Team Automata for CSCW

Maurice H. ter Beek[1], Clarence A. Ellis[2],
Jetty Kleijn[1] and Grzegorz Rozenberg[1,2]

15 June 2001

Abstract

Team automata have been proposed as a formal framework for modelling both the conceptual and the architectural level of groupware systems. They are defined in terms of component automata together with an interconnection mechanism which is based on shared actions (synchronizations). Components can be combined in a loose or more tight fashion depending on which actions are to be shared, and when. The formal set-up makes it possible to distinguish between, e.g., master-slave and peer-to-peer synchronizations and to classify team automata based on the mode of synchronization. Since a team automaton can be used as a component in a higher-level team, the framework allows for the representation of hierarchical systems. As an example, using a spatial access metaphor, we will consider some access control strategies in the context of team automata.

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 of such conceptual ideas is enhanced when they are accompanied by formal versions which require an exact formulation of basic features. Such formalizations make analysis possible and properties can be rigourously proved. With the increase of the complexity of groupware systems, abstractions tend to be especially useful. Thus, CSCW has a need for developing a precise and consistent terminology. Moreover, at the architectural level, CSCW needs a rigorous framework to describe, compare and contrast groupware systems.

Team automata have been introduced in [13] explicitly for the specification and analysis of CSCW phenomena and groupware systems. The set-up of the model makes it possible to clarify and capture precisely notions related to coordination 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". 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.

Team automata thus fit nicely with the needs and the philosophy of groupware systems. Moreover, thanks to the formal automata theoretic set-up, 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 model has proved useful in various CSCW modelling areas. A spectrum from hardware components to protocols for interacting groups of people can be modelled by team automata.

Team automata are an extension of and inspired by Input/Output automata (see, e.g., [23]). They are related to Vector Controlled Concurrent Systems (see, e.g., [18], [20] and [21]), Petri nets (see, e.g., [29] and [30]), and other models of concurrent and collaborative systems ([28]).

This paper surveys some aspects of team automata as investigated in [3] and employed in [4], where access control mechanisms are considered in the context of the team automata model. Proofs of claims made in the sequel can mostly be found in those papers. A few definitions in a very condensed form are given only to avoid ambiguities and may be skipped on first reading.

[1]Leiden Institute of Advanced Computer Science, Universiteit Leiden, P.O. Box 9512, 2300 RA, Leiden, The Netherlands, {mtbeek,kleijn,rozenber}@liacs.nl
[2]Department of Computer Science, University of Colorado, Boulder, CO 80309-0430, U.S.A., skip@colorado.edu

The organization of the paper is as follows. First team automata are introduced, followed by an exposition of their possibilities to define hierarachies by iteration. Next we focus on different types of synchronization. Then the question of how to (uniquely) construct team automata satisfying certain synchronization constraints is considered. To conclude this more or less technical survey team automata are compared with I/O automata and Vector Controlled Concurrent Systems. All this is accompanied by a few small examples intended to illuminate some of the technical details. In a final section we present a more realistic example which demonstrates how some well-known access control policies can be given a rigorous formal description in terms of synchronizations in team automata. The conclusion of the paper reflects on the use of team automata in the design of CSCW systems.

2 Some notation

The following notations are used frequently throughout this exposition. 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 $\{(v_{i_1}, v_{i_2}, \ldots) \mid v_{i_j} \in V_{i_j}$, 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 $\mathcal{I} = \emptyset$, then $\prod_{i\in\mathcal{I}} V_i = \emptyset$. 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 $\mathrm{proj}_J(a) = \prod_{j\in J} \mathrm{proj}_j(a)$. We use $\mathrm{proj}_i^{[2]}$ and $\mathrm{proj}_J^{[2]}$ as shorthand notations for double projections, thus $\mathrm{proj}_i^{[2]}(a,b) = (\mathrm{proj}_i(a), \mathrm{proj}_i(b))$ and $\mathrm{proj}_J^{[2]}(a,b) = (\mathrm{proj}_J(a), \mathrm{proj}_J(b))$.

3 Team automata

A team automaton consists of component automata, combined in a coordinated way such that they can perform shared actions. Within a team, component automata can simultaneously participate in an action (i.e. synchronize on this action) or remain idle. The choice for a specific interconnection strategy (which components synchronize on what actions, and when) is based on what one wants to model, and this choice gives the team automata framework a high level of flexibility.

The basic concept underlying both team and component automata is a labelled transition system, 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 as described by the labelled transitions in its transition relation. These transition systems come equipped with a set of initial states from which they can start their executions.

Definition An *initialized labelled transition system* is a construct $\mathcal{A} = (Q, \Sigma, \delta, I)$, where Q is its set of *states*, possibly infinite, Σ its alphabet of *actions*, such that $Q \cap \Sigma = \emptyset$, $\delta \subseteq Q \times \Sigma \times Q$ its set of (*labelled*) *transitions* and $I \subseteq Q$ its set of *initial states*.

Let $a \in \Sigma$. The set of *a-transitions* of $\mathcal{A}$ is denoted by δ_a and is defined as $\delta_a = \{(q,q') \mid (q,a,q') \in \delta\}$. An a-transition $(q,q) \in \delta_a$ is called a *loop* (on a) and a is *enabled* in an automaton at a state q if there is a state q' such that $(q,q') \in \delta_a$.

The set of (finite) *computations* of $\mathcal{A}$ is defined as $\{q_0 a_1 q_1 a_2 \cdots a_n q_n \mid q_0 \in I,\ n \geq 0$ and $(q_i, a_{i+1}, q_{i+1}) \in \delta$, for all $0 \leq i < n\}$.

A component automaton is an initialized labelled transition system (ilts for short) 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 [13]: "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.

Definition 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 Σ_{inp}, Σ_{out} and Σ_{int} are mutually disjoint alphabets called the *input*, *output* and *internal* alphabet of $\mathcal{C}$, respectively.

The *computations* of a component automaton are the computations of the underlying ilts. Given a computation one may choose to focus on certain details while filtering away others. In this way, records are made of computations and a certain notion of behaviour can be chosen. A standard behavioural notion obtained in this way is the *language* of the system, which is derived from the computations by deleting the states while preserving the actions. In general however, given the different roles actions may have in a component automaton one may also opt, e.g., for

information on only external actions. In this exposition, we will mainly refer to computations and only occasionally say something about behaviour.

We now turn to the definition of a team automaton formed from component automata. Let us fix $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$, a collection of component automata. Here $\mathcal{I}$ is a (possibly infinite) set of positive integers used 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.

When composing a team automaton from $\mathcal{S}$, we require first of all that the internal actions of these components are private, i.e. uniquely associated to one component automaton. This is formally expressed by stating that, for all $i \in \mathcal{I}$, $\Sigma_{i,int} \cap \bigcup_{j \in \mathcal{I} \setminus \{i\}} \Sigma_j = \emptyset$. Thus no internal action of any component automaton from $\mathcal{S}$ may appear as an action in any of the other component automata in $\mathcal{S}$. If this is the case, then $\mathcal{S}$ is called a *composable system*. Note that every subset of a composable system is again a composable system.

So let us assume for the sequel that $\mathcal{S}$ is a composable system. The state space of any team automaton $\mathcal{T}$ formed from $\mathcal{S}$ is 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.

Definition Let $a \in \bigcup_{i \in \mathcal{I}} \Sigma_i$. The *complete transition space* of a in $\mathcal{S}$ is denoted by $\Delta_a(\mathcal{S})$ and is defined as $\Delta_a(\mathcal{S}) = \{(q, q') \in \prod_{i \in \mathcal{I}} Q_i \times \prod_{i \in \mathcal{I}} Q_i \mid \exists j \in \mathcal{I} : proj_j{}^{[2]}(q, q') \in \delta_{j,a}$ and $(\forall i \in \mathcal{I} : [proj_i{}^{[2]}(q, q') \in \delta_{i,a}]$ or $[proj_i(q) = proj_i(q')])\}$.

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). $\Delta_a(\mathcal{S})$ 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 $\Delta_a(\mathcal{S})$. Consequently, the transformation of the state of the team automaton is defined by the local state changes of the components that participate in the action that is executed. When defining $\mathcal{T}$, for each action a, a specific subset of its complete transition space has to be chosen. For an internal action however, each component retains all its possibilities to execute that action and change state. Note that since $\mathcal{S}$ is a composable system, synchronizations on internal actions never involve more than one component.

The alphabets of actions of any team automaton 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 the team automaton. Each action which is output for one or more of the component automata is an output action of the team automaton. 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 [13] 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 A *team automaton over* $\mathcal{S}$ is a construct $\mathcal{T} = (\prod_{i \in \mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i \in \mathcal{I}} I_i)$ such that $\Sigma_{int} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,int}$, $\Sigma_{out} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,out}$, $\Sigma_{inp} = (\bigcup_{i \in \mathcal{I}} \Sigma_{i,inp}) \setminus \Sigma_{out}$, 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 $a \in \Sigma_{int}$.

The definition of the alphabets of actions of a team automaton over $\mathcal{S}$ together with the composability of $\mathcal{S}$ guarantee consistency in the sense that in a team automaton every action appears exclusively as an input, output or internal action. As a consequence, every team automaton is again a component automaton, which in its turn could be used as a component in a higher-level team, an issue to which we return later. Since team automata are component automata, their behaviour can be described using computations.

Summarizing, we note that 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. We are thus not concerned with so-called reduced automata, in which all states and transitions are useful, a notion which depends on the transition relation. Due to the freedom of choosing a δ_a for each external action a, a composable system 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 the composable system.

When designing a system as a team automaton, one chooses a specific transition relation with a specific protocol in mind. Certain synchronizations be-

tween component automata may be excluded even if the action could occur according to the current local states. 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., [23], [17] and [5]). 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 model. Leaving the set of transitions of a team automaton as a modelling choice is perhaps the most important feature of team automata.

Example 1 Let us now consider a simple example, in full described in [3] but originally from [13], where it was presented to illustrate the concept of team automata.

Consider the three component automata $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$, as depicted in Figure 1.

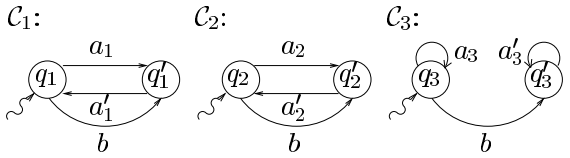

Fig. 1: Three component automata

Each $\mathcal{C}_i$ has two states q_i, q'_i with q_i as its only initial state. The actions a_i and a'_i are the internal actions of $\mathcal{C}_i$ and all a_i and a'_i are distinct symbols and different from b. Hence $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ is a composable system.

Action b is a common observable action of the three component automata. We choose b to be an output action of $\mathcal{C}_1$ and $\mathcal{C}_2$ and an input action of $\mathcal{C}_3$.

Now the transition relations are as depicted in Figure 1. Thus $\delta_{i,b} = \{(q_i, q'_i)\}$, for $i = 1, 2, 3$, and for $j = 1, 2$ we have $\delta_{j,a_j} = \{(q_j, q'_j)\}$ and $\delta_{j,a'_j} = \{(q'_j, q_j)\}$, and $\delta_{3,a_3} = \{(q_3, q_3)\}$ and $\delta_{3,a'_3} = \{(q'_3, q'_3)\}$.

Clearly, each team automaton $\mathcal{T}$ over $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ is of the form $\mathcal{T} = (\prod_{1 \leq i \leq 3} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \{(q_1, q_2, q_3)\})$ with $\Sigma_{inp} = \emptyset$, $\Sigma_{out} = \{b\}$, $\Sigma_{int} = \{a_1, a'_1, a_2, a'_2, a_3, a'_3\}$ and only δ as a variable parameter. In fact, since all a_i and a'_i are internal actions, $\delta_a = \Delta_a(\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\})$, for each $a \in \{a_1, a'_1, a_2, a'_2, a_3, a'_3\}$, and there is only a choice of which synchronizations on b to include in δ.

We consider different options leading to the three team automata $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$.

First of all, one could require that all and only those synchronizations on b are allowed that involve all components that have b as an (observable) action. In this case we set $\delta_b = \{((q_1, q_2, q_3), (q'_1, q'_2, q'_3))\}$. Hence in every computation of the resulting team automaton $\mathcal{T}_1$, action b is executed at most once.

Another possibility is the scenario that $\mathcal{C}_1$ and $\mathcal{C}_2$ synchronize on their output action b and, moreover, that the input action b of $\mathcal{C}_3$ cannot be executed unless it is caused by the output action b of the other components. Now one can make even more distinctions:

if $\mathcal{C}_3$ has to execute b whenever the other components do, we are back in the first situation;

if $\mathcal{C}_3$ has to execute b whenever the other components do, provided it is ready to do so (enabled), then we have $\delta_b = \{((q_1, q_2, q_3), (q'_1, q'_2, q'_3)), ((q_1, q_2, q'_3), (q'_1, q'_2, q'_3))\}$, and thus the resulting team automaton $\mathcal{T}_2$ has computations with any number of synchronizations on b, but $\mathcal{C}_3$ is involved only in the first one;

if $\mathcal{C}_3$ has an option to execute b or not whenever the other components do, then we have $\delta_b = \{((q_1, q_2, q_3), (q'_1, q'_2, q'_3)), ((q_1, q_2, q'_3), (q'_1, q'_2, q'_3)), ((q_1, q_2, q_3), (q'_1, q'_2, q_3)), \}$, and in this case team automaton $\mathcal{T}_3$ has computations with any number of synchronizations on b, with $\mathcal{C}_3$ involved in at most one of them.

Of course, there are many more possibilities, like dropping the requirement that $\mathcal{C}_1$ and $\mathcal{C}_2$ should always synchronize on b. Hence even this small example shows a range of possibilities for the design of connections between the components in a system.

A final observation to be made in this section is that within the formalization of a team automaton over a composable system, no explicit information on loops is provided. That is to say, in general one cannot distinguish whether or not a component automaton with a loop on a in its current local state takes part in the team automaton's synchronization on a. This component may either 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.

Example 2 Consider the three component automata $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$, as depicted in Figure 2(a).

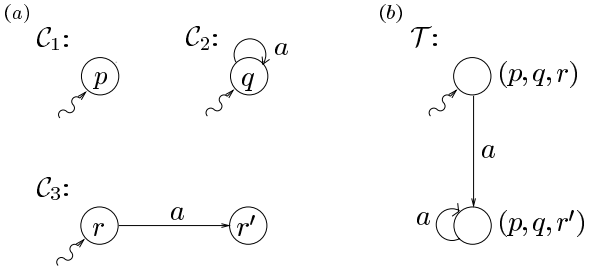

Fig. 2: Three component automata and a team automaton

$\mathcal{C}_1$ and $\mathcal{C}_2$ each have only one state, p and q, respectively, which are their initial states. $\mathcal{C}_3$ has two states, r and r', of which r is its initial state. Both $\mathcal{C}_2$ and $\mathcal{C}_3$ have $\{a\}$ as their output alphabet, while all other alphabets of the three component automata are empty. Hence $\{\mathcal{C}_1,\mathcal{C}_2,\mathcal{C}_3\}$ is a composable system. The transition relations are as depicted in Figure 2(a).

Consider the team automaton $\mathcal{T} = (\{(p,q,r), (p,q,r')\}, (\varnothing,\{a\},\varnothing), \delta, \{(p,q,r)\})$, with $\delta_a = \Delta_a(\{\mathcal{C}_1,\mathcal{C}_2,\mathcal{C}_3\}$, over $\{\mathcal{C}_1,\mathcal{C}_2,\mathcal{C}_3\}$. Now one might wonder which component automata participate when the a-transitions of this team are executed.

First consider the execution of the loop on a in $\mathcal{T}$. Clearly $\mathcal{C}_1$ does not participate as it cannot execute a at all. Also $\mathcal{C}_3$ does not participate since a is not enabled in r' (there is no a-transition leaving r'). However, since in every transition of a team automaton at least one component is required to participate, it must thus be the case that $\mathcal{C}_2$ does participate by executing its loop on a.

Secondly, consider the a-transition from (p,q,r) to (p,q,r') in $\mathcal{T}$. Clearly $\mathcal{C}_1$ is not involved. On the other hand, $\mathcal{C}_3$ is responsible for the local state change from r to r' and thus participates by executing a. But what about $\mathcal{C}_2$ — does it participate by executing its loop on a or does it remain idle during this execution of a by the team?

As we have seen in Example 2, information on the actual execution of loops by the components is lacking in the definition of a team automaton. Nevertheless, in order to relate the computations of a team to those taking place in the components one can simply apply projections. 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.

Definition 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 $\mathcal{S}$ and let $j \in \mathcal{I}$. The *projection on $\mathcal{C}_j$ of a computation w* of $\mathcal{T}$ is denoted by $\pi_{\mathcal{C}_j}(w)$ and is defined recursively as follows. (1) If $w = q$, for some $q \in \prod_{i\in\mathcal{I}} I_i$, then $\pi_{\mathcal{C}_j}(w) = \mathrm{proj}_j(q)$. (2) If $w = w'qaq'$, for some computation $w'q$ of $\mathcal{T}$, $q,q' \in \prod_{i\in\mathcal{I}} Q_i$ and $a \in \Sigma_{inp}\cup\Sigma_{out}\cup\Sigma_{int}$, then $\pi_{\mathcal{C}_j}(w) = \pi_{\mathcal{C}_j}(w'q)$ if $\mathrm{proj}_j{}^{[2]}(q,q') \notin \delta_{j,a}$ and $\pi_{\mathcal{C}_j}(w) = \pi_{\mathcal{C}_j}(w'q)a\mathrm{proj}_j(q')$ if $\mathrm{proj}_j{}^{[2]}(q,q') \in \delta_{j,a}$.

Since computations of team automata are sequences of synchronizations, a 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 is used in a computation of that team.

4 Subteams and iteration

For this and the next two sections we let $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$ as before be a fixed but arbitrary, composable system and we 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 $\mathcal{S}$.

By focussing 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 roles 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 component automata are considered which have this action as an output action.

Definition Let $J \subseteq \mathcal{I}$. 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 $a \in \Sigma_J = \bigcup_{j\in J}\Sigma_j$, $(\delta_J)_a = proj_J{}^{[2]}(\delta_a) \cap \Delta_a(\{\mathcal{C}_j \mid j \in J\})$.

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 component automata in $\{\mathcal{C}_j \mid j \in J\}$. Hence in each transition of the subteam at least one of the component automata is actively involved. This is formalized by the intersection of $proj_J{}^{[2]}(\delta_a)$ with $\Delta_a(\{\mathcal{C}_j \mid j \in J\})$, for each action a, because in each transition in this complete transition space at least one component from $\{\mathcal{C}_j \mid j \in \mathcal{J}\}$ is active.

Since $\{\mathcal{C}_j \mid j \in J\}$ is a composable system, it is clear that the subteam $SUB_J(\mathcal{T})$ of $\mathcal{T}$ is again a team automaton (over $\{\mathcal{C}_j \mid j \in J\}$). Subteams can thus be used as components and, as we will see, they can be used to iteratively define the team automaton from which they are derived.

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 straightforward to extend the definition of the projection of computations of $\mathcal{T}$ on a component $\mathcal{C}_j$ to projection on a subteam $SUB_{\mathcal{J}}(\mathcal{T})$ (it suffices to replace proj_j by proj_J). Also in this case the projection on $SUB_{\mathcal{J}}(\mathcal{T})$ of a team computation yields computations of $SUB_{\mathcal{J}}(\mathcal{T})$.

Until now we directly defined team automata from the component automata in $\mathcal{S}$, but other routes are also feasible. In particular, one might (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.

Example 3 Suppose that the composable system $\mathcal{S}$ consists of seven component automata. Thus $\mathcal{S} = \{\mathcal{C}_i \mid 1 \leq i \leq 7\}$. Let $\mathcal{T}_{1-7}$ be a team automaton over $\mathcal{S}$. Thus $\mathcal{T}_{1-7}$ has the form $(\prod_{1\leq i\leq 7} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{1\leq i\leq 7} I_i)$. The structure of $\mathcal{T}_{1-7}$ relative to the components in $\mathcal{S}$ is depicted in the tree of Figure 3(a).

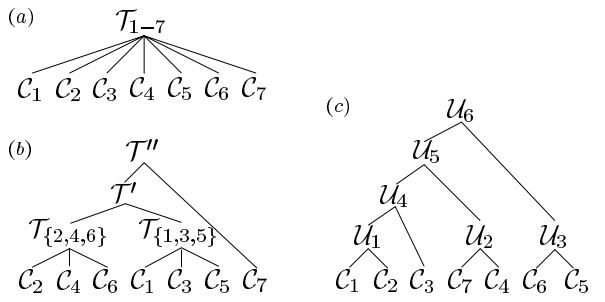

Fig. 3: Team automata constructed from $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_7\}$

Now recall that every subset of a composable system is itself a composable system. Thus one could form a team automaton $\mathcal{T}_{\{2,4,6\}}$ over $\{\mathcal{C}_2, \mathcal{C}_4, \mathcal{C}_6\}$ and a team automaton $\mathcal{T}_{\{1,3,5\}}$ over $\{\mathcal{C}_1, \mathcal{C}_3, \mathcal{C}_5\}$. Then a new team can be formed from $\mathcal{T}_{\{2,4,6\}}$ and $\mathcal{T}_{\{1,3,5\}}$ since — as observed earlier — every team automaton is a component automaton and, moreover, $\mathcal{C}'_1 = \mathcal{T}_{\{2,4,6\}}$ and $\mathcal{C}'_2 = \mathcal{T}_{\{1,3,5\}}$ together form a composable system $\mathcal{S}' = \{\mathcal{C}'_1, \mathcal{C}'_2\}$. That $\mathcal{S}'$ is a composable system is a consequence of the composability of $\{C_i \mid 1 \leq i \leq 7\}$ and the definition of team automata by which the internal actions of a team are exactly the internal actions of its components. Hence, the internal actions of $\mathcal{T}_{\{2,4,6\}}$ do not occur as actions in $\mathcal{T}_{\{1,3,5\}}$ and vice versa. In general, we have that teams over disjoint subsets of components from a composable system form again a composable system.

Now assume that a team automaton $\mathcal{T}'$ over $\mathcal{S}'$ has been constructed. Then, with the same reasoning as above, $\mathcal{T}'$ and $\mathcal{C}_7$ together form a composable system $\mathcal{S}'' = \{\mathcal{C}''_1, \mathcal{C}''_2\}$, with $\mathcal{C}''_1 = \mathcal{T}'$ and $\mathcal{C}''_2 = \mathcal{C}_7$ and we can define a team automaton $\mathcal{T}''$ over $\mathcal{S}''$. The structure of such $\mathcal{T}''$ relative to the components in $\mathcal{S}$, is depicted in the tree of Figure 3(b).

In Figure 3(c), the tree for yet another route for constructing a team automaton $\mathcal{U}_6$, starting from the components in $\mathcal{S}$, is depicted.

Let $\mathcal{T}''$ be a team automaton over $\mathcal{S}''$ specified as $\mathcal{T}'' = (P'', (\Sigma''_{inp}, \Sigma''_{out}, \Sigma''_{int}), \delta'', I'')$, for some $\delta'' \subseteq P'' \times \Sigma'' \times P''$ where $\Sigma'' = \Sigma''_{inp} \cup \Sigma''_{out} \cup \Sigma''_{int}$. Then by the definition of team automata we immediately have $\Sigma''_{int} = ((\bigcup_{i\in\{2,4,6\}} \Sigma_{i,int}) \cup (\bigcup_{i\in\{1,3,5\}} \Sigma_{i,int})) \cup \Sigma_{7,int} = \bigcup_{1\leq i\leq 7} \Sigma_{i,int}$. Likewise, $\Sigma''_{out} = \bigcup_{1\leq i\leq 7} \Sigma_{i,out}$ and $\Sigma''_{inp} = (\bigcup_{1\leq i\leq 7} \Sigma_{i,inp}) \setminus \bigcup_{1\leq i\leq 7} \Sigma_{i,out}$. Thus $\mathcal{T}''$ has the same input, output and internal actions as any team formed directly over $\mathcal{S}$. Its set P'' of states, however, differs from the set of states of a team over $\mathcal{S}$ by its nested structure and its ordering. By definition, $P'' = ((\prod_{i\in\{2,4,6\}} Q_i) \times (\prod_{i\in\{1,3,5\}} Q_i)) \times Q_7 = ((Q_2 \times Q_4 \times Q_6) \times (Q_1 \times Q_3 \times Q_5)) \times Q_7$. Similarly, $I'' = ((I_2 \times I_4 \times I_6) \times (I_1 \times I_3 \times I_5)) \times I_7$.

Also the team automaton $\mathcal{U}_6$ has the same input, output and internal actions as any team formed directly over $\mathcal{S}$. The set of states of $\mathcal{U}_6$ is $(((Q_1 \times Q_2) \times Q_3) \times (Q_7 \times Q_4)) \times (Q_6 \times Q_5)$.

Example 3 illustrates that, given the composable system $\mathcal{S}$, one may form teams over disjoint subsets of components from $\mathcal{S}$. These teams together with the component automata not involved in any of these teams are again a composable system, which can then be used as the basis for the formation of still higher-level teams. This leads to the following definition of an iterated team automaton.

Definition A team automaton is an *iterated team automaton over* $\mathcal{S}$ if it is either a team automaton over $\mathcal{S}$ or a team automaton over $\{\mathcal{T}_j \mid j \in \mathcal{J}\}$, where each $\mathcal{T}_j$ is an iterated team automaton over $\{\mathcal{C}_i \mid i \in \mathcal{I}_j\}$, for some $\mathcal{I}_j \subseteq \mathcal{I}$, and $\{\mathcal{I}_j \mid j \in \mathcal{J}\}$ forms a partition of $\mathcal{I}$.

The notion of an iterated team automaton is a generalization of the notion of a team automaton: every team over a given composable system $\mathcal{S}$ is also an iterated team over $\mathcal{S}$. Conversely, a team formed iteratively over a composable system has, when compared with the teams formed directly over $\mathcal{S}$, the same alphabet of actions — including the distribution over input, output and internal actions — and "essentially" the same state space, transition space and set of initial states. The only difference lies in the ordering and grouping of states coming from different components. In [3] this difference has been formalized using functions to unpack and reorder Cartesian products. For the purpose of this exposition it is sufficient if we simply *reorder* their state spaces (with respect to $\mathcal{S}$).

Given a composable system $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$, by reordering we can identify the state space of ev-

ery iterated team automaton over $\mathcal{S}$ with $\prod_{i\in\mathcal{I}} Q_i$ and its set of initial states with $\prod_{i\in\mathcal{I}} I_i$. For a state p of an iterated team automaton over $\mathcal{S}$, we write $\langle p\rangle$ to denote its reordered version in $\prod_{i\in\mathcal{I}} Q_i$. As an example, let $p = ((p_1,p_2,p_3),(p_4,p_5,p_6),p_7)$ be a state of the iterated team depicted in Figure 3(b). Then $\langle p\rangle = (p_4,p_1,p_5,p_2,p_6,p_3,p_7)$, an element of $\prod_{1\leq i\leq 7} Q_i$.

When reordering the state space, the transition spaces obviously have to be modified accordingly. This means that rather than considering transitions (p,a,q) we reorder the states involved and consider $(\langle p\rangle,a,\langle q\rangle)$. We can thus relate the transitions of an iterated team to transitions of teams formed directly over $\mathcal{S}$.

Let $\mathcal{T}'$ be a team automaton over $\{\mathcal{T}_j \mid j\in\mathcal{J}\}$, where each $\mathcal{T}_j$ is an iterated team automaton over $\{\mathcal{C}_i \mid i\in\mathcal{I}_j\}$, for some $\mathcal{I}_j\subseteq\mathcal{I}$, such that $\{\mathcal{I}_j \mid j\in\mathcal{J}\}$ forms a partition of $\mathcal{I}$. It is not too difficult to prove that, for each action a in $\mathcal{S}$, its complete transition space in $\{\mathcal{T}_j \mid j\in\mathcal{J}\}$ is after reordering included in its complete transition space in $\mathcal{S}$. Hence, the transitions of any team over $\{\mathcal{T}_j \mid j\in\mathcal{J}\}$ are the transitions of a team over $\mathcal{S}$. Consequently, iteration in the construction of a team 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 transition space.

Conversely, however, one cannot simply apply an inverse of the reordering to translate the team automaton $\mathcal{T}$ directly constructed from $\mathcal{S}$, into a team constructed according to a prescribed iteration from $\mathcal{S}$ with still essentially the same transitions. That this is in general not possible follows immediately from the observation that iterated teams, like teams, are equipped with only a subset of all possible synchronizations. Thus, 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 $\{\mathcal{T}_j \mid j\in\mathcal{J}\}$ may provide less transitions for the forming of a team than $\{\mathcal{C}_i \mid i\in\mathcal{I}\}$ does. This problem is avoided by imposing the natural condition that for each j, the transitions of the subteam of $\mathcal{T}$ determined by $\mathcal{I}_j$ are reordered transitions of $\mathcal{T}_j$.

Hence we may conclude that team automata are naturally suited to describe hierarchical systems. Both subteams and iterated teams can be treated as team automata including the considerations concerning their computations and behaviour and it suffices to consider only the relationship between subteams and team automata.

5 Synchronizations

In this and the next section $\Sigma_{ext} = \bigcup_{i\in\mathcal{I}}(\Sigma_i \setminus \Sigma_{i,int})$ is the set of external actions of the team automaton $\mathcal{T}$ over $\mathcal{S}$, and in fact of any team automaton over $\mathcal{S}$.

This section discusses various natural ways of synchronizing transitions within team automata. First we focus on the individual actions of a team automaton and distinguish three natural modes of synchronization. We consider actions that are never used in synchronizations between multiple components and actions on which all components that have 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).

Let a be an action of $\mathcal{T}$.

a is *free* in $\mathcal{T}$ 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.

a is *action-indispensable* (ai for short) in $\mathcal{T}$ if 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).

a is *state-indispensable* (si for short) in $\mathcal{T}$ if 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.

In Example 1 we have that b is ai and not free in $\mathcal{T}_1$, it is si, not ai and not free in $\mathcal{T}_2$ and it is not si and not free in $\mathcal{T}_3$.

As noted before, information on the actual execution of loops by components is missing in the transition relation of a team automaton. Therefore, in the above classification of actions 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. In Example 2 we thus have that action a is not free and not ai, but it is si.

Since an internal action of $\mathcal{T}$ is an action of only one of the component automata, it is immediate that every internal action is free, ai and si in $\mathcal{T}$. Furthermore, every action that is ai in $\mathcal{T}$ also satisfies the weaker requirement of being si. So ai implies si, and this is in fact the only dependency among the properties free, ai and si.

The property of an action being free, ai or si in $\mathcal{T}$ is inherited by all subteams of $\mathcal{T}$ to which this action belongs. Thus if $J\subseteq\mathcal{I}$ and $a\in\Sigma_J=\bigcup_{j\in J}\Sigma_j$, then a is free (ai or si, respectively) in $SUB_J(\mathcal{T})$ whenever a is free (ai or si, respectively) in $\mathcal{T}$. The converse is not

true in general. Consider, e.g., the team automaton $\mathcal{T}_3$ defined in Example 1. The action b is neither free, nor ai nor si in $\mathcal{T}_3$, although it is free, ai and si in the subteam determined by $\{3\}$, which is a copy of $\mathcal{C}_3$.

In fact, an action that is not free (ai or si) in a team automaton, can be made free (ai or si, respectively) in a subteam by dropping those components that caused it not to be free (ai or si, respectively) in $\mathcal{T}$. Observe that in every subteam determined by a single component, every action is free, ai and si. The properties of being free, ai or si are carried over from a subteam to the team as a whole if all components that the action in question belongs to, are included in the subteam.

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 any behaviour of such automata, when projected on a component or subteam, yields the corresponding behaviour of that component or subteam. Thus, for such automata, we can extend earlier observation on computations to behaviours: from a team behaviour a component (subteam) behaviour can immediately be extracted by deleting all actions that do not belong to that component (or subteam)! This is not possible when the execution of an action does not involve all components to which it belongs.

Until now we have discussed synchronizations while ignoring whether the action was input, output or internal in certain components. Next we turn to the different roles an action may have in different components. Since internal actions 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, and thus we consider only external actions for the rest of this section. First we separate their output role and their input role. Since no external action of any team automaton over $\mathcal{S}$ will ever be both an input and an output action for one and the same component, we can define disjoint input and output domains for each external action.

For $a \in \Sigma_{ext}$ we have $\{j \in \mathcal{I} \mid a \in \Sigma_{j,inp}\}$ as the *input domain* of a in $\mathcal{S}$, and $\{j \in \mathcal{I} \mid a \in \Sigma_{j,out}\}$ as the *output domain* of a in $\mathcal{S}$. These two domains each determine a subteam of $\mathcal{T}$, to which we refer, respectively, as the *input subteam* of a in $\mathcal{T}$ denoted by $SUB_{a,inp}(\mathcal{T})$ and the *output subteam* of a in $\mathcal{T}$ denoted by $SUB_{a,out}(\mathcal{T})$. If no confusion arises we omit $\mathcal{T}$ and simply write $SUB_{a,inp}$ and $SUB_{a,out}$. Note that for every external action, at least one of its input domain and its output domain is not empty. In case the input (or output) domain of an external action is empty, then its input (output) subteam is the trivial automaton $(\varnothing, (\varnothing, \varnothing, \varnothing), \varnothing, \varnothing)$.

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.

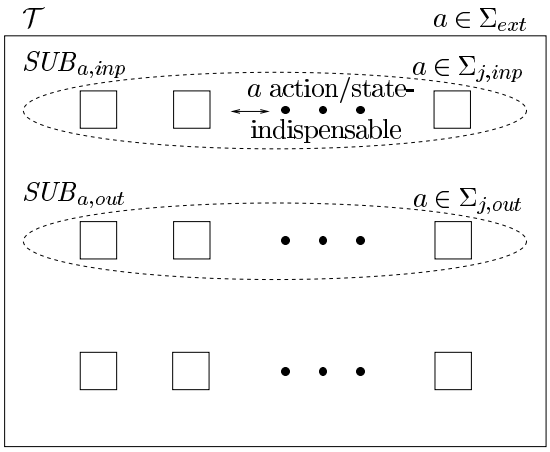

Fig. 4: A team automaton with a strong/weak input peer-to-peer action a

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 can be executed). 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.

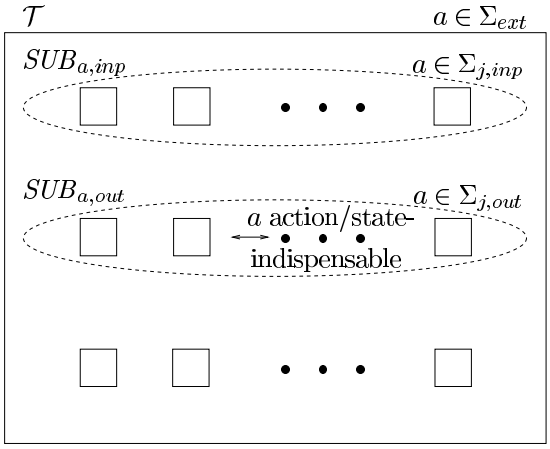

Fig. 5: A team automaton with a strong/weak output peer-to-peer action a

Since ai implies si, it is immediately clear that every action which is strong input (output) peer-to-peer in a team automaton, is also weak input (output) peer-to-peer in that team automaton. This however does not hold the other way around. For the output case this follows from Example 2 since a is weak output peer-to-peer but not strong output peer-to-peer in $\mathcal{T}$.

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. Formally, we have that a is *strong master-slave* if it is master-slave and its input subteam moreover participates in every a-transition of $\mathcal{T}$. For a to be *weak master-slave*, we require that it is master-slave and that its input subteam moreover participates in every a-transition of $\mathcal{T}$ whenever it can. Thus if an action is strong master-slave in a team automaton, then it is also weak master-slave.

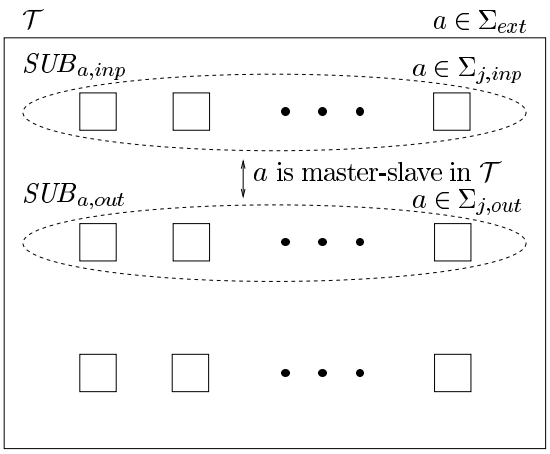

Fig. 6: A team automaton with a master-slave action a

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 the definition of subteams 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. 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.

In these master-slave definitions, 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.

6 Constructing 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 synchronizations. In general, however, these conditions 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. Note that since the transition relation of an internal action is by definition fixed to be the complete transition space of that action in $\mathcal{S}$, there is no need to explicitly specify predicates for internal actions. Hence, for the rest of this section we assume that a is an external action of any team automaton over $\mathcal{S}$. Once predicates have been chosen for all external actions, the team automaton determined by these predicates is unique. The construction of a team automaton satisfying certain conditions on its

synchronizations thus amounts to the choice of appropriate predicates for each of its external actions.

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 [13] and generalizes the classical approach to define synchronized systems (see, e.g., [11], [17], [23] and [26]) 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.

In case no constraints are imposed on the synchonizations on a, all a-transitions are allowed since nothing is required, and thus no transition is forbidden. In this case the predicate is simply $\Delta_a(\mathcal{S})$.

If a should be free, ai, or si in $\mathcal{S}$, then *all and only* those a-transitions are included that respect the specified property of a. Thus we have a predicate *is-free in* $\mathcal{S}$ for a, which consists of *all* a-transitions from $\Delta_a(\mathcal{S})$ in which only one component automaton is active. The predicate *is-action-indispensable in* $\mathcal{S}$ for a is (for later use) denoted by $\mathcal{R}_a^{ai}(\mathcal{S})$ and consists of *all* a-transitions from $\Delta_a(\mathcal{S})$ in which all components that have a as an action are active. The predicate *is-state-indispensable in* $\mathcal{S}$ for a consists of *all* a-transitions from $\Delta_a(\mathcal{S})$ in which every component that has a as an action is active provided a is enabled in the state of the component. It can easily be shown that each of these three predicates 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.

First assume that the input domain of a is not empty. The predicate *is-strong-input-peer-to-peer in* $\mathcal{S}$ for a consists of *all* a-transitions from $\Delta_a(\mathcal{S})$ in which all components from the input domain of a are active. The predicate *is-weak-input-peer-to-peer in* $\mathcal{S}$ for a consists of *all* a-transitions from $\Delta_a(\mathcal{S})$ in which all components from the input domain of a in which a is currently enabled are active.

In case the output domain of a is not empty, the predicates *is-strong-output-peer-to-peer in* $\mathcal{S}$ for a and *is-weak-output-peer-to-peer in* $\mathcal{S}$ for a are defined in an analogous way.

Since these predicates define the largest sets of a-transitions satisfying the specified constraints, again the aim to describe unique sets of a-transitions is achieved.

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 the output role of actions. This time, however, the predicates describe synchronizations *between* components from the input and components from the output domains.

Recall that a is master-slave in a team automaton if it is an output action and its output subteam participates in every synchronization on a. The predicate *is-master-slave in* $\mathcal{S}$ for a includes all and only those a-transitions in which a is executed by at least one of the components in its output domain.

For a to be strong master-slave or weak master-slave, there is the additional requirement that a should (in the strong or the weak sense) also be executed by its input subteam. The predicate *is-strong-master-slave in* $\mathcal{S}$ for a consists of all a-transitions in which a appears at least once in its output role and, moreover, if the input domain of a is not empty, then also at least one component from the input domain of a is involved.

The predicate *is-weak-master-slave in* $\mathcal{S}$ for a consists of all a-transitions in which a appears at least once in its output role and, moreover, if a component from the input domain of a is ready to perform a, then also at least one component from the input domain of a is involved.

Each of these three (strong/weak) master-slave predicates guarantees that a is indeed (strong/weak) master-slave in every team automaton over $\mathcal{S}$ having that predicate for its a-transitions. Furthermore, both the master-slave predicate and the strong master-slave predicate are the largest set of a-transitions satisfying the specified constraint. Thus, in these cases we have again a uniquely defined set of a-transitions. 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. Hence, a can be weak master-slave

in the 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.

Hence, except for weak master-slave synchronization, each of the various modes of synchronization introduced in the previous section gives rise to a predicate that is the unqiue 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.

7 Comparison

Team automata are related both to I/O automata and to Vector Controlled Concurrent Systems. Actually, as we demonstrate next, one may view team automata as a model somewhere in between those two.

7.1 I/O Automata

As mentioned in the Introduction, team automata are an extension of Input/Output automata (I/O automata for short). In this section we briefly discuss the relationship between I/O automata and team automata and we show how I/O automata fit into the framework of team automata.

I/O automata were introduced in [34] (see also [22] and [23]) for modelling 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., [24] and [27]).

Originally, I/O automata are defined in terms of labelled transition systems together with an associated equivalence relation over the set of actions used to define so-called fair computations. In [34] I/O automata without such equivalence relations are called safe I/O automata and in [15] 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 [34], is rather restrictive and may hinder a realistic modelling of certain types of interactions. This is the main motivation given in [13] 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 $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$ of I/O automata, a new I/O automaton can be constructed provided $\mathcal{S}$ 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 $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$ 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 $\mathcal{S}$ 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 $\mathcal{S}$ 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.

Definition Let $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$ be a compatible system of I/O automata. Then the *team I/O automaton* over $\mathcal{S}$ is the team automaton $\mathcal{T}$ over $\mathcal{S}$ with transition relation δ such that $\delta_a = \mathcal{R}_a^{ai}(\mathcal{S})$, for all actions a of $\mathcal{S}$.

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 $\mathcal{S}$ of I/O automata can be constructed by iteration. Any iterated team I/O automaton corresponds to the team I/O automaton over $\mathcal{S}$ after reordering its state space.

Conversely, if $\mathcal{T}$ is the team I/O automaton over a compatible system $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$ of I/O automata, then every subteam of $\mathcal{T}$ determined by some $J \subseteq \mathcal{I}$, is the team I/O automaton over $\{\mathcal{C}_j \mid j \in J\}$. 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 $\mathcal{S}$. Then it is again an input action of the team I/O automaton over $\mathcal{S}$ 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 [34], 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.

7.2 Vector control

Team automata are compositions of component automata which work together through synchronizations on certain actions. These synchronizations are labelled 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. This subsection shows how vector actions can be employed to this aim and then discusses how team automata fit in a theory of vector controlled concurrent systems.

For the sequel we let Σ denote the union $\Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}$ of the input, output and internal alphabet of a given 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 $\mathcal{S}$.

By the definition of team automata 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.

This switch is feasible since for each transition (q, a, q') the global state change from q to q', caused by the occurrence of this transition, is described in terms of changes in the local states of the components involved. If $proj_j(q) \neq proj_j(q')$, then $proj_j{}^{[2]}(q, q') \in \delta_{j,a}$ and the j-th component is involved. In that case we set $proj_j(\alpha) = a$. If $proj_j(q) = proj_j(q')$ and $proj_j{}^{[2]}(q, q') \notin \delta_{j,a}$, then the j-th component is not involved and we set $proj_j(\alpha) = \lambda$. There is however — again — the problem of loops. If $proj_j(q) = proj_j(q')$ and $proj_j{}^{[2]}(q, q') \in \delta_{j,a}$, then it is unclear whether or not the j-th component is involved. Following the maximal interpretation we assume it is and set $proj_j(\alpha) = a$.

Following this procedure 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, one may 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, one chooses vector transitions from the *complete vector transition space* $\Delta_a^v(\mathcal{S})$ of a in $\mathcal{S}$ which describes all possible vec-

tor transitions for a.

Definition Let $a \in \bigcup_{i\in\mathcal{I}} \Sigma_i$. The *complete vector transition space* of a in $\mathcal{S}$ is denoted by $\Delta_a^v(\mathcal{S})$ and is defined as $\Delta_a^v(\mathcal{S}) = \{(q,\alpha,q') \mid (q,q') \in \Delta_a(\mathcal{S}),\ \alpha \in \prod_{i\in\mathcal{I}}(\{a\} \cup \{\lambda\})$ and $(\forall i \in \mathcal{I} : [\text{if } proj_i(\alpha) = a, \text{ then } (proj_i(q), proj_i(q')) \in \delta_{i,a}]$ and $[\text{if } proj_i(\alpha) = \lambda, \text{ then } proj_i(q) = proj_i(q')])\}$.

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 $\mathcal{S}$. Observe that 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, all vector representatives of external actions have *at least* one entry which is not λ.

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

Definition A *vector team automaton over* $\mathcal{S}$ is a construct $\mathcal{V} = (\prod_{i\in\mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta^v, \prod_{i\in\mathcal{I}} I_i)$ such that $\Sigma_{int} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,int}$, $\Sigma_{out} = \bigcup_{i\in\mathcal{I}} \Sigma_{i,out}$, $\Sigma_{inp} = (\bigcup_{i\in\mathcal{I}} \Sigma_{i,inp}) \setminus \Sigma_{out}$, and $\delta^v \subseteq \prod_{i\in\mathcal{I}} Q_i \times [\prod_{i\in\mathcal{I}}(\{a\} \cup \{\lambda\})] \times \prod_{i\in\mathcal{I}} Q_i$, is such that for all $a \in \Sigma$, $\delta^v \cap \prod_{i\in\mathcal{I}} Q_i \times [\prod_{i\in\mathcal{I}}(\{a\} \cup \{\lambda\})] \times \prod_{i\in\mathcal{I}} Q_i \subseteq \Delta_a^v(\mathcal{S})$ and moreover $\Delta_a^v(\mathcal{S}) \subseteq \delta^v$ if $a \in \Sigma_{int}$.

Example 4 In Figure 10 two vector team automata over the composable system $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ of Example 2 are given.

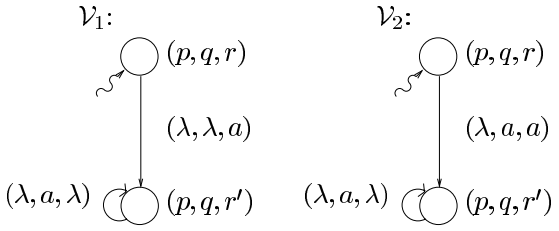

Fig. 10: Two vector team automata

Note that in both vector team automata it is clear which components participate in each of its vector transitions. This contrasts with the team automaton $\mathcal{T}$ of Example 2.

By replacing each transition (q,α,q') of a vector team automaton $\mathcal{V}$ by the flat transition (q,a,q') if α is a vector representative of the action a, one obtains a *flattened version* of the vector team automaton. This is a normal 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. As an example, note that both vector team automata of Example 4 have the team automaton $\mathcal{T}$ of Example 2 as their flattened version.

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 [20] and [21] a framework is proposed for the study of vector controlled systems. The approach there is based on the synchronization as modelled in the vector firing sequence semantics of path expressions and COSY (see, e.g., [17]) and is related to the work of Arnold and Nivat (see, e.g., [1]) and the coordination of cooperating automata by synchronization on multisets in [2]. The model of Vector Controlled Concurrent Systems (VCCS for short) 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. To conclude this section we will make this claim more concrete and describe a particular method of vector synchronization within VCCS which is applicable to team automata and based on Petri nets (see, e.g., [29]). We assume the reader to be familiar with the basic notions of Petri nets.

Let $\mathcal{V}$ be a vector team automaton over $\mathcal{S}$, specified as $\mathcal{V} = (\prod_{i\in\mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta^v, \prod_{i\in\mathcal{I}} I_i)$. Then $\mathcal{V}$ has an immediate interpretation as a Petri net $\mathcal{N}_\mathcal{V}$ of a special form. This net has a place for each state of each of the component automata and, for each vector transition of the team, a Petri net transition (which we will call an event from now on in order to avoid confusion). Let (q,α,q') be a vector transition of $\mathcal{V}$ and let t be its associated event. For all $i \in \mathcal{I}$, whenever $proj_i(\alpha)$ is not λ we have $proj_i(q)$ as an input place of t and $proj_i(q')$ as an output place of t. Thus only the components actively involved in the vector transition (q,α,q') participate in the event t and have their local state changed by t according to (q,α,q'). The event t is labelled by α. The initial markings of $\mathcal{N}_\mathcal{V}$ are defined by the initial states of $\mathcal{V}$. An initial marking corresponding to $q \in \prod_{i\in\mathcal{I}} I_i$ assigns one token to $proj_i(q)$ for each $i \in \mathcal{I}$. This defines the structure and the labelling of $\mathcal{N}_\mathcal{V}$. Its dynamics is defined by the ordinary firing rule for Petri nets.

Example 5 In Figure 11 the Petri net $\mathcal{N}_{\mathcal{V}_2}$ obtained by applying the above construction to the vector team automaton $\mathcal{V}_2$ of Example 4 is given.

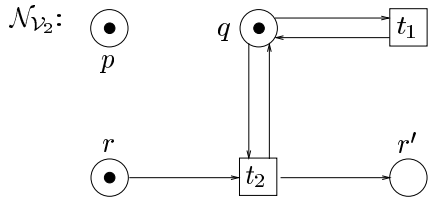

Fig. 11: A Petri net

Event t_1 is associated to $((p,q,r'),(\lambda,a,\lambda),(p,q,r'))$ and is labelled by (λ,a,λ), whereas event t_2 is associated to $((p,q,r),(\lambda,a,a),(p,q,r'))$ and labelled by (λ,a,a).

The Petri net $\mathcal{N}_{\mathcal{V}}$ is a composition of state machines (sequential Petri net versions of transition systems). The synchronization constraints formulated in δ^v, the composition of the state machines and the labelling of the events, are mutually consistent. This makes $\mathcal{N}_{\mathcal{V}}$ an *Individual Token Net Controller* (ITNC for short) as defined in [20] and further investigated in [19]. ITNCs are control mechanisms for vector synchronization based on state machine decomposable nets. They are designed in such a way 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.

To conclude, we note that vector team automata like ITNCs (see [19]) 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., [25] and [9]).

8 Spatial Access Control

A vital component of any groupware system is security and information access control. In typical electronic file systems, access rights such as read-access and write-access are often allocated to users on some informal basis such as "need to know", ownership, or ad hoc lists of accessors.

Within groupware systems there are typically needs for more refined access rights, such as the right to scroll a document that is being synchronously edited by a group in real time. Furthermore, the granularity of access must sometimes be more fine-grained and flexible, as within a software development team (see [16] for an example of the use of team automata when modelling software development teams). Moreover, it is important to control access meta-rights. For example, it may be useful for an author to grant another team member the right to grant document access to other non-team members (delegation). Various models have been proposed to meet such requirements (see, e.g., [32], [31], and [33]).

In [4] it is demonstrated how team automata can be used for modelling access control mechanisms presented through the metaphor of spatial access control ([6] and [8]), which is based upon the virtual reality metaphor of places and spaces ([7]). Each space is represented by a component automaton, dynamic access changes are represented by joint external actions, while resource accesses within a space can be represented by internal actions.

Using an example from [4] we now show how team automata can be used to formally describe some of the key issues of access control.

In security literature, authentication deals with verification that the user is truly the person represented, whereas authorization deals with validation that the user has access to the given resource. Here we are only concerned with authorization. We thus assume that when the user logs into the system there is an authentication check. Then whenever the user tries to read or write a file, only authorization checking occurs, and he or she is either allowed the access, or not.

Consider a file F, which might be any data or document that is stored electronically within a typical file system, and a user who can be granted and revoked read and write access rights to F. The file system keeps track of which users have which access rights to F. Three types of access rights are possible for a file: null access (implying the user can neither read nor write the file), read access (implying the user cannot write the file), and full access (implying the user can read and write — i.e. edit — the file).

Furthermore, we assume the existence of an assistant system administrator, who can change the user's rights. Hence this assistant has the right to grant and revoke access by the user to F. The actions $m(r)$, $\underline{m}(r)$, $m(w)$, and $\underline{m}(w)$ model the operations of "being granted read access", "being revoked read access", "being granted write access", and "being revoked write access", respectively. The rights to grant and revoke are legitimate rights, but they are not directly applied to F. They are in fact meta operations.

Finally, we assume that the right to grant and revoke rights to a file can ifself be granted and revoked.

Hence, if there is a head of system adminstration, who can allow (and disallow) the assistant system administrator to grant and revoke, then this head has meta meta rights. The head thus has the meta meta right to grant and revoke the assistant's meta rights to grant and revoke the user's access rights to F. A typical action of the head is $\underline{m}^2(w)$, which revokes the assistant's right to grant and revoke write access to the user. The complicated (recursive) situations that may arise in this fashion depend on the chosen access control policy and next we demonstrate how they can unambiguously and concisely be defined in terms of team automata.

Figure 7 shows a component automaton M^0 modelling three levels — A, B and C — corresponding to null access, read access, and full (write) access, respectively. The status of the user directly determines the level he or she operates on and the granting and revoking of access rights is identified with changing levels.

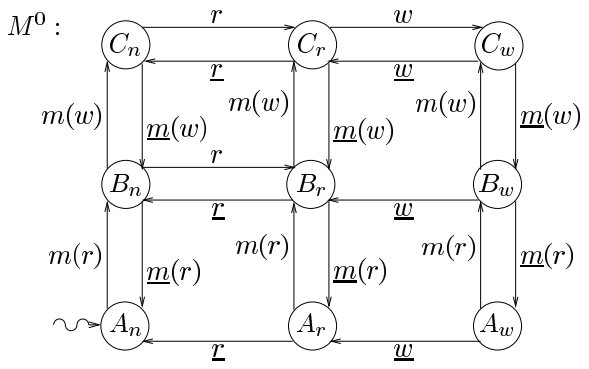

Fig. 7: Component automaton M_0

In M^0 the user thus moves in two dimensions: vertically between levels A, B, and C — indicating the dynamic change in access rights he or she has for F — and horizontally between the states "null", "reading", and "writing" — indicating the current activities of the user with respect to F. Notice that, e.g., the state B_w meaning that the user is writing while having read access but no write access, can only be reached from C_w by an action $\underline{m}(w)$ or from A_w by an action $m(r)$. Hence this state B_w can be entered only when the user is writing while his status changes.

Since this status change is imposed on the user by the assistant system administrator, the granting and revoking of access rights clearly are passive actions from the user's point of view. For this reason we have chosen all actions of granting and revoking access rights in M^0 to be input actions, while all actions of reading and writing are output actions. Note that M^0 is not input-enabled, which immediately implies that M^0 is not an I/O automaton.

By including $(C_w, \underline{m}(w), B_w)$ we have chosen to model *delayed revocation*. Delayed revocation, as opposed to immediate revocation, means that the user can continue his current activity even when his or her rights have been revoked. He or she can do so until he or she wants to restart this activity, at which moment an authorization check is done to decide if he or she has the right to restart this activity.

If we instead would have included $(C_w, \underline{m}(w), B_r)$, then we would have modelled *immediate revocation* which avoids delays. This means that if a user is reading when his or her reading right is revoked, then the file immediately disappears from view, while if a user is writing when his or her writing right is revoked, then the edit is interrupted and writing is terminated in the middle of the current activity. In some applications, this is overly disruptive and unfriendly.

The notion of meta clearly extends to arbitrary layers. An interesting question now arises as to the effect of revocation: should revocation of a meta right also revoke the rights that were passed on to others? This is the issue of *shallow revocation* versus *deep revocation*.

Shallow revocation means that a revoke action does not revoke any of the rights that were previously passed on to others, whereas deep revocation means that a revoke action does revoke all rights previously passed on. Now recall the action $\underline{m}^2(w)$ of the head, which revokes the assistant's right to grant and revoke write access to the user. Assume moreover that before the assistant lost this right he or she has granted the user write access. If the action $\underline{m}^2(w)$ is executed, then the effect is that either only the assistant loses the right to grant and revoke access to the user (shallow revocation) or also the user loses the write access he or she was granted by the assistant (deep revocation). Team automata can be used to model shallow, deep, or even hybrid revocation. Shallow revocation is often the easiest to model, whereas deep revocation is known as a big challenge to model and implement ([10]). We now show how deep revocation can be modelled using team automata.

Figure 8 shows a component automaton capturing one layer (layer k) of a multi-layer meta access specification for our example of read and write access. We have already seen layer 0, viz. component automaton M^0. For each value of $k \geq 1$ there are corresponding component automata that are directly related to layer k (viz. M^{k-1} at layer $k-1$ and M^{k+1} at layer $k+1$). For each such component automaton M^k, the horizontal actions $m^k(r)$, $\underline{m}^k(r)$, $m^k(w)$ and $\underline{m}^k(w)$ are output actions, whereas the vertical actions $m^{k+1}(r)$, $\underline{m}^{k+1}(r)$, $m^{k+1}(w)$ and $\underline{m}^{k+1}(w)$ are input actions. Note that this means that also for each $k > 0$ the component automaton M^k is not an I/O automaton. It is immediate that $\{M^0, M^1, \ldots, M^n \mid n \geq 1\}$ is a composable system. Since each output action is uniquely associated to a component automaton it is even compatible.

Finally, for $k = 0$ we identify r with $m^0(r)$, $\underline{r}$ with $\underline{m}^0(r)$, w with $m^0(w)$ and $\underline{w}$ with $\underline{m}^0(w)$. Similarly, $m(r) = m^1(r)$, $\underline{m}(r) = \underline{m}^1(r)$, $m(w) = m^1(w)$ and $\underline{m}(w) = \underline{m}^1(w)$.

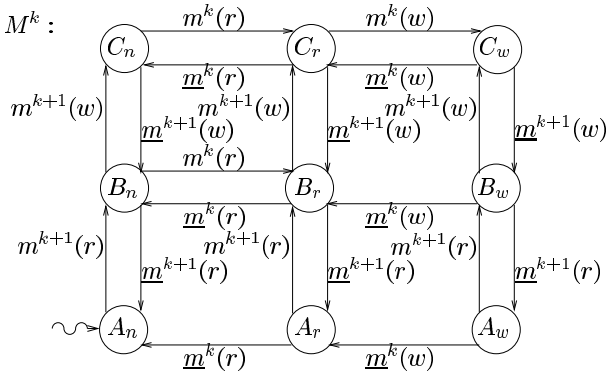

Fig. 8: Component automaton M^k: meta access at layer k

We can now define a multi-layered structure by composing a team automaton over the composable system $\{M^0, M^1, \ldots, M^n \mid n \geq 1\}$. By the choice of the actions it is clear that only contiguous component automata M^{k-1} and M^k can synchronize on common actions. Hence, due to the results on iterative constructions of team automata in Section 4 we can specify the team automaton by describing the interaction of two automata M^{k-1} and M^k and requiring that the resulting team automaton is a subteam of the final overall team automaton modelling the desired multi-layered structure.

In Figure 9 a team automaton T^k_{k-1} over M^{k-1} and M^k, representing layer $k-1$ and layer k of this layered structure, is depicted (in fact, only the reachable part is drawn). The transition relation of this team T^k_{k-1} is chosen with the modelling of deep revocation in mind. Before we motivate our choice, please note that in Figure 9 we have added superscripts to distinguish the states in M^k from the states in M^{k-1}, e.g., state B_r of M^k from state B_r of M^{k-1}.

Now we motivate the chosen transition relation of T^k_{k-1} in more detail. Recall our example of a head, an assistant and a user. Let M^2 represent the actions of the head, M^1 those of the assistant and M^0 those of the user. Now consider the head in state B^2_r. Then Figure 9 (with $k = 2$) tells us that the assistant must be in one of the three states B^1_n, B^1_r or B^1_w. Assume that the head reached this state B^2_r by performing action $m^2(r)$ from B^2_n, while the assistant was in state A^1_n having no rights to grant and revoke reading rights. Action $m^2(r)$ is an output action of M^2 and an input action of M^1, and our transition relation forces M^1 to transition from A^1_n to B^1_n. The interpretation is that the head granted the assistant the right to do read grants and revokes (to the user for file F).

Similarly, automaton M^k can revoke the right to grant and to revoke read access from M^{k-1} at any time by performing output action $\underline{m}^k(r)$, and thus forcing M^{k-1} to perform this action — this time as an input action — as well. Continuing our example, now for $k = 3$, this means that while in state B^2_r, the head's read granting right may be revoked by action $\underline{m}^3(r)$ at any time. If this happens, the head is forced into the state A^2_r, which has only one possible output action, viz. $\underline{m}^2(r)$, leading to A^2_n. Whenever that action $\underline{m}^2(r)$ occurs it revokes the assistant's right to change the user's read access by forcing the assistant to transition from B^1_n to A^1_n.

Two general rules of activity are modelled in our team automaton over $\{M^0, M^1, \ldots, M^n \mid n \geq 1\}$.

First, when a component automaton M^k, with $1 \leq k \leq n$, transitions right (grant) or left (revoke), then the component automaton M^{k-1} must transition upward (gaining some access right) or downward (losing some access right). This is achieved as follows. Each of the output actions $m^k(r)$, $\underline{m}^k(r)$, $m^k(w)$ and $\underline{m}^k(w)$ of M^k is an input action of M^{k-1}. We choose the transition relation of the team automaton T^k_{k-1} over M^k and M^{k-1} such that all of these output actions are master-slave synchronizations in T^k_{k-1}. Note that all other actions of T^k_{k-1} are free.

Secondly, the component automaton M^{k-1} may be forced to transition downward, from where it will eventually transition to the left by executing one of its output actions. All these output actions of M^{k-1} are themselves involved (as "masters") in a master-slave synchronization with the input actions of the same name of component automaton M^{k-2} (as "slaves"). This transition to the left again forces a downward transition of M^{k-2}, and so on until M^0 on layer 0. Hence, as promised, we indeed model deep revocation.

9 Conclusion

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 behaviours. Consequently, coordination policies and their effect on behaviour 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. Modelling 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,

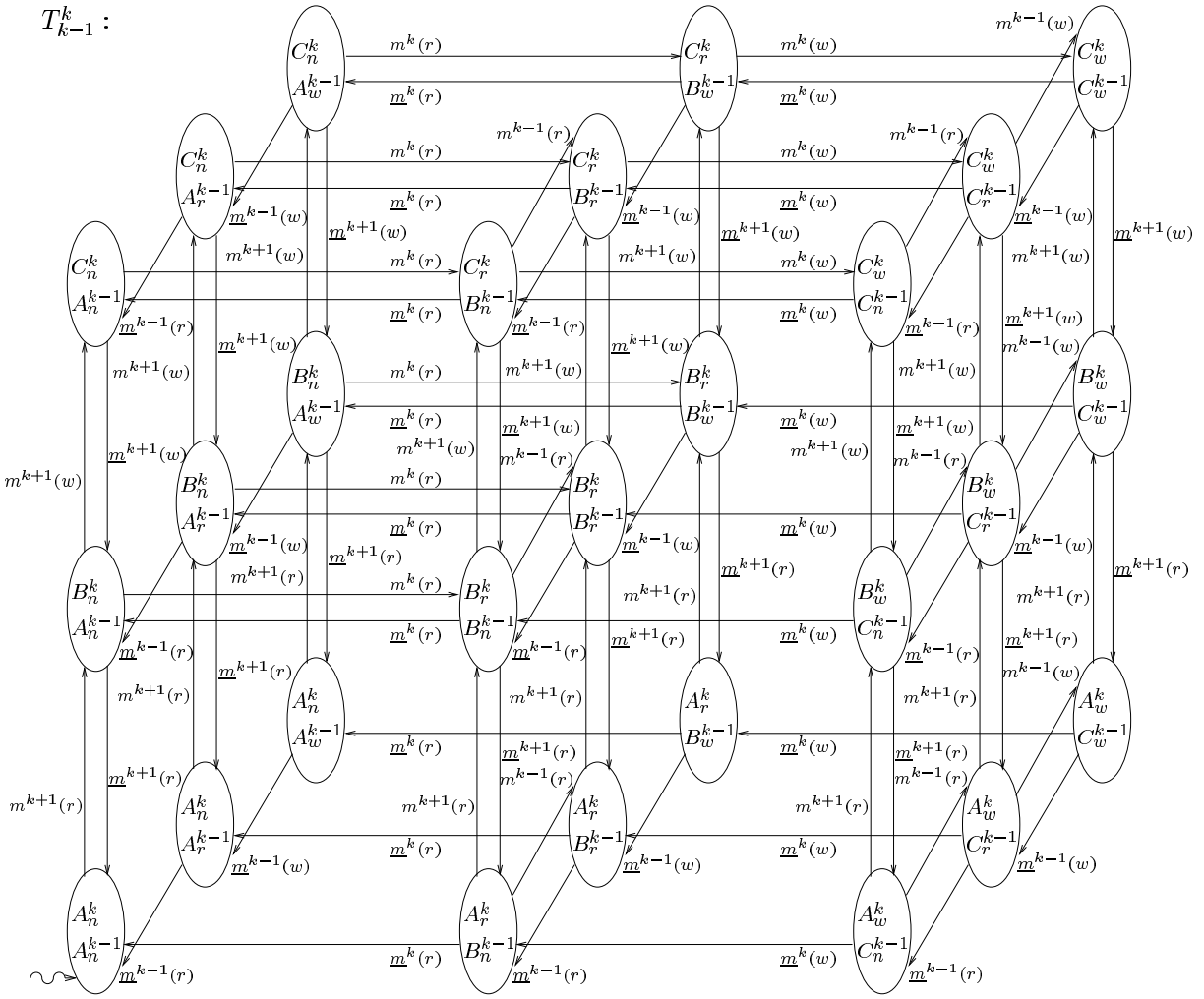

Fig. 9: Team automaton T^k_{k-1} over M^{k-1} and M^k

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 rigourous definitions provide the possibility of formal analysis tools for proving crucial design properties, without first having to implement the design.

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 labelled 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.

In this paper we have presented only a survey, based mainly on the work in [3]. An example from [4] has been used to illustrate the use of team automata to model multi-layered access control requirements for collaborative environments. Also distributed access control, where the supervisory work of granting and revoking access rights is administered by multiple agents, can be conveniently modelled by team automata. The administrative supervisors who must agree on any change of access rights can have an action which is output to all of them and input to the user. Synchronization on that action requires all supervisors to participate. By including all transitions with at least one of the supervisors being active, we can model the case of approval being required by any one of them. In [13] a shared application example is discussed and an architecture for the GROVE document editor (described in [12]) is presented using the design possibilities provided by team automata.

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 execution of a corresponding input action. They can be used to model asynchronous cooperation, as in workflow systems, to enact certain modules (see, e.g., [14]). On the other hand there are peer-to-peer synchronizations, in which all participants are considered equal. They model the group collaboration aspect that frequently occurs in synchronous groupware. Thus, exact descriptions can be given of certain groupware notions which may otherwise have an ambiguous interpretation. For example the descriptions of cooperation and collaboration of team automata as proposed in [13] can be given formal, mathematically precise definitions using the concepts of master-slave and peer-to-peer synchronizations. In fact, more and finer distinctions can be made on basis of the two different formalizations (ai or si) of the obligation for components to participate in the execution of a certain action. Combinations of cooperation and collaboration, called hybrids in [13], are also easy to define.

Team automata have been informally compared to I/O automata and concurrent systems with vector control. This has demonstrated that team automata are more flexible than I/O automata, both by imposing less requirements on the role of the actions and by the option to choose a transition relation. Hence it has become possible, e.g., for components to be unavailable for communications and to model output peer-to-peer synchronizations. Of course one has to pay for this. To describe the behaviour of a team automaton in terms of its component behaviours is much more difficult when one cannot assume that the composition is based on maximal sets of ai synchronizations, as is the case for I/O automata. This is a topic currently under investigation as part of the Ph.D. research of the first author. Also, the power of the different modes of synchronization should be investigated and compared.

With respect to their synchronizations team automata can be embedded in the more general theories of systems with vector control and, in particular, state machine decomposable nets with vector labels. In this way, results from the areas of Vector Controlled Concurrent Systems, Petri nets and trace theory, next to those from automata theory, become available.

References

[1] A. Arnold, Synchronized Behaviours of Processes and Rational Relations. *Acta Informatica* 17 (1982), 21 – 29.

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

[3] M.H. ter Beek, C.A. Ellis, J. Kleijn, and G. Rozenberg, Synchronizations in Team Automata for Groupware Systems. Technical Report TR-99-12, Leiden Institute of Advanced Computer Science, Universiteit Leiden, 1999.

[4] M.H. ter Beek, C.A. Ellis, J. Kleijn, and G. Rozenberg, Team Automata for Spatial Access Control. To appear in *Proceedings of the ECSCW 2001 European Conference on Computer Supported Cooperative Work, Bonn, Germany*, Kluwer Academic Publishers, Dordrecht, 2001. (A preliminary version appeared as Technical Report TR-01-03, Leiden Institute of Advanced Computer Science, Universiteit Leiden, 2001.)

[5] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM* 31, 3 (1984), 560 – 599.

[6] A. Bullock and S. Benford, Access Control in Virtual Environments. In *Proceedings of the VRST'97 ACM Symposium on Virtual Reality Software and Technology, Lausanne, Switzerland* (D. Thalmann, S. Feiner, and G. Singh, eds.), ACM Press, 1997, 29 – 35.

[7] A. Bullock, *SPACE: Spatial Access Control in Collaborative Virtual Environments*. Ph.D. thesis. Department of Computer Science, University of Nottingham, 1998.

[8] A. Bullock and S. Benford, An access control framework for multi-user collaborative environments. In *Proceedings of the GROUP'99 International ACM SIGGROUP Conference on Supporting Group Work, Phoenix, Arizona*, ACM Press, 1999, 140 – 149.

[9] V. Diekert and G. Rozenberg, *Book of Traces*, World Scientific, Singapore, 1995.

[10] P. Dewan and H. Shen, Flexible Meta Access-Control for Collaborative Applications. In *Proceedings of the CSCW'98 ACM Conference on Computer Supported Cooperative Work, Seattle, Washington* (E. Churchill, D. Snowdon, and G. Golovchinsky, eds.), ACM Press, 1998, 247 – 256.

[11] C. Duboc, Mixed Product and Asynchronous Automata. *Theoretical Computer Science* 42 (1986), 183 – 199.

[12] C.A. Ellis, S.J. Gibbs and G. Rein, Design and Use of a Group Editor. In *Engineering for Human Computer Interaction* (G. Cockton, ed.), North-Holland Publishing Company, Amsterdam, 1990.

[13] C.A. Ellis, Team Automata for Groupware Systems. In *Proceedings of the GROUP'97 International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge, Phoenix, Arizona* (J. Clifford, B. Lindsday, and D. Maier, eds.), ACM Press, 1997, 415 – 424.

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

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

[16] P.J. 't Hoen and M.H. ter Beek, A Conflict-Free Strategy for Team-Based Model Development. In *Proceeedings of the PDTSD'00 International Workshop on Process support for Distributed Team-based Software Development in conjunction with the SCI 2000 World MultiConference on Systemics, Cybernetics and Informatics, Orlando, Florida* (B. Sanchez, R. Hammel II, M. Soriano, and P. Tiako, eds.), International Institute of Informatics and Systemics, 2000, 720 – 725.

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

[18] N.W. Keesmaat, *Vector Controlled Concurrent Systems*. Ph.D. thesis, Leiden University, 1996.

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

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

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

[22] N.A. Lynch and M.R. Tuttle, Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, British Colombia, Canada*,1987, 137 – 151.

[23] N.A. Lynch and M.R. Tuttle, An Introduction to Input/Output Automata. *CWI Quarterly* 2,3 (1989), 219 – 246. (Also appeared as Technical Memo MIT/LCS/TM-373, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1988.)

[24] N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Mateo, California, 1996.

[25] A. Mazurkiewicz, Basic Notions of Trace Theory. In *Lecture Notes in Computer Science* 354, Springer-Verlag, Berlin, 1989, 285 – 363.

[26] R. Morin, Decompositions of Asynchronous Systems. In *Proceedings of the CONCUR'98 International Conference on Concurrency Theory, Nice, France* (D. Sangiorgi and R. De Simone, eds), *Lecture Notes in Computer Science* 1466, Springer-Verlag, Berlin, 1998, 549 – 564.

[27] O. Müller, *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. Ph.D. thesis, Technische Universität München, 1998.

[28] G.J. Nutt, *Operating Systems: A Modern Perspective*, Addison-Wesley Publishers, Reading, Massachusetts, 1997.

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

[30] *Lectures on Petri Nets II: Applications* (W. Reisig and G. Rozenberg, eds.), *Lecture Notes in Computer Science* 1492, Springer-Verlag, Berlin, 1998.

[31] T. Rodden, Populating the Application: A Model of Awareness for Cooperative Applications. In *Proceedings of the CSCW'96 ACM Conference on Computer Supported Cooperative Work, Boston, Massachusetts* (M. Ackerman, ed.), ACM Press, 1996, 87 – 96.

[32] H. Shen and P. Dewan, Access Control for Collaborative Environments. In *Proceedings of the CSCW'92 ACM Conference on Computer Supported Cooperative Work, Toronto, Canada* (J. Turner and R. Kraut, eds.), ACM Press, 1992, 51 – 58.

[33] K. Sikkel, A Group-based Authorization Model for Cooperative Systems. In *Proceedings of the ECSCW'97 European conference on Computer Supported Cooperative Work, Lancaster, UK* (J. Hughes, W. Prinz, T. Rodden, and K. Schmidt, eds.), Kluwer Academic Publishers, 1997, 345 – 360.

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