Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 195 (2008) 41–55

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Calculus for Team Automata [⋆]

Maurice H. ter Beek[1]

Istituto di Scienza e Tecnologie dell'Informazione, CNR
via G. Moruzzi 1, 56124 Pisa, Italy

Fabio Gadducci[2]

Dipartimento di Informatica, Università di Pisa
via Buonarroti 2, 56125 Pisa, Italy

Dirk Janssens[3]

Department of Mathematics and Computer Science, University of Antwerp
Middelheimlaan 1, 2020 Antwerpen, Belgium

Abstract

Team automata are a formalism for the component-based specification of reactive, distributed systems. Their main feature is a flexible technique for specifying coordination patterns among systems, thus extending I/O automata. Furthermore, for some patterns the language recognized by a team automaton can be specified via those languages recognized by its components.
We introduce a process calculus tailored over team automata. Each automaton is described by a process, such that its associated (fragment of a) labeled transition system is bisimilar to the original automaton. The mapping is moreover denotational, since the operators defined on processes are in a bijective correspondence with a chosen family of coordination patterns and that correspondence is preserved by the mapping.
We thus extend to team automata a few classical results on I/O automata and their representation by process calculi. Moreover, besides providing a language for expressing team automata, we widen the family of coordination patterns for which an equational characterization of the language associated to a composite automaton can be provided. The latter result is obtained by providing a set of axioms, in ACP-style, for capturing bisimilarity in our calculus.

Keywords: Team automata, process calculi

1 Introduction

Team automata have originally been introduced in the context of Computer Supported Cooperative Work (CSCW for short) to formalize the conceptual and archi-

⋆ Research partially supported by the EU FP6-IST IP 16004 SENSORIA (*Software Engineering for Service-Oriented Overlay Computers*).

1 Email: maurice.terbeek@isti.cnr.it

2 Email: gadducci@di.unipi.it

3 Email: dirk.janssens@ua.ac.be

1571-0661/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2007.08.022

tectural levels of groupware systems [3,11,14]. As shown in [5], they represent an extension of I/O automata [15,16], and since their introduction they have proved their usefulness in various application fields [2,6,7]. Team automata form a mathematical framework enabling the recast of notions like communication, coordination and cooperation in reactive, distributed systems. The model allows to separately specify the components of a system, to describe their interactions and to reuse the system as a component of a higher-level automaton, thus supporting a modular approach to system design. Its main feature is a flexible technique for specifying coordination patterns among distributed systems, extending classical I/O automata.

During the stepwise development of a system, it is desirable to have the possibility to decompose an abstract high-level specification of a large, complex design into a more concrete low-level specification. In order to guarantee that the decomposition is correct, it is necessary to prove that the chosen model is compositional, i.e., that the specification of a large system can be obtained from specifications of its components [13]. Unfortunately, as we show in Proposition 2.8 of this paper, for some of the coordination patterns employed so far it is not possible to capture the behavior of a (finite!) team automaton (intended as the language it recognizes) in terms of its components, by resorting only to set-theoretic operations on languages.

In order to overcome this difficulty, we introduce a calculus for team automata. Our proposal recalls those calculi that have been defined for (probabilistic) I/O automata [10,19,20], and the aim is to transfer the technology involving the equational characterization of behavioral equivalences on processes to team automata, in order to obtain a characterization of the relevant coordination patterns. The main idea underlying process algebras like the ACP [8], the CCS [17] and the CSP [12] frameworks is to use a set of operators, each one representing an architectural feature, for the inductive presentation of a complex system. Our calculus for team automata is essentially an enrichment of CSP, and its behavioral semantics is axiomatized by suitably adapted operators from ACP. Each team automaton is described by a process, in such a way that its associated (fragment of a) labeled transition system is behaviorally equivalent (namely, *bisimilar* [17]) to the original automaton. Furthermore, the mapping is denotational, since the operators on processes are in a bijective correspondence with a chosen family of coordination patterns, and that correspondence is preserved by the mapping.

One of our results is thus the extension to team automata of some classical results on I/O automata and their representation by process calculi. Another result concerns the compositionality of team automata. In [2,4] it was shown that certain team automata that are defined by a coordination pattern are compositional, in the sense that their languages can be obtained from the languages of their constituting automata. Besides proving that this characterization does not hold for all coordination patterns devised so far (even in the presence of acyclic automata: See the already mentioned Proposition 2.8), we use our calculus to provide some preliminary results on how to nevertheless obtain the language of a team automaton defined by a coordination pattern directly from its components. Hence, a compositionality result does exist, even if the manipulation of the languages of the components does not suffice. By providing a set of axioms, in ACP-style, to capture bisimilarity in our calculus, we thus enlarge the family of coordination patterns for which an equational characterization of the language associated to a team automaton can be provided. This axiomatization is sound and complete for finite processes only (i.e., equivalently, for acyclic automata) as is typical for all the calculi for describing automata that we are aware of (compare, e.g., the situation for (probabilistic) I/O automata, as reported in [10,19]). The search for a set of axioms for possibly recursive processes is currently under development.

The paper is organized as follows. Section 2 recalls the main definitions concerning team automata. Then, the syntax and the operational semantics of our calculus for team automata, as well as an equational theory for bisimulation over finite processes, are given in Section 3. Section 4 presents an encoding from team automata to processes that preserves bisimulation equivalence, while Section 5 offers an axiomatic characterization for language equivalence, thus partly solving (since the encoding preserves the composition patterns on automata) our modularity issues. Finally, Section 6 concludes the paper, hinting at possible future work.

2 A quick look at team automata

Roughly speaking, a team automaton consists of component automata — ordinary automata without final states and with a distinction of their sets of actions into input, output and internal actions — combined in such a way that they can perform shared actions. During each clock tick the components within a team can simultaneously participate in one instantaneous action (i.e., synchronize on this action) or remain idle. Component automata can thus be combined in a loose or more tight fashion, depending on which actions are to be synchronized and when.

Notations and terminology used in this article are initially fixed, after which team automata are introduced, slightly adapting the usual definition of team automata [3]. First, each automaton is assumed to have a unique initial state: This is not a real limitation, but it eases some constructions. Second, the usual distinction between input, output and internal actions in component and team automata is discarded. In [10,19,20] the distinction of the set of actions of I/O automata into input, output and internal actions is taken into account. For team automata, however, this distinction is much less important since — contrary to I/O automata — team automata are not required to be input enabling and synchronizations between output actions are not prohibited [3,5]. Hence in team automata the consideration of input and output actions does not have any syntactic significance. As a result, taking these actions into account would not affect our calculus. Moreover, it would not be easy to extend our calculus in order to deal with internal actions.

2.1 Some notation

We start by saying that for the sake of readability we often denote the set $\{1, \ldots, n\}$ by $[n]$; thus $[0]$ is the empty set. For $j \in [n]$, the (cartesian) product of sets V_i is denoted either by $\prod_{i \in [n]} V_i$ or by $V_1 \times \cdots \times V_n$, while the projection proj_j :

$\prod_{i\in[n]} V_i \to V_j$ is defined by $\mathrm{proj}_j((a_1,\ldots,a_n)) = a_j$. The set difference of sets V and W is denoted by $V \setminus W$ and, if V is finite, its cardinality is denoted by $\#V$.

Let $f : A \to A'$ and $g : B \to B'$ be two functions. Then $f\times g : A\times B \to A'\times B'$ is defined as $(f \times g)(a, b) = (f(a), g(b))$, and $f^{[2]}$ is a shorthand for $f \times f$.

Definition 2.1 A *labeled transition system* (LTS) is a triple (Q, Σ, δ), for a set Q of *states*, a set Σ of *actions* (with $Q\cap\Sigma = \varnothing$) and a set $\delta \subseteq Q \times \Sigma \times Q$ of *transitions*.

The set δ_a of *a-transitions* of $\mathcal{A}$ is defined as $\delta_a = \{ (q, q') \mid (q, a, q') \in \delta \}$ and an a-transition $(q, a, q') \in \delta$ may also be written as $q \xrightarrow{a} q'$. Action a is said to be *enabled in* $\mathcal{A}$ *at* state $q \in Q$, denoted by $a\ \mathrm{en}_{\mathcal{A}}\ q$, if there exists $q' \in Q$ such that $(q, q') \in \delta_a$. An a-transition $(q, q) \in \delta_a$ is called a *loop* (on a).

2.2 Synchronizations and team automata

Definition 2.2 A *(component) automaton* $\mathcal{C}$ is a *finite*, *rooted* LTS, i.e., a quadruple (Q, Σ, δ, q_0), where (Q, Σ, δ) is an LTS with finite Q and Σ and an *initial state* $q_0 \in Q$.

The set $\mathbb{C}(\mathcal{C})$ of *computations* of $\mathcal{C}$ is defined as $\mathbb{C}(\mathcal{C}) = \{ q_0a_1q_1a_2\cdots a_nq_n \mid n \geq 0$ and $(q_{i-1}, a_i, q_i) \in \delta$ for all $i \in [n] \}$.

The *language* $\mathbb{L}(\mathcal{C})$ of $\mathcal{C}$ is the set of sequences of symbols in Σ obtained by the obvious restriction (discarding the symbols in Q) of the computations in $\mathbb{C}(\mathcal{C})$.

In the sequel, we let $\mathcal{S} = \{ \mathcal{C}_i \mid i \in [n] \}$ be an arbitrary but fixed set of automata, with $n \geq 0$ and each $\mathcal{C}_i$ specified as $\mathcal{C}_i = (Q_i, \Sigma_i, \delta_i, q_{0i})$, and we let $\Sigma = \bigcup_{i\in[n]} \Sigma_i$.

A team automaton over $\mathcal{S}$ has the cartesian product of the state spaces of its components as its state space and its actions are those of its components. Its transition relation, however, is not fixed by those of its components: It is defined by choosing certain synchronizations of actions, while excluding others.

Definition 2.3 Let $a \in \Sigma$. The set $\Delta_a(\mathcal{S})$ of *synchronizations* of a is defined as $\Delta_a(\mathcal{S}) = \{ (q, q') \in \prod_{i\in[n]} Q_i \times \prod_{i\in[n]} Q_i \mid [\exists j \in [n] : \mathrm{proj}_j{}^{[2]}(q, q') \in \delta_{j,a}] \wedge [\forall i \in [n] : [\mathrm{proj}_i{}^{[2]}(q, q') \in \delta_{i,a}] \vee [\mathrm{proj}_i(q) = \mathrm{proj}_i(q')]] \}$.

The set $\Delta_a(\mathcal{S})$ contains all possible combinations of a-transitions of the components constituting $\mathcal{S}$, with all non-participating components remaining idle. It is explicitly required that in every synchronization at least one component participates. The state change of a team automaton over $\mathcal{S}$ is thus defined by the local state changes of the components constituting $\mathcal{S}$ that participate in the action of the team being executed. Hence, when defining a team automaton over $\mathcal{S}$, a specific subset of $\Delta_a(\mathcal{S})$ must be chosen for each action a. This defines an explicit communication pattern between those components constituting the team.

Definition 2.4 A *team automaton* $\mathcal{T}$ over $\mathcal{S}$ is a quadruple (Q, Σ, δ, q_0), with $Q = \prod_{i\in[n]} Q_i$, $\Sigma = \bigcup_{i\in[n]} \Sigma_i$, $\delta \subseteq Q \times \Sigma \times Q$ such that $\delta_a = \{ (q, q') \mid (q, a, q') \in \delta \} \subseteq \Delta_a(\mathcal{S})$ for all $a \in \Sigma$ and $q_0 = \prod_{i\in[n]} q_{0i}$.

2.3 Coordination patterns

In [3] several coordination patterns for the synchronizations of a team automaton were defined, each leading to a uniquely defined team automaton. These patterns fix the synchronizations of a team by defining — per action a — certain conditions on the a-transitions to be chosen from $\Delta_a(\mathcal{S})$, thus determining a unique subset of $\Delta_a(\mathcal{S})$ as the set of a-transitions of the team. Once such subsets have been chosen for all actions, the team automaton they define is unique.

Definition 2.5 Let $\mathcal{R}_a(\mathcal{S}) \subseteq \Delta_a(\mathcal{S})$, for all $a \in \Sigma$, and let $\mathcal{R}_\Sigma = \{\, \mathcal{R}_a(\mathcal{S}) \mid a \in \Sigma \,\}$. Then (Q, Σ, δ, q_0) is the $\mathcal{R}_\Sigma$-*team automaton* over $\mathcal{S}$ if $\delta_a = \mathcal{R}_a(\mathcal{S})$ for all $a \in \Sigma$.

In this notation we usually discard Σ if no confusion can arise. Here we consider three coordination patterns, based on those actions of a team automaton $\mathcal{T}$ that are *free*, *ai* or *si*. An action a is *free* in $\mathcal{T}$ if none of its a-transitions is brought about by a synchronization of a by two or more components from $\mathcal{S}$, action a is *action-indispensable* (*ai* for short) in $\mathcal{T}$ if all its a-transitions are brought about by a synchronization of all components from $\mathcal{S}$ sharing a and action a is *state-indispensable* (*si* for short) in $\mathcal{T}$ if all its a-transitions are brought about by a synchronization of all components from $\mathcal{S}$ in which a is currently enabled.

Definition 2.6 Let $a \in \Sigma$. Then we define the sets

- $\mathcal{R}^{no}_a(\mathcal{S}) = \Delta_a(\mathcal{S})$;
- $\mathcal{R}^{free}_a(\mathcal{S}) = \{\, (q, q') \in \Delta_a(\mathcal{S}) \mid \#\{\, i \in [n] \mid a \in \Sigma_i \, \wedge \, \mathrm{proj}_i{}^{[2]}(q, q') \in \delta_{i,a} \,\} = 1 \,\}$;
- $\mathcal{R}^{ai}_a(\mathcal{S}) = \{\, (q, q') \in \Delta_a(\mathcal{S}) \mid \forall\, i \in [n] : \, a \in \Sigma_i \Rightarrow \mathrm{proj}_i{}^{[2]}(q, q') \in \delta_{i,a} \,\}$;
- $\mathcal{R}^{si}_a(\mathcal{S}) = \{\, (q, q') \in \Delta_a(\mathcal{S}) \mid \forall\, i \in [n] : \, [a \in \Sigma_i \wedge a \ \mathrm{en}_{\mathcal{C}_i}\, q] \ \Rightarrow \mathrm{proj}_i{}^{[2]}(q, q') \in \delta_{i,a} \,\}$.

Each of these subsets of $\Delta_a(\mathcal{S})$ thus defines, for a given action $a \in \Sigma$, *all* transitions from $\Delta_a(\mathcal{S})$ that satisfy a certain type of synchronization. In the case of *no* constraints, this means that all a-transitions are allowed since nothing is required, and hence no transition is forbidden. In the other three cases, *all and only those* a-transitions are included that respect the specified property of a.

Before presenting an example to illustrate the notions defined so far, we provide shorthand notations for three specific types of team automata that we will use in the sequel. Let $n = 2$ (i.e., we consider $\mathcal{S} = \{\mathcal{C}_1, \mathcal{C}_2\}$) and let $\Gamma \subseteq \Sigma$. Then

- $\mathcal{C}_1 \;|||^{f}_{\Gamma}\; \mathcal{C}_2$ defines the $\mathcal{R}^{no}_{\Sigma\setminus\Gamma} \cup \mathcal{R}^{free}_{\Gamma}$-team automaton over $\mathcal{S}$;
- $\mathcal{C}_1 \;|||^{ai}_{\Gamma}\; \mathcal{C}_2$ defines the $\mathcal{R}^{no}_{\Sigma\setminus\Gamma} \cup \mathcal{R}^{ai}_{\Gamma}$-team automaton over $\mathcal{S}$;
- $\mathcal{C}_1 \;|||^{si}_{\Gamma}\; \mathcal{C}_2$ defines the $\mathcal{R}^{no}_{\Sigma\setminus\Gamma} \cup \mathcal{R}^{si}_{\Gamma}$-team automaton over $\mathcal{S}$.

Example 2.7 Consider the component automata $\mathcal{C}_1 = (\{p, p'\}, \{b\}, \{(p, b, p')\}, p)$ and $\mathcal{C}_2 = (\{q, q'\}, \{a, b\}, \{(q, b, q), (q, a, q')\}, q)$. They are depicted in Figure 1.

In Figure 2 we have depicted $\mathcal{C}_1 \;|||^{f}_{\{b\}}\; \mathcal{C}_2$, $\mathcal{C}_1 \;|||^{ai}_{\{b\}}\; \mathcal{C}_2$ and $\mathcal{C}_1 \;|||^{si}_{\{b\}}\; \mathcal{C}_2$. Note that $\mathcal{C}_1 \;|||^{f}_{\{b\}}\; \mathcal{C}_2$ has no b-transition from (p, q) to (p', q). In fact, this team automaton is different from the $\mathcal{R}^{no}_{\{b\}}$-team automaton over $\{\mathcal{C}_1, \mathcal{C}_2\}$ due to the loop on b in $\mathcal{C}_2$

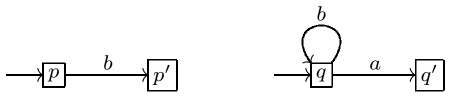

Fig. 1. From left to right: component automata $\mathcal{C}_1$ and $\mathcal{C}_2$.

(and this fact is going to be further explored in Section 3).

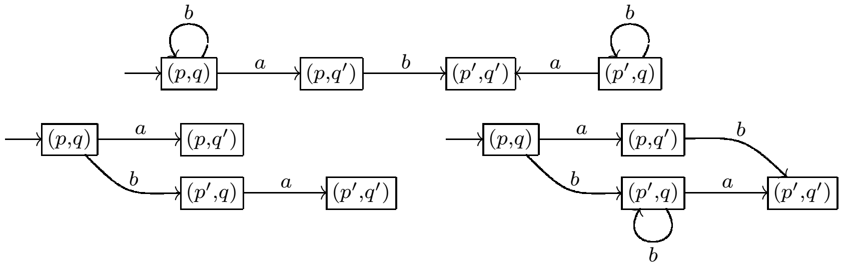

Fig. 2. Clockwise from top: team automata $\mathcal{C}_1 \; |||^{f}_{\{b\}} \; \mathcal{C}_2$, $\mathcal{C}_1 \; |||^{si}_{\{b\}} \; \mathcal{C}_2$ and $\mathcal{C}_1 \; |||^{ai}_{\{b\}} \; \mathcal{C}_2$.

2.4 *A negative result*

A team automaton over $\mathcal{S}$ is said to satisfy *compositionality* if its behavior (i.e., its language) can be described in terms of that of its constituting component automata: There exists a set-theoretic operation that when applied to the languages of the automata in $\mathcal{S}$, the language of a particular team over $\mathcal{S}$ results. In [2,4] it was shown that the construction of team automata according to certain patterns of synchronization, e.g., the ones leading to $\mathcal{R}^{free}$- and $\mathcal{R}^{ai}$-team automata, guarantees compositionality. In [2] it is moreover claimed that a similar result for the case of $\mathcal{R}^{si}$-team automata "seems impossible due to the simple fact that the behavior of component automata is stripped from all state information". This is proved below.

Proposition 2.8 *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two component automata. Then there exists no set-theoretic operation $|||$ on languages such that $\mathbb{L}(\mathcal{C}_1 \; |||^{si}_{\Sigma} \; \mathcal{C}_2) = \mathbb{L}(\mathcal{C}_1) \; ||| \; \mathbb{L}(\mathcal{C}_2)$.*

The proof is by counterexample. Consider the component automata in Figure 3. Then $\mathbb{L}(\mathcal{D}_2) = \mathbb{L}(\mathcal{D}_3)$, while $\mathbb{L}(\mathcal{D}_1 \; |||^{si}_{\Sigma} \; \mathcal{D}_2) = \mathbb{L}(\mathcal{D}_1 \; |||^{si}_{\Sigma} \; \mathcal{D}_3) \cup \{abc\}$.

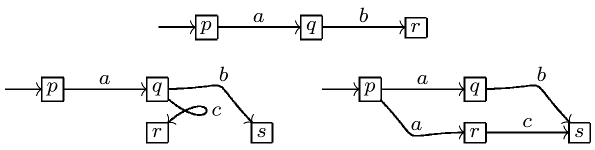

Fig. 3. Clockwise from top: component automata $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$.

In Section 5 we show that the calculus for team automata that we are going to introduce does provide a recipe to obtain the language of an acyclic $\mathcal{R}^{si}$-team automaton without actually constructing the team automaton from its constituting components: Translate the component automata to processes, perform the so-called eager parallel composition operation defined below, derive the normal form of the resulting process term and its associated regular language is the desired language.

3 A CSP-like process calculus

In this section we first introduce a simple process calculus: It is essentially an enrichment of Hoare's CSP [12]. We later present the associated operational semantics, also providing a set of axioms for recasting bisimilarity.

3.1 The syntax

We discuss now the syntax of our calculus. It is based on the *process algebra* paradigm: Each operator represents a basic feature of a possibly distributed system, and each system is thus defined inductively by composing the operators.

Definition 3.1 Let A be a countable set of *actions*, ranged over by $a, b, \ldots$, and let X be a countable set of *agent variables*, ranged over by $x, y, \ldots$, with $\wp_f(A)$ — the finite subsets of A — ranged over by L. *Agents* and *processes* are thus terms built from actions and variables according to the syntax

$$M ::= \mathsf{nil} \mid a.x \mid a.P \mid a.M \mid M + M \mid rec_x.M$$
$$P ::= M_c \mid P \parallel_L^f P \mid P \parallel_L^{ai} P \mid P \parallel_L^{si} P$$

As usual, a variable x is *free* if it does not occur inside the scope of a rec_x operator. The set of *(sequential) agents* is ranged over by $M, N, \ldots$, and for its subset of *closed* agents (i.e., containing no free variables) the subscript $\_c$ is added. The set of *processes* is denoted by $\mathcal{P}$ and ranged over by $P, Q, \ldots$, and a process is *finite* if it contains no occurrence of a recursion operator.

The constant nil represents the terminated process. The *action prefix* $a.P$ can perform an atomic action a and then evolve to P. Summation $+$ denotes *non-deterministic choice*: $M + N$ behaves either as M or as N, the choice being triggered by the execution of an action. The intended meaning of the *recursion* operator $rec_x.M$ is the process defined by the equation $x = M$, with the further restriction implicitly ensured by the syntax, namely that only closed terms may be inserted into a parallel composition operator: This assumption corresponds to what are usually called *size-bounded* processes, and it is formalized by Proposition 3.3 below.

There are three different notions of parallel composition. Basically, $P \parallel_L^{ai} Q$ means that processes P and Q must evolve synchronously with respect to all actions in L, while they may evolve independently of each other with respect to actions $a \notin L$, i.e., the actions in L are synchronized according to the *ai* type of synchronization. Similarly for its *eager* version: Also in $P \parallel_L^{si} Q$ both processes must synchronize on the actions in L, but now a process may in any case evolve with any action that is not offered at the moment by the other process, i.e., the actions in L are synchronized according to the *si* type of synchronization. Finally, in $P \parallel_L^f Q$ the two processes may synchronize on actions $a \notin L$, but both processes must evolve independently of each other for all actions in L, in which case a further restriction

Table 1
The operational semantics for $\mathcal{P}$.

$$act:\ \frac{-}{a.P \xrightarrow{a} P} \qquad sum:\ \frac{M \xrightarrow{a} M'}{M+N \xrightarrow{a} M'} \qquad rec:\ \frac{M[rec_x.M/x] \xrightarrow{a} N}{rec_x.M \xrightarrow{a} N}$$

$$par^f:\ \frac{P \xrightarrow{a} P',\ Q \xrightarrow{a} Q'}{P \parallel^f_L Q \xrightarrow{a} P' \parallel^f_L Q'}\ a \notin L \qquad asyn^f:\ \frac{P \xrightarrow{a} P'}{P \parallel^f_L Q \xrightarrow{a} P' \parallel^f_L Q}\ a \notin L$$

$$par^{ai}:\ \frac{P \xrightarrow{a} P',\ Q \xrightarrow{a} Q'}{P \parallel^{ai}_L Q \xrightarrow{a} P' \parallel^{ai}_L Q'} \qquad asyn^{ai}:\ \frac{P \xrightarrow{a} P'}{P \parallel^{ai}_L Q \xrightarrow{a} P' \parallel^{ai}_L Q}\ a \notin L$$

$$par^{si}:\ \frac{P \xrightarrow{a} P',\ Q \xrightarrow{a} Q'}{P \parallel^{si}_L Q \xrightarrow{a} P' \parallel^{si}_L Q'} \qquad asyn^{si}:\ \frac{P \xrightarrow{a} P'}{P \parallel^{si}_L Q \xrightarrow{a} P' \parallel^{si}_L Q}\ a \notin L$$

$$asyn^f_L:\ \frac{P \xrightarrow{a} P',\ Q \not\xrightarrow{a} Q}{P \parallel^f_L Q \xrightarrow{a} P' \parallel^f_L Q}\ a \in L \qquad asyn^{si}_L:\ \frac{P \xrightarrow{a} P',\ Q \not\xrightarrow{a}}{P \parallel^{si}_L Q \xrightarrow{a} P' \parallel^{si}_L Q}\ a \in L$$

is imposed in case one of the processes may loop: In order to faithfully mimic the *free* type of synchronization for all actions in L, a process may independently evolve with an action $a \in L$ only if the other process cannot evolve with a loop on a. This condition seems peculiar in the context of process calculi, but it is a consequence of the lack of explicit information on loops in team automata, i.e., in general it is impossible to distinguish whether or not a component with a loop on a in its current local state participates in the synchronization of the team on a. In [3] this led to the adoption of the maximal interpretation of the components' participation: Given a team transition (q, a, q') it is assumed that the jth component participates in this transition by executing $(\mathrm{proj}_j(q), a, \mathrm{proj}_j(q'))$ whenever $\mathrm{proj}^{[2]}(q, q') \in \delta_{j,a}$, whereas otherwise no transition takes place in the jth component (see Example 2.7).

3.2 The operational semantics

The operational semantics of the calculus is described by the LTS $(\mathcal{P}, A, \rightarrow)$, where $\rightarrow \subseteq \mathcal{P} \times A \times \mathcal{P}$ is defined in the so-called SOS style [18] as the least relation that satisfies the set of axioms and inference rules of Table 1 (where we omitted the symmetric rules for the choice operator and for the three parallel composition operators). Note also the negative premises occurring in the last two rules, namely $asyn^f_L$ and $asyn^{si}_L$: $Q \not\xrightarrow{a}$ means that from Q there is no outgoing transition labeled with a and $Q \not\xrightarrow{a} Q$ means that from Q there is no outgoing transition labeled with a that results in a cycle. Due to the restricted structure of the processes, and since the inference rules increase the size of a process, the least transition relation is well-defined. The semantics of a process $P \in \mathcal{P}$, denoted by $LTS(P)$, is defined as the rooted LTS $LTS(P) = (\mathcal{P}, A, \rightarrow, P)$.

Example 3.2 Consider the simple sequential agents $M = b.\mathsf{nil}$ and $N = rec_x(b.x + a.\mathsf{nil})$. Their associated rooted LTS's are depicted in Figure 4.

Let $L = \{b\}$. Then, the LTS's resulting from the application of the three parallel composition operators to M and N are depicted in Figure 5.

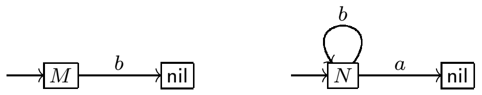

Fig. 4. From left to right: $LTS(M)$, $M = b.\mathsf{nil}$, and $LTS(N)$, $N = rec_x(b.x + a.\mathsf{nil})$.

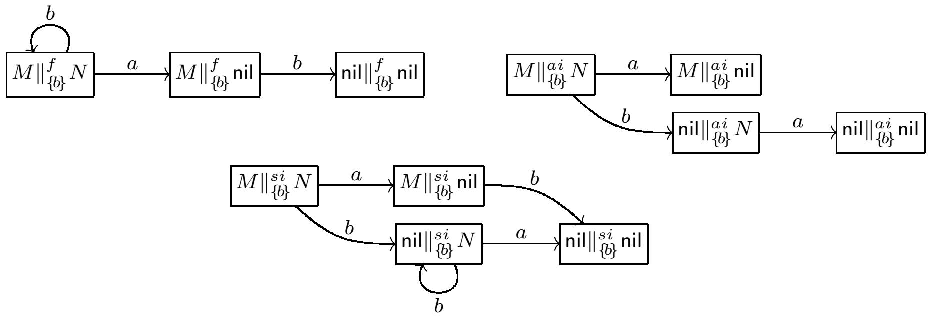
$M\|^f_{\{b\}}N$
$M\|^f_{\{b\}}\mathsf{nil}$
$\mathsf{nil}\|^f_{\{b\}}\mathsf{nil}$
$M\|^{ai}_{\{b\}}N$
$M\|^{ai}_{\{b\}}\mathsf{nil}$
$\mathsf{nil}\|^{ai}_{\{b\}}N$
$\mathsf{nil}\|^{ai}_{\{b\}}\mathsf{nil}$
$M\|^{si}_{\{b\}}N$
$M\|^{si}_{\{b\}}\mathsf{nil}$
$\mathsf{nil}\|^{si}_{\{b\}}N$
$\mathsf{nil}\|^{si}_{\{b\}}\mathsf{nil}$
a
b

Fig. 5. Clockwise from top: the LTS's for $M \|^f_{\{b\}} N$, $M \|^{ai}_{\{b\}} N$ and $M \|^{si}_{\{b\}} N$.

The next section focuses on an equational presentation for *bisimulation* equivalence, equating those processes exhibiting the same (non-deterministic) *observational* behavior. The result below states a property of our operational semantics, making precise the previous remark on size-bounded processes.

Proposition 3.3 *Let P be a process. Then, the rooted* LTS *$LTS(P)$ is finite.*

In other words, no syntactic explosion of a process during its evolution may occur, because only closed terms may be inserted into parallel composition operators.

3.3 Axioms for bisimulation

The aim of this section is to introduce a finite equational theory for bisimulation, which will later form the basis for the characterization of the language associated to a process (hence, to an automaton). First we define the notion of bisimulation.

Definition 3.4 Let $(Q_A, \Sigma_A, \delta_A)$ and $(Q_B, \Sigma_B, \delta_B)$ be LTS's. A relation $R \subseteq Q_A \times Q_B$ is a *bisimulation* if, whenever $(p, q) \in R$, then for any $a \in \Sigma_A \cup \Sigma_B$ holds

(i) if $p \xrightarrow{a} p'$, then $q \xrightarrow{a} q'$ for some $q' \in Q_B$ such that $(p', q') \in R$;

(ii) if $q \xrightarrow{a} q'$, then $p \xrightarrow{a} p'$ for some $p' \in Q_A$ such that $(p', q') \in R$.

Two states $q \in Q_A$ and $q' \in Q_B$ are said to be *bisimilar*, denoted by $q \simeq q'$, if there exists a bisimulation R such that $(q, q') \in R$. Two rooted LTS's $(Q_1, \Sigma_1, \delta_1, q_1)$ and $(Q_2, \Sigma_2, \delta_2, q_2)$ are said to be *bisimilar* if $q_1 \simeq q_2$. Two processes P and Q are said to be *bisimilar* if $LTS(P)$ and $LTS(Q)$ are.

It often occurs that bisimilarity (the largest bisimulation) is not a congruence with respect to the operators of the calculus, whenever there are rules containing negative premises. In fact, two of the SOS rules of Table 1 have negative premises, and the set of rules of our calculus does not fit the general *ntyft/ntyxt* format [9].

Table 2
Axioms for the choice operator.

$$M + M = M \qquad M + N = N + M$$

$$(M + N) + O = M + (N + O) \qquad M + \mathsf{nil} = M$$

Table 3
Axioms for the parallel composition operators.

$$P \parallel_L Q = P \mathbin{\lfloor\!\lfloor}_L Q + Q \mathbin{\lfloor\!\lfloor}_L P + P \mid_L Q \qquad (P + Q) \mid_L R = P \mid_L R + Q \mid_L R$$

$$(P + Q) \mathbin{\lfloor\!\lfloor}_L R = P \mathbin{\lfloor\!\lfloor}_L R + Q \mathbin{\lfloor\!\lfloor}_L R \qquad R \mid_L (P + Q) = R \mid_L P + R \mid_L Q$$

$$a.P \mathbin{\lfloor\!\lfloor}_L^f Q = a.(P \parallel_L^f Q) \qquad \mathsf{nil} \mathbin{\lfloor\!\lfloor}_L P = \mathsf{nil}$$

$$a.P \mid_L^f a.Q = \begin{cases} a.(P \parallel_L^f Q) & \text{if } a \notin L \\ \mathsf{nil} & \text{otherwise} \end{cases} \qquad \mathsf{nil} \mid_L P = \mathsf{nil} = P \mid_L \mathsf{nil}$$

$$a.P \mathbin{\lfloor\!\lfloor}_L^{ai} Q = \begin{cases} a.(P \parallel_L^{ai} Q) & \text{if } a \notin L \\ \mathsf{nil} & \text{otherwise} \end{cases} \qquad a.P \mid_L b.Q = \mathsf{nil}$$

$$a.P \mathbin{\lfloor\!\lfloor}_L^{si} Q = \begin{cases} a.(P \parallel_L^{si} Q) & \text{if } a \notin L \cap \mathrm{En}(Q) \\ \mathsf{nil} & \text{otherwise} \end{cases} \qquad a.P \mid_L^{\{ai,si\}} a.Q = a.(P \parallel_L^{\{ai,si\}} Q)$$

The main problem is the $asyn_L^f$ rule, since it contains an explicit hypothesis on the target state of the negative premise. It is in fact easy to see that the process $P = rec_x.a.x$ and its unfolding $Q = a.rec_x.a.x$ are bisimilar, while $P \parallel_{\{a\}}^f R$ and $Q \parallel_{\{a\}}^f R$ are not, for $R = a.b.\mathsf{nil}$. However, the problem disappears whenever we restrict our attention to finite processes, since the negative premise of the rule is always void. Thus, in the remaining of the section we consider finite processes only.

Our starting point for a finite equational theory for bisimulation is the solution routinely adopted in the ACP framework [8], i.e., to use suitable auxiliary operators (usually $\lfloor\!\lfloor$ and $|$) to split the parallel composition operator ($\parallel$) into its possible behaviors: Either an asynchronous evolution ($\lfloor\!\lfloor$) or a forced synchronization ($|$). For our calculus of finite processes this leads to the axioms concerning the choice and parallel composition operators reported in Tables 2 and 3, respectively. Concerning parallel composition, the lack of a superscript (either f, ai or si) means that the law holds for each of the three operators. Furthermore, given a process $P \in \mathcal{P}$, the predicate $\mathrm{En}(P)$ is defined as $\mathrm{En}(P) = \{ a \in A \mid \exists Q \in \mathcal{P} : P \xrightarrow{a} Q \}$.

Proposition 3.5 *Let P, Q be finite processes. Then P and Q are bisimilar if and only if they are equated by the axioms of Tables 2 and 3.*

Since the set of SOS rules of our calculus of finite processes can be transformed into a set of so-called *smooth* GSOS rules, we could as well have used the general procedure described in [1] to automatically generate a complete axiomatization for bisimulation. We however chose to provide a direct, intuitive set of axioms.

Note that the equations of Tables 2 and 3 can be oriented from left to right, so that they induce a rewriting system, modulo the so-called AC (associativity and commutativity) axioms for the choice operator $+$. So, two finite processes are bisimilar if they have the same (modulo AC) *normal form* (i.e., a process obtained from the original one and such that no further rewrite can be performed from it).

4 From automata to processes

The aim of this section is to present an encoding from automata to processes such that bisimulation equivalence is preserved. To this end, we now extend the usual definition of automata by assigning a specific set of states to be considered as entry points for the recursion operator.

Definition 4.1 Let X be a set of state variables. Then an automaton over X is a pair $\langle \mathcal{A}, f\rangle$, where $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ is an automaton and $f : X \to Q$ is an injective (possibly partial) function.

So, for the rest of this section we assume that for each automaton a set of its states is uniquely labeled by an element in X.

It is now possible to define our encoding from automata to processes.

Definition 4.2 Let $\langle \mathcal{A}, f\rangle$ be an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ over $X_{\mathcal{A}}$. Then the algorithm obtained by repeatedly applying the three steps below inductively defines an *essentially unique* — up to the choice of variables — process $\mathit{Exp}(\langle \mathcal{A}, f\rangle)$.

- If q_0 has no outgoing transitions, then

$$\mathit{Exp}(\langle \mathcal{A}, f\rangle) = \begin{cases} x & \text{if } f(x) = q_0, \text{ for some } x \in X_{\mathcal{A}}, \text{ and} \\ \mathsf{nil} & \text{otherwise;} \end{cases}$$

- If q_0 has $n > 0$ outgoing transitions (q_0, a_i, q_i) and no incoming ones, then

$$\mathit{Exp}(\langle \mathcal{A}, f\rangle) = \sum_{i\in\{1,\ldots,n\}} a_i.\mathit{Exp}(\langle \mathcal{A}_i, f\rangle)$$

  for automata $\mathcal{A}_i = (Q \setminus \{q_0\}, \Sigma, \delta \setminus \{\, (q_0, a, q) \mid a \in \Sigma, q \in Q \,\}, q_i)$ over $X_{\mathcal{A}}$;

- If q_0 has $n > 0$ outgoing transitions (q_0, a_i, q_i) and some incoming ones, then

$$\mathit{Exp}(\langle \mathcal{A}, f\rangle) = \mathit{rec}_x.\left(\sum_{i\in\{1,\ldots,n\}} a_i.\mathit{Exp}(\langle \mathcal{A}_i, f_x\rangle)\right)$$

  for a new variable x, automata $\mathcal{A}_i = (Q, \Sigma, \delta \setminus \{\, (q_0, a, q) \mid a \in \Sigma, q \in Q \,\}, q_i)$ over $X_{\mathcal{A}} \cup \{x\}$ and function f_x extending f such that $f_x(x) = q_0$.

Note that we have implicitly used the fact that the operator $+$ is commutative and associative, up to bisimulation (see the equations in Table 2). Note also that the second rule we introduced would actually not be needed: We added it just to associate a finite process to an acyclic automaton.

Proposition 4.3 *Let $\langle \mathcal{A}, f \rangle$ be an automaton over $X_{\mathcal{A}}$ and let $Exp(\langle \mathcal{A}, f \rangle)$ be its essentially unique process. Then $\mathcal{A}$ is bisimilar to $LTS(Exp(\langle \mathcal{A}, f \rangle))$.*

The proof can be given by coinductive arguments, by associating to the root of $\mathcal{A}$ the state $Exp(\langle \mathcal{A}, f \rangle)$, and to each state q_i all the processes $Exp(\langle \mathcal{A}_i, g \rangle)$ arising during the translation, and such that q_i is the root of $\mathcal{A}_i$.

Example 4.4 Consider component automata $\mathcal{C}_1 = (\{p, p'\}, \{b\}, \{(p, b, p')\}, \{p\})$ and $\mathcal{C}_2 = (\{q, q'\}, \{a, b\}, \{(q, b, q), (q, a, q')\}, \{q\})$ from Example 2.7 as automata over the sets of state variables $X_{\mathcal{C}_1}$ and $X_{\mathcal{C}_2}$, respectively.

By Definition 4.2 we obtain that $Exp(\langle \mathcal{C}_1', f_1 \rangle) = \mathsf{nil}$ and that $Exp(\langle \mathcal{C}_1, f_1 \rangle) = b.Exp(\langle \mathcal{C}_1', f_1 \rangle)$, with $\mathcal{C}_1' = (\{p'\}, \{b\}, \varnothing, p')$; thus $Exp(\langle \mathcal{C}_1, f_1 \rangle) = b.\mathsf{nil}$. Moreover, $\mathcal{C}_1$ trivially is bisimilar to $LTS(b.\mathsf{nil})$.

Again by Definition 4.2, $Exp(\langle \mathcal{C}_2, f_2 \rangle) = rec_x.(\, b.x + a.Exp(\langle \mathcal{C}_2', f_2' \rangle)\,)$, with $\mathcal{C}_2' = (\{q, q'\}, \{a, b\}, \varnothing, q')$ and $f_2'(x) = q$, and $Exp(\langle \mathcal{C}_2', f_2' \rangle) = \mathsf{nil}$; thus $Exp(\langle \mathcal{C}_2, f_2 \rangle) = rec_x.(b.x + a.\mathsf{nil})$. Finally, $\mathcal{C}_2$ trivially is bisimilar to $LTS(rec_x.(b.x + a.\mathsf{nil}))$.

It is worth noting that the encoding presented in Definition 4.2 is compositional: The parallel composition of two automata, according to any of the coordinaton patterns, is mapped into a process that is bisimilar to the parallel composition, according to the corresponding operator, of the encoding of the underlying automata.

Proposition 4.5 *Let $\mathcal{A}$ and $\mathcal{B}$ be automata, with alphabet Σ_A and Σ_B, respectively; let $\langle \mathcal{A}, \varnothing \rangle$ and $\langle \mathcal{B}, \varnothing \rangle$ be the associated automata over an empty set of state variables; and let $\mathcal{A}' = LTS(Exp(\langle \mathcal{A}, \varnothing \rangle))$ and $\mathcal{B}' = LTS(Exp(\langle \mathcal{B}, \varnothing \rangle))$ be the automata resulting from the encoding of the process expressions. Then, $\mathcal{A} \parallel_L^{\{f,si\}} \mathcal{B}$ is bisimilar to $\mathcal{A}' \parallel_L^{\{f,si\}} \mathcal{B}'$ and $\mathcal{A} \parallel_L^{ai} \mathcal{B}$ is bisimilar to $\mathcal{A}' \parallel_{L \cap \Sigma_A \cap \Sigma_B}^{ai} \mathcal{B}'$ for any set of names L.*

The further restriction for the *action-indispensable* coordination pattern is due to the loss of distinction between alphabets in the labeled transition system associated to the calculus. Indeed, the alphabet a component automaton is defined over is relevant only for the *ai*-coordination pattern, while for the other two patterns only the actions that actually occur in each automaton are relevant. Consider, e.g., the component automata $\mathcal{C}_1$ and $\mathcal{C}_2$ in Figure 1, and check the difference between the automaton $\mathcal{C}_1 \,|||_{\{a,b\}}^{ai}\, \mathcal{C}_2$, according to the choice of either $\{b\}$ or $\{a, b\}$ for Σ_1.

5 Equations for (finite) languages

Consider the equational presentation for bisimulation offered in Section 3. In particular, note how the normal form of a finite process corresponds to a regular expression, obtained by using the set of actions of the calculus as the alphabet and action prefixing and non-deterministic choice as operations. This intuition can be exploited to obtain an equational presentation for the language of a team automaton.

The correspondence between regular expressions and languages is a staple of theoretical computer science, so we do not repeat it here. We simply let $\mathcal{L}_P$ denote the language of a process P, which is easily derived from its normal form. Moreover, we let $\widehat{\mathcal{L}}$ denote the prefix-closed extension of a language $\mathcal{L}$ over Σ, i.e.,

$$\widehat{\mathcal{L}} = \{\, \alpha \in \Sigma^* \mid \exists\, \beta \in \Sigma^* : \, \alpha\beta \in \mathcal{L} \,\}.$$

As a direct corollary of Proposition 4.3 we thus obtain the following result.

Proposition 5.1 *Let $\mathcal{A}$ be an automaton. Then $\mathbb{L}(\mathcal{A}) = \widehat{\mathcal{L}}_{Exp(\langle \mathcal{A}, \varnothing \rangle)}$.*

This result suggests that our calculus can be used to derive the language of a team automaton. This is not surprising, since bisimulation is always finer than language equivalence: This property is just considered in an environment that is slightly more complex than usual, since our calculus contains three different operators for parallel composition. The previous result moreover suggests the use of equational laws to distill a normal form that is simpler than the original automaton.

Proposition 5.2 *Let P, Q be finite processes. Then $\widehat{\mathcal{L}}_P = \widehat{\mathcal{L}}_Q$ if and only if the normal forms of P and Q are equated by the axioms of $+$ (Table 2) and the axiom*

$$a.M + a.N = a.(M + N).$$

Also this equation can be interpreted as a left-to-right rewriting rule, allowing the further reduction of the normal form of a process. However, it is important to realize that this axiom could not simply have been added to Tables 2 and 3, since *critical pairs* would have arisen due to this axiom's incompatibility with the distributivity axioms of eager parallel composition.

Example 5.3 Consider the three automata $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$ used in the counterexample concerning Proposition 2.8, as shown in Figure 3. Ignoring the above axiom, then clearly their associated processes D_1, D_2 and D_3 have the normal forms $a.b.\mathsf{nil}$, $a.b.\mathsf{nil} + a.c.\mathsf{nil}$ and $a.(b.\mathsf{nil} + c.\mathsf{nil})$, respectively. Should the above axiom have been added to the set of equations in Tables 2 and 3, then clearly D_2 would be equated to D_3 and thus $D_1 \parallel^{si}_{\Sigma} D_2$ would have the same normal form (hence recognize the same language) as $D_1 \parallel^{si}_{\Sigma} D_3$, which is not the case: Instead, the normal form for $a.b.\mathsf{nil} \parallel^{si}_{\Sigma} a.b.\mathsf{nil} + a.c.\mathsf{nil}$ is $a.b.\mathsf{nil} + a.b.c.\mathsf{nil} + a.c.b.\mathsf{nil}$, reduced to $a.(b.c.\mathsf{nil} + c.b.\mathsf{nil})$; while the normal form for $a.b.\mathsf{nil} \parallel^{si}_{\Sigma} a.(b.\mathsf{nil} + c.\mathsf{nil})$ is $a.b.\mathsf{nil} + a.c.b.\mathsf{nil}$, reduced to $a.(b.\mathsf{nil} + c.b.\mathsf{nil})$. The associated languages are easily derived.

The situation so far is thus quite satisfactory for finite processes (i.e., equivalently, for acyclic automata): In order to prove the equivalence of two team automata with respect to the language they recognize, it is sufficient to consider the associated processes and analyze their normal forms. Moreover, it is relevant that the mapping from team automata to processes preserves, up to bisimulation, the three composition patterns that were considered in this paper: This result ensures that the procedure devised so far for obtaining the normal form is modular.

6 Conclusions and future work

In this paper we introduced a process calculus for team automata, extending some classical results on I/O automata. As a side-effect, we widened the family of team automata that guarantee a degree of compositionality by providing a way to obtain the language of a (finite) $\mathcal{R}^{si}$-team automaton from its components. While this language cannot be obtained through a direct manipulation of the languages of the component automata, the resulting degree of modularity favors the use of team automata in component-based system design.

Future work in this direction should lead to compositionality results also for other types of team automata. A first step in this direction could be to extend our calculus with parallel composition operators that mimic the various peer-to-peer and master-slave patterns of synchronization for team automata as introduced in [3], as well as mixtures of the synchronizations defined for team automata. As a matter of fact, [2,4] contain compositionality results not only for $\mathcal{R}^{\mathit{free}}$- and $\mathcal{R}^{ai}$-team automata, but also for team automata constructed according to a mixture of the *free* and *ai* synchronizations. It is important to recall, however, that the various peer-to-peer and master-slave patterns of synchronization make use of the distinction of the set of actions of team automata into input, output and internal actions. This means that in order to tackle the above issues, our calculus should first be extended to take this distinction into account.

Our correspondence results between automata and processes (as summed up by the two propositions in Section 4) relate the behavior of *possibly cyclic* automata and *possibly recursive* processes. We restrained however from tackling the axiomatization of recursive processes in our paper. It would be relatively easy to come up with a complete set of equational laws for those recursive processes not containing the parallel operators, since they basically boil down to regular expressions equipped with a Kleene star operator. On the other hand, the lack of a complete set of axioms for recursive processes is a common trait for all the calculi proposed for automata that we are aware of: Compare, e.g., the situation for (possibly probabilistic) I/O automata, as reported in [10,19]. We hope that our syntactical restriction will suffice to obtain a relatively small set of equations which is complete, but we leave this topic as the subject of future work.

Lastly, in order to be really useful in practical applications of team automata, it would be worthwhile to study the complexity of the algorithms introduced in this paper, e.g., what is the cost of checking the bimisimilarity between two automata, or of obtaining the language of a team automaton via its translation into a process.

Acknowledgement

We are grateful to Jetty Kleijn and to the anonymous referees for their useful comments on a preliminary version of this paper.

References

[1] Aceto, L., B. Bloom and F. W. Vaandrager, *Turning SOS rules into equations*, Information and Computation **111** (1994), pp. 1–52.

[2] ter Beek, M. H., "Team Automata—A Formal Approach to the Modeling of Collaboration Between System Components," Ph.D. thesis, Leiden University (2003).

[3] ter Beek, M. H., C. A. Ellis, J. Kleijn and G. Rozenberg, *Synchronizations in team automata for groupware systems*, Computer Supported Cooperative Work **12** (2003), pp. 21–69.

[4] ter Beek, M. H. and J. Kleijn, *Team automata satisfying compositionality*, in: K. Araki, S. Gnesi and D. Mandrioli, editors, *International Symposium of Formal Methods Europe*, Lect. Notes in Comp. Sci. **2805** (2003), pp. 381–400.

[5] ter Beek, M. H. and J. Kleijn, *Modularity for teams of I/O automata*, Information Processing Letters **95** (2005), pp. 487–495.

[6] ter Beek, M. H., G. Lenzini and M. Petrocchi, *Team automata for security: A survey*, in: F. R. and G. Zavattaro, editors, *International Workshop on Security Issues in Coordination Models, Languages, and Systems*, Electr. Notes in Theor. Comp. Sci. **128** (2005), pp. 105–119.

[7] ter Beek, M. H., G. Lenzini and M. Petrocchi, *A team automaton scenario for the analysis of security properties of communication protocols*, Journal of Automata, Languages and Combinatorics (2006), to appear.

[8] Bergstra, J. A. and J. W. Klop, *Process Algebra for Synchronous Communication*, Information and Control **60** (1984), pp. 109–137.

[9] Bol, R. N. and J. F. Groote, *The meaning of negative premises in transition system specifications*, Journal of the ACM **43** (1996), pp. 863–914.

[10] De Nicola, R. and R. Segala, *A process algebraic view of input/output automata*, Theoretical Computer Science **138** (1995), pp. 391–423.

[11] Ellis, C. A., *Team automata for groupware systems*, in: *International Conference on Supporting Group Work* (1997), pp. 415–424.

[12] Hoare, C. A. R., "Communicating Sequential Processes," Prentice Hall, 1985.

[13] Jonsson, B., *Compositional specification and verification of distributed systems*, ACM Transactions on Programming Languages and Systems **16** (1994), pp. 259–303.

[14] Kleijn, J., *Team automata for CSCW: A survey*, in: *Petri Net Technology for Communication-Based Systems*, Lect. Notes in Comp. Sci. **2472**, Springer, 2003 pp. 295–320.

[15] Lynch, N. A., "Distributed Algorithms," Morgan Kaufmann, 1996.

[16] Lynch, N. A. and M. R. Tuttle, *An introduction to input/output automata*, CWI Quarterly **2** (1989), pp. 219–246.

[17] Milner, R., "A Calculus of Communicating Systems," Lect. Notes in Comp. Sci. **92**, Springer, 1980.

[18] Plotkin, G., *A structural approach to operational semantics*, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University (1981).

[19] Stark, E. W., R. Cleaveland and S. A. Smolka, *A process-algebraic language for probabilistic I/O automata*, in: R. Amadio and D. Lugiez, editors, *International Conference on Concurrency Theory*, Lect. Notes in Comp. Sci. **2761** (2003), pp. 193–207.

[20] Vaandrager, F. W., *On the relationship between process algebra and input/output automata (extended abstract)*, in: *Symposium on Logic in Computer Science* (1991), pp. 387–398.