and is based solely on the transition relations of the components forming the team.

Definition 4. Let $a \in \bigcup_{i \in \mathcal{I}} \Sigma_i$. Then the *complete transition space* of a in $\mathcal{S}$ is denoted by $\Delta_a(\mathcal{S})$ and is defined as

$$\Delta_a(\mathcal{S}) = \{(q, q') \in \textstyle\prod_{i \in \mathcal{I}} Q_i \times \prod_{i \in \mathcal{I}} Q_i \mid \exists j \in \mathcal{I} : \ \mathrm{proj}_j{}^{[2]}(q, q') \in \delta_{j,a}$$
$$\text{and } (\forall i \in \mathcal{I} : [\mathrm{proj}_i{}^{[2]}(q, q') \in \delta_{i,a}] \text{ or } [\mathrm{proj}_i(q) = \mathrm{proj}_i(q')])\}. \quad \square$$

Thus the complete transition space $\Delta_a(\mathcal{S})$ consists of all possible combinations of a-transitions from components of $\mathcal{S}$, with all non-participating components remaining idle. It is an explicit requirement that at least one component is

active, i.e. performs an a-transition. The transitions in $\Delta_a(\mathcal{S})$ are referred to as *synchronizations* (on a).

This $\Delta_a(\mathcal{S})$ is called the *complete* transition space of $\mathcal{S}$ because whenever a team automaton $\mathcal{T}$ is constructed from $\mathcal{S}$, then for each action a, all a-transitions of $\mathcal{T}$ come from $\Delta_a(\mathcal{S})$. The transformation of the state of $\mathcal{T}$ is defined by the local state changes of the components participating in the action being executed. When defining $\mathcal{T}$, for each action a, a specific subset of $\Delta_a(\mathcal{S})$ has to be chosen. By restricting the set of allowed transitions, a certain kind of interaction between the component automata can be enforced. For an internal action, however, each component retains all its possibilities to execute that action and change state. Since $\mathcal{S}$ is a composable system, synchronizations on internal actions thus never involve more than one component.

The alphabets of actions of any team automaton $\mathcal{T}$ formed from $\mathcal{S}$ are uniquely determined by the alphabets of actions of the components of $\mathcal{S}$. The internal actions of the components will be the internal actions of $\mathcal{T}$. Each action which is output for one or more of the component automata is an output action of $\mathcal{T}$. Hence an action that is an output action of one component and also an input action of another component, is considered an output action of the team. The input actions of the component automata that do not occur at all as an output action of any of the component automata, are the input actions of the team. The reason for this construction of alphabets is again based on the intuitive idea of Ellis (1997) that when relating an input action a of a component automaton to an output action a of another component, the input may be thought of as being caused by the output. On the other hand, output actions remain observable as output to other automata.

Definition 5. Let $\mathcal{S}$ be a composable system. Then $\mathcal{T} = (\prod_{i\in\mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i\in\mathcal{I}} I_i)$ is a *team automaton over* $\mathcal{S}$ if

$\Sigma_{int} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,int}$,
$\Sigma_{out} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,out}$,
$\Sigma_{inp} = (\bigcup_{i\in\mathcal{I}} \Sigma_{i,inp}) \setminus \Sigma_{out}$ and
$\delta \subseteq \prod_{i\in\mathcal{I}} Q_i \times \Sigma \times \prod_{i\in\mathcal{I}} Q_i$, where $\Sigma = \Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}$, is such that for all $a \in \Sigma$,
$\delta_a \subseteq \Delta_a(\mathcal{S})$, and moreover $\delta_a = \Delta_a(\mathcal{S})$ if $a \in \Sigma_{int}$. □

All team automata over a given composable system thus have the same set of states, the same alphabet of actions – including the distribution over input, output and internal actions – and the same set of initial states. They only differ by the choice of the transition relation, which is based on but not fixed by the transition relations of the component automata.

For a given action $a \in \bigcup_{i\in\mathcal{I}} \Sigma_i$, a team automaton $\mathcal{T}$ over $\mathcal{S}$ has as its a-transitions a selection δ_a chosen from all possible synchronizations on a, as collected in $\Delta_a(\mathcal{S})$. In the case that $\delta_a = \Delta_a(\mathcal{S})$, no combination of a-transitions from the components is excluded in $\mathcal{T}$. In general, however, δ_a is a subset of $\Delta_a(\mathcal{S})$, which implies that certain combinations of a-transitions from some components

may be forbidden by the construction of $\mathcal{T}$ even though a is enabled in each of the current local states of these components.

Due to the freedom of choosing a δ_a for each external action a, a composable system $\mathcal{S}$ does not uniquely define a single team automaton. Instead, a flexible framework is provided within which one can construct a variety of team automata over $\mathcal{S}$, all of which differ solely by the choice of the transition relation. Hence by choosing a specific subset of $\Delta_a(\mathcal{S})$ we fix a specific team automaton.

As we will see later, fixed strategies for choosing transition relations in a predetermined way can be described, which lead to uniquely defined team automata. An example of such a strategy is the rule to include, for all actions a, all and only those a-transitions in which all component automata participate that have a as one of their actions. This leaves no choice for the transition relation, and thus leads to a unique team automaton. Constructing the transition relation according to this particular strategy is very natural and often presupposed implicitly in the literature (see, e.g., Lynch and Tuttle, 1989; Janicki and Laurer, 1992). Note that the freedom of the team automata model to choose transition relations offers the flexibility to distinguish even the smallest nuances in the meaning of one's design. Leaving the set of transitions of a team automaton as a modeling choice is perhaps the most important feature of team automata.

Finally, an important observation at this point is that within the formalization given here of a team automaton, no explicit information on loops is provided. That is to say, in general one cannot distinguish whether or not a component with a loop on a in its current local state participates in the team's synchronization on a. This component may have been idle or, after having participated in the action a starting from the global state, it may have returned to its original local state. Nevertheless, in order to relate the computations of a team to those taking place in the components we simply apply projections. Recall that computations are determined by the consecutive execution of transitions, starting from the initial state. Consider a team transition (q, a, q'). We now stipulate that the j-th component participates in this transition by executing $(\mathrm{proj}_j(q), a, \mathrm{proj}_j(q'))$ whenever $\mathrm{proj}^{[2]}(q, q') \in \delta_{j,a}$. Otherwise no transition takes place in the j-th component. We thus resolve the problem of loops by implicitly assuming that the presence of a component's loop in a transition of a team implies execution of that loop. This may be considered as a 'maximal' interpretation of the components' participation.

Since computations of team automata are sequences of synchronizations, this projection on the j-th component yields computations of that j-th component. Hence team computations are composites of the components' computations. However, due to the fact that the transition relation of a team automaton is only required to be a subset of the complete transition space, not every computation of a component of a team will be part of a computation of that team.

The following example illustrates the definition of team automata. Note that here and in the following examples vectors may be written vertically, even though in the text they are written horizontally.

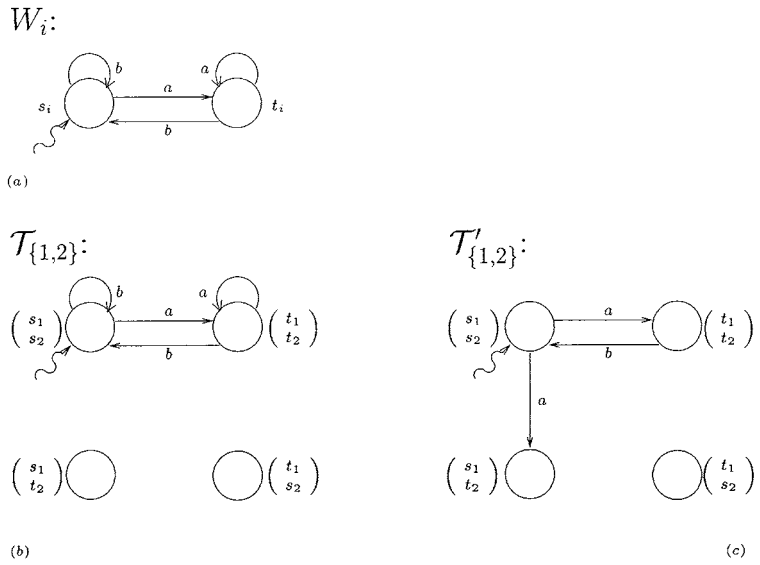

Figure 1. Component automata W_i, and team automata $\mathcal{T}_{\{1,2\}}$ and $\mathcal{T}'_{\{1,2\}}$.

Example 1. For $i \in \{1, 2\}$, let $W_i = (\{s_i, t_i\}, (\varnothing, \{a, b\}, \varnothing), \delta_i, \{s_i\})$, where $\delta_i = \{(s_i, b, s_i), (s_i, a, t_i), (t_i, a, t_i), (t_i, b, s_i)\}$, be two component automata, each modeling a **w**heel of a car. They are depicted in Figure 1(a).

The states s_i indicate that the i-th wheel is **s**tanding **s**till, while the states t_i indicate that the i-th wheel **t**urns. Then the result of **a**ccelerating, modeled by the action a, makes the wheel turn. The result of **b**raking, modeled by the action b, causes the wheel to stand still. Naturally, the wheel is initially standing still.

We show how W_1 and W_2 can form a team (an axle). Clearly, $\{W_1, W_2\}$ is a composable system. Team automaton $\mathcal{T}_{\{1,2\}}$ is depicted in Figure 1(b). It has four states of which (s_1, s_2) is its only initial state. It has no other actions than (the output actions) a and b. We require that the two wheels W_1 and W_2 accelerate and break in unison, so we choose $\delta_{\{1,2\}} = \{((s_1, s_2), b, (s_1, s_2)), ((s_1, s_2), a, (t_1, t_2)), ((t_1, t_2), a, (t_1, t_2)), ((t_1, t_2), b, (s_1, s_2))\}$. Note that only the transition relation had to be chosen, all other elements follow from Definition 5.

As mentioned before, explicit information on loops is lacking. From the execution of the transition $((s_1, s_2), b, (s_1, s_2))$ one can thus only conclude, by Definition 4, that at least one of the components s_1 and s_2 participated in this team action b, i.e. either (s_1, b, s_1) or (s_2, b, s_2) or both were executed. From the description of $\mathcal{T}_{\{1,2\}}$ one cannot conclude which component executed b and whether a component remained idle. However, as said before, we consider a maximal interpretation of the components' involvements in team transitions, in the sense that we assume that if a component could have participated in a team transition a by executing a loop on this action a, then it indeed has done so.

Finally, by choosing a different transition relation such as, e.g., $\delta'_{\{1,2\}} = \{((s_1, s_2), a, (s_1, t_2)), ((t_1, t_2), b, (s_1, s_2))\}$, another team automaton over $\{W_1, W_2\}$ is defined, which we denote by $\mathcal{T}'_{\{1,2\}}$. Apart from its transition relation, $\mathcal{T}'_{\{1,2\}}$ contains the same elements as $\mathcal{T}_{\{1,2\}}$. $\mathcal{T}'_{\{1,2\}}$ is depicted in Figure 1(c). If we assume that a flat tire is modeled by a wheel that cannot accelerate, then in $\mathcal{T}'_{\{1,2\}}$ the wheel W_1 has a flat tire. $\mathcal{T}'_{\{1,2\}}$ ends up in a deadlock after the execution of a, since one doesn't drive far with a flat tire. Note also that action b can never be executed in $\mathcal{T}'_{\{1,2\}}$. □

Before turning to a further formal investigation of the notion of a team automaton, we make two additional observations.

First, it should be noted that in the definition of a team over $\mathcal{S} = \{\mathcal{C}_i | i \in \mathcal{I}\}$ we have implicitly used the ordering on $\mathcal{S}$ induced by $\mathcal{I}$. Every team automaton over $\mathcal{S}$ has $\prod_{i \in \mathcal{I}} Q_i$ as its set of states and thus, if $\mathcal{I} = \{i_1, i_2, \ldots\}$ with $i_1 < i_2 < \cdots$, then every state q of $\mathcal{T}$ is a tuple $(q_{i_1}, q_{i_2}, \ldots)$ with $q_{i_j} \in Q_{i_j}$. This is convenient in concrete situations, but note that changing the order of the components in $\mathcal{S}$ leads to formally different state spaces. As an example, consider two component automata $\mathcal{C}_4$ and $\mathcal{C}_7$ with sets of states Q_4 and Q_7, respectively. Assume that $\mathcal{S} = \{\mathcal{C}_i | i \in \{4, 7\}\}$ is composable. Then also $\mathcal{S}' = \{D_j | j \in \{1, 2\}\}$ with $D_1 = \mathcal{C}_7$ and $D_2 = \mathcal{C}_4$ is composable. Teams over $\mathcal{S}$ have $\{(q, q') | q \in Q_4, q' \in Q_7\}$ as their state space, whereas teams over $\mathcal{S}'$ have $\{(q', q) | q \in Q_4, q' \in Q_7\}$ as their state space. In Section 4 we will come back to the ordering within state spaces in a more general setup.

Secondly, neither in the definition of an ilts, nor in the definition of a component automaton nor in the definition of a team automaton, have we required that states or actions have to be useful in the sense that they should be reachable or executable, respectively, in at least one computation starting from the initial state of the system. The lack of such extra conditions allows for a smooth and general definition of a team automaton, with the full cartesian product of the state sets of the components as the team's state space and an arbitrary selection of synchronizations as its transitions. Although it is often quite natural to consider so-called reduced automata, in which all states and transitions are useful, we will not do so here because it is not of immediate relevance to the topic of this paper.

Consistency in the sense that in a team automaton every action appears exclusively as an input, output or internal action, is guaranteed by the composability condition and Definition 5 (which ensures that input and output actions remain distinct). As a consequence, every team automaton is again a component automaton, which in its turn could be used as a component in a new team.

Lemma 1. Every team automaton is a component automaton.

Proof. Follows directly from Definitions 3 and 5. □

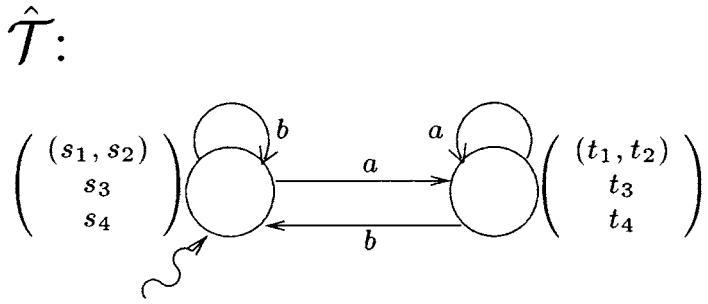

Figure 2. Team automaton $\hat{\mathcal{T}}$.

Note, however, that even though a team automaton over a composable system consisting of just one component automaton $\{\mathcal{C}_i\}$ is again a component automaton, such a team automaton is different from its only constituting component automaton. Even when Q_i and $\prod Q_i$ are identified, the transition relation of the team automaton may be properly included in the transition relation of the component automaton.

Example 2. (Example 1 continued) We form a team automaton (a car) over three component automata (an axle and two wheels).

For $i \in \{3, 4\}$, let $W_i = (\{s_i, t_i\}, (\{a, b\}, \varnothing, \varnothing), \delta_i, \{s_i\})$, in which $\delta_i = \{(s_i, b, s_i), (s_i, a, t_i), (t_i, a, t_i), (t_i, b, s_i)\}$, be two component automata modeling the third and the fourth **w**heel of a car. Note that in these automata, the actions a and b are input actions, whereas they are output actions in W_1 and W_2. The transition systems underlying W_3 and W_4 are as depicted in Figure 1(a).

Clearly $\{\mathcal{T}_{\{1,2\}}, W_3, W_4\}$, with $\mathcal{T}_{\{1,2\}}$ as in Example 1, is a composable system. A possible team automaton over this system is $\hat{\mathcal{T}}$, with $\hat{\delta} = \{(((s_1, s_2), s_3, s_4), b, ((s_1, s_2), s_3, s_4)), (((s_1, s_2), s_3, s_4), a, ((t_1, t_2), t_3, t_4)), (((t_1, t_2), t_3, t_4), a, ((t_1, t_2), t_3, t_4)), (((t_1, t_2), t_3, t_4), b, ((s_1, s_2), s_3, s_4))\}$. It is depicted in Figure 2. Note that fourteen states of $\hat{\mathcal{T}}$ are omitted from this drawing. $\hat{\mathcal{T}}$ has no other labeled transitions than those depicted. Hence none of the non-depicted states can ever be reached by a computation of $\hat{\mathcal{T}}$.

Now let us consider the aforementioned maximal interpretation of the components' involvements in team actions. Then a loop on b by $\hat{\mathcal{T}}$ means that each of the three components performs a loop on b (and likewise for action a). If we assume the same within the first component, which we recall to be a team over two component automata, then this means that the four wheels are now synchronized in such a way that they always move in unison. □

By focusing on a subset of the components in $\mathcal{S}$, a subteam within $\mathcal{T}$ can be distinguished. Its transitions are restrictions of the transitions of $\mathcal{T}$ to the components in the subteam. Its actions are of course the actions of the components involved. To allow for the use of the subteam as an independent team over a

subset of $\mathcal{S}$, its actions are classified without the context provided by $\mathcal{T}$. Hence, whether an action is input, output or internal for the subteam only depends on its role in the components forming the subteam rather than on how it is classified in $\mathcal{T}$. This means in particular that an action which is an output action of $\mathcal{T}$ is an input action for the subteam whenever this action is an input action of at least one of the components of the subteam and no components are considered which have this action as an output action.

Definition 6. Let $\mathcal{T} = (\prod_{i\in\mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i\in\mathcal{I}} I_i)$ be a team automaton over the composable system $\mathcal{S}$ and let $J \subseteq \mathcal{I}$. Then the *subteam of* $\mathcal{T}$ *determined by* J is denoted by $SUB_J(\mathcal{T})$ and is defined as
$SUB_J(\mathcal{T}) = (\prod_{j\in J} Q_j, (\Sigma_{J,inp}, \Sigma_{J,out}, \Sigma_{J,int}), \delta_J, \prod_{j\in J} I_j)$, where,
$\Sigma_{J,int} = \bigcup_{j\in J} \Sigma_{j,int}$,
$\Sigma_{J,out} = \bigcup_{j\in J} \Sigma_{j,out}$,
$\Sigma_{J,inp} = (\bigcup_{j\in J} \Sigma_{j,inp}) \setminus \Sigma_{J,out}$ and for all $a \in \Sigma_J = \bigcup_{j\in J} \Sigma_j$,
$(\delta_J)_a = \mathrm{proj}_J{}^{[2]}(\delta_a) \cap \Delta_a(\{\mathcal{C}_j | j \in J\})$. □

The transition relation of a subteam of $\mathcal{T}$ determined by some $J \subseteq \mathcal{I}$, is obtained by restricting the transition relation of $\mathcal{T}$ to synchronizations between the components in $\{\mathcal{C}_j | j \in J\}$. Hence in each transition of the subteam at least one of the component automata is actively involved. This is formalized by the intersection of $\mathrm{proj}_J{}^{[2]}(\delta_a)$ with $\Delta_a(\{\mathcal{C}_j | j \in J\})$, for each action a, as in each transition in this complete transition space at least one component from $\{\mathcal{C}_j | j \in J\}$ is active.

Note that if $J = \varnothing$, then $SUB_J(\mathcal{T})$ is the trivial automaton.

Clearly, the subteam $SUB_{\mathcal{I}}(\mathcal{T})$ of $\mathcal{T}$ determined by $\mathcal{I}$, i.e. by all components, is the team itself. However, a subteam determined by a single component $j \in \mathcal{I}$ in general differs from $\mathcal{C}_j$, even if the difference between Q_j and $\prod Q_j$ is ignored. This is due to the possibility that within $\mathcal{T}$ not all transitions from the complete transition spaces $\Delta_a(\mathcal{S})$ are used and hence some a-transitions from $\Delta_a(\{\mathcal{C}_j\})$ may be missing. Thus if $\mathcal{I} \neq \{j\}$, then for each action $a \in \Sigma_{\{j\}} = \Sigma_j$ we only have $\mathrm{proj}_j{}^{[2]}((\delta_{\{j\}})_a) \subseteq \delta_{j,a}$ and not necessarily an equality.

It is not hard to see that subteams satisfy the requirements of a team automaton.

Lemma 2. Let $\mathcal{T} = (\prod_{i\in\mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i\in\mathcal{I}} I_i)$ be a team automaton over the composable system $\mathcal{S}$ and let $J \subseteq \mathcal{I}$. Then
$SUB_J(\mathcal{T})$ is a team automaton over $\{\mathcal{C}_j | j \in J\}$.

Proof. We already noted that every subset of a composable system is again a composable system. The states, initial states and alphabets of $SUB_J(\mathcal{T})$ as given in Definition 6 satisfy the requirements of Definition 5 for teams over $\{\mathcal{C}_j | j \in J\}$. Finally, $(\delta_J)_a \subseteq \Delta_a(\{\mathcal{C}_j | j \in J\})$ by Definition 6. □

Thus, given a subteam $SUB_J(\mathcal{T})$ of $\mathcal{T}$, we may write $a\ \mathrm{en}_{SUB_J(\mathcal{T})}\ q$ if action a is *enabled* (in $SUB_J(\mathcal{T})$) at state $q \in \prod_{j\in J} Q_j$ of $SUB_J(\mathcal{T})$.

It is straightforward to extend the projection of computations of $\mathcal{T}$ on a component $\mathcal{C}_j$ to projection on a subteam $SUB_J(\mathcal{T})$ (it suffices to use proj_J instead of proj_j). Also in this case the projection on $SUB_J(\mathcal{T})$ of a team computation yields computations of $SUB_J(\mathcal{T})$.

According to Lemma 2 a subteam of a team automaton is again a team automaton and thus, by Lemma 1, also a component automaton. In the next section we will consider the dual approach and use team automata as component automata in 'larger' team automata. It will be shown that subteams can be used as components to iteratively define the team automaton they are derived from.

4. Teams over teams

In this section we show that team automata are naturally suited to describe hierarchical systems by demonstrating how to iteratively build teams from teams, and how to consider subteams as components in an iterated definition of a team automaton.

Given a composable system $\mathcal{S}$, there may be several ways of forming a team over $\mathcal{S}$. Until now we directly defined teams over $\mathcal{S}$, but other routes are also feasible. We might first (iteratively) form teams from (disjoint) subsets of $\mathcal{S}$ and then use these as components for a higher-level team, until after a finite number of such iterations all components from $\mathcal{S}$ have been used. This is shown in Example 2 and Figure 2, where four wheels are combined by first connecting two of them (to form an axle) and then attaching the other two to the result. This section shows that whatever route chosen, the resulting *iterated* team can always be regarded as a team over $\mathcal{S}$: it will always have the same alphabet of actions – including the distribution over input, output and internal actions – and it will have essentially the same state space, transition space and set of initial states, as any team formed directly over $\mathcal{S}$.

We will shortly begin this section by proving that the composability condition is preserved in the process of iteration. However, first we make the following remark.

Remark. To improve the readability of this section we have omitted some formal definitions and the longer, more technical proofs. The interested reader can find them in the Appendix.

Lemma 3. Let $\mathcal{S}$ be a composable system. Let $\{\mathcal{I}_j | j \in \mathcal{J}\}$, where $\mathcal{J} \subseteq \mathbb{N}$, form a partition of $\mathcal{I}$. Let, for each $j \in \mathcal{J}$, $\mathcal{T}_j$ be a team automaton over $\mathcal{S}_j = \{\mathcal{C}_i | i \in \mathcal{I}_j\}$. Then

$\{\mathcal{T}_j | j \in \mathcal{J}\}$ is a composable system.

Proof. Denote for each $\mathcal{T}_j$, $j \in \mathcal{J}$, by Γ_j its set of actions and by $\Gamma_{j,int}$ its internal alphabet. By Definition 5, we have $\Gamma_{j,int} = \bigcup_{i \in \mathcal{I}_j} \Sigma_{i,int}$ and $\Gamma_j = \bigcup_{i \in \mathcal{I}_j} \Sigma_i$, for all $j \in \mathcal{J}$. By the composability of $\mathcal{S}$, $\Sigma_{i,int} \cap \bigcup_{l \in \mathcal{I} \setminus \{i\}} \Sigma_l = \varnothing$, for

all $i \in \mathcal{I}$. Since the $\mathcal{I}_j$ are mutually disjoint it now follows immediately that, for all $j \in \mathcal{J}$, $\Gamma_{j,int} \cap \bigcup_{l \in \mathcal{J} \setminus \{j\}} \Gamma_l = \varnothing$. Hence $\{\mathcal{T}_j | j \in \mathcal{J}\}$ is a composable system. □

Thus, given a composable system, one may form teams over disjoint subsets of the system. These teams together with the component automata not involved in any of these teams are, by Lemma 3, again a composable system, which can then be used as the basis for the formation of still higher-level teams.

Example 3. Consider a composable system $\mathcal{S} = \{C_i | i \in [7]\}$, with $\mathcal{C}_i = (Q_i, (\Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}), \delta_i, I_i)$, for $i \in [7]$. Let $\mathcal{T}_{1-7} = (\prod_{i\in[7]} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i\in[7]} I_i)$ be a team automaton over $\mathcal{S}$. As δ is not relevant for the moment, it is not specified any further. Recall that all other parameters of $\mathcal{T}_{1-7}$ are uniquely defined by Definition 5. The structure of this team automaton relative to $\mathcal{S}$, is depicted in the tree of Figure 3(a).

Next consider the team automaton $\mathcal{T}_{\{2,4,6\}}$ over $\{\mathcal{C}_2, \mathcal{C}_4, \mathcal{C}_6\}$, and the team automaton $\mathcal{T}_{\{1,3,5\}}$ over $\{\mathcal{C}_1, \mathcal{C}_3, \mathcal{C}_5\}$. Let $\mathcal{T}_{\{2,4,6\}}$ be specified as $\mathcal{T}_{\{2,4,6\}} = (P'_1, (\Sigma'_{1,inp}, \Sigma'_{1,out}, \Sigma'_{1,int}), \delta'_1, I'_1)$ and $\mathcal{T}_{\{1,3,5\}}$ as $\mathcal{T}_{\{1,3,5\}} = (P'_2, (\Sigma'_{2,inp}, \Sigma'_{2,out}, \Sigma'_{2,int}), \delta'_2, I'_2)$. As proved in Lemma 3, these two teams form a composable system $\mathcal{S}' = \{\mathcal{C}'_1, \mathcal{C}'_2\}$, with $\mathcal{C}'_1 = \mathcal{T}_{\{2,4,6\}}$ and $\mathcal{C}'_2 = \mathcal{T}_{\{1,3,5\}}$. Let $\mathcal{T}'$ be a team automaton over $\mathcal{S}'$ and let $\mathcal{T}'$ be specified as $\mathcal{T}' = (P''_1, (\Sigma''_{1,inp}, \Sigma''_{1,out}, \Sigma''_{1,int}), \delta''_1, I''_1)$. This $\mathcal{T}'$ and $\mathcal{C}_7$ form again a composable system $\mathcal{S}'' = \{\mathcal{C}''_1, \mathcal{C}''_2\}$, with $\mathcal{C}''_1 = \mathcal{T}'$ and $\mathcal{C}''_2 = \mathcal{C}_7$. Let $\mathcal{T}''$ be a team automaton over $\mathcal{S}''$ and let $\mathcal{T}''$ be specified as $\mathcal{T}'' = (P'', (\Sigma''_{inp}, \Sigma''_{out}, \Sigma''_{int}), \delta'', I'')$, for some $\delta'' \subseteq P'' \times \Sigma'' \times P''$, with $\Sigma'' = \Sigma''_{inp} \cup \Sigma''_{out} \cup \Sigma''_{int}$.

By Definition 5, $P'' = P''_1 \times Q_7 = (\prod_{i\in\{1,2\}} P'_i) \times Q_7 = ((\prod_{i\in\{2,4,6\}} Q_i) \times (\prod_{i\in\{1,3,5\}} Q_i)) \times Q_7 = ((Q_2 \times Q_4 \times Q_6) \times (Q_1 \times Q_3 \times Q_5)) \times Q_7$. Similarly, $I'' = ((I_2 \times I_4 \times I_6) \times (I_1 \times I_3 \times I_5)) \times I_7$. Furthermore, $\Sigma''_{int} = \Sigma''_{1,int} \cup \Sigma_{7,int} = (\bigcup_{i\in\{1,2\}} \Sigma'_{i,int}) \cup \Sigma_{7,int} = ((\bigcup_{i\in\{2,4,6\}} \Sigma_{i,int}) \cup (\bigcup_{i\in\{1,3,5\}} \Sigma_{i,int})) \cup \Sigma_{7,int} = \bigcup_{i\in[7]} \Sigma_{i,int}$. Likewise, $\Sigma''_{out} = \bigcup_{i\in[7]} \Sigma_{i,out}$ and $\Sigma''_{inp} = (\bigcup_{i\in[7]} \Sigma_{i,inp}) \setminus \bigcup_{i\in[7]} \Sigma_{i,out}$.

Thus $\mathcal{T}''$ has the same actions as any team formed directly over $\mathcal{S}$. Its set of states, however, differs from the set of states of a team over $\mathcal{S}$ by its nested structure and its ordering. In Figure 3(b), the construction tree of $\mathcal{T}''$ is depicted.

In Figure 3(c), the construction tree of yet another route for constructing a team automaton, starting from the components in $\mathcal{S}$, is depicted. The set of states of this team $\mathcal{U}_6$ is $(((Q_1 \times Q_2) \times Q_3) \times (Q_7 \times Q_4)) \times (Q_6 \times Q_5)$. □

In order to describe in a precise way the relationship between a team obtained by iteratively forming teams over teams and a team formed directly from a given set of components, we need formal notions enabling us to describe the construction and the parsing of vectors with vectors as components. Let $\mathcal{D} = \{D_j | j \in J\}$ be an indexed set, with $J \subseteq \mathbb{N}$ and $J \neq \varnothing$. Then the set $\mathcal{V}(\mathcal{D})$ is defined as consisting of all finitely nested combinations of elements from $\mathcal{D}$ provided each D_j is used at most once. The domain of an element V from $\mathcal{V}(\mathcal{D})$ consists of the indices of

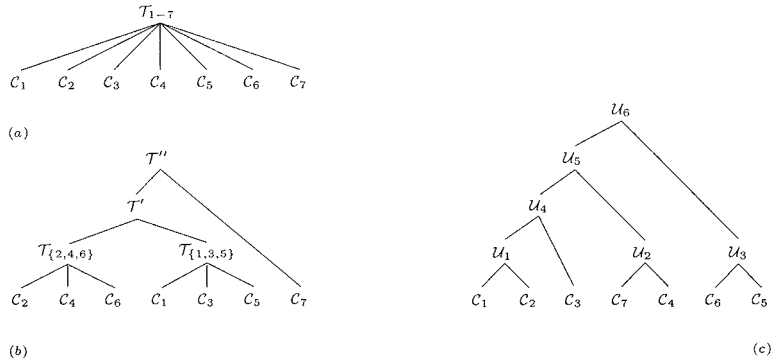

Figure 3. Three constructions of team automata starting from $\{C_i | i \in [7]\}$.

the sets in $\mathcal{D}$ combined to form V. (See the Appendix for the formal definition of $\mathcal{V}(\mathcal{D})$.) Every element of $\mathcal{V}(\mathcal{D})$ thus describes a finitely nested cartesian product of sets from $\mathcal{D}$, while its domain gives the information which D_j have been used.

Example 4. (Example 3 continued) Let $\mathcal{Q} = \{Q_i | i \in [7]\}$.
The set of states $P_2' = \prod_{i\in\{1,3,5\}} Q_i$ is an element of $\mathcal{V}(\mathcal{Q})$ with domain $\{1, 3, 5\}$.

Also $P_1'' = P_1' \times P_2' = \prod_{i\in\{2,4,6\}} Q_i \times \prod_{i\in\{1,3,5\}} Q_i$ is an element of $\mathcal{V}(\mathcal{Q})$. Its domain is $\{2, 4, 6\} \cup \{1, 3, 5\} = \{1, 2, 3, 4, 5, 6\}$.

For $P'' = P_1'' \times Q_7 \in \mathcal{V}(\mathcal{Q})$, $\mathrm{dom}(P_1'' \times Q_7) = \{1, 2, 3, 4, 5, 6, 7\}$. □

Given an element v of a nested cartesian product V from $\mathcal{V}(\mathcal{D})$ with domain $J' \subseteq J$, we want to rearrange v in such a way that the 'corresponding' element of $\prod_{j\in J'} D_j$ results. This *reordering* of an element $v \in V$ relative to the construction of V is denoted by $\langle v \rangle_V$. (See the Appendix for the formal definition of $\langle v \rangle_V$.)

Example 5. (Example 4 continued) Recall that $P'' = ((Q_2 \times Q_4 \times Q_6) \times (Q_1 \times Q_3 \times Q_5)) \times Q_7$. Assume that $q = (((x, m, l), (e, a, p)), e) \in P''$. Then the reordering $\langle q \rangle_{P''}$ of q relative to the construction of P'' is (e, x, a, m, p, l, e). □

The information about the construction of $V \in \mathcal{V}(\mathcal{D})$ is necessary in order to obtain a faithful reordering of the entries from $\bigcup_{j\in J} D_j$ in $\mathcal{V}$.

Example 6. Let $\mathcal{Q} = \{Q_i | i \in [3]\}$. Let $a \in Q_1$ and let $b, c \in Q_2 \cap Q_3$. Now assume we want to reorder $q = (a, (b, c))$. Then we need to know whether we are dealing with a construction $Q_1 \times (Q_2 \times Q_3) \in \mathcal{V}(\mathcal{Q})$, which would mean that the faithful reordering of q is (a, b, c), or with a construction $Q_1 \times (Q_3 \times Q_2) \in \mathcal{V}(\mathcal{Q})$, which would result in (a, c, b) as the faithful reordering of q. □

Only if $D_i \cap D_j = \varnothing$ for any two sets of states of a composable system, reordering could be simplified. This has never been a condition though.

Reordering all elements of a nested cartesian product V over sets from $\mathcal{D}$ (relative to the construction of V) results in the cartesian product (over sets from $\mathcal{D}$) according to J, as is formally stated in the following lemma.

Lemma 4. If $V \in \mathcal{V}(\mathcal{D})$ and $\mathrm{dom}(V) = J'$, then $\{\langle v \rangle_V | v \in V\} = \prod_{j \in J'} D_j$. □

Now we are ready to return to the issue of iteratively forming a team, given a composable system of team automata. We begin by generalizing the notion of a team automaton.

Definition 7. Let $\mathcal{S}$ be a composable system. Then $\mathcal{T}$ is an *iterated team automaton over* $\mathcal{S}$ if either
(1) $\mathcal{T}$ is a team automaton over $\mathcal{S}$ or
(2) $\mathcal{T}$ is a team automaton over $\{\mathcal{T}_j | j \in \mathcal{J}\}$, where each $\mathcal{T}_j$ is an iterated team automaton over $\{\mathcal{C}_i | i \in \mathcal{I}_j\}$, for some $\mathcal{I}_j \subseteq \mathcal{I}$, and $\{\mathcal{I}_j | j \in \mathcal{J}\}$ forms a partition of $\mathcal{I}$. □

We see that iterated team automata indeed are a generalization of team automata: every team over a given composable system may also be viewed as an iterated team over that system. But, as announced in the beginning of this section, teams formed iteratively over a composable system are essentially teams over that system. Intuitively, the only difference lies in the ordering and grouping of the components from the composable system. In the remainder of this section, we will formalize this statement.

The following lemma shows that the states and initial states of an iterated team over a composable system are *upto a reordering* the same as the states and initial states of any team over that system.

Lemma 5. Let $\mathcal{T} = (P, (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int}), \gamma, J)$ be an iterated team automaton over the composable system $\mathcal{S}$. Let $\mathcal{Q} = \{Q_i | i \in \mathcal{I}\}$. Then
(1) $P \in \mathcal{V}(\mathcal{Q})$ and $\mathrm{dom}(P) = \mathcal{I}$,
(2) $\{\langle q \rangle_P | q \in P\} = \prod_{i \in \mathcal{I}} Q_i$ and
(3) $\{\langle q \rangle_P | q \in J\} = \prod_{i \in \mathcal{I}} I_i$. □

Consequently we consider the actions and transitions of iterated teams. The actions of an iterated team over a composable system are the same as the actions of any team over that system. Furthermore, the transitions of any team over $\{\mathcal{T}_j | j \in \mathcal{J}\}$ are – after reordering – the transitions of a team over $\mathcal{S}$.

Lemma 6. Let $\mathcal{T} = (P, (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int}), \gamma, J)$ be an iterated team automaton over the composable system $\mathcal{S}$. Then
(1) $\Gamma_{int} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,int}$,
$\Gamma_{out} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,out}$, and
$\Gamma_{inp} = (\bigcup_{i \in \mathcal{I}} \Sigma_{i,inp}) \setminus \Gamma_{out}$ and
(2) $\{(\langle q \rangle_P, \langle q' \rangle_P) | (q, q') \in \gamma_a\} \subseteq \Delta_a(\mathcal{S})$, for all $a \in \Gamma_{inp} \cup \Gamma_{out} \cup \Gamma_{int}$. □

Note that Lemma 6(2) states that for each action a its complete transition space in $\{\mathcal{T}_j | j \in \mathcal{J}\}$ is included – after reordering – in its complete transition space

in $\mathcal{S}$. Iteration in the construction of a team thus does not lead to an increase of the possibilities for synchronization. In other words, every iterated team over a composable system can be interpreted as a team over that system by reordering its state space and its transition space.

Definition 8. Let $\mathcal{T} = (Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, I)$ be an iterated team automaton over the composable system $\mathcal{S}$. Then the team automaton $\langle\langle \mathcal{T} \rangle\rangle_{\mathcal{S}}$ over $\mathcal{S}$ is defined as

$$\langle\langle \mathcal{T} \rangle\rangle_{\mathcal{S}} = (\{\langle q \rangle_Q | q \in Q\}, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \{(\langle q \rangle_Q, a, \langle q' \rangle_Q) | q, q' \in Q,\ (q, a, q') \in \delta\}, \{\langle q \rangle_I | q \in I\}). \quad \square$$

From Lemmata 5 and 6 we conclude that $\langle\langle \mathcal{T} \rangle\rangle_{\mathcal{S}}$ is indeed a team automaton over $\mathcal{S}$ whenever $\mathcal{T}$ is an iterated team over $\mathcal{S}$. In fact, $\langle\langle \mathcal{T} \rangle\rangle_{\mathcal{S}}$ is the interpretation of $\mathcal{T}$ as a team automaton over $\mathcal{S}$ by reordering. Since their only difference is the ordering of the components of their state spaces, it is immediate that $\mathcal{T}$ and $\langle\langle \mathcal{T} \rangle\rangle_{\mathcal{S}}$ have *upto a reordering* the same computations.

Clearly, the converse of the inclusion of Lemma 6(2) does not hold in general since teams, and hence also iterated teams, are equipped with only a subset of all possible synchronizations. Moreover, a given intermediate team $\mathcal{T}_j$ over a subsystem $\mathcal{S}_j$ of $\mathcal{S}$ may have a transition relation that is properly included in the complete transition space of $\mathcal{S}_j$. As a consequence, a composable system $\{\mathcal{T}_j | j \in \mathcal{J}\}$ may provide less transitions for the forming of a team than $\{\mathcal{C}_i | i \in \mathcal{I}\}$ does. However, there is a natural condition that guarantees that for a given arbitrary team $\mathcal{T}$ over $\mathcal{S}$ and given iterated teams $\mathcal{T}_j$ over subsystems $\mathcal{S}_j = \{\mathcal{C}_i | i \in \mathcal{I}_j\}$, where the $\mathcal{I}_j$ form a partition of $\mathcal{I}$, one can still obtain a team $\hat{\mathcal{T}}$ over the composable system consisting of the $\mathcal{T}_j$, such that $\langle\langle \hat{\mathcal{T}} \rangle\rangle_{\mathcal{S}} = \mathcal{T}$. This condition requires that each of the $\mathcal{T}_j$ has at least all transitions – after reordering – of the corresponding subteam of $\mathcal{T}$ determined by $\mathcal{I}_j$. In fact, when loops are ignored, this is a necessary and sufficient condition for obtaining an iterated version of a given team over $\mathcal{S}$. Formally, we have the following result.

Theorem 1. Let $\mathcal{T} = (Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, I)$ be a team automaton over the composable system $\mathcal{S}$ and let $\{\mathcal{I}_j | j \in \mathcal{J}\}$, where $\mathcal{J} \subseteq \mathbb{N}$, form a partition of $\mathcal{I}$. Let, for each $j \in \mathcal{J}$, $\mathcal{T}_j = (P_j, (\Gamma_{j,inp}, \Gamma_{j,out}, \Gamma_{j,int}), \gamma_j, J_j)$ be an iterated team over $\{\mathcal{C}_i | i \in \mathcal{I}_j\}$. Then

(1) if $(\delta_{\mathcal{I}_j})_a \subseteq \{(\langle q \rangle_{P_j}, \langle q' \rangle_{P_j}) | (q, q') \in \gamma_{j,a}\}$, for all $a \in \Gamma_{j,inp} \cup \Gamma_{j,out} \cup \Gamma_{j,int}$ for all $j \in \mathcal{J}$, then there exists a team automaton $\hat{\mathcal{T}}$ over $\{\mathcal{T}_j | j \in \mathcal{J}\}$ such that $\langle\langle \hat{\mathcal{T}} \rangle\rangle_{\mathcal{S}} = \mathcal{T}$ and

(2) if $\hat{\mathcal{T}}$ is a team automaton over $\{\mathcal{T}_j | j \in \mathcal{J}\}$, then $\langle\langle \hat{\mathcal{T}} \rangle\rangle_{\mathcal{S}} = \mathcal{T}$ implies that $(\delta_{\mathcal{I}_j})_a \setminus \{(p, p) | (p, p) \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_j\})\} \subseteq \{(\langle q \rangle_{P_j}, \langle q' \rangle_{P_j}) | (q, q') \in \gamma_{j,a}\}$, for all $a \in \Gamma_{j,inp} \cup \Gamma_{j,out} \cup \Gamma_{j,int}$ for all $j \in \mathcal{J}$. $\square$

Thus, not only can every iterated team over $\mathcal{S}$ be considered as a team directly constructed from $\mathcal{S}$ by Definition 8, but according to Theorem 1 also every team

automaton can be iteratively constructed from its subteams. Consequently, both subteams and iterated teams can be treated as team automata including the considerations concerning their computations and behavior and it suffices to consider in the sequel only the relationship between subteams and team automata.

5. Synchronizations

An important – perhaps even the most important – feature of the team automata framework is the high level of flexibility that is obtained by leaving the set of transitions of a team automaton as a modeling choice. This choice for a specific interconnection strategy (which components synchronize on what actions, and when) is based on what one wants to model. In this section we discuss various natural types of synchronization within team automata.

Notation 2. In the sequel we assume that $\mathcal{S}$ is a composable system and fix a team automaton $\mathcal{T} = (\prod_{i\in\mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i\in\mathcal{I}} I_i)$ over $\mathcal{S}$, for which we let Σ denote $\Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}$ and $\Sigma_{ext} = \Sigma_{inp} \cup \Sigma_{out}$. The component automata $\mathcal{C}_i$ are as specified in Notation 1. □

First we focus on the individual actions of a team automaton and distinguish three different modes of synchronizing on an action. We consider actions that are never used in synchronizations between multiple components and actions on which all components having this action have to synchronize. The latter case is weakened by requiring participation only if the components are ready (in the right state) to execute that action.

Definition 9. The set of *free* actions of $\mathcal{T}$ is denoted by $Free(\mathcal{T})$ and is defined as

$$Free(\mathcal{T}) = \{a \in \Sigma | (q, q') \in \delta_a \Rightarrow \#\{i \in \mathcal{I} | a \in \Sigma_i \wedge \text{proj}_i{}^{[2]}(q, q') \in \delta_{i,a}\} = 1\}.$$

The set of *action-indispensable* (*ai* for short) actions of $\mathcal{T}$ is denoted by $AI(\mathcal{T})$ and is defined as

$$AI(\mathcal{T}) = \{a \in \Sigma | \forall i \in \mathcal{I} : (a \in \Sigma_i \wedge (q, q') \in \delta_a) \Rightarrow \text{proj}_i{}^{[2]}(q, q') \in \delta_{i,a}\}.$$

The set of *state-indispensable* (*si* for short) actions of $\mathcal{T}$ is denoted by $SI(\mathcal{T})$ and is defined as

$$SI(\mathcal{T}) = \{a \in \Sigma | \forall i \in \mathcal{I} : (a \in \Sigma_i \wedge (q, q') \in \delta_a \wedge a \text{ en}_{\mathcal{C}_i} \text{proj}_i(q)) \Rightarrow \text{proj}_i{}^{[2]}(q, q') \in \delta_{i,a}\}. \quad \square$$

Intuitively, an action a is a free action of $\mathcal{T}$ if no a-transition of $\mathcal{T}$ is brought about by a simultaneous execution of a by two or more components. Thus, whenever a is executed by $\mathcal{T}$ only one component is active in this execution.

If a is action-indispensable, all components which have a as one of their actions are involved in every execution of a by $\mathcal{T}$. This means that $\mathcal{T}$ cannot perform an a

if one of the components to which a belongs, is not ready for it (a is not enabled in that component at the current local state).

Finally, state-indispensable is a weak version of action-indispensable: if a is state-indispensable, all executions of a by $\mathcal{T}$ involve all components in which a is currently enabled. In this case $\mathcal{T}$ does not have to 'wait' with the execution of a until all components of which a is an action are ready for it.

Recall that information on the actual execution of loops by components is missing in the transition relation of a team automaton. Therefore, in Definition 9 and its intuitive explanation, the presence of loops on a in components is treated as if a is actually executed, which is in accordance with the maximal interpretation of the components' involvements adopted before.

It is immediate that every action that is *ai* in $\mathcal{T}$ also satisfies the weaker requirement of being *si*. Furthermore, in case a is an internal action of one of the components, it is not an action of any other component and thus it is trivially *free*, *ai* and *si*.

Lemma 7. (1) $AI(\mathcal{T}) \subseteq SI(\mathcal{T})$, and (2) $\Sigma_{int} \subseteq Free(\mathcal{T}) \cap AI(\mathcal{T})$.

Proof. We only prove (2). Let $a \in \Sigma_{int}$. From Definition 3 it follows that, for all $(q, q') \in \delta_a$, there exists a unique $i \in \mathcal{I}$ such that $(\text{proj}_i(q), a, \text{proj}_i(q')) \in \delta_i$. Hence, all cases in Definition 9 coincide, which completes the proof. □

As shown in the following example, Lemma 7(1) describes the only dependency among *free*, *ai* and *si*.

Example 7. The combination of the properties of being *free*, *ai* and *si*, leads in principle to eight different types of actions in a team automaton. However, by Lemma 7(1), *ai* implies *si*, which eliminates the combinations ⟨*free*, *ai*, not *si*⟩ and ⟨not *free*, *ai*, not *si*⟩. Each of the remaining combinations is feasible, as we now demonstrate.

Let $\mathcal{C}_1 = (\{q, q'\}, (\varnothing, \{a\}, \varnothing), \{(q, a, q')\}, \{q\})$ and $\mathcal{C}_2 = (\{r, r'\}, (\varnothing, \{a\}, \varnothing), \{(r, a, r')\}, \{r\})$ be as depicted in Figure 4(a).

They clearly form a composable system, as both have an empty alphabet of internal actions. From this composable system we construct five team automata $\mathcal{T}^i = (\{(q, r), (q, r'), (q', r), (q', r')\}, (\varnothing, \{a\}, \varnothing), \delta^i, \{(q, r)\}), i \in [5]$, where

- $\delta^1 = \{((q, r), a, (q', r)), ((q, r), a, (q', r'))\}$; now a is not *free*, since both automata execute a in the second transition, while a is not *si*, and thus also not *ai*, since $\mathcal{C}_2$ does not execute a in the first transition although it is in a state ready to perform a,
- $\delta^2 = \{((q, r'), a, (q', r')), ((q, r), a, (q', r'))\}$; now a is not *free* because of the second transition, it is not *ai* because of the first transition, but a is *si*, since both automata take part in the execution of a whenever they are ready to do so,

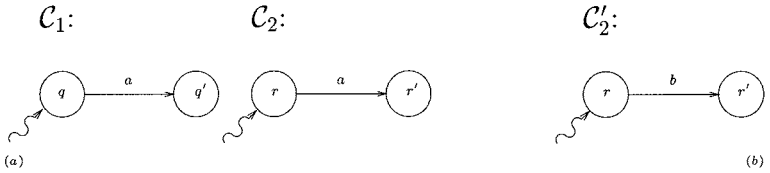

Figure 4. Component automata $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_2'$.

$\delta^3 = \{((q, r), a, (q', r'))\}$; now a is not *free*, since in the given transition a is executed by both automata, which also implies that a is *ai* and thus *si*,

$\delta^4 = \{((q, r), a, (q', r))\}$; now a is *free*, since only one automaton is involved in the a-transition, but a is not *si* and thus also not *ai*, since $\mathcal{C}_2$ does not synchronize on a even though it is ready to perform a,

$\delta^5 = \{((q, r'), a, (q', r'))\}$; now a is *free* for the same reason as in the previous case, a is not *ai*, since $\mathcal{C}_2$ does have a in its alphabet but does not execute a, and a is *si* since the second automaton cannot execute a in state r'.

The only case that remains to be illustrated by an example team automaton is the case in which a set of actions is *free*, *ai* and *si* at the same time. This case cannot occur in any team automaton over $\{\mathcal{C}_1, \mathcal{C}_2\}$ with a nonempty transition relation, because if a is *free* then only one automaton performs a, whereas with $a \in \Sigma_1 \cap \Sigma_2$ the action a can only be *ai* if $\mathcal{C}_1$ and $\mathcal{C}_2$ perform a together.

However, in the team automaton $\mathcal{C}_1$ (or $\mathcal{C}_2$) a is by definition *free*, *ai* and *si*. Moreover, the team automaton $\mathcal{T}'$ over $\{\mathcal{C}_1, \mathcal{C}_2'\}$, with $\mathcal{C}_2' = (\{r, r'\}, (\varnothing, \{b\}, \varnothing), \{(r, b, r')\}, \{r\})$ – depicted in Figure 4(b) – and defined by $\mathcal{T}' = (\{(q, r), (q, r'), (q', r), (q', r')\}, (\varnothing, \{a, b\}, \varnothing), \gamma, \{(q, r)\})$, with $\gamma = \delta^4$, is yet another example of a team automaton in which a is *free*, *ai* and *si*. □

As we show next, the property of an action a being *free*, *ai*, or *si* in $\mathcal{T}$ is inherited by all subteams of $\mathcal{T}$ which have a as one of their actions.

Lemma 8. Let $J \subseteq \mathbb{1}$, $a \in \Sigma_J = \bigcup_{j \in J} \Sigma_j$ and $X \in \{Free, AI, SI\}$. Then
if $a \in X(\mathcal{T})$, then $a \in X(SUB_J(\mathcal{T}))$.

Proof. The set of a-transitions of $SUB_J(\mathcal{T})$, $(\delta_J)_a$, consists of a-transitions of $\mathcal{T}$ projected on the components determined by J. Thus, whenever $(q, q') \in (\delta_J)_a$, then there exists a $(p, p') \in \delta_a$ such that $\text{proj}_J{}^{[2]}(p, p') = (q, q')$.

Let $X = Free$. Assume $a \notin Free(SUB_J((\mathcal{T}))$. Then there exists a $(q, q') \in (\delta_J)_a$ such that $\#\{i \in J | \text{proj}_i{}^{[2]}(q, q') \in \delta_{i,a}\} > 1$. Using the observation above, we thus conclude that then there exists a $(p, p') \in \delta_a$ that also violates the requirement for the set of free actions.

In a similar fashion we can prove the lemma for $X = AI$ and for $X = SI$, i.e. if there exists a transition in $SUB_J(\mathcal{T})$ violating the *ai* (*si*) requirement for a, then the observation implies that there exists an 'extension' of this transition in $\mathcal{T}$ that also violates the *ai* (*si*) requirement for a. □

The converse of this lemma does not hold in general, as can be concluded from Example 7, where the action a is neither *free*, nor *ai* nor *si* in the team automaton $\mathcal{T}^1$, although it is *free*, *ai* and *si* in its subteam determined by {2}, which is a copy of $\mathcal{C}_2$. The reason for this resides in the fact that an action a (that is not *free* in a team automaton $\mathcal{T}$) is *free* in a subteam of $\mathcal{T}$, if the restriction to a subset of the components leads to dropping those components from $\mathcal{T}$ that caused a not to be *free* in $\mathcal{T}$. The same reasoning can be applied in case a is *ai* or *si*.

The converse of Lemma 8, however, does hold if a is an internal action. This is due to the fact that always only one component automaton is involved in the execution of an internal action. More general, whenever an action only belongs to components which are included in a subteam, the properties of being *free*, *ai* or *si* are carried over from that subteam to the team as a whole. This leads to the following result.

Lemma 9. Let $J \subseteq \mathcal{I}$, $a \in \Sigma_J = (\bigcup_{j\in J} \Sigma_j) \setminus (\bigcup_{i\in\mathcal{I}\setminus J} \Sigma_i)$ and $X \in \{Free, AI, SI\}$. Then

$a \in X(\mathcal{T})$ if and only if $a \in X(SUB_J(\mathcal{T}))$.

Proof. By Lemma 8, we only have to prove the 'if' direction.

By the condition on a it follows that $(p, p') \in \delta_a$ implies $\text{proj}_J{}^{[2]}(p, p') \in (\delta_J)_a$ and that, for all $i \in \mathcal{I} \setminus J$, $a \notin \Sigma_i$.

Assume $a \notin X(\mathcal{T})$. Then there exists a transition $(p, p') \in \delta_a$ that violates the requirement for a to be in $X(\mathcal{T})$, and from the observation above we conclude that this violation occurs in the J part, i.e. in $SUB_J(\mathcal{T})$. Hence $\text{proj}_J{}^{[2]}(p, p')$ is a transition in $SUB_J(\mathcal{T})$ violating the requirement for a to be in $X(SUB_J(\mathcal{T}))$. □

To conclude this section, we observe that in a team automaton where every action is *ai*, every transition involves all components that have the action to be executed in their alphabet. This implies that for such automata we can extend the earlier observation on computations to languages: from the language of a team, a language of a component (subteam) can immediately be extracted by deleting all actions that do not belong to that component (or subteam)! In fact, this is not restricted to languages but holds for any behavioral notion obtained by preserving only a specific type of actions (e.g. only external actions). Note that this is not possible when the execution of an action does not involve all components to which it belongs.

6. Peer-to-peer and master-slave synchronizations

Until now we have discussed synchronizations while ignoring whether the action was input, output or internal in certain components. For internal actions which belong to only one component, distinguishing between their roles in different

components is indeed not very relevant. External actions, however, may be input to some components, and output to other components. We now investigate synchronizations relating to the different roles an action may have in different components.

First we separate the output role of external actions from their input role. Given an external action, we locate its input and output domain within $\mathcal{I}$, and then use these domains to define subteams. Finally, we define two specific modes of synchronization relating the input subteam and the output subteam.

Definition 10. Let $a \in \Sigma_{ext}$. Then
(1) $\mathcal{I}_{a,inp}(\mathcal{S}) = \{j \in \mathcal{I} | a \in \Sigma_{j,inp}\}$ is the *input domain* of a in $\mathcal{S}$ and
(2) $\mathcal{I}_{a,out}(\mathcal{S}) = \{j \in \mathcal{I} | a \in \Sigma_{j,out}\}$ is the *output domain* of a in $\mathcal{S}$. □

No external action of any team automaton $\mathcal{T}$ will ever be both an input and an output action for one component. Thus, for each $j \in \mathcal{I}$, $\Sigma_{j,inp} \cap \Sigma_{j,out} = \varnothing$, and consequently $\mathcal{I}_{a,inp}(\mathcal{S}) \cap \mathcal{I}_{a,out}(\mathcal{S}) = \varnothing$ for all $a \in \Sigma_{ext}$. That is, the input domain and the output domain of an external action are always disjoint.

Note that, by Definition 5, $a \in \Sigma_{out}$ if and only if $\mathcal{I}_{a,out}(\mathcal{S}) \neq \varnothing$, while $a \in \Sigma_{inp}$ if and only if $\mathcal{I}_{a,inp}(\mathcal{S}) \neq \varnothing$ and $\mathcal{I}_{a,out}(\mathcal{S}) = \varnothing$.

In order to simplify notation, we make no explicit references to $\mathcal{S}$ as this is the fixed composable system we are working with. Furthermore, for all $a \in \Sigma_{ext}$, we use $SUB_{a,inp}(\mathcal{T})$ to denote $SUB_{\mathcal{I}_{a,inp}}(\mathcal{T})$, the *input subteam* of a (in $\mathcal{T}$), and we use $SUB_{a,out}(\mathcal{T})$ to denote $SUB_{\mathcal{I}_{a,out}}(\mathcal{T})$, the *output subteam* of a (in $\mathcal{T}$). If no confusion arises we even omit the $\mathcal{T}$ and simply write $SUB_{a,inp}$ and $SUB_{a,out}$.

Note that the input domain and the output domain of an external action may be empty. However, for every external action, at least one of these domains is nonempty. In case the input (output) domain is empty, then its input (output) subteam is the trivial automaton.

Example 8. In Figure 5, the structure of the team automaton $\mathcal{T}$, with respect to one of its external actions a, is depicted. Indicated are the input subteam $SUB_{a,inp}$ and the output subteam $SUB_{a,out}$ in $\mathcal{T}$. In this figure, the square boxes denote component automata. $\mathcal{T}$ may also contain components that do not have a as an external action. □

Having determined for each external action a its input and its output subteam, we can now identify certain modes of synchronization relating to a in its role as input or output action. First we look within these subteams in which by definition a has only one role and all components are peers, in the sense that they are on an equal footing with respect to a. We say that an input (output) action a is input (output) peer-to-peer, if every execution of a involving components of that subteam requires the participation of all.

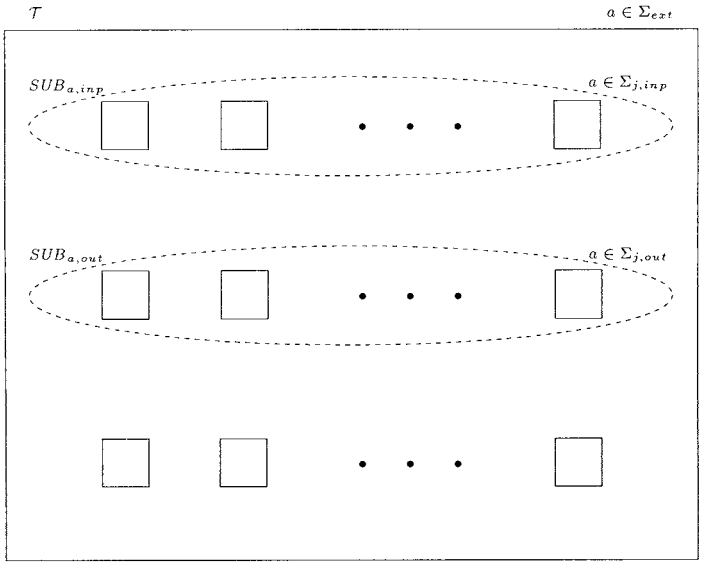

Figure 5. A team automaton $\mathcal{T}$ with subteams $SUB_{a,inp}$ and $SUB_{a,out}$.

This obligation to participate can be explained in a strong and in a weak sense. Strong simply means that no synchronizations on a can take place unless all components in the input (output) domain of a take part. Weak means that synchronizations on a involve all of the components in the input (output) domain of a which are ready to execute a (in a state in which a is enabled). Thus the notion of strong requires that a is *ai* in its input (output) subteam, while the notion of weak requires that a is *si* in its input (output) subteam.

This is formally defined as follows.

Definition 11. Let $a \in \Sigma_{ext}$. Then
(1) a is *strong input peer-to-peer* (in $\mathcal{T}$) if $a \in AI(SUB_{a,inp})$,
(2) a is *weak input peer-to-peer* (in $\mathcal{T}$) if $a \in SI(SUB_{a,inp})$,
(3) a is *strong output peer-to-peer* (in $\mathcal{T}$) if $a \in AI(SUB_{a,out})$ and
(4) a is *weak output peer-to-peer* (in $\mathcal{T}$) if $a \in SI(SUB_{a,out})$. □

We should remark here that an external action a that is not an input action (implying that $\mathcal{I}_{a,inp} = \varnothing$ and that $SUB_{a,inp}$ is the trivial automaton) cannot be strong or weak input peer-to-peer (since the trivial automaton has no actions, and thus no *ai* or *si* actions). Similarly, if a is strong or weak output peer-to-peer in $\mathcal{T}$, then it must be the case that it is an output action in at least one component.

Note that whenever an action is strong input (output) peer-to-peer in a team automaton, then by Lemma 7(1) it is also weak input (output) peer-to-peer in that team automaton. This does not hold the other way around, however, as can be deduced from Example 7.

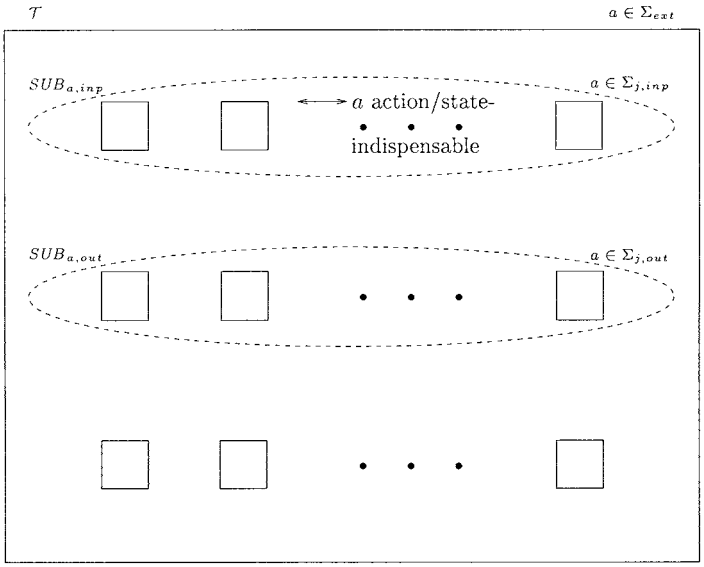

Figure 6. A team automaton $\mathcal{T}$ with a strong/weak input peer-to-peer action a.

Example 9. (Example 8 continued) As depicted in Figures 6 and 7, strong and weak input (output) peer-to-peer synchronizations relate to synchronizations within the corresponding input (output) subteam. □

Next we define synchronizations between the input and output subteams of the external action a. Here the idea is that input actions ('slaves') are driven by output actions ('masters'). This means that if a is an output action, then its input counterpart can never take place without being triggered (the slave never proceeds on its own). Consequently, the input subteam of an output action a cannot execute a unless a is also executed as an output action (by its output subteam). It is however possible that a is executed as an output action without its simultaneous execution as an input action. We say that a is *master-slave* if it is an output action and its output subteam participates in every a-transition of $\mathcal{T}$.

In addition one could require that a in its role of input action *has to* synchronize with a as an output action (the slave has to follow the master). Since the obligation of the slave to follow the master may again be formulated in two different ways, we obtain notions of strong and weak master-slave actions. When guided by the *ai* principle, we get a strong notion of master-slave synchronization, while the *si* principle leads to a weak notion of master-slave synchronization. We say that a is *strong master-slave* if it is master-slave and its input subteam moreover participates in every a-transition of $\mathcal{T}$. We call *a weak master-slave* if it is master-slave and its input subteam moreover participates in every a-transition of $\mathcal{T}$ whenever it can.

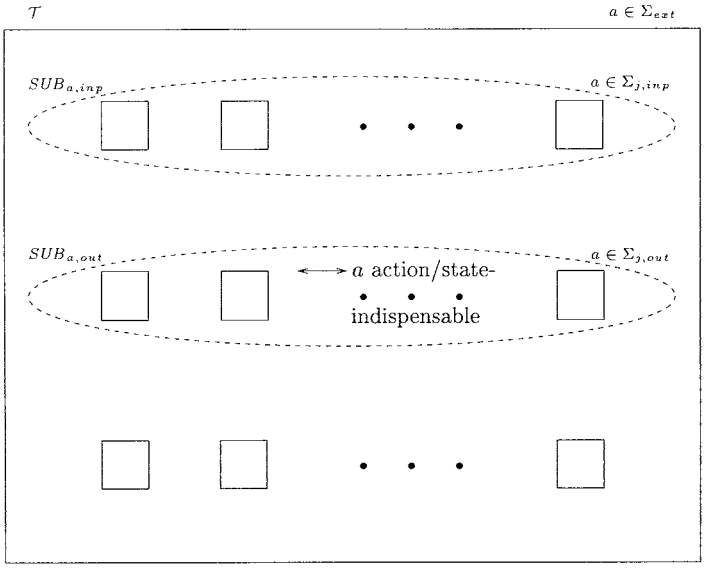

Figure 7. A team automaton $\mathcal{T}$ with a strong/weak output peer-to-peer action a.

Definition 12. Let $a \in \Sigma_{out}$, and let $J = \mathbb{1}_{a,out}$ and $K = \mathbb{1}_{a,inp}$. Then
(1) a is *master-slave* (in $\mathcal{T}$) if
$\text{proj}_J{}^{[2]}(\delta_a) \subseteq (\delta_J)_a$,
(2) a is *strong master-slave* (in $\mathcal{T}$) if
(1) holds, and $K \neq \varnothing$ implies that $\text{proj}_K{}^{[2]}(\delta_a) \subseteq (\delta_K)_a$, and
(3) a is *weak master-slave* (in $\mathcal{T}$) if
(1) holds, and $K \neq \varnothing$ implies that
$((q, q') \in \delta_a \wedge a \text{ en}_{SUB_K} \text{proj}_K(q)) \Rightarrow \text{proj}_K{}^{[2]}(q, q') \in (\delta_K)_a$. □

Note that if an action is strong master-slave in a team automaton, then it is also weak master-slave.

For a to be master-slave, we require it to occur at least once as an output action ($\mathbb{1}_{a,out} \neq \varnothing$), i.e. a can act as a master. Otherwise we could have slaves without a master. A master without slaves is allowed: $\mathbb{1}_{a,out} \neq \varnothing$ and $\mathbb{1}_{a,inp} = \varnothing$. In that case, a is trivially strong and weak master-slave, since there are no slaves that do not follow the master.

Note that in the master-slave definitions above, input subteams and output subteams are treated as given entities (black boxes). Clearly, one can combine the master-slave synchronizations with additional requirements on the synchronizations taking place within the subteams. Thus one may prescribe a master-slave synchronization on an action a which is in addition input peer-to-peer. Then *all* components with a as an input have to follow the output.

Example 10. (Example 8 continued) If for an external action a of $\mathcal{T}$, $SUB_{a,out}$ is involved in all a-transitions of $\mathcal{T}$, then a is a master-slave action. If, moreover,

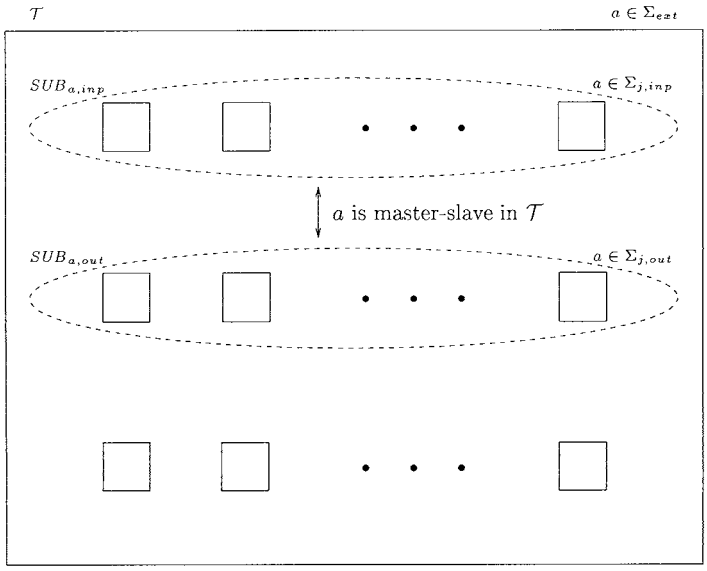

Figure 8. A team automaton $\mathcal{T}$ with a master-slave synchronization a.

$SUB_{a,inp}$ 'has to' participate in every a-transition of $\mathcal{T}$, then a is a strong or weak master-slave action in $\mathcal{T}$. The idea of master-slave synchronization is sketched in Figure 8.

One can thus see that whereas every peer-to-peer synchronization is defined within subteams, master-slave synchronizations are defined between input and output subteams. □

Since the definition of a being master-slave in $\mathcal{T}$ guarantees that the output subteam of a is actively involved in every a-transition of $\mathcal{T}$, it follows immediately from Definition 6 that the a-transitions of the output subteam of a are precisely the projections of the a-transitions of $\mathcal{T}$ on the output domain of a. Similarly, in case a is strong master-slave we have in addition that the a-transitions of the input subteam of a are precisely the projections of the a-transitions of $\mathcal{T}$ on the input domain of a. This leads to the following lemma.

Lemma 10. (1) Let a be master-slave in $\mathcal{T}$ and let $J = \mathcal{I}_{a,out}$. Then
$\text{proj}_J{}^{[2]}(\delta_a) = (\delta_J)_a$.
(2) Let a be strong master-slave in $\mathcal{T}$ and let $K = \mathcal{I}_{a,inp}$. Then
$\text{proj}_K{}^{[2]}(\delta_a) = (\delta_K)_a$.

Proof. (1) By Definition 6, $(\delta_J)_a = \text{proj}_J{}^{[2]}(\delta_a) \cap \Delta_a(\{\mathcal{C}_j | j \in J\})$. Since a is master-slave, we have $\text{proj}_J{}^{[2]}(\delta_a) \subseteq (\delta_J)_a$, for $J = \mathcal{I}_{a,out}$. Hence in this case $(\delta_J)_a = \text{proj}_J{}^{[2]}(\delta_a)$.

(2) Analogous to (1). Note $K = \varnothing$ implies $\text{proj}_K{}^{[2]}(\delta_a) = \varnothing = (\varnothing)_a$. □

Note that in case a is weak master-slave, there may be a-transitions in $\mathcal{T}$ in which the input subteam, even when it is not trivial, is not actively involved. In those cases, a is executed as an output action by $\mathcal{T}$ without simultaneous execution of a as an input action.

Example 11. (Example 2 continued) In this example we show that the car $\hat{\mathcal{T}}$ is actually a two wheel drive. Recall that we assume a maximal interpretation of the components' involvements in loops.

We now set $\mathcal{T}_1 = \mathcal{T}_{\{1,2\}}$, $\mathcal{T}_2 = W_3$ and $\mathcal{T}_3 = W_4$. Then $\hat{\mathcal{T}}$ is a team automaton over $\{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3\}$. Actions a and b are output actions in $\mathcal{T}_1$, whereas they are input actions in both $\mathcal{T}_2$ and $\mathcal{T}_3$. Hence, $\mathcal{U}_{a,out} = \{1\}$ and $\mathcal{U}_{a,inp} = \{2, 3\}$.

Actions a and b are both strong master-slave in $\hat{\mathcal{T}}$. For a, this can be concluded from $\text{proj}_{\{1\}}{}^{[2]}(\hat{\delta}_a) = \{((s_1, s_2), (t_1, t_2)), ((t_1, t_2), (t_1, t_2))\} = (\hat{\delta}_{\{1\}})_a$ and $\text{proj}_{\{2,3\}}{}^{[2]}(\hat{\delta}_a) = \{((s_3, s_4), (t_3, t_4)), ((t_3, t_4), (t_3, t_4))\} = (\hat{\delta}_{\{2,3\}})_a$ (thus satisfying (1) and (2) of Definition 12), whereas one can verify this for b in a similar fashion. We thus conclude that $\hat{\mathcal{T}}$ models a two wheel drive, in the sense that one axle (the input subteam of a and b) only turns and halts as a reaction to the other axle (the output subteam of a and b), i.e. the first axle is the 'slave' of the latter axle.

It is easy to model a four wheel drive (using two new wheels, though) by making a and b output actions rather than input actions in W_3 and W_4. Then all four wheels always turn or halt in unison. Actions a and b still are master-slave. Moreover, it is easy to see that actions a and b are both strong (and thus also weak) output peer-to-peer in $\hat{\mathcal{T}}'$.

Finally, if we would want to model a car more faithfully, we would allow the front axle to turn and halt without having the back axle doing the same. It is left to the reader to choose appropriate wheels and an appropriate transition relation, and to consequently also construct a team automaton modeling such a car. □

7. A case study

In Ellis (1997), a simple example was presented to illustrate the concept of team automata. Here a rigorous treatment of this example in the new formal framework is given.

Consider the three component automata depicted in Figure 9. Formally, $\mathcal{C}_i = (Q_i, (\Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}), \delta_i, I_i)$ where, for $i \in [3]$,

$Q_i = \{q_i, q_i'\}$,
$\Sigma_{1,inp} = \Sigma_{2,inp} = \Sigma_{3,out} = \varnothing$,
$\Sigma_{1,out} = \Sigma_{2,out} = \Sigma_{3,inp} = \{b\}$,
$\Sigma_{i,int} = \{a_i, a_i'\}$, with all a_i, a_i' distinct symbols different from b,
$\delta_{i,b} = \{(q_i, q_i')\}$,
$\delta_{j,a_j} = \{(q_j, q_j')\}$ and $\delta_{j,a_j'} = \{(q_j', q_j)\}$, for $j \in [2]$,

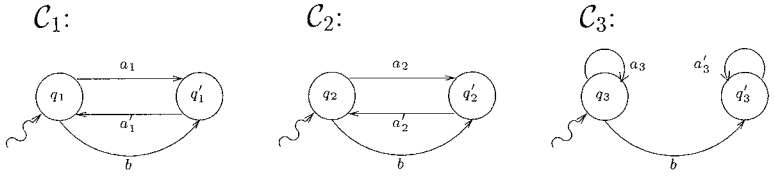

Figure 9. Component automata $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$.

$\delta_{3,a_3} = \{(q_3, q_3)\}$ and $\delta_{3,a_3'} = \{(q_3', q_3')\}$ and
$I_i = \{q_i\}$.

Hence $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ is a composable system.

Two slightly different team automata $\mathcal{T}$ and $\mathcal{T}'$ over this system are defined next. All parameters of these teams, except for the set of labeled transitions, are predetermined by $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$. In fact, only the b-transitions can be varied as all the other actions are internal. The first team automaton ($\mathcal{T}$) is the one spelled out in Ellis (1997), whereas the second one ($\mathcal{T}'$) is the one discussed in the text in Ellis (1997).

Let $\mathcal{T} = (\prod_{i\in[3]} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \{(q_1, q_2, q_3)\})$ and let $\mathcal{T}' = (\prod_{i\in[3]} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta', \{(q_1, q_2, q_3)\})$, where

$\Sigma_{inp} = \varnothing$,
$\Sigma_{out} = \{b\}$,
$\Sigma_{int} = \{a_1, a_1', a_2, a_2', a_3, a_3'\}$ and
δ and δ' are defined by
$\delta_a = \delta_a' = \Delta_a(\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\})$, for each $a \in \{a_1, a_1', a_2, a_2', a_3, a_3'\}$,
$\delta_b = \{((q_1, q_2, q_3), (q_1', q_2', q_3'))\}$ and
$\delta_b' = \{((q_1, q_2, q_3), (q_1', q_2', q_3')), ((q_1, q_2, q_3'), (q_1', q_2', q_3'))\}$.

Thus in $\mathcal{T}$ there is only one b-transition that can take place, and it involves all three components by requiring the i-th component to be in state q_i, for each $i \in [3]$, i.e. this transition is a simultaneous execution of b by all three components. In $\mathcal{T}'$, however, next to this b-transition just described, there is another b-transition that can take place and it involves only the first two components, while the third component is in state q_3', in which b is not enabled. Hence, this transition is a simultaneous execution of b by the first two components only.

It is easy to check that $Free(\mathcal{T}) = Free(\mathcal{T}') = AI(\mathcal{T}') = \Sigma_{int}$ and $AI(\mathcal{T}) = SI(\mathcal{T}) = SI(\mathcal{T}') = \Sigma$. Thus in $\mathcal{T}$, b is both *si* and *ai*, while in $\mathcal{T}'$, b is *si* but not *ai*. This is because $\mathcal{T}'$ has a b-transition in which $\mathcal{C}_3$ does not participate, even though $\mathcal{C}_3$ contains b in its (input) alphabet.

Note that in $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$, the input domain $\mathcal{I}_{b,inp}$ of b is $\{3\}$ and the output domain $\mathcal{I}_{b,out}$ of b is $\{1, 2\}$. The subteams of $\mathcal{T}$ and $\mathcal{T}'$ determined by $\{1, 2\}$ are exactly the same: $SUB_{\{1,2\}}(\mathcal{T}) = SUB_{\{1,2\}}(\mathcal{T}')$. This is because $\text{proj}_{\{1,2\}}{}^{[2]}(\delta_c) = \text{proj}_{\{1,2\}}{}^{[2]}(\delta_c')$, for each $c \in \{a_1, a_1', a_2, a_2', b\}$. Also $SUB_{\{3\}}(\mathcal{T}) = SUB_{\{3\}}(\mathcal{T}')$, since $\text{proj}_{\{3\}}{}^{[2]}(\delta_c) = \text{proj}_{\{3\}}{}^{[2]}(\delta_c')$, for each $c \in \{a_3, a_3'\}$, and $\text{proj}_{\{3\}}{}^{[2]}(\delta_b) \cap \Delta_b(\{\mathcal{C}_3\}) = \text{proj}_{\{3\}}{}^{[2]}(\delta_b') \cap \Delta_b(\{\mathcal{C}_3\}) = \{(q_3, q_3')\}$.

Since b is *ai* in $\mathcal{T}$, Lemma 8 implies that b is also *ai* in both $SUB_{\{1,2\}}(\mathcal{T}) = SUB_{\{1,2\}}(\mathcal{T}')$ and $SUB_{\{3\}}(\mathcal{T}) = SUB_{\{3\}}(\mathcal{T}')$. From this it follows that b is both strong input peer-to-peer and strong output peer-to-peer in $\mathcal{T}$, as well as in $\mathcal{T}'$.

Furthermore, action b is master-slave in both $\mathcal{T}$ and $\mathcal{T}'$, since we have $\text{proj}_{\{1,2\}}{}^{[2]}(\delta_b) = \text{proj}_{\{1,2\}}{}^{[2]}(\delta'_b) = \{((q_1, q_2), (q'_1, q'_2))\} \subseteq \{((q_1, q_2), (q'_1, q'_2))\} = (\delta_{\{1,2\}})_b = (\delta'_{\{1,2\}})_b$, i.e. the output subteam of b participates in every b-transition of the teams. In fact, b is even strong master-slave in $\mathcal{T}$, as b is master-slave and $\text{proj}_{\{3\}}{}^{[2]}(\delta_b) = \{(q_3, q'_3)\} \subseteq \{(q_3, q'_3)\} = (\delta_{\{3\}})_b$, i.e. also the input subteam of b participates in every b-transition of $\mathcal{T}$. This implies that b is also weak master-slave in $\mathcal{T}$. However, $\text{proj}_{\{3\}}{}^{[2]}(\delta'_b) = \{(q_3, q'_3), (q'_3, q'_3)\} \nsubseteq \{(q_3, q'_3)\} = (\delta'_{\{3\}})_b$, and b is thus not strong master-slave in $\mathcal{T}'$. Since q_3 is the only state of $\mathcal{C}_3$ at which b is enabled in $\mathcal{C}_3$, b is weak master-slave in $\mathcal{T}'$.

Hence, the fact that $\mathcal{T}$, as opposed to $\mathcal{T}'$, does not allow an output action b to take place without a 'slave' input action b (b is *ai* in $\mathcal{T}$, but not *ai* in $\mathcal{T}'$), leads to b being strong master-slave in $\mathcal{T}$, and only weak master-slave in $\mathcal{T}'$.

To understand that despite the similarities, this subtle difference due to the distinction between *ai* and *si*, may lead to different languages for $\mathcal{T}$ and $\mathcal{T}'$, it is sufficient to show that $ba'_1a'_2b \in \mathbf{L}_{\mathcal{T}'}$, while no word with two b's is contained in $\mathbf{L}_{\mathcal{T}}$. That $ba'_1a'_2b \in \mathbf{L}_{\mathcal{T}'}$ is proved by the computation $(q_1, q_2, q_3)b(q'_1, q'_2, q'_3)a'_1(q_1, q'_2, q'_3)a'_2(q_1, q_2, q'_3)b(q'_1, q'_2, q'_3) \in \mathbf{C}_{\mathcal{T}'}$, whereas in δ the execution of b from the initial state (q_1, q_2, q_3) always leads to (q'_1, q'_2, q'_3), after which (q_1, q_2, q_3) (the only state from which b can be executed) has become unreachable.

8. The construction and classification of team automata

Our discussion until now has been analytical, in the sense that we have investigated transition relations to determine whether or not they satisfy the conditions inherent to certain modes of synchronization. As we have seen before, however, these conditions in general do not lead to uniquely defined team automata. To make the model of team automata of any use for applications in the field of groupware systems, it is necessary to be able to unambiguously construct a team automaton according to the specification of the required mode of synchronization (per action). We now turn to the question of how to define specific team automata satisfying certain constraints on synchronizations.

Synchronization requirements for an action a are conditions on the a-transitions to be chosen from $\Delta_a(\mathcal{S})$, the complete transition space of a in $\mathcal{S}$. Together these conditions should determine a unique subset $\mathcal{R}_a$ which will be the set of a-transitions in the team automaton. We will refer to subsets of $\Delta_a(\mathcal{S})$ as *predicates* for a. Once predicates have been chosen for all external actions, the team automaton over $\mathcal{S}$ defined by these predicates is unique. Note that for internal actions the transition relation is by definition fixed to equal the complete transition

space of that action in $\mathcal{A}$. Hence the construction of a team automaton satisfying certain conditions on its synchronizations, amounts to describing the appropriate predicates for its external actions.

The following generic definition formalizes this setup. The composable system $\mathcal{A}$ and the team automaton $\mathcal{T}$ over $\mathcal{A}$ are as fixed before. Recall that $\Sigma_{ext} = \bigcup_{i \in \mathcal{I}} (\Sigma_i \setminus \Sigma_{i,int})$ is the set of external actions of $\mathcal{T}$ and of any other team automaton over $\mathcal{A}$.

Definition 13. For all $a \in \Sigma_{ext}$, let $\mathcal{R}_a(\mathcal{A}) \subseteq \Delta_a(\mathcal{A})$ and let $\mathcal{R} = \{\mathcal{R}_a(\mathcal{A}) | a \in \Sigma_{ext}\}$. Then $\mathcal{T}$ is the *$\mathcal{R}$-team automaton over $\mathcal{A}$* if $\delta_a = \mathcal{R}_a(\mathcal{A})$ for all $a \in \Sigma_{ext}$. □

A natural way of fixing a predicate for a given mode of synchronization is to apply a maximality principle. That is, to include everything that is not forbidden, i.e. is in accordance with the synchronization constraints. This is the intuitive approach of Ellis (1997) and generalizes the classical approach to define synchronized systems (see, e.g., Duboc, 1986; Lynch and Tuttle, 1989; Janicki and Laurer, 1992) from *ai* to other modes of synchronization. Thus when a team automaton is to be constructed according to a specification of synchronization conditions for its external actions, the strategy is to include as many transitions as possible without violating the specification while checking that the result is unique.

For the case of no constraints, and for the case of constraints relating to the different modes of synchronization defined in Definition 9, this leads to the following predicates.

Definition 14. Let $a \in \Sigma_{ext}$. Then the predicate
no-constraints in $\mathcal{A}$ for a is denoted by $\mathcal{R}_a^{no}(\mathcal{A})$ and is defined as
$\mathcal{R}_a^{no}(\mathcal{A}) = \Delta_a(\mathcal{A})$,
is-free in $\mathcal{A}$ for a is denoted by $\mathcal{R}_a^{free}(\mathcal{A})$ and is defined as
$\mathcal{R}_a^{free}(\mathcal{A}) = \{(q, q') \in \Delta_a(\mathcal{A}) |$
$\#\{i \in \mathcal{I} | a \in \Sigma_i \wedge \text{proj}_i{}^{[2]}(q, q') \in \delta_{i,a}\} = 1\}$,
is-action-indispensable in $\mathcal{A}$ for a is denoted by $\mathcal{R}_a^{ai}(\mathcal{A})$ and is defined as
$\mathcal{R}_a^{ai}(\mathcal{A}) = \{(q, q') \in \Delta_a(\mathcal{A}) | \forall i \in \mathcal{I} : a \in \Sigma_i \Rightarrow \text{proj}_i{}^{[2]}(q, q') \in \delta_{i,a}\}$ and
is-state-indispensable in $\mathcal{A}$ for a is denoted by $\mathcal{R}_a^{si}(\mathcal{A})$ and is defined as
$\mathcal{R}_a^{si}(\mathcal{A}) = \{(q, q') \in \Delta_a(\mathcal{A}) | \forall i \in \mathcal{I} :$
$(a \in \Sigma_i \wedge a \text{ en}_{\mathcal{C}_i} \text{proj}_i(q)) \Rightarrow \text{proj}_i{}^{[2]}(q, q') \in \delta_{i,a}\}$. □

Each of these predicates selects, for a given external action a, *all* transitions from its complete transition space $\Delta_a(\mathcal{A})$ that obey a certain mode of synchronization. In the case of *no-constraints* for a, this means that all a-transitions are allowed since nothing is required, and thus no transition is forbidden. In the other three cases, *all and only* those a-transitions are included that respect the specified property of a.

Lemma 11. Let $a \in \Sigma_{ext}$. Then
(1) $a \in \mathit{Free}(\mathcal{T})$ if and only if $\delta_a \subseteq \mathcal{R}_a^{free}(\mathcal{S})$,
(2) $a \in \mathit{AI}(\mathcal{T})$ if and only if $\delta_a \subseteq \mathcal{R}_a^{ai}(\mathcal{S})$ and
(3) $a \in \mathit{SI}(\mathcal{T})$ if and only if $\delta_a \subseteq \mathcal{R}_a^{si}(\mathcal{S})$.

Proof. This follows immediately from Definitions 9 and 14. □

Thus each of the three predicates $\mathcal{R}_a^{free}(\mathcal{S})$, $\mathcal{R}_a^{ai}(\mathcal{S})$ and $\mathcal{R}_a^{si}(\mathcal{S})$ defines the largest and hence unique transition relation in $\Delta_a(\mathcal{S})$ in which a is *free*, *ai* or *si*, respectively.

Next we consider the constraints relating to peer-to-peer synchronizations. In this case we have to distinguish between the input and output role an external action may have in $\mathcal{S}$. Thus the predicates have to refer to the input and output domains of a in $\mathcal{S}$. Moreover, we have to distinguish between strong (*ai*) and weak (*si*) synchronizations. This leads to four predicates, each of which includes all and only those transitions from $\Delta_a(\mathcal{S})$ in which all component automata given by the input or output domain, respectively, are forced (in the weak or in the strong sense) to participate.

Recall that, for an external action a, $\mathcal{I}_{a,inp}(\mathcal{S}) = \{i \in \mathcal{I} | a \in \Sigma_{i,inp}\}$ is the input domain of a in $\mathcal{S}$ and $\mathcal{I}_{a,out}(\mathcal{S}) = \{i \in \mathcal{I} | a \in \Sigma_{i,out}\}$ is the output domain of a in $\mathcal{S}$. As before, we may simply write $\mathcal{I}_{a,inp}$ and $\mathcal{I}_{a,out}$, since $\mathcal{S}$ has been fixed.

Definition 15. (1) Let $a \in \bigcup_{i \in \mathcal{I}} \Sigma_{i,inp}$. Then the predicate
is-strong-input-peer-to-peer in $\mathcal{S}$ for a is denoted by $\mathcal{R}_a^{sipp}(\mathcal{S})$ and is defined as

$$\mathcal{R}_a^{sipp}(\mathcal{S}) = \{(q,q') \in \Delta_a(\mathcal{S}) | \\ \mathrm{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q,q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\}) \Rightarrow \\ \mathrm{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q,q') \in \mathcal{R}_a^{ai}(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})\}$$

and
is-weak-input-peer-to-peer in $\mathcal{S}$ for a is denoted by $\mathcal{R}_a^{wipp}(\mathcal{S})$ and is defined as

$$\mathcal{R}_a^{wipp}(\mathcal{S}) = \{(q,q') \in \Delta_a(\mathcal{S}) | \\ \mathrm{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q,q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\}) \Rightarrow \\ \mathrm{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q,q') \in \mathcal{R}_a^{si}(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})\}.$$

(2) Let $a \in \bigcup_{i \in \mathcal{I}} \Sigma_{i,out}$. Then the predicate
is-strong-output-peer-to-peer in $\mathcal{S}$ for a is denoted by $\mathcal{R}_a^{sopp}(\mathcal{S})$ and is defined as

$$\mathcal{R}_a^{sopp}(\mathcal{S}) = \{(q,q') \in \Delta_a(\mathcal{S}) | \\ \mathrm{proj}_{\mathcal{I}_{a,out}}{}^{[2]}(q,q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,out}\}) \Rightarrow \\ \mathrm{proj}_{\mathcal{I}_{a,out}}{}^{[2]}(q,q') \in \mathcal{R}_a^{ai}(\{\mathcal{C}_i | i \in \mathcal{I}_{a,out}\})\}$$

and
is-weak-output-peer-to-peer in $\mathcal{S}$ for a is denoted by $\mathcal{R}_a^{wopp}(\mathcal{S})$ and is defined as

$$\mathcal{R}_a^{wopp}(\mathcal{S}) = \{(q,q') \in \Delta_a(\mathcal{S}) | \\ \mathrm{proj}_{\mathcal{I}_{a,out}}{}^{[2]}(q,q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,out}\}) \Rightarrow \\ \mathrm{proj}_{\mathcal{I}_{a,out}}{}^{[2]}(q,q') \in \mathcal{R}_a^{si}(\{\mathcal{C}_i | i \in \mathcal{I}_{a,out}\}).$$ □

One should be aware at this point that we are not discussing the properties of a given team automaton over $\mathcal{A}$, with a fixed transition relation determining the transitions in the input and output subteams of a. Thus, in Definition 15, we relate to the complete transition spaces of a in the subsystems determined by the input and output domain of a. Each predicate includes all and only those transitions from $\Delta_a(\mathcal{A})$, for which all component automata given by the input or output domain, respectively, are forced (in the weak or in the strong sense) to participate in the execution of a by any of these component automata.

As before, the aim is to describe by means of the predicates unique and maximal sets of a-transitions satisfying the specified constraints. That this has been achieved through Definition 15, follows from the next lemma.

Lemma 12. Let $a \in \Sigma_{ext}$. Then
(1) a is strong input peer-to-peer in $\mathcal{T}$ if and only if $\delta_a \subseteq \mathcal{R}_a^{sipp}(\mathcal{A})$,
(2) a is weak input peer-to-peer in $\mathcal{T}$ if and only if $\delta_a \subseteq \mathcal{R}_a^{wipp}(\mathcal{A})$,
(3) a is strong output peer-to-peer in $\mathcal{T}$ if and only if $\delta_a \subseteq \mathcal{R}_a^{sopp}(\mathcal{A})$ and
(4) a is weak output peer-to-peer in $\mathcal{T}$ if and only if $\delta_a \subseteq \mathcal{R}_a^{wopp}(\mathcal{A})$.

Proof. We only prove (1). The other cases are proved analogously.

First, assume that a is strong input peer-to-peer in $\mathcal{T}$. Hence, according to Definition 11(1), $a \in AI(SUB_{a,inp})$, meaning that $a \in \Sigma_{inp}$ and a is *ai* in the subteam of $\mathcal{T}$ determined by the input domain of a. According to Definition 6, the a-transitions of this subteam are $(\delta_{\mathcal{I}_{a,inp}})_a = \text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(\delta_a) \cap \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})$. Now, by Lemma 11(2), $a \in AI(SUB_{a,inp})$ implies that $(\delta_{\mathcal{I}_{a,inp}})_a \subseteq \mathcal{R}_a^{ai}(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})$. Thus, for all $(q, q') \in \Delta_a(\mathcal{A})$, whenever $\text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q, q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})$, then $\text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q, q') \in \mathcal{R}_a^{ai}(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})$. Consequently, according to Definition 15(1), $\delta_a \subseteq \mathcal{R}_a^{sipp}(\mathcal{A})$.

Next, assume that $\delta_a \subseteq \mathcal{R}_a^{sipp}(\mathcal{A})$. By Definition 11(1), we now have to prove that a is *ai* in $SUB_{a,inp}$. Thus consider an arbitrary pair $(p, p') \in (\delta_{\mathcal{I}_{a,inp}})_a$. Since $(p, p') \in (\delta_{\mathcal{I}_{a,inp}})_a = \text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(\delta_a) \cap \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})$, there is a $(q, q') \in \delta_a \subseteq \Delta_a(\mathcal{A})$ for which $\text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q, q') = (p, p')$. From $\delta_a \subseteq \mathcal{R}_a^{sipp}(\mathcal{A})$, we infer that $(p, p') \in \mathcal{R}_a^{ai}(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})$. Hence $(\delta_{\mathcal{I}_{a,inp}})_a \subseteq \mathcal{R}_a^{ai}(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})$, and thus, by Lemma 11(2), a is *ai* in $SUB_{a,inp}$. □

Finally, we turn to master-slave synchronizations. As in the case of the peer-to-peer predicates, we have to distinguish between the input and output role of actions. This time, however, the predicates describe synchronizations *between* the components from the input domain and the components from the output domain.

Definition 16. Let $a \in \bigcup_{i \in \mathcal{I}} \Sigma_{i,out}$. Then the predicate
is-master-slave in $\mathcal{A}$ for a is denoted by $\mathcal{R}_a^{ms}(\mathcal{A})$ and is defined as

$$\mathcal{R}_a^{ms}(\mathcal{A}) = \{(q, q') \in \Delta_a(\mathcal{A}) | \text{proj}_{\mathcal{I}_{a,out}}{}^{[2]}(q, q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,out}\})\},$$

is-strong-master-slave in $\mathcal{S}$ for a is denoted by $\mathcal{R}_a^{sms}(\mathcal{S})$ and is defined as

$$\mathcal{R}_a^{sms}(\mathcal{S}) = \mathcal{R}_a^{ms}(\mathcal{S}) \cap \{(q, q') \in \Delta_a(\mathcal{S}) | \text{ if } \mathcal{I}_{a,inp} \neq \varnothing, \text{ then } \text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q, q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})\}$$

and *is-weak-master-slave in* $\mathcal{S}$ for a is denoted by $\mathcal{R}_a^{wms}(\mathcal{S})$ and is defined as

$$\mathcal{R}_a^{wms}(\mathcal{S}) = \mathcal{R}_a^{ms}(\mathcal{S}) \cap \{(q, q') \in \Delta_a(\mathcal{S}) | \text{ if } \mathcal{I}_{a,inp} \neq \varnothing, \text{ then } (\exists i \in \mathcal{I}_{a,inp} : a \text{ en}_{\mathcal{C}_i}\text{proj}_i(q)) \Rightarrow \text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q, q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})\}. \quad \square$$

The predicate *is-master-slave in* $\mathcal{S}$ for a includes all and only those a-transitions, in which a appears at least once in its output role. For the predicates *is-strong-master-slave in* $\mathcal{S}$ and *is-weak-master-slave in* $\mathcal{S}$, there is the additional requirement that a should also be executed by the components from its input domain. In the strong case, this obligation is strict in the sense that if the input domain of a is not empty, then always at least one component from the input domain of a participates in every a-transition included in the predicate. In the weak case, this obligation has to be met only when at least one component from the input domain of a is ready to perform a.

Each of these three (strong/weak) master-slave predicates guarantees that a is indeed (strong/weak) master-slave in every team automaton over $\mathcal{S}$ with that predicate for its a-transitions. Both the master-slave predicate and the strong master-slave predicate moreover are the largest set of a-transitions satisfying the specified constraint. However, it is not necessarily the case that every set of a-transitions by which a is weak master-slave is contained in the weak master-slave predicate. This difference stems from the fact that the predicate refers to components from the input domain of a rather than an input subteam. There is no way out and in fact the maximality principle is not applicable, because to define a subteam with transitions, a team automaton including the transition relation should have been defined already. Since a subteam only contains a selection of all possible a-transitions, it may happen that a is enabled in a component of the input subteam, but not in the subteam. Thus a can be weak master-slave in team automaton $\mathcal{T}$ when δ_a contains transitions in which the input subteam of a does not participate, while a is currently enabled in a component of this subteam.

Lemma 13. Let $a \in \bigcup_{i \in \mathcal{I}} \Sigma_{i,out}$. Then
(1) a is master-slave in $\mathcal{T}$ if and only if $\delta_a \subseteq \mathcal{R}_a^{ms}(\mathcal{S})$,
(2) a is strong master-slave in $\mathcal{T}$ if and only if $\delta_a \subseteq \mathcal{R}_a^{sms}(\mathcal{S})$ and
(3) if $\delta_a \subseteq \mathcal{R}_a^{wms}(\mathcal{S})$, then a is weak master-slave in $\mathcal{T}$.

Proof. (1) First, assume that a is master-slave in $\mathcal{T}$. Hence, by Lemma 10(1), $\text{proj}_{\mathcal{I}_{a,out}}{}^{[2]}(\delta_a) = (\delta_{\mathcal{I}_{a,out}})_a$. By Definition 6, $(\delta_{\mathcal{I}_{a,out}})_a = \text{proj}_{\mathcal{I}_{a,out}}{}^{[2]}(\delta_a) \cap \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,out}\})$, and thus $\text{proj}_{\mathcal{I}_{a,out}}{}^{[2]}(\delta_a) \subseteq \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,out}\})$. Thus, by Definition 16, $\delta_a \subseteq \mathcal{R}_a^{ms}(\mathcal{S})$.

Next, assume that $\delta_a \subseteq \mathcal{R}_a^{ms}(\mathcal{S})$. Then, by Definition 12(1), we have to prove that $\text{proj}_{\mathcal{I}_{a,out}}{}^{[2]}(\delta_a) \subseteq (\delta_{\mathcal{I}_{a,out}})_a$. By Definition 6, we thus have to prove

$\text{proj}_{\mathcal{I}_{a,out}}{}^{[2]}(\delta_a) \subseteq \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,out}\})$. This follows immediately from Definition 16.

(2) Assume that $\mathcal{I}_{a,inp} \neq \varnothing$ (otherwise there is nothing to prove). As in the proof of (1) for $\mathcal{I}_{a,out}$, it is easy to prove that $\text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(\delta_a) \subseteq (\delta_{\mathcal{I}_{a,inp}})_a$ if and only if $\delta_a \subseteq \{(q,q') \in \Delta_a(\mathcal{S}) | \text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q,q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})\}$. By using Definition 12(2), we thus infer that a is strong master-slave in $\mathcal{T}$ if and only if $\delta_a \subseteq \mathcal{R}_a^{ms}(\mathcal{S})$ and $\delta_a \subseteq \{(q,q') \in \Delta_a(\mathcal{S}) | \text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q,q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})\}$. Thus, according to Definition 16, we are ready.

(3) Assume that $\mathcal{I}_{a,inp} \neq \varnothing$ (otherwise there is nothing to prove), and assume moreover that $\delta_a \subseteq \mathcal{R}_a^{wms}(\mathcal{S})$. Then, by Definition 12(3), we have to prove that if $(q,q') \in \delta_a$ and $a \ \text{en}_{SUB_{a,inp}} \ \text{proj}_{\mathcal{I}_{a,inp}}(q)$, then $\text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q,q') \in (\delta_{\mathcal{I}_{a,inp}})_a$. Definition 16 implies that, for all $(q,q') \in \delta_a$, if there is an $i \in \mathcal{I}_{a,inp}$ for which $a \ \text{en}_{\mathcal{C}_i} \ \text{proj}_i(q)$, then $\text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q,q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})$. Since $a \ \text{en}_{SUB_{a,inp}} \ \text{proj}_{\mathcal{I}_{a,inp}}(q)$ implies that then there is an $i \in \mathcal{I}_{a,inp}$ for which $a \ \text{en}_{\mathcal{C}_i} \ \text{proj}_i(q)$, we now have that if $(q,q') \in \delta_a$ and $a \ \text{en}_{SUB_{a,inp}} \ \text{proj}_{\mathcal{I}_{a,inp}}(q)$, then $\text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q,q') \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_{a,inp}\})$. In this case, Definition 6 implies that $(\delta_{\mathcal{I}_{a,inp}})_a = \text{proj}_{\mathcal{I}_{a,inp}}{}^{[2]}(q,q')$, and thus we are ready. □

Hence, except for weak master-slave synchronization, each of the various modes of synchronization introduced in Sections 5 and 6 gives rise to a predicate that is the unique maximal representative among all transition relations satisfying the constraints implied by the mode of synchronization. Consequently, once for each external action one of these modes has been chosen for its synchronizations, a unique team automaton can be constructed using the maximality principle. Finally, observe that in the formalizations of the predicates there is no need to refer to a team automaton, its subteams, and their transition relations.

In case all external actions are required to be of the same type (e.g. *ai*) then this team automaton is called *homogeneous*. Thus, one may consider, e.g., the $\{\mathcal{R}_a^{ai}(\mathcal{S}) | a \in \Sigma_{ext}\}$-team automaton as *the maximal-ai team automaton* over $\mathcal{S}$. If the actions may be of different types, the $\{\mathcal{R}_a(\mathcal{S}) | a \in \Sigma_{ext}\}$-team automaton is called *heterogeneous*.

We now have various tools available for the classification of team automata over $\mathcal{S}$. We can analyze the properties of the transition relation of a given team automaton, or we can set requirements leading to the definition of the maximal team automaton. Furthermore, we can consider properties at the team level, or at the level of subteams. In particular, we now have tools allowing formal and precise definitions of various basic groupware notions.

Team automata can be classified on basis of the properties of their transition relation or by imposing conditions on the construction of the transition relation. One way of viewing the team automaton model is as having a two way mechanism to model a spectrum of group interactions. On the one hand there are master-slave synchronizations, in which output as a master may force the concurrent execu-

tion of a corresponding input action. They can be used to model asynchronous cooperation, as in workflow systems, to enact certain modules (see, e.g., Ellis and Nutt, 1993). On the other hand we have peer-to-peer synchronizations, in which all participants are considered equal. They model the group collaboration aspect that frequently occurs in synchronous groupware. Exact descriptions of certain groupware notions which may otherwise have an ambiguous interpretation can thus be given. For example, Ellis (1997) makes the following distinction between cooperation and collaboration within the team automaton model:

> "A Team Automaton is defined to be *cooperating* if it is structured so that one of its components is the active master, and all the others are passive slaves."

and

> "A Team Automaton is defined to be *collaborating* if it is structured so that all of the automata are active peers."

To this it is added that the master-slave mechanism is referred to as *passive cooperation*, since the master is never blocked waiting for a slave. This contrasts with the peer-to-peer mechanism, in which blocking may occur when not all of the participants are ready to perform the action, and which is called *active collaboration*.

Clearly, the framework presented in this paper allows for more and finer distinctions. This is mainly due to the uniform approach towards the formalization of the notion of obligation for components to participate in the execution of a certain action, which is here independent of the role of that action (input or output, master, slave or peer).

Thus, we have provided two global interpretations of collaboration through the notions of *ai* (comparable to the adjective 'active' above, as blocking may occur) and *si*. Here the input role an action may have is not yet separated from its output role. When this distinction is made we arrive at the four notions of weak/strong input/output peer-to-peer.

Cooperation is formalized through the notions of (weak/strong) master-slave synchronizations. When an action is master-slave, it cannot be executed as an input action without being simultaneously executed as an output action. In the strong case, all slaves (the components having the action as an input action) should participate in the action, whereas in the weak case all components that are ready for that action should participate in the synchronization (which corresponds to the 'passive' cooperation mentioned above). Note that as is made precise in this paper, the master in a master-slave synchronization may be a subteam rather than a single component. As argued in Section 4, there is no essential difference between a subteam of a team automaton and a component which itself may have been obtained as a team. Similarly the slaves may be one or more components or one or more subteams.

This viewpoint also easily allows combinations of cooperation and collaboration, called hybrids in Ellis (1997). One may, e.g., have a master-slave synchro-

nization in which within the master (subteam) the synchronizations are strong output peer-to-peer, while the subteam of the slaves exhibits weak input peer-to-peer synchronization (all slaves that can, participate) or strong input peer-to-peer synchronization (all slaves have to take part).

Finally, observe that these considerations on cooperation and collaboration all relate to the synchronization modes of a single external action. These notions can also be lifted to the level of the team as a whole, either in a homogeneous way or in an heterogeneous way. In the first case there is one type of cooperation or collaboration (the same for all actions) including the identity of the master, the slaves, the input domain, the output domain, etc. In the second case, each external action can have its own cooperation or collaboration specification.

Given requirements for each external action, one may follow the approach outlined in the first part of this section to construct a unique automaton team with the appropriate combinations of cooperating and collaborating synchronizations.

Thus the theory presented has led to a flexible framework that allows one to precisely classify, describe and construct many different incarnations of cooperation and collaboration. Which of these may be of use in applications, is for practice to decide.

This concludes the display of our framework for precisely and consistently defining terminology at the conceptual level of groupware systems. In the remainder of the paper we focus on its use at the architectural level of groupware systems.

9. Teams as architectural building blocks

As we have seen, a team automaton over a composable system is itself a component automaton that can be used in further constructions of team automata. Team automata can thus be used as building blocks. From the point of view of groupware systems two things are important.

First, one should be able to construct unique team automata of a certain specified type. With this we dealt in the previous section. Secondly, before a team is used as a building block, it may be necessary to internalize certain external actions in order to prohibit their further use on a higher level of the construction. In this section, we explore these architectural considerations in more detail.

Hiding makes certain external actions of an automaton invisible to other automata by turning these external actions into internal actions.

Definition 17. Let $\mathcal{C} = (P, (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int}), \gamma, J)$ be a component automaton and let $\Theta \subseteq \Gamma_{inp} \cup \Gamma_{out}$. Then the component automaton $hide_\Theta(\mathcal{C})$ is defined as

$$hide_\Theta(\mathcal{C}) = (P, (\Gamma_{inp} \setminus \Theta, \Gamma_{out} \setminus \Theta, \Gamma_{int} \cup \Theta), \gamma, J). \quad \square$$

Composability is in general not preserved by hiding, because the composability condition requires that the internal actions of the components belong to one component only, whereas external actions are not subject to such a restriction.

Thus, for our composable system $\mathcal{S} = \{\mathcal{C}_i | i \in \mathcal{I}\}$ and subsets Θ_i of the external actions of each $\mathcal{C}_i$, the system $\mathcal{S}' = \{hide_{\Theta_i}(\mathcal{C}_i) | i \in \mathcal{I}\}$ is composable if and only if, for all $i \in \mathcal{I}$, $\Theta_i \cap \bigcup_{j \in \mathcal{I} \setminus \{i\}} \Sigma_{j,ext} = \varnothing$.

Note that since hiding does not preserve composability, a team automaton after hiding is not necessarily a team automaton over its original components after hiding.

The external actions to be hidden are those that should only be used for communications with certain components and with no others.

Definition 18. A pair $\mathcal{C}_i, \mathcal{C}_j$, with $i, j \in \mathcal{I}$, of component automata is *communicating* (in $\mathcal{S}$) if there exists an $a \in (\Sigma_{i,ext} \cup \Sigma_{j,ext})$ such that

$$a \in (\Sigma_{i,inp} \cap \Sigma_{j,out}) \cup (\Sigma_{j,inp} \cap \Sigma_{i,out}).$$

Such an a is called a *communicating* action (in $\mathcal{S}$). By Σ_{com} we denote the set of all *communicating actions* (in $\mathcal{S}$). □

Note that the communicating relation between components, i.e. the set of all pairs of communicating automata over component automata, is symmetric and irreflexive. Furthermore, note that the fact that an action is communicating, does not imply that a team automaton over $\mathcal{S}$ will actually have a synchronization involving this action as a communication, i.e. in its two roles of input and output. The communicating property is based solely on alphabets and thus by no means related to transition relations.

With the hide operation we can internalize all communicating actions of a team automaton before this automaton is used to build a higher-level team. Then the automaton is closed with respect to its communications to the outside world.

Definition 19. The (*communication*) *closed* version of $\mathcal{T}$ is denoted by $\boxed{\mathcal{T}}$ and is defined as

$$\boxed{\mathcal{T}} = hide_{\Sigma_{com}}(\mathcal{T}).$$ □

Rather than the team itself, we may now use its closed version in a new construction. If we do this, then only those external actions that do not have a matching external action within the team are external actions of the closed version of the team. The other external actions have been reclassified as internal actions. Before exploring this in the next section, we need to solve one more technical detail.

In practice one often wants to work with several copies of an automaton. In our model, however, more than one copy of an automaton in a set of components in general means this set does not satisfy the composability condition. Renaming the actions of an automaton solves this problem. Modulo renaming, these copies all have the same computations (and thus also the same language).

A function $f : A \to A'$ is injective if $f(a_1) \neq f(a_2)$ whenever $a_1 \neq a_2$, f is surjective if for every $a' \in A'$ there exists an $a \in A$ such that $f(a) = a'$, and f is a bijection if f is injective and surjective.

Definition 20. Let $\mathcal{C} = (P, (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int}), \gamma, J)$ be a component automaton, Γ' an alphabet and $h : \Gamma_{inp} \cup \Gamma_{out} \cup \Gamma_{int} \rightarrow \Gamma'$ a bijection. Then the *renaming* $h(\mathcal{C})$ of $\mathcal{C}$ according to h is the component automaton

$$h(\mathcal{C}) = (P, (h(\Gamma_{inp}), h(\Gamma_{out}), h(\Gamma_{int})), \gamma', J),$$

where $\gamma' = \{(q, h(a), q') | (q, a, q') \in \gamma\}$. □

In practice, renaming might best be defined to generate new names which are disjoint from the domain set, e.g., by requiring Γ' to be disjoint from $\Gamma_{inp} \cup \Gamma_{out} \cup \Gamma_{int}$. Note that uniqueness of the renamings of component automata is guaranteed by requiring h to be a bijection.

It is clear that, apart from the use of new names, the properties of component automata are not changed by a renaming. Hence the proof of the following lemma is omitted.

Lemma 14. Let h be a bijection such that $h(\mathcal{T})$ is a renaming of the team automaton $\mathcal{T}$ over $\mathcal{S}$. Then

(1) $\mathbf{C}_{h(\mathcal{T})} = \bar{h}(\mathbf{C}_{\mathcal{T}})$, where $\bar{h}$ is the extension of h to $\Sigma \cup Q$, defined by $\bar{h}(q) = q$ for all $q \in Q$,
(2) $\mathbf{L}_{h(\mathcal{T})} = h(\mathbf{L}_{\mathcal{T}})$ and
(3) if an action a is X in $\mathcal{T}$, then $h(a)$ is X in $h(\mathcal{T})$, where X is *free*, *ai*, *si*, strong output peer-to-peer, strong input peer-to-peer, weak output peer-to-peer, weak input peer-to-peer, master-slave, strong master-slave, or weak master-slave. □

10. The GROVE document editor architecture modeled by team automata

In this section we show how the design issues discussed in the previous sections can be used to model a specific groupware architecture.

In Ellis (1997), the distributed architecture of the GROVE document editor (see Ellis et al., 1990), depicted here in Figure 10, is discussed. We show how to model this architecture, using a formal description in terms of team automata. In the process we point out where the notions introduced in the previous section come into play. The formal relations between component automata follow from definitions in the previous sections.

Consider a user interface automaton $\mathcal{C}_1$, a keeper automaton $\mathcal{C}_2$, an application automaton $\mathcal{C}_3$ and a coordination automaton $\mathcal{C}_4$, together forming a composable system $\mathcal{S} = \{\mathcal{C}_i | i \in [4]\}$. Only the pairs $\mathcal{C}_i, \mathcal{C}_{i+1}$, $i \in [3]$, are communicating. All external actions of $\mathcal{C}_2$ and $\mathcal{C}_3$ are communicating in $\mathcal{S}$. $\mathcal{C}_1$ has external actions that are not communicating in $\mathcal{S}$, but intended to be used solely for interaction with the users. $\mathcal{C}_4$ has external actions to be used for communication with the communication automaton $\mathcal{C}_5$, which is to be added in a later stage. However, the non-communicating actions of $\mathcal{C}_1$ are different from those of $\mathcal{C}_4$.

The architecture requires all components in $\mathcal{S}$ to synchronize on all communications, thus we construct the $\{\mathcal{R}_a^{ai}(\mathcal{S}) | a \in \Sigma_{ext}\}$-team automaton $\mathcal{T}$ over $\mathcal{S}$.

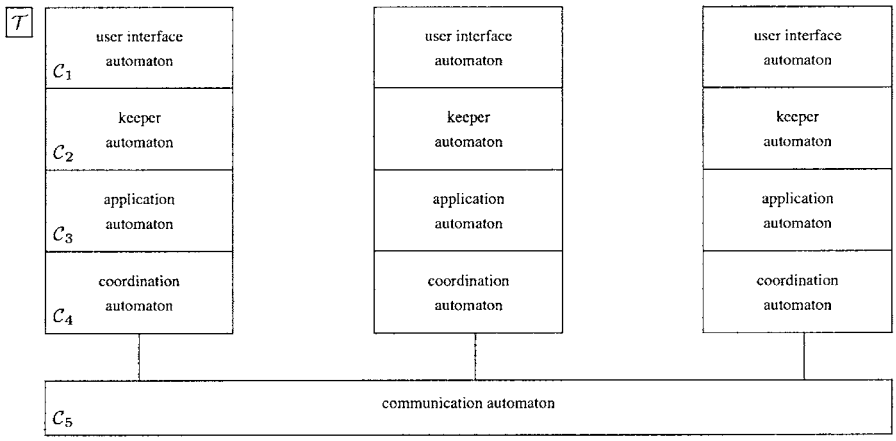

Figure 10. The GROVE document editor architecture.

Then this team automaton $\mathcal{T}$ is closed, resulting in its closed version $\boxed{\mathcal{T}}$. Now all communicating external actions are internal in $\boxed{\mathcal{T}}$. In this way we prohibit further synchronizations involving a component of $\mathcal{S}$. The only remaining external actions are those of $\mathcal{C}_1$ and those of $\mathcal{C}_4$.

Next we introduce renamings of $\boxed{\mathcal{T}}$ satisfying two conditions. First, the sets of actions of the renamings should be mutually disjoint to avoid unwanted synchronizations of their user interfaces and of actions to be used for the interaction with the communication automaton $\mathcal{C}_5$. Note that this condition ensures that these renamings form a composable system $\mathcal{S}'$. Secondly, the external actions of $\boxed{\mathcal{T}}$ originating from the coordination automaton $\mathcal{C}_4$ should be renamed in such a way that they will communicate with actions from $\mathcal{C}_5$.

Finally, to obtain the desired automaton modeling the GROVE document editor architecture, we define a team automaton over $\mathcal{S}'' = \{\mathcal{C}_5\} \cup \mathcal{S}'$. Since we want $\mathcal{C}_5$ to communicate with all copies of $\boxed{\mathcal{T}}$, we construct the $\{\mathcal{R}_a^{ai}(\mathcal{S}'')|a \in \Sigma_{ext}\}$-team automaton over $\mathcal{S}''$, which thus results in all communicating actions being synchronized.

It is clear that the iterated way in which we have constructed this automaton guarantees that no undesired synchronizations between, e.g., a keeper automaton and the communication automaton can take place. Not only all communication between the communication automaton and any of the renamings of $\boxed{\mathcal{T}}$ takes place via their coordination automata, but also there are no interactions between the renamings of $\mathcal{T}$. This is conveniently modeled by the communication closure. Moreover, the explicit construction used to form the final team makes all communications mandatory.

11. A comparison with I/O automata and Petri nets

Team automata are an extension of *Input/Output automata* (I/O automata for short) and they are also related to a specific Petri net based model called *Vector Controlled Concurrent Systems* (VCCS for short). Actually, as we briefly sketch in this section, one may view team automata as a model somewhere in between those two.

We begin this section with a discussion of the relationship between I/O automata and team automata, which ends by showing how I/O automata fit into the framework of team automata.

I/O automata were introduced in Tuttle (1987) (see also Lynch and Tuttle, 1989) for modeling distributed discrete event systems consisting of components that operate concurrently. Since then they have been used extensively as a formal model for the verification of distributed algorithms (see, e.g., Lynch, 1996).

Originally, I/O automata are defined in terms of labeled transition systems together with an associated equivalence relation over the set of actions used to define so-called fair computations. In Tuttle (1987) I/O automata without such equivalence relations are called safe I/O automata and in Gawlick et al. (1994) they are referred to as unfair. Here we are not concerned with fairness and we only consider safe or unfair I/O automata, to which we will simply refer as I/O automata.

The model of I/O automata has a single notion of automaton composition which, as already noted in Tuttle (1987), is rather restrictive and may hinder a realistic modeling of certain types of interactions. This is the main motivation given in Ellis (1997) for introducing team automata for groupware systems as a generalization of I/O automata.

An I/O automaton is an ilts together with a classification of its actions as input, output or internal. Input and output actions form the interface between the automaton and its environment, including other I/O automata. Within a composition, automata which share an action a have to perform a simultaneously (synchronize on a). The intention is that simultaneous execution models a communication from the automata of which a is an output action to the automata of which a is an input action. In fact, the execution of an input action is thought of as the notification of the arrival of output from another automaton. With these considerations in mind, I/O automata are formally defined as component automata, but with the additional condition that they should be *input-enabled*. This means that, whatever the current state of the automaton, it is always capable of receiving any of its potential inputs. Thus, in every state of the automaton, every input action of that automaton is enabled.

Given a collection $\mathcal{S} = \{\mathcal{C}_i | i \in \mathcal{I}\}$ of I/O automata, a new I/O automaton can be constructed if $\mathcal{S}$ satisfies two conditions. These conditions only relate to the role of the actions and for them it is irrelevant whether or not the $\mathcal{C}_i$ are input-enabled. We can thus assume that $\mathcal{S} = \{\mathcal{C}_i | i \in \mathcal{I}\}$ is as before a collection of component automata. The first condition is that $\mathcal{S}$ should be composable. Hence, as for the

definition of a team automaton, it is required that the internal actions of any of the component automata belong uniquely to that component. Secondly, there is the idea that two components cannot be expected to synchronize on an output action. Rather than complicating the notion of composition itself, this is prohibited by the requirement that the output actions of the automata in $\mathcal{A}$ should be disjoint. This means that every external action can be output in at most one of the component I/O automata. Formally, for all $i \in \mathcal{I}$, $\Sigma_{i,out} \cap \bigcup_{j \in \mathcal{I} \setminus \{i\}} \Sigma_{j,out} = \emptyset$. If $\mathcal{A}$ satisfies both conditions, then we call it a *compatible system*. Note that every subset of a compatible system is again a compatible system.

Finally, the composition of I/O automata into a new automaton is defined according to the intuitive explanation above that automata which share an action have to synchronize on a. In terms of our framework this means that a team automaton is constructed, in which every action is ai. Moreover (although this is only implicit in the explanation) all synchronizations which do not violate this condition have to be included (maximality). Hence the constructed team automaton is unique. Formally, if $\mathcal{A} = \{\mathcal{C}_i | i \in \mathcal{I}\}$ is a compatible system of I/O automata, then the *team I/O automaton* over $\mathcal{A}$ is the team automaton $\mathcal{T}$ over $\mathcal{A}$ with transition relation δ such that $\delta_a = \mathcal{R}_a^{ai}(\mathcal{A})$, for all actions a of $\mathcal{A}$.

Since this composition of a team automaton preserves input-enabledness, it follows that every team I/O automaton is again an I/O automaton and hence can be used to iteratively define higher-level team I/O automata. Together with our earlier observation that subsets of a compatible system are compatible systems, this implies that the team I/O automaton over a compatible system $\mathcal{A}$ of I/O automata can be constructed by iteration. Any iterated team I/O automaton corresponds to the team I/O automaton over $\mathcal{A}$ after reordering its state space.

Conversely, if $\mathcal{T}$ is the team I/O automaton over a compatible system $\mathcal{A} = \{\mathcal{C}_i | i \in \mathcal{I}\}$ of I/O automata, then every subteam of $\mathcal{T}$ determined by some $J \subseteq \mathcal{I}$, is the team I/O automaton over $\{\mathcal{C}_j | j \in J\}$. This follows from our earlier remark that the property of an action being *ai* in a team automaton is inherited by all of its subteams.

Another consequence of composition on basis of maximal *ai* predicates is that every output action is strong master-slave. This provides a formal description of the idea that output is always received by those component automata that have its input counterpart as an action. Since I/O automata are input-enabled, it is even the case that the output automaton does not have to wait until the input automata are ready for the communication. It is however worthwhile to notice that it may be the case that an external action appears only as an input action in the system $\mathcal{A}$. Then it is again an input action of the team I/O automaton over $\mathcal{A}$ and can be used as such in a higher-level team. Note that since all input (output) actions are *ai* in an I/O team automaton, they are also strong input (output) peer-to-peer.

The I/O automaton model thus fits seamlessly in the team automata model and so results and notions from team automata become available for I/O automata.

In particular, a framework is provided in which the underlying concepts of I/O automata can be given a broader perspective and compared with other ideas. For instance, the possibility to define the language of a team I/O automaton directly (without actually considering the team) from the languages of its components is an important property in the theory of I/O automata. This property is already implied by the maximal *ai* construction for general team automata. Also the idea of subteams and iterative construction only marginally investigated for I/O automata, are now immediately available from the team automata framework. Team automata however allow more types of synchronizations, which is convenient when formally designing a system. As remarked in Tuttle (1987), for some designs it may be a disadvantage that the composition of I/O automata implies that output actions can always be traced back to a unique sender.

We now turn to a very brief comparison of team automata with Petri nets (see, e.g., Reisig and Rozenberg, 1998) or better, with models based on Petri nets.

Team automata are compositions of component automata working together through synchronizations on certain actions. These synchronizations are labeled transitions which describe state changes caused by global actions of the team. As a consequence, the operational semantics in terms of computations of team automata is of a sequential nature and does not reflect the fact that they are distributed systems. By switching from global actions to actions with local information on the participation of the components it is however possible to make the potential concurrency within a team visible. We now show how vector actions can be employed to this end and consequently we discuss how team automata fit into a theory of vector controlled concurrent systems.

By Definition 5, each transition of $\mathcal{T}$ is of the form (q, a, q') with $a \in \Sigma$ and $q, q' \in \prod_{i \in \mathcal{I}} Q_i$. We now switch from transitions (q, a, q') to *vector transitions* (q, α, q'), where α is an element of $\prod_{i \in \mathcal{I}}(\{a\} \cup \{\lambda\})$, i.e. a vector with for each component a corresponding entry which is either a or λ. If an entry of α is a, then this indicates that the corresponding component takes part in the synchronization on a, while if it is λ, then that component is not involved. By performing this switch (with once again the maximal interpretation of the components' involvement in case of loops) we can transform $\mathcal{T}$ into a *vector team automaton* over $\mathcal{S}$ which has vector transitions rather than 'flat' transitions.

On the other hand, we can also directly define a vector team automaton over $\mathcal{S}$ by translating the required synchronizations straight away into vector transitions. In that case, for each action a, we choose vector transitions from the *complete vector transition space* $\Delta_a^v(\mathcal{S})$ of a in $\mathcal{S}$ which describes all possible vector transitions for a. If $(q, \alpha, q') \in \Delta_a^v(\mathcal{S})$, then α is called a *vector representation* of a in $\mathcal{S}$ or a *vector action* of $\mathcal{S}$. Due to the composability of $\mathcal{S}$ every internal action has only one vector representative and this representative has exactly one entry which is not λ. Furthermore we of course require all vector representatives of external actions to have *at least* one entry which is not λ. A vector team automaton over $\mathcal{S}$ can now

be defined exactly as an ordinary team automaton over $\mathcal{S}$, except that its transition relation consists of vector transitions.

Note that in vector team automata it is clear which components participate in each of its vector transitions. Recall from Example 1 that this contrasts with 'ordinary' team automata. In fact, by replacing each transition (q, α, q') of a vector team automaton by the flat transition (q, a, q') if α is a vector representative of the action a, we obtain a *flattened version* of the vector team automaton. This is an ordinary team automaton which models essentially the same synchronizations. However, information on the role of loops is lost. In fact, each vector team automaton has a unique flattened version, whereas there may be many vector team automata that have the same flattened version. In this sense, vector team automata have more expressive power than ordinary team automata.

Vector team automata are an example of a model of distributed systems consisting of sequential components, the cooperation of which is controlled by synchronization vectors. In such systems one deals with independently operating processes that from time to time synchronize their actions with others. Vectors of actions describe which processes are involved in such a cooperation. No other actions are allowed in the system than those described by the vectors. Thus also individual actions of the processes appear as vectors with a single non-λ entry. In Keesmaat et al. (1990, 1991) a framework is proposed for the study of vector controlled systems. The approach there is based on the synchronization as modeled in the vector firing sequence semantics of path expressions and COSY (see, e.g., Janicki and Laurer, 1992) and is related to the work of Arnold and Nivat (see, e.g., Arnold, 1982) and the coordination of cooperating automata by synchronization on multisets in Badouel et al. (1999).

The VCCS model allows to specify, in addition to the component processes and the synchronization vectors, also a control mechanism to determine when synchronizations are to be used. In team automata and vector team automata, the synchronizations that can take place are state dependent and hence they fit in the VCCS framework. In ter Beek et al. (2001a) we make this claim concrete by describing a particular method of vector synchronization within VCCS which is applicable to team automata and based on Petri nets. This method is a straightforward construction to obtain a Petri net of a specific form out of a vector team automaton. In fact, this specific Petri net is an *Individual Token Net Controller* (ITNC for short) as defined in Keesmaat et al. (1990) and further studied in Keesmaat and Kleijn (1997). ITNCs are control mechanisms for vector synchronization based on state machine decomposable nets. They are designed such that they can record the progress and control the synchronizations of sequential processes. The vector labels in an ITNC are the synchronization vectors and they do not have to be uniform in the sense that all non-λ entries are instances of the same action. Thus ITNCs allow more types of synchronization than team automata. However,

ITNCs are not concerned with the distinction of actions into input, output and internal actions.

To conclude, we note that vector team automata like ITNCs (see Keesmaat and Kleijn, 1997) allow a concurrent operational semantics according to the intuition that synchronizations that involve disjoint sets of components are independent and can be executed concurrently. This can be formalized also in terms of an independence relation (over transitions or over vector actions) similar to the independence relation used for Mazurkiewicz traces (see, e.g., Mazurkiewicz, 1989; Diekert and Rozenberg, 1995; and also the asynchronous automata of Zielonka, 1987).

12. Discussion

Within the field of CSCW one deals with systems intended to support groups of people working together in collaborative projects. Such systems are often distributed and conceived as consisting of agents cooperating in a coordinated way, which leads to complex interactive behavior. Consequently, coordination policies and their effect on behavior are key issues for CSCW. There is a need for models which help to clarify basic notions and to develop new notions of collaboration. Team automata provide a formal, yet flexible framework for the description and analysis of protocols and groupware systems. Modeling a system as a team automaton in the early phases of design forces one to consider the intended communications and synchronizations in detail, which leads to a better understanding of the functionality of the system and to explicit and unambiguous design choices. This forms the basis of further design and implementation, especially since the team automata framework allows a modular (iterative) construction and can be used also at the architectural level of system specification. At the same time the mathematically rigorous definitions provide the possibility of formal analysis tools for proving crucial design properties, without first having to implement the design. This paper proposes formal definitions and notions relevant for the design, but does not yet consider behavioral aspects which may be useful for the analysis of groupware systems.

Team automata model the logical architecture of a design. They abstract from concrete data, configurations and actions, and they describe the system solely in terms of a state-action diagram (transition system), the role of actions (input, output or internal) and synchronizations.

To model a system as a team automaton, first the components have to be identified. Each of them should be given a description in the form of a labeled transition system, an easy to understand well-known model. Based on the idea of shared action, these components can be connected in order to work together. Within each component, a distinction has to be made between internal actions (not available for synchronization with other components) and external actions (which can be used to synchronize components and are subject to synchronization restrictions).

Next, for each external action separately, a decision is made as to how the components should synchronize on this action. Assigning different roles to an external action makes it possible to describe different types of synchronization, such as, e.g., communications in which an action has both an input and an output role. If the action is supposed to be a 'passive' action, which may not be under the local control of the component, then it can be designated as an input action of that component. Otherwise it is an output action. If such distinction between the roles of an external action is not necessary, then the choice is arbitrary. A natural option would be to make it an output action in all components in which it occurs.

Once the synchronization constraints for each external action have been determined, one may apply, e.g., a maximality principle to construct a unique team automaton satisfying all constraints.

Similarly, the architecture of the system can be described as a team automaton with team automata as building blocks (components). System properties can then be considered both at the team level and at the level of subteams.

Thus, the team automata framework supports the design by making explicit the role of actions and the choice of transitions governing the coordination of the component automata. The crucial feature is the freedom of choice for the synchronizations collected in the transition relation of a team automaton.

Appendix

Here we list the definitions and proofs omitted from Section 4.

Definition A1. $\mathcal{V}(\mathcal{D})$ is the smallest set $\mathcal{V}$ such that:

(1) $D_j \in \mathcal{V}$, for each $j \in J$.
Set $\mathrm{dom}(D_j) = \{j\}$.

(2) If $\{V_l | l \in L\} \subseteq \mathcal{V}$, with $L \subseteq \mathbb{N}$ and $L \neq \varnothing$, then $\prod_{l \in L} V_l \in \mathcal{V}$ provided that, for all $k \neq l \in L$, $\mathrm{dom}(V_k) \cap \mathrm{dom}(V_l) = \varnothing$.
Set $\mathrm{dom}(\prod_{l \in L} V_l) = \bigcup_{l \in L} \mathrm{dom}(V_l)$. □

This (recursive) definition provides a description of how products of products are constructed. Given an element v of a nested cartesian product V from $\mathcal{V}(\mathcal{D})$ the function u_V, defined next, locates recursively for each j in the domain $\mathrm{dom}(V)$ of V the element in the position of D_j according to the construction of V. Since each D_j, with $j \in \mathrm{dom}(V)$, is used exactly once in the construction of V, its position is unique. Thus u_V **u**npacks v and on basis of this unpacking the resulting elements of $\bigcup_{j \in \mathrm{dom}(V)} D_j$ are ordered in $\langle v \rangle_V$.

Definition A2. Let $V \in \mathcal{V}(\mathcal{D})$ be such that $\mathrm{dom}(V) = J'$ for some $J' \subseteq J$. Then the function $u_V : V \times J' \to \bigcup_{j \in J'} D_j$ is defined as follows:

(1) If $J' = \{j\}$ and $V = D_j$, then $u_V(v, j) = v$ for all $v \in V$.

(2) If $V = \prod_{l \in L} V_l$, with $V_l \in \mathcal{V}(\mathcal{D})$ for all $l \in L$, then, for all $v \in V$ and $j \in J'$, $u_V(v, j) = u_{V_k}(\mathrm{proj}_k(v), j)$, where $k \in L$ is such that $j \in \mathrm{dom}(V_k)$.

The *reordering* of an element $v \in V$ relative to the construction of V is denoted by $\langle v \rangle_V$ and is defined as

$\langle v \rangle_V = \prod_{j \in J'} u_V(v, j)$. □

Lemma 4. If $V \in \mathcal{V}(\mathcal{D})$ and $\mathrm{dom}(V) = J'$, then $\{\langle v\rangle_V | v \in V\} = \prod_{j\in J'} D_j$.

Proof. ($\subseteq$) Let $v \in V$. By Definition A2, $\langle v\rangle_V = \prod_{j\in J'} u_V(v, j)$. Now we only have to prove that $u_V(v, j) \in D_j$, for all $j \in J'$. We do this by structural induction. If $J' = \{j\}$ and $V = D_j$, then $u_V(v, j) = v \in V = D_j$. Next assume that $V = \prod_{l\in L} V_l$, with $V_l \in \mathcal{V}(\mathcal{D})$ for all $l \in L$. Then, by Definition A2, for all $j \in J'$, $u_V(v, j) = u_{V_k}(\mathrm{proj}_k(v), j)$, where k is such that $j \in \mathrm{dom}(V_k)$. Since each $V_k \in \mathcal{V}(\mathcal{D})$, the depth of its nesting is strictly less than the depth of the nesting in V. Thus, by the induction hypothesis, $u_{V_k}(\mathrm{proj}_k(v), j) \in D_j$, for all $j \in \mathrm{dom}(V_k)$, which completes this direction of the proof.

($\supseteq$) Let $d \in \prod_{j\in J'} D_j$. Then we only have to prove that there exists a $v \in V$ such that $\langle v\rangle_V = d$, or equivalently, that there exists a $v \in V$ such that, for all $j \in J'$, $u_V(v, j) = \mathrm{proj}_j(d)$. We do this by structural induction. Assume that $J' = \{j\}$ and $V = D_j$. Now set $v = \mathrm{proj}_j(d)$. Then $u_V(v, j) = v = \mathrm{proj}_j(d)$. Next assume that $V = \prod_{l\in L} V_l$. Then, from the induction hypothesis, it follows that for all $l \in L$, $\{\langle v_l\rangle_{V_l} | v_l \in V_l\} = \prod_{j\in J_l} D_j$ where $J_l = \mathrm{dom}(V_l)$. Hence, for all $l \in L$ and for all $j \in J_l$, we have a $v_l \in V_l$ such that $u_{V_l}(v_l, j) = \mathrm{proj}_j(d) \in D_j$. Let $v \in V$ be such that, for all $l \in L$, $\mathrm{proj}_l(v) = v_l$ with $v_l \in V_l$. Then, for all $j \in J'$, $u_V(v, j) = u_{V_l}(\mathrm{proj}_l(v), j)$, where l is such that $j \in \mathrm{dom}(V_l)$. Since, for all $l \in L$, $u_{V_l}(\mathrm{proj}_l(v), j) = u_{V_l}(v_l, j) = \mathrm{proj}_j(d)$, this completes also this direction of the proof. □

Lemma 5. Let $\mathcal{T} = (P, (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int}), \gamma, J)$ be an iterated team automaton over the composable system $\mathcal{S}$. Let $\mathcal{Q} = \{Q_i | i \in \mathcal{I}\}$. Then
(1) $P \in \mathcal{V}(\mathcal{Q})$ and $\mathrm{dom}(P) = \mathcal{I}$,
(2) $\{\langle q\rangle_P | q \in P\} = \prod_{i\in\mathcal{I}} Q_i$ and
(3) $\{\langle q\rangle_P | q \in J\} = \prod_{i\in\mathcal{I}} I_i$.

Proof. If $\mathcal{T}$ is a team automaton over $\mathcal{S}$, then $P = \prod_{i\in\mathcal{I}} Q_i$ and $J = \prod_{i\in\mathcal{I}} I_i$. By Definition A1(2), $P \in \mathcal{V}(\mathcal{Q})$ and $\mathrm{dom}(P) = \bigcup_{i\in\mathcal{I}} \mathrm{dom}(Q_i) = \mathcal{I}$. By Lemma 4, $\{\langle q\rangle_P | q \in P\} = \prod_{i\in\mathcal{I}} Q_i$. Since, according to Definition A2, for all $q \in P$, $\langle q\rangle_P = \prod_{i\in\mathcal{I}} u_P(q, i) = \prod_{i\in\mathcal{I}} u_{Q_i}(\mathrm{proj}_i(q), i) = \prod_{i\in\mathcal{I}} \mathrm{proj}_i(q) = q$, it follows that $\{\langle q\rangle_P | q \in J\} = \{q | q \in \prod_{i\in\mathcal{I}} I_i\} = \prod_{i\in\mathcal{I}} I_i$.

Now assume that $\mathcal{T}$ is an iterated team over $\mathcal{S}$. Hence $\mathcal{T}$ is a team over a composable system $\{\mathcal{T}_j | j \in \mathcal{J}\}$, where $\mathcal{J} \subseteq \mathbb{N}$, $\{\mathcal{I}_j | j \in \mathcal{J}\}$ forms a partition of $\mathcal{I}$, and each $\mathcal{T}_j$ is an iterated team over $\{\mathcal{C}_i | i \in \mathcal{I}_j\}$. Let, for $j \in \mathcal{J}$, $\mathcal{T}_j$ be specified as $\mathcal{T}_j = (P_j, (\Gamma_{j,inp}, \Gamma_{j,out}, \Gamma_{j,int}), \gamma_j, J_j)$. Hence $P = \prod_{j\in\mathcal{J}} P_j$ and $J = \prod_{j\in\mathcal{J}} J_j$. As induction hypothesis we assume that, for all $j \in \mathcal{J}$, $P_j \in \mathcal{V}(\mathcal{Q})$ with $\mathrm{dom}(P_j) = \mathcal{I}_j$, and $\{\langle q\rangle_{P_j} | q \in J_j\} = \prod_{i\in\mathcal{I}_j} I_i$. Since $\{\mathcal{I}_j | j \in \mathcal{J}\}$ forms a partition of $\mathcal{I}$, we immediately have $P = \prod_{j\in\mathcal{J}} P_j \in \mathcal{V}(\mathcal{Q})$ and $\mathrm{dom}(P) = \bigcup_{j\in\mathcal{J}} \mathrm{dom}(P_j) = \bigcup_{j\in\mathcal{J}} \mathcal{I}_j = \mathcal{I}$. By Lemma 4, $\{\langle q\rangle_P | q \in P\} = \prod_{i\in\mathcal{I}} Q_i$. Finally, $q \in J$ if and only if $\mathrm{proj}_j(q) \in J$, for all $j \in \mathcal{J}$. By the induction hypothesis, for all $j \in \mathcal{J}$, $\mathrm{proj}_j(q) \in J_j$ if and only if $\langle \mathrm{proj}_j(q)\rangle_{P_j} = \prod_{i\in\mathcal{I}_j} u_{P_j}(\mathrm{proj}_j(q), i) \in \prod_{i\in\mathcal{I}_j} I_i$. Thus, $q \in J$ if and only if, for all $j \in \mathcal{J}$ and for all $i \in \mathcal{I}_j$, $u_{P_j}(\mathrm{proj}_j(q), i) \in I_i$. Since, for all $q \in P$, $\langle q\rangle_P = \prod_{i\in\mathcal{I}} u_P(q, i) = \prod_{i\in\mathcal{I}} u_{P_{k_i}}(\mathrm{proj}_{k_i}(q), i)$, where $k_i \in \mathcal{J}$ is such that $i \in \mathrm{dom}(P_{k_i})$, it follows that $\{\langle q\rangle_P | q \in J\} = \prod_{i\in\mathcal{I}} I_i$. □

Lemma 6. Let $\mathcal{T} = (P, (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int}), \gamma, J)$ be an iterated team automaton over the composable system $\mathcal{S}$. Then

(1) $\Gamma_{int} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,int}$,
$\Gamma_{out} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,out}$ and
$\Gamma_{inp} = (\bigcup_{i \in \mathcal{I}} \Sigma_{i,inp}) \setminus \Gamma_{out}$, and
(2) $\{(\langle q \rangle_P, \langle q' \rangle_P) | (q, q') \in \gamma_a\} \subseteq \Delta_a(\mathcal{S})$, for all $a \in \Gamma_{inp} \cup \Gamma_{out} \cup \Gamma_{int}$.

Proof. If $\mathcal{T}$ is a team over $\mathcal{S}$, then (1) follows immediately from Definition 5. In that case, also (2) follows from Definition 5 because, as in the proof of Lemma 5, $\langle q \rangle_P = q$, for all $q \in P$.

Now assume that $\mathcal{T}$ is a team over $\{\mathcal{T}_j | j \in \mathcal{J}\}$, where $\mathcal{J} \subseteq \mathbb{N}$, and each $\mathcal{T}_j = (P_j, (\Gamma_{j,inp}, \Gamma_{j,out}, \Gamma_{j,int}), \gamma_j, J_j)$ is an iterated team over $\{\mathcal{C}_i | i \in \mathcal{I}_j\}$, with $\{\mathcal{I}_j | j \in \mathcal{J}\}$ forming a partition of $\mathcal{I}$. Assume furthermore inductively that, for all $j \in \mathcal{J}$, $\Gamma_{j,int} = \bigcup_{i \in \mathcal{I}_j} \Sigma_{i,int}$, $\Gamma_{j,out} = \bigcup_{i \in \mathcal{I}_j} \Sigma_{i,out}$ and $\Gamma_{j,inp} = (\bigcup_{i \in \mathcal{I}_j} \Sigma_{i,inp}) \setminus \Gamma_{j,out}$. Then $\Gamma_{int} = \bigcup_{j \in \mathcal{J}} \Gamma_{j,int} = \bigcup_{j \in \mathcal{J}} \bigcup_{i \in \mathcal{I}_j} \Sigma_{i,int} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,int}$, by Definition 5, and because $\{\mathcal{I}_j | j \in \mathcal{J}\}$ forms a partition of $\mathcal{I}$. Similarly, $\Gamma_{out} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,out}$. Finally, $\Gamma_{inp} = (\bigcup_{j \in \mathcal{J}} \Gamma_{j,inp}) \setminus \Gamma_{out}$, by Definition 5. Hence $\Gamma_{inp} = (\bigcup_{j \in \mathcal{J}} ((\bigcup_{i \in \mathcal{I}_j} \Sigma_{i,inp}) \setminus \Gamma_{j,out})) \setminus \Gamma_{out} = (\bigcup_{j \in \mathcal{J}} ((\bigcup_{i \in \mathcal{I}_j} \Sigma_{i,inp}) \setminus \bigcup_{i \in \mathcal{I}_j} \Sigma_{i,out})) \setminus \Gamma_{out} = (\bigcup_{i \in \mathcal{I}} \Sigma_{i,inp}) \setminus \Gamma_{out}$.

Next consider the transitions of $\mathcal{T}$. Let $a \in \Gamma_{inp} \cup \Gamma_{out} \cup \Gamma_{int}$. As $\mathcal{T}$ is a team over $\{\mathcal{T}_j | j \in \mathcal{J}\}$, we know that $\gamma_a \subseteq \Delta_a(\{\mathcal{T}_j | j \in \mathcal{J}\})$. We have to prove that upto the reordering relative to the construction of P, every a-transition of $\mathcal{T}$ is an element of the complete transition space of a in $\mathcal{S}$. In order to prove this, we make inductively the following assumption. For all $j \in \mathcal{J}$, $\{(\langle p \rangle_{P_j}, \langle p' \rangle_{P_j}) | (p, p') \in \gamma_{j,a}\} \subseteq \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_j\})$.

Before continuing, we make the following auxiliary observation:
Let $q \in P$. By Lemma 5, $\langle q \rangle_P \in \prod_{i \in \mathcal{I}} Q_i$, thus $\langle q \rangle_P = \prod_{i \in \mathcal{I}} \text{proj}_i(\langle q \rangle_P)$. Let $i \in \mathcal{I}$. By Definition A2, $\text{proj}_i(\langle q \rangle_P) = u_P(q, i) = u_{P_j}(\text{proj}_j(q), i)$, where j is such that $i \in \mathcal{I}_j$. Now $\text{proj}_j(q) \in P_j$ and hence, again by Lemma 5, $\langle \text{proj}_j(q) \rangle_{P_j} \in \prod_{i \in \mathcal{I}_j} Q_i$. By Definition A2 once again, $\text{proj}_i(\langle \text{proj}_j(q) \rangle_{P_j}) = u_{P_j}(\text{proj}_j(q), i)$, whenever $i \in \mathcal{I}_j$. Hence, $\text{proj}_i(\langle q \rangle_P) = \text{proj}_i(\langle \text{proj}_j(q) \rangle_{P_j})$, for all $q \in P$, $i \in \mathcal{I}_j$, and $j \in \mathcal{J}$. This ends the observation.

Let now $(q, q') \in \gamma_a$. In order to prove that $(\langle q \rangle_P, \langle q' \rangle_P) \in \Delta_a(\mathcal{S})$, we verify the two conditions in Definition 4. First we prove there exists an $i \in \mathcal{I}$ such that $\text{proj}_i^{[2]}(\langle q \rangle_P, \langle q' \rangle_P) \in \delta_{i,a}$. Let $j \in \mathcal{J}$ be such that $\text{proj}_j^{[2]}(q, q') \in \gamma_{j,a}$. Such a j exists, because $\gamma_a \subseteq \Delta_a(\{\mathcal{T}_j | j \in \mathcal{J}\})$. By the induction hypothesis, $(\langle \text{proj}_j(q) \rangle_{P_j}, \langle \text{proj}_j(q') \rangle_{P_j}) \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_j\})$. Hence, by Definition 4, there exists an $i \in \mathcal{I}_j$ such that $\text{proj}_i^{[2]}(\langle \text{proj}_j(q) \rangle_{P_j}, \langle \text{proj}_j(q') \rangle_{P_j}) \in \delta_{i,a}$. Thus, by our observation above, for this i, $\text{proj}_i^{[2]}(\langle q \rangle_P, \langle q' \rangle_P) \in \delta_{i,a}$, as desired. Secondly, we prove that, for all $i \in \mathcal{I}$, either $\text{proj}_i^{[2]}(\langle q \rangle_P, \langle q' \rangle_P) \in \delta_{i,a}$ or $\text{proj}_i(\langle q \rangle_P) = \text{proj}_i(\langle q' \rangle_P)$. Let $i \in \mathcal{I}$ and $j \in \mathcal{J}$ be such that $i \in \mathcal{I}_j$. As $\{\mathcal{I}_j | j \in \mathcal{J}\}$ forms a partition of $\mathcal{I}$ such j exists and is unique. As $\gamma_a \subseteq \Delta_a(\{\mathcal{T}_j | j \in \mathcal{J}\})$, Definition 4 implies either $\text{proj}_j^{[2]}(q, q') \in \gamma_{j,a}$ or $\text{proj}_j(q) = \text{proj}_j(q')$. If $\text{proj}_j^{[2]}(q, q') \in \gamma_{j,a}$, then $(\langle \text{proj}_j(q) \rangle_{P_j}, \langle \text{proj}_j(q') \rangle_{P_j}) \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_j\})$, by the induction hypothesis. Hence, by Definition 4, either $\text{proj}_i^{[2]}(\langle \text{proj}_j(q) \rangle_{P_j}, \langle \text{proj}_j(q') \rangle_{P_j}) \in \delta_{i,a}$, which – by the above auxiliary observation – implies that $\text{proj}_i^{[2]}(\langle q \rangle_P, \langle q' \rangle_P) \in \delta_{i,a}$, or $\text{proj}_i(\langle \text{proj}_j(q) \rangle_{P_j}) = \text{proj}_i(\langle \text{proj}_j(q') \rangle_{P_j})$, which – again by the above auxiliary observation – implies that $\text{proj}_i(\langle q \rangle_P) = \text{proj}_i(\langle q' \rangle_P)$. If $\text{proj}_j(q) = \text{proj}_j(q')$, then

$\text{proj}_i(\langle q\rangle_P) = u_{P_j}(\text{proj}_j(q), i) = u_{P_j}(\text{proj}_j(q'), i) = \text{proj}_i(\langle q'\rangle_P)$, which completes the proof. □

Theorem 1. Let $\mathcal{T} = (Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, I)$ be a team automaton over the composable system $\mathcal{S}$ and let $\{\mathcal{I}_j | j \in \mathcal{J}\}$, where $\mathcal{J} \subseteq \mathbb{N}$, form a partition of $\mathcal{I}$. Let, for each $j \in \mathcal{J}$, $\mathcal{T}_j = (P_j, (\Gamma_{j,inp}, \Gamma_{j,out}, \Gamma_{j,int}), \gamma_j, J_j)$ be an iterated team over $\{\mathcal{C}_i | i \in \mathcal{I}_j\}$. Then

(1) if $(\delta_{\mathcal{I}_j})_a \subseteq \{(\langle q\rangle_{P_j}, \langle q'\rangle_{P_j}) | (q, q') \in \gamma_{j,a}\}$, for all $a \in \Gamma_{j,inp} \cup \Gamma_{j,out} \cup \Gamma_{j,int}$ for all $j \in \mathcal{J}$, then there exists a team automaton $\hat{\mathcal{T}}$ over $\{\mathcal{T}_j | j \in \mathcal{J}\}$ such that $\langle\langle \hat{\mathcal{T}}\rangle\rangle_{\mathcal{S}} = \mathcal{T}$ and

(2) if $\hat{\mathcal{T}}$ is a team automaton over $\{\mathcal{T}_j | j \in \mathcal{J}\}$, then $\langle\langle \hat{\mathcal{T}}\rangle\rangle_{\mathcal{S}} = \mathcal{T}$ implies that $(\delta_{\mathcal{I}_j})_a \setminus \{(p, p) | (p, p) \in \Delta_a(\{\mathcal{C}_i | i \in \mathcal{I}_j\})\} \subseteq \{(\langle q\rangle_{P_j}, \langle q'\rangle_{P_j}) | (q, q') \in \gamma_{j,a}\}$, for all $a \in \Gamma_{j,inp} \cup \Gamma_{j,out} \cup \Gamma_{j,int}$ for all $j \in \mathcal{J}$.

Proof. Let $\hat{\mathcal{T}} = (P, (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int}), \gamma, J)$ be an arbitrary team automaton over $\{\mathcal{T}_j | j \in \mathcal{J}\}$.

First we make an auxiliary observation similar to the one in the proof of Lemma 6. Let $q \in P$ and let $j \in \mathcal{J}$. Then $\text{proj}_{\mathcal{I}_j}(\langle q\rangle_P) = \langle \text{proj}_j(q)\rangle_{P_j}$, since $P = \prod_{j\in\mathcal{J}} P_j$ and, by Lemma 5, $\prod_{i\in\mathcal{I}_j} Q_i = \{\langle q\rangle_{P_j} | q \in P_j\}$.

(1) Assume that $(\delta_{\mathcal{I}_j})_a \subseteq \{(\langle q\rangle_{P_j}, \langle q'\rangle_{P_j}) | (q, q') \in \gamma_{j,a}\}$. By Lemmata 5 and 6, we know that $Q = \{\langle q\rangle_P | q \in P\}$, $(\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}) = (\Gamma_{inp}, \Gamma_{out}, \Gamma_{int})$ and $I = \{\langle q\rangle_J | q \in J\}$, respectively. Thus it only remains to prove that the transition relation γ for $\hat{\mathcal{T}}$ can be chosen such that $\delta = \{(\langle q\rangle_P, a, \langle q'\rangle_P) | q, q' \in P, (q, a, q') \in \gamma\}$. Hence, using the injectivity of reordering, we define γ simply by $\gamma_a = \{(q, q') \in \prod_{j\in\mathcal{J}} P_j \times \prod_{j\in\mathcal{J}} P_j | (\langle q\rangle_P, \langle q'\rangle_P) \in \delta_a\}$, for all $a \in \Gamma_{inp} \cup \Gamma_{out} \cup \Gamma_{int}$ and prove this is indeed the transition relation of a team over $\{\mathcal{T}_j | j \in \mathcal{J}\}$.

Let $(p, p') \in \gamma_a$. First we prove there exists a $j \in \mathcal{J}$ such that $\text{proj}_j{}^{[2]}(p, p') \in \gamma_{j,a}$. As $(\langle p\rangle_P, \langle p'\rangle_P) \in \delta_a$, there exists an $i \in \mathcal{I}$ such that $\text{proj}_j{}^{[2]}(\langle p\rangle_P, \langle p'\rangle_P) \in \delta_{i,a}$. Let j be such that $i \in \mathcal{I}_j$. Then it follows that $\text{proj}_{\mathcal{I}_j}{}^{[2]}(\langle p\rangle_P, \langle p'\rangle_P) \in (\delta_{\mathcal{I}_j})_a$. Since $(\delta_{\mathcal{I}_j})_a \subseteq \{(\langle q\rangle_{P_j}, \langle q'\rangle_{P_j}) | (q, q') \in \gamma_{j,a}\}$, there exists an $(r, r') \in \gamma_{j,a}$ such that $(\langle r\rangle_{P_j}, \langle r'\rangle_{P_j}) = \text{proj}_{\mathcal{I}_j}{}^{[2]}(\langle p\rangle_P, \langle p'\rangle_P)$. Thus, by the observation above, $(\langle r\rangle_{P_j}, \langle r'\rangle_{P_j}) = (\langle \text{proj}_j(p)\rangle_{P_j}, \langle \text{proj}_j(p')\rangle_{P_j})$. Since reordering is injective, it follows that $r = \text{proj}_j(p)$ and $r' = \text{proj}_j(p')$ and thus $\text{proj}_j{}^{[2]}(p, p') = (r, r') \in \gamma_{j,a}$.

It now remains to prove that, for all $j \in \mathcal{J}$, either $\text{proj}_j(p) = \text{proj}_j(p')$ or $\text{proj}_j{}^{[2]}(p, p') \in \gamma_{j,a}$. Let $j \in \mathcal{J}$ be such that $\text{proj}_j(p) \neq \text{proj}_j(p')$. Then we only have to prove that $\text{proj}_j{}^{[2]}(p, p') \in \gamma_{j,a}$. Since $(p, p') \in \gamma_a$, we have $(\langle p\rangle_P, \langle p'\rangle_P) \in \delta_a$. By the observation above, $\text{proj}_{\mathcal{I}_j}(\langle p\rangle_P) = \langle \text{proj}_j(p)\rangle_{P_j}$ and $\text{proj}_{\mathcal{I}_j}(\langle p'\rangle_P) = \langle \text{proj}_j(p')\rangle_{P_j}$. Since reordering is an injective operation, we infer $\text{proj}_{\mathcal{I}_j}(\langle p\rangle_P) \neq \text{proj}_{\mathcal{I}_j}(\langle p'\rangle_P)$. Hence, $\text{proj}_{\mathcal{I}_j}{}^{[2]}(\langle p\rangle_P, \langle p'\rangle_P) \in (\delta_{\mathcal{I}_j})_a$. Since $(\delta_{\mathcal{I}_j})_a \subseteq \{(\langle q\rangle_{P_j}, \langle q'\rangle_{P_j}) | (q, q') \in \gamma_{j,a}\}$, it follows that $\text{proj}_j{}^{[2]}(p, p') \in \gamma_{j,a}$.

(2) Now assume that $\langle\langle \hat{\mathcal{T}}\rangle\rangle_{\mathcal{S}} = \mathcal{T}$. Let $j \in \mathcal{J}$ and $a \in \Gamma_{inp} \cup \Gamma_{out} \cup \Gamma_{int}$ be fixed. Let $(p, p') \in (\delta_{\mathcal{I}_j})_a$ be such that $p \neq p'$. By Definition 6, there is a pair $(r, r') \in \delta_a$ such that $\text{proj}_{\mathcal{I}_j}{}^{[2]}(r, r') = (p, p')$. Since $\langle\langle \hat{\mathcal{T}}\rangle\rangle_{\mathcal{S}} = \mathcal{T}$, there are

$(\hat{r}, \hat{r}') \in \gamma_a$ such that $(\langle \hat{r} \rangle_P, \langle \hat{r}' \rangle_P) = (r, r')$. By the observation above, $(p, p') = \text{proj}_{\mathcal{I}_j}{}^{[2]}(r, r') = (\langle \text{proj}_j(\hat{r}) \rangle_{P_j}, \langle \text{proj}_j(\hat{r}') \rangle_{P_j})$ and thus the only thing left to prove here is that $(\text{proj}_j(\hat{r}), \text{proj}_j(\hat{r}')) \in \gamma_{j,a}$. Assume to the contrary that this is not the case. Then the fact that $\hat{\mathcal{T}}$ is a team over $\{\mathcal{T}_j \mid j \in \mathcal{J}\}$ together with $(\hat{r}, \hat{r}') \in \gamma_a$ implies that $\text{proj}_j(\hat{r}) = \text{proj}_j(\hat{r}')$ and thus $p = p'$. A contradiction. Thus $(\text{proj}_j(\hat{r}), \text{proj}_j(\hat{r}')) \in \gamma_{j,a}$. □

Acknowledgements

We are grateful to the anonymous referees for their suggestions to improve an earlier version of this paper.

References

Arnold, A. (1982): Synchronized Behaviours of Processes and Rational Relations. *Acta Informatica*, vol. 17, pp. 21–29.

Badouel, E., Ph. Darondeau, D. Quichaud and A. Tokmakoff (1999): Modelling Dynamic Agent Systems with Cooperating Automata. Publication Interne 1253, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes.

Beek, M.H. ter, C.A. Ellis, J. Kleijn and G. Rozenberg (2001a): Team Automata for CSCW. In H. Weber, H. Ehrig and W. Reisig (eds.): *Proceedings of the 2nd International Colloquium on Petri Net Technologies for Modelling Communication Based Systems. Berlin, Germany, September 14 to 15, 2001*. Berlin: Fraunhofer Institute for Software and Systems Engineering, pp. 1–20. (Also appeared as Technical Report TR-01-07, Leiden Institute of Advanced Computer Science, Universiteit Leiden, 2001.)

Beek, M.H. ter, C.A. Ellis, J. Kleijn and G. Rozenberg (2001b): Team Automata for Spatial Access Control. In W. Prinz, M. Jarke, Y. Rogers, K. Schmidt, and V. Wulf (eds.): *ECSCW 2001. Proceedings of the Seventh European Conference on Computer Supported Cooperative Work, Bonn, Germany, September 16 to 20, 2001*. Dordrecht: Kluwer Academic Publishers, pp. 59–77. (Also appeared as Technical Report TR-01-03, Leiden Institute of Advanced Computer Science, Universiteit Leiden, 2001.)

Diekert, V. and G. Rozenberg (1995): *Book of Traces*. Singapore: World Scientific.

Duboc, C. (1986): Mixed Product and Asynchronous Automata. *Theoretical Computer Science*, vol. 42, pp. 183–199.

Ellis, C.A., S.J. Gibbs and G. Rein (1990): Design and Use of a Group Editor. In G. Cockton (ed.): *Engineering for Human Computer Interaction*. Amsterdam: North-Holland Publ. Co., pp. 13–25.

Ellis, C.A. (1997): Team Automata for Groupware Systems. In S.C. Hayne and W. Prinz (eds.): *GROUP'97. Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge, Phoenix, Arizona, November 16 to 19, 1997*. New York: ACM Press, pp. 415–424.

Ellis, C.A. and G.J. Nutt (1993): Modelling and Enactment of Workflow Systems. In M. Ajmone Marsan (ed.): *Proceedings of the International Conference on Application and Theory of Petri Nets, Chicago, U.S.A.. Lecture Notes in Computer Science*, vol. 691. Berlin: Springer-Verlag, pp. 1–16.

Gawlick, R., R. Segala, F.F. Søgaard-Andersen and N. Lynch (1994): Liveness in Timed and Untimed Systems. In S. Abiteboul and E. Shamir (eds.): *ICALP'94. Proceedings of the International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science*, vol. 820. Berlin: Springer-Verlag, pp.166–177. (A full version appeared as Technical Report MIT/LCS/TR-587, Massachusetts Institute of Technology, Cambridge, Massachusetts.)

Janicki, R. and P.E. Laurer (1992): *Specification and Analysis of Concurrent Systems, The COSY Approach. EATCS Monograhps on Theoretical Computer Science*. Berlin: Springer-Verlag.

Keesmaat, N.W. and H.C.M. Kleijn (1997): Net-Based Control versus Rational Control: The Relation between ITNC Vector Languages and Rational Relations. *Acta Informatica*, vol. 34, pp. 23–57.

Keesmaat, N.W., H.C.M. Kleijn and G. Rozenberg (1990): Vector Controlled Concurrent Systems, Part I: Basic Classes. *Fundamenta Informaticae*, vol. 13, pp. 275–316.

Keesmaat, N.W., H.C.M. Kleijn and G. Rozenberg (1991): Vector Controlled Concurrent Systems, Part II: Comparisons. *Fundamenta Informaticae*, vol. 14, pp. 1–38.

Lynch, N.A. and M.R. Tuttle (1989): An Introduction to Input/Output Automata. *CWI Quarterly*, vol. 2, no. 3, pp. 219–246. (Also appeared as Technical Memo MIT/LCS/TM-373, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1988.)

Lynch, N.A. (1996): *Distributed Algorithms*. San Mateo, California: Morgan Kaufmann Publishers.

Mazurkiewicz, A. (1989): Basic Notions of Trace Theory. In *Lecture Notes in Computer Science*, vol. 354. Berlin: Springer-Verlag, pp. 285–363.

Reisig, W. and G. Rozenberg (eds.) (1998): *Lectures on Petri Nets I: Basic Models*. *Lecture Notes in Computer Science*, vol. 1491. Berlin: Springer-Verlag.

Sun, C. and C.A. Ellis (1998): Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In J. Grudin and S.E. Poltrock (eds.): *CSCW'98. Proceedings of the ACM Conference on Computer Supported Cooperative Work, Seattle, Washington, November 14 to 18, 1998*. New York: ACM Press, pp. 59–68.

Tuttle, M.R. (1987): *Hierarchical Correctness Proofs for Distributed Algorithms*. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts. (Also appeared as Technical Report MIT/LCS/TR-387, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1987.)

Zielonka, W. (1987): Notes on Finite Asynchronous Automata. *RAIRO Informatique Théoretique et Applications*, vol. 21, pp. 99–135.