A calculus for team automata*

Maurice H. ter Beek[1], Fabio Gadducci[2], and Dirk Janssens[3]

[1] Istituto di Scienza e Tecnologie dell'Informazione, CNR
via G. Moruzzi 1, 56124 Pisa, Italy
`maurice.terbeek@isti.cnr.it`
[2] Dipartimento di Informatica, Università di Pisa
via Buonarroti 2, 56125 Pisa, Italy
`gadducci@di.unipi.it`
[3] Department of Mathematics and Computer Science, University of Antwerp
Middelheimlaan 1, 2020 Antwerpen, Belgium
`dirk.janssens@ua.ac.be`

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 distributed systems, extending classical I/O automata. Furthermore, for some of these patterns, the language recognised by a team automaton can be specified in terms of the languages recognised by its components.
The present paper introduces a process calculus tailored over team automata. Each automaton is thus described by a process, in such a way that its associated (fragment of a) labelled transition system is bisimilar to the original automaton. Furthermore, the mapping is proved to be 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.
The paper thus extends to team automata some classical results on I/O automata and their representation by process calculi. Moreover, besides providing a language for expressing team automata and their composition, we widen the family of coordination patterns for which an equational characterisation 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: Compositional specification, process calculi, team automata.

1 Introduction

Team automata have originally been introduced in the context of Computer Supported Cooperative Work (CSCW for short) to formalise the conceptual and architectural levels of groupware systems [2, 9, 12]. As shown in [4], they represent an extension of classical I/O automata [13, 14], and since their introduction

* Work partly supported by the EU projects HPRN-CT-2002-00275 SegraVis (*Syntactic and Semantic Integration of Visual Modelling Techniques*) and IST-3-016004-IP-09 Sensoria (*Software Engineering for Service-Oriented Overlay Computers*).

they have proved their usefulness also in various application fields [1, 5]. Team automata form a mathematical framework to capture notions like communication, coordination and cooperation in reactive, distributed systems. The model allows one to separately specify the components of a system, to describe their interactions and to reuse the system as a component of a higher-level team automaton, thus supporting a compositional approach to system design. Such an approach benefits from stepwise development: An abstract high-level specification of a large, complex design is decomposed into a more concrete low-level specification by step-by-step refinement. To guarantee correct decompositions it is important that the chosen model is compositional, i.e., a specification of a large system can be obtained from specifications of its components [11].

In this paper we introduce a calculus for specifying team automata, similar in spirit to the calculi that have been defined for (probabilistic) I/O automata [8, 17, 18]. The main idea underlying process algebras like the ACP framework [6], Hoare's CSP [10] and Milner's CCS [15] is the use of basic processes and a set of fundamental operators to inductively construct more complex processes. Our calculus for team automata is essentially an enrichment of CSP, and its observational semantics is axiomatised by some suitably adapted operators from the ACP framework. Each automaton is described by a process, in such a way that its associated (fragment of a) labelled transition system is bisimilar to the original automaton. Furthermore, the mapping is proved to be 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 [1, 3] it was shown that certain team automata that are defined by a coordination pattern guarantee compositionality, in the sense that their languages can be obtained from the languages of their constituting automata. Furthermore, it was claimed that one specific other type of team automata does not. Besides proving the latter claim, we will use our calculus for team automata to show how to obtain the language of such a specific type of team automaton defined by a coordination pattern directly from its components. Thus, 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 enlarge the family of coordination patterns for which an equational characterisation of the language associated to a team automaton can be provided.

The paper is organised as follows. Section 2 introduces team automata. The syntax and the operational semantics of our calculus for team automata, as well as a finitary equational theory for bisimulation, are introduced in Section 3. Section 4 presents an encoding from processes to automata, and back, which preserves the bisimulation equivalence, while Section 5 offers a characterisation, via a suitable axiomatisation, for language equivalence, thus partly solving (since the encoding preserves the parallel operators on automata) our modularity issues. Finally, Section 6 concludes the paper, hinting at possible future work.

2 Team automata

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 a coordinated way such that they can perform shared actions. During each clock tick the components within a team can simultaneously participate in one instantaneous action, i.e., synchronise 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 synchronised and when.

We now fix some notations and terminology used in this article, after which we introduce team automata. However, we slightly adapt the usual definition of team automata [2]. First, we assume each automaton to have a unique initial state. This is of course not a real limitation, but it will ease some of the constructions below. Second, we discard the usual distinction between input, output and internal actions in component and team automata. In [8, 17, 18] 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 need not be input enabling and synchronisations between output actions may occur. Hence in team automata the consideration of input and output actions does not have any syntactic significance. As a result, taking input and output actions into account thus would not affect our calculus. On the other hand, it would not be difficult to extend our calculus in order to deal with internal actions.

For convenience we sometimes denote the set $\{1, \ldots, n\}$ by $[n]$; thus $[0] = \varnothing$. The (cartesian) product of sets V_i, with $i \in [n]$, is denoted either by $\prod_{i\in[n]} V_i$ or by $V_1 \times \cdots \times V_n$. For $j \in [n]$, $\mathrm{proj}_j : \prod_{i\in[n]} V_i \to V_j$ is defined by $\mathrm{proj}_j((a_1, \ldots, a_n)) = a_j$. The set difference of sets V and W is denoted by $V \setminus W$. For a finite set V, its cardinality is denoted by $\#V$. The empty word (a sequence of symbols) is denoted by λ.

Let $\varGamma$ and $\varSigma$ be sets of symbols. The morphism $\mathrm{pres}_{\varGamma,\varSigma} : \varGamma^* \to \varSigma^*$, defined by $\mathrm{pres}_{\varGamma,\varSigma}(a) = a$ if $a \in \varSigma$ and $\mathrm{pres}_{\varGamma,\varSigma}(a) = \lambda$ otherwise, preserves the symbols from $\varSigma$ and erases all other symbols. We discard $\varGamma$ when no confusion can arise.

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

Definition 1. *A* labelled transition system *(lts for short) is a triple $\mathcal{A} = (Q, \varSigma, \delta)$, with a set Q of* states, *a set $\varSigma$ of actions (satisfying $Q \cap \varSigma = \varnothing$) and a set $\delta \subseteq Q \times \varSigma \times Q$ of* labelled 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).

Definition 2. *A* (component) automaton *is a* rooted *lts, i.e., a quadruple $\mathcal{C} = (Q, \varSigma, \delta, q_0)$, where $(Q, \varSigma, \delta)$ is an lts and $q_0 \in Q$ is the* initial state.

The set $\mathbb{C}(\mathcal{C})$ *of* computations *of* $\mathcal{C}$ *is defined as* $\mathbb{C}(\mathcal{C}) = \{\, q_0 a_1 q_1 a_2 \cdots a_n q_n \mid n \geq 0$ *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 defined as* $\mathbb{L}(\mathcal{C}) = \mathit{pres}_{\Sigma}(\mathbb{C}(\mathcal{C}))$.

For the sequel we let $\mathcal{S} = \{\, \mathcal{C}_i \mid i \in [n]\,\}$ be an arbitrary but fixed set of component 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 based on but not fixed by those of its components: The transition relation of a team automaton over $\mathcal{S}$ is defined by choosing certain synchronisations of actions of its components, while excluding others.

Definition 3. *Let* $a \in \Sigma$. *The set* $\Delta_a(\mathcal{S})$ *of* synchronisations *of* a *is defined as* $\Delta_a(\mathcal{S}) = \{\, (q, q') \in \prod_{i \in [n]} Q_i \times \prod_{i \in [n]} Q_i \mid [\exists\, j \in [n] : \mathit{proj}_j^{[2]}(q, q') \in \delta_{j,a}] \wedge [\forall\, i \in [n] : [\mathit{proj}_i^{[2]}(q, q') \in \delta_{i,a}] \vee [\mathit{proj}_i(q) = \mathit{proj}_i(q')]]\,\}$.

The set $\Delta_a(\mathcal{S})$ thus 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 synchronisation 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 a certain kind of communication between the components constituting the team.

Definition 4. *A* team automaton *over* $\mathcal{S}$ *is a quadruple* $\mathcal{T} = (Q, \Sigma, \delta, q_0)$, *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* $q_0 = \prod_{i \in [n]} q_{0i}$.

In [2] several strategies for choosing the synchronisations of a team automaton were defined, each leading to a uniquely defined team automaton. These strategies fix the synchronisations 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 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* $\mathcal{T} = (Q, \Sigma, \delta, q_0)$ *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 when no confusion can arise. The subsets mentioned above are based on those actions of $\mathcal{T}$ 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 synchronisation of a by two or more components from $\mathcal{S}$, a is *action-indispensable* (*ai* for short) in $\mathcal{T}$ if all its a-transitions are brought about by a synchronisation of all components from $\mathcal{S}$ sharing a and a is *state-indispensable* (*si* for short) in $\mathcal{T}$ if all its a-transitions are brought about by a synchronisation of all components from $\mathcal{S}$ in which a is currently enabled.

Definition 6. *Let $a \in \Sigma$. Then the set*

- $\mathcal{R}^{no}_a(\mathcal{S})$ *is defined as* $\mathcal{R}^{no}_a(\mathcal{S}) = \Delta_a(\mathcal{S})$*;*
- $\mathcal{R}^{free}_a(\mathcal{S})$ *is defined as* $\mathcal{R}^{free}_a(\mathcal{S}) = \{\, (q,q') \in \Delta_a(\mathcal{S}) \mid \#\{\, i \in [n] \mid a \in \Sigma_i \wedge proj_i^{[2]}(q,q') \in \delta_{i,a} \,\} = 1 \,\}$*;*
- $\mathcal{R}^{ai}_a(\mathcal{S})$ *is defined as* $\mathcal{R}^{ai}_a(\mathcal{S}) = \{\, (q,q') \in \Delta_a(\mathcal{S}) \mid \forall\, i \in [n] : [a \in \Sigma_i \Rightarrow proj_i^{[2]}(q,q') \in \delta_{i,a}] \,\}$*;*
- $\mathcal{R}^{si}_a(\mathcal{S})$ *is defined as* $\mathcal{R}^{si}_a(\mathcal{S}) = \{\, (q,q') \in \Delta_a(\mathcal{S}) \mid \forall\, i \in [n] : [\,[a \in \Sigma_i \wedge a\ en_{\mathcal{A}_i}\ proj_i(q)] \Rightarrow proj_i^{[2]}(q,q') \in \delta_{i,a}] \,\}$*.*

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 obey to a certain type of synchronisation. In the case of *no* constraints, this means that all a-transitions are allowed since nothing is required, and thus no transition is forbidden. In the other three cases, *all and only those* a-transitions are included that respect the specified property of a.

Before presenting an example to illustrate the notions defined so far, we define 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

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

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

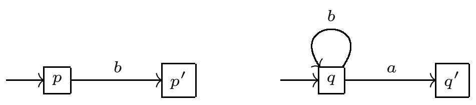

Fig. 1. The component automata $\mathcal{C}_1$ (left) and $\mathcal{C}_2$ (right).

In Figure 2 we have depicted $\mathcal{C}_1 \parallel^f \mathcal{C}_2$, $\mathcal{C}_1 \parallel^{ai}_{\{b\}} \mathcal{C}_2$ and $\mathcal{C}_1 \parallel^{si}_{\{b\}} \mathcal{C}_2$. Note that $\mathcal{C}_1 \parallel^f \mathcal{C}_2$ is different from the $\mathcal{R}^{no}_{\{a,b\}}$-team automaton over $\{\mathcal{C}_1, \mathcal{C}_2\}$.

A team automaton over $\mathcal{S}$ is said to satisfy *compositionality* if its behaviour can be described in terms of that of its constituting component automata: There exists a language-theoretic operation such that when it is applied to the languages of the component automata in $\mathcal{S}$, the language of a particular team over $\mathcal{S}$ results. In [1,3] it was shown that the construction of team automata according to certain types of synchronisation, e.g., the ones leading to $\mathcal{R}^{free}$- and $\mathcal{R}^{ai}$-team automata, guarantee compositionality. In [1] 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 behaviour of component automata is stripped from all state information". Here we prove this statement.

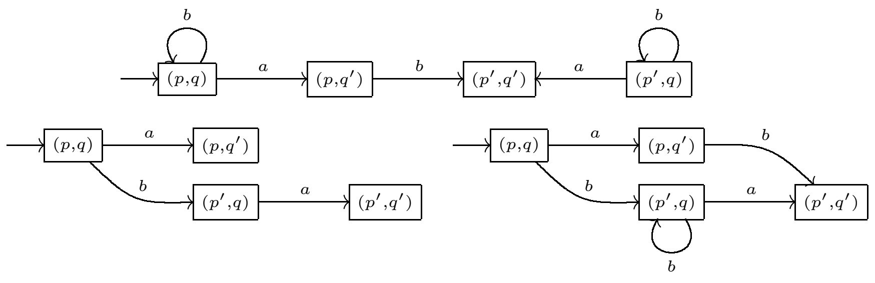

Fig. 2. Clockwise from top, the automata $\mathcal{C}_1 \|^f \mathcal{C}_2$, $\mathcal{C}_1 \|^{si}_{\{b\}} \mathcal{C}_2$ and $\mathcal{C}_1 \|^{ai}_{\{b\}} \mathcal{C}_2$.

Proposition 1. *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two component automata. Then there exists no language-theoretic operation $|||$ 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. Let us consider the team automata in Figure 3. Now, we have that $\mathbb{L}(\mathcal{D}_2) = \mathbb{L}(\mathcal{D}_3)$, while $\mathbb{L}(\mathcal{D}_1 \|^{si}_{\Sigma} \mathcal{D}_2) = \mathbb{L}(\mathcal{D}_1 \|^{si}_{\Sigma} \mathcal{D}_3) \cup \{abc\}$.

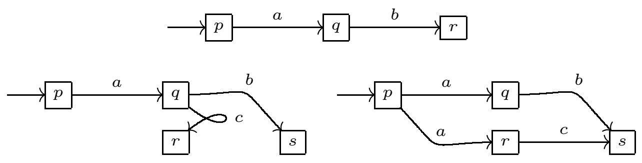

Fig. 3. Clockwise from top, the automata $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$

In Section 5 the calculus for team automata that we are about to introduce does provide a way to obtain the language of an $\mathcal{R}^{si}$-team automaton directly from its components, i.e., without actually constructing the automaton.

3 A CSP-like process calculus

In this section we introduce a simple process calculus, essentially an enrichment of Hoare's CSP [10], and then present the associated operational semantics.

3.1 Syntax and operational semantics

We assume countable sets of *actions* A, ranged over by $a, b, \ldots$, and *agent variables* X, ranged over by $x, y, \ldots$, with $\wp_f(A)$ — the finite subsets of A — ranged over by L. Terms are built from actions and variables according to the syntax

$$M ::= \mathsf{nil} \mid a.x \mid a.P \mid M + M \mid rec_x.M$$

$$P ::= M_c \mid P \parallel P \mid P \parallel_L P \mid P \parallel^e_L P$$

As usual, a variable x is *free* if it does not occure inside the scope of a rec_x operator. The set of *(sequential) agents* is ranged over by $M, N, \ldots$, and for its subsets of *closed* agents 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: $P + Q$ behaves either as P or as Q, 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 are inserted into a process context: This assumption corresponds to what are usually called size-bounded processes, and it is formalised by Proposition 2 below.

There are three different notions of parallel composition. Basically, $P \parallel_L Q$ means that processes P and Q must evolve synchronously with respect to each action $a \in L$, while they may evolve independently of each other with respect to actions $a \notin L$, i.e., the actions in L are synchronised according to the *ai* type of synchronisation. Similarly for its *eager* version: Also in $P \parallel^e_L Q$ both processes must synchronise on the actions in L, but now a process may in any case evolve with any action which is not offered at the moment by the other process, i.e., the actions in L are synchronised according to the *si* type of synchronisation. Finally, in $P \parallel Q$ each of the two processes may only evolve independently of each other, but a further restriction is imposed in the case that one of the processes may loop: In order to faithfully mimic the *free* type of synchronisation for all actions, a process may independently evolve with an action a only if the other process cannot evolve with a loop on a. However peculiar this condition may seem at first, it is a direct consequence of the fact that in team automata no explicit information on loops is provided, i.e., in general one cannot distinguish whether or not a component with a loop on a in its current local state participates in the synchronisation of the team on a. In [2] this led to the adoption of the maximal interpretation of the components' participation. That is, given a team transition (q, a, q') it is thus assumed that the jth component participates in this transition by executing $(\mathrm{proj}_j(q), a, \mathrm{proj}_j(q'))$ whenever $\mathrm{proj}^{[2]}(q, q') \in \delta_{j,a}$. Otherwise, no transition takes place in the jth component.

The operational semantics of this calculus is described by the lts $\mathcal{T} = (\mathcal{P}, A, \rightarrow)$, where $\rightarrow \subseteq \mathcal{P} \times A \times \mathcal{P}$ is defined in the so-called SOS style [16] as the least relation that satisfies the set of axioms and inference rules of Table 1. In Table 1 we have omitted the symmetric rules for the choice operator and for the three parallel composition operators. Moreover, given a process $P \in \mathcal{P}$, the predicates $\mathrm{Loop}(P)$ and $\mathrm{En}(P)$ are defined as

- $\mathrm{Loop}(P) = \{\, a \in A \mid P \xrightarrow{a} P \,\}$ and
- $\mathrm{En}(P) = \{\, a \in A \mid \exists\, Q \in \mathcal{P} :\, P \xrightarrow{a} Q \,\}$.

Finally, the semantics of a process $P \in \mathcal{P}$, denoted by $LTS(P)$, is defined as the rooted lts $LTS(P) = (\mathcal{P}, A, \rightarrow, P)$.

$$act : \frac{-}{a.P \xrightarrow{a} P} \qquad\qquad rec : \frac{M[{}^{rec_x.M}/_x] \xrightarrow{a} N}{rec_x.M \xrightarrow{a} N}$$

$$sum : \frac{M \xrightarrow{a} M'}{M + N \xrightarrow{a} M'} \qquad\qquad par : \frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}\ a \notin \mathrm{Loop}(Q)$$

$$par_L : \frac{P \xrightarrow{a} P',\ Q \xrightarrow{a} Q'}{P \parallel_L Q \xrightarrow{a} P' \parallel_L Q'}\ a \in L \qquad\qquad par_a : \frac{P \xrightarrow{a} P'}{P \parallel_L Q \xrightarrow{a} P' \parallel_L Q}\ a \notin L$$

$$par^e_L : \frac{P \xrightarrow{a} P',\ Q \xrightarrow{a} Q'}{P \parallel^e_L Q \xrightarrow{a} P' \parallel^e_L Q'}\ a \in L \qquad\qquad par^e_a : \frac{P \xrightarrow{a} P'}{P \parallel^e_L Q \xrightarrow{a} P' \parallel^e_L Q}\ a \notin L \cap \mathrm{En}(Q)$$

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

Example 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.

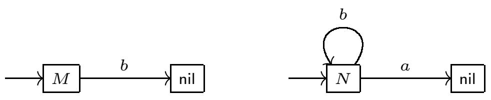

Fig. 4. $LTS(M = b.\mathsf{nil})$ (left) and $LTS(N = rec_x(b.x + a.\mathsf{nil}))$ (right).

Let $L = \{b\}$. Hence, no constraints are imposed on a. Then the lts's corresponding to the application of the three parallel composition operators to M and N are depicted in Figure 5.

Note that nil acts as the identity for both $\parallel$ and $\parallel^e_L$, while it is a sort of annihilator for $\parallel_L$. Indeed, the next section focuses on an equational presentation for *bisimulation* equivalence, equating those processes exhibiting the same (nondeterministic) behaviour. The result below states a property of our operational semantics, making precise the previous remark on size-bounded processes.

Proposition 2. *Let P be a process. Then the rooted lts $LTS(P)$ is finite.*

In other terms, there is no syntactical explosion of a process, during its evolution, because only closed terms may be inserted into a parallel operator.

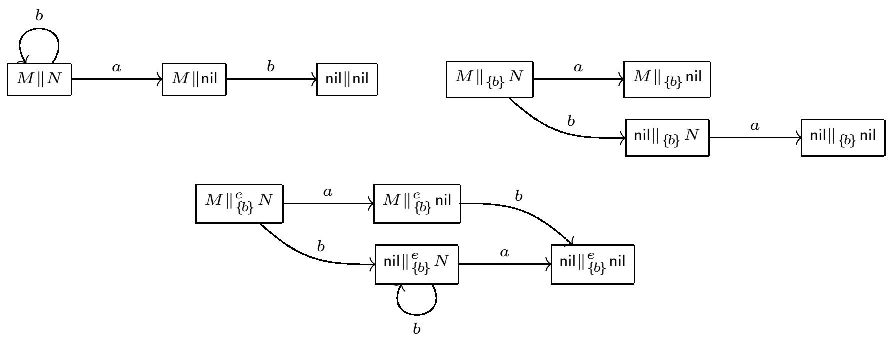
$M \| N$
$M \| \mathsf{nil}$
$\mathsf{nil} \| \mathsf{nil}$
$M \|_{\{b\}} N$
$M \|_{\{b\}} \mathsf{nil}$
$\mathsf{nil} \|_{\{b\}} N$
$\mathsf{nil} \|_{\{b\}} \mathsf{nil}$
$M \|^e_{\{b\}} N$
$M \|^e_{\{b\}} \mathsf{nil}$
$\mathsf{nil} \|^e_{\{b\}} N$
$\mathsf{nil} \|^e_{\{b\}} \mathsf{nil}$
a
b

Fig. 5. Clockwise from top, the lts's for $M \parallel N$, $M \parallel_{\{b\}} N$ and $M \parallel^e_{\{b\}} N$.

3.2 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 characterisation of the language associated to a process (hence, to an automaton). First we define the notion of bisimulation.

Definition 7. *Let $\mathcal{T} = (Q, \Sigma, \delta)$ be an lts. A relation $R \subseteq Q \times Q$ is a* bisimulation *if it is symmetric and for each $(p, q) \in R$ and $a \in \Sigma$ holds that*

– *whenever $p \xrightarrow{a} p'$, then $q \xrightarrow{a} q'$ for some $q' \in Q$ such that $(p', q') \in R$.*

Two states $q, q' \in Q$ 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 $\mathcal{T}_1 = (Q_1, \Sigma_1, \delta_1, q_1)$ and $\mathcal{T}_2 = (Q_2, \Sigma_2, \delta_2, q_2)$ are* bisimilar *if $q_1 \simeq q_2$.*

Our starting point for a finite equational theory for bisimulation is the solution routinely adopted in the ACP framework [6], i.e., the use of suitable auxiliary operators (usually $\llfloor$ and $|$) to split the parallel composition operator ($\|$) into its possible behaviours: either an asynchronous evolution ($\llfloor$) or a forced synchronisation ($|$). For our calculus this leads to the axioms of Tables 2 and 3, where an equation with occurrences of (e) is intended to hold both in case each occurrence is replaced by e and in case each occurrence is simply removed.

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

Note that the equations can be oriented from left to right, so that they actually induce a rewriting system, modulo the so-called ACI (associativity, commutativity and identity) axioms for the choice operator. So, two finite processes are bisimilar if they have the same *normal form*. Concerning recursive processes, we offer some preliminary considerations in the following sections.

$$P + P = P \qquad P + Q = Q + P$$

$$(P + Q) + R = P + (Q + R) \qquad P + \mathsf{nil} = P$$

$$P \parallel Q = P \lfloor\!\lfloor\, Q + Q \lfloor\!\lfloor\, P \qquad (P + Q) \lfloor\!\lfloor\, R = P \lfloor\!\lfloor\, R + Q \lfloor\!\lfloor\, R$$

$$\mathsf{nil} \lfloor\!\lfloor\, P = \mathsf{nil} \qquad a.P \lfloor\!\lfloor\, Q = \begin{cases} a.(P \parallel Q) & \text{if } a \notin \mathrm{Loop}(Q) \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

Table 2. Axioms for choice and asynchronous parallel composition.

$$P \parallel_L^{(e)} Q = P \lfloor\!\lfloor_L^{(e)} Q + Q \lfloor\!\lfloor_L^{(e)} P + P \,|_L^{(e)}\, Q \qquad (P + Q) \,|_L^{(e)}\, R = P \,|_L^{(e)}\, R + Q \,|_L^{(e)}\, R$$

$$R \,|_L^{(e)}\, (P + Q) = R \,|_L^{(e)}\, P + R \,|_L^{(e)}\, Q \qquad \mathsf{nil} \,|_L^{(e)}\, P = \mathsf{nil} = P \,|_L^{(e)}\, \mathsf{nil}$$

$$(P + Q) \lfloor\!\lfloor_L^{(e)} R = P \lfloor\!\lfloor_L^{(e)} R + Q \lfloor\!\lfloor_L^{(e)} R \qquad \mathsf{nil} \lfloor\!\lfloor_L^{(e)} P = \mathsf{nil}$$

$$a.P \,|_L^{(e)}\, b.Q = \mathsf{nil} \qquad a.P \,|_L^{(e)}\, a.Q = \begin{cases} a.(P \parallel_L^{(e)} Q) & \text{if } a \in L \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

$$a.P \lfloor\!\lfloor_L Q = \begin{cases} a.(P \parallel_L Q) & \text{if } a \notin L \\ \mathsf{nil} & \text{otherwise} \end{cases} \qquad a.P \lfloor\!\lfloor_L^{e} Q = \begin{cases} a.(P \parallel_L^{e} Q) & \text{if } a \notin L \cap \mathrm{En}(Q) \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

Table 3. Axioms for (eager) synchronous parallel composition.

4 From processes to automata, and back

The aim of this section is to present an encoding from processes to automata, such that the 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 8. *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 (but not necessarily total) function.*

So, for the rest of this section we assume that for each automaton a set of its states is uniquely labelled by an element in X. This extension allows the definition of a few operations on automata, mimicking process composition.

Definition 9. *Let $\langle \mathcal{A}, f \rangle$ be an automaton over $X_{\mathcal{A}}$. Then $a \cdot \mathcal{A}$ is the automaton over $X_{\mathcal{A}}$ obtained by creating a new initial state and adding an a-transition from this new initial state to the original initial state of $\mathcal{A}$.*

Next, let $\langle \mathcal{B}, g \rangle$ an automaton over $X_{\mathcal{B}}$. Then $\mathcal{A} + \mathcal{B}$ is the automaton over $X_{\mathcal{A}} \cup X_{\mathcal{B}}$ obtained by first taking the disjoint union of $\mathcal{A}$ and $\mathcal{B}$ and then coalescing the roots and those states labelled by the same variable.

Finally, let $x \in X_{\mathcal{A}}$. Then $\mathcal{A} \uparrow_x$ is the automaton over $X_{\mathcal{A}} \setminus \{x\}$ obtained by coalescing the root with the state labelled by x (if it exists).

It is now possible to define our encoding from processes to automata. For the sake of simplicity, we assume all the variables that are used in the occurrences of the recursion operator to be different.

Definition 10. *Let P be a process and let X be a set of variables (not including those occurring bound in P). Then the automaton $[\![P]\!]_X$ is defined by*

- $[\![\mathsf{nil}]\!]_X = \langle \mathcal{A}_{\mathsf{nil}}, \varnothing \rangle$, *where* $\mathcal{A}_{\mathsf{nil}} = (\{\mathsf{nil}\}, \varnothing, \varnothing, \mathsf{nil})$;
- $[\![x]\!]_X = \langle \mathcal{A}_x, f \rangle$, *where* $\mathcal{A}_x = (\{q_0\}, \varnothing, \varnothing, q_0)$ *and* $f(x) = q_0$;
- $[\![a.P]\!]_X = a \cdot [\![P]\!]_X$;
- $[\![M + N]\!]_X = [\![M]\!]_X + [\![N]\!]_X$;
- $[\![rec_x.M]\!]_X = [\![M]\!]_{X \cup \{x\}} \uparrow_x$;
- $[\![P \parallel Q]\!]_X = [\![P]\!]_\varnothing \parallel^f [\![Q]\!]_\varnothing$;
- $[\![P \parallel_L Q]\!]_X = [\![P]\!]_\varnothing \parallel_L^{ai} [\![Q]\!]_\varnothing$;
- $[\![P \parallel_L^e Q]\!]_X = [\![P]\!]_\varnothing \parallel_L^{si} [\![Q]\!]_\varnothing$.

We let $[\![P]\!]$ stand for $[\![P]\!]_\varnothing$. Now, please note that the fragment of the lts $LTS(P)$ associated to a process P actually defines an automaton, with the reachable processes as its states and its labelled transitions defined accordingly.

Proposition 4. *Let P be a process. Then $LTS(P)$ and $[\![P]\!]$ are bisimilar automata.*

The proof is given by coinductive arguments, exhibiting a suitable bisimulation between the two lts's. In this case, we relate each process P to $[\![P]\!]_X$ for any set X of variables containing those occurring free in P.

Clearly, also the inverse path can be followed, namely, obtaining a process out of an automaton.

Definition 11. *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 $Exp(\langle \mathcal{A}, f \rangle)$.*

- *If q_0 has no outgoing transitions, then*

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

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

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

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 $n \geq 0$ outgoing transitions (q_0, a_i, q_i) and some incoming ones, then*

$$Exp(\langle \mathcal{A}, f\rangle) = rec_x.\left(\sum_{i\in\{1,\dots,n\}} a_i.Exp(\langle \mathcal{A}_i, g\rangle)\right)$$

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 g extending f such that $g(x) = q_0$.

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 is actually not needed: we added it just to associate a finite process to an acyclic automaton.

Proposition 5. *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 $LTS(Exp(\langle \mathcal{A}, f\rangle))$.*

Once more, the proof is 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 $\mathcal{A}_i$.

It is noteworthy that the encoding of Definition 11 can further be proved to be compositional, up to bisimulation, with respect to the automata operators of parallel compositions.

Example 3. Consider the 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 1 as automata over $X_{\mathcal{C}_1}$ and $X_{\mathcal{C}_2}$, respectively.

By Definition 11, $Exp(\langle \mathcal{C}_1, f_1\rangle) = b.Exp(\langle \mathcal{C}_1', f_1\rangle)$, with $\mathcal{C}_1' = (\{p'\}, \{b\}, \varnothing, p')$, and $Exp(\langle \mathcal{C}_1', f_1\rangle) = \mathsf{nil}$; thus $Exp(\langle \mathcal{C}_1, f_1\rangle) = b.\mathsf{nil}$. Moreover, $\mathcal{C}_1$ trivially is bisimilar to $LTS(b.\mathsf{nil})$.

By Definition 11, $Exp(\langle \mathcal{C}_2, f_2\rangle) = rec_x.(\,b.x + a.Exp(\langle \mathcal{C}_2', f_2'\rangle)\,)$, with $\mathcal{C}_2' = (\{p, p'\}, \{a, b\}, \varnothing, p')$ and $f_2'(x) = p$, 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 $LTS(rec_x.(b.x + a.\mathsf{nil}))$.

5 Equations for (finite) languages

Let us consider again the equational presentation for bisimulation offered in the previous section. In particular, note how the normal form associated to a finite process intuitively corresponds to a regular expression, obtained using as alphabet the set of actions of the calculus as well as composition and non-deterministic choice. The aim of this section is to transform this view into an equational presentation for the language of a team automaton.

The correspondence between regular expressions and languages is a staple of theoretical computer science, and we do not repeat it here. We simply denote in the following by $\mathcal{L}_P$ the language associated to the normal form of a process P. Moreover, we denote by $\widehat{\mathcal{L}}$ the prefix-closed extension of a language $\mathcal{L}$ over Σ, namely

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

Proposition 6. *Let P be a finite process. Then the language of $[\![P]\!]$ coincides with $\widehat{\mathcal{L}}_P$.*

The previous result suggests the use of our calculus also for deriving the language associated to an automaton. This is not surprising, since bisimulation is finer than language equivalence: the language of an automaton corresponds to the set of paths originating from its root, hence, to the set of sequences of transitions of the associated process (and vice versa).

Furthermore, the above characterisation suggests the use of equational laws in order to distill a normal form that is simpler than the original automaton.

Proposition 7. *Let P, Q be finite processes. Then the languages of $[\![P]\!]$ and $[\![Q]\!]$ coincide iff the normal forms of P and Q are equated by using the ACI axioms of $+$ (see Table 2) and the axiom*

$$a.P + a.Q = a.(P + Q).$$

Once more, note how the equation can be interpreted as a left to right rewriting rule, obtaining for each process a further reduced normal form. It is important to realise that this axiom could not be simply added to the set of equations in Tables 2 and 3, since *critical pairs* would arise because it is not compatible with the distributivity of eager parallel composition.

Example 4. Let us consider the automata $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$ used for providing the counterexample concerning Proposition 1, as shown in Figure 3. If we ignore 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. But if the axiom had 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 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 good for finite processes, i.e., equivalently, for acyclic automata. In order to prove the language-theoretic equivalence of two team automata, it is sufficient to consider the associated processes, and then analyse their normal forms. It is interesting that the mapping to processes preserves the operators of parallel composition on automata, up to bisimulation, so that the procedure for obtaining the normal form becomes modular.

5.1 Tackling recursive processes

The situation is less satisfactory for recursive processes. This is far from surprising, since in general no finite set (or schemata) of axioms is available for bisimulation equivalence on calculi containing parallel composition operators.

We could define a set of axioms completely characterising what are usually called *basic process algebras*: In our case, the subset of sequential processes containing only the constant nil, prefixing, sum and recursion (see, e.g., [7] for a recent survey in ACP-style). For the full calculus, either *conditional* axioms are considered or processes are described by a set of suitable equations, admitting a unique solution. We follow the following path, by stating the result below.

Proposition 8. *Let P be a process, containing free only the variable x. Then the equation $x = P$ admits a unique solution, up to bisimulation.*

A solution for an equation $x = P$ is a process Q that is bisimilar to $P[{}^{Q}/_{x}]$. Similar results are standard in most process calculi for so-called guarded choice processes, i.e., such that each occurrence of a sum operator is inside prefixing. Our result is thus more restricted, since processes such as $rec_x.(a.x \parallel b.\mathsf{nil})$ are explicitly forbidden. This is not restrictive for our aims, since it is enough to model team automata, and it is a consequence of the semantics of eager parallel composition, in particular its negative premises in the inference rules describing its behaviour.

The proposition above tells us that, despite the lack of a complete axiomatisation, a recursive process is characterised by a regular expression, and thus it can be used for a modular description of the language associated to the process.

6 Conclusion

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

Future work in this direction should lead to compositionality results for other types of team automata, thus widening the family of team automata that guarantee a degree of compositionality even further. 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 types of synchronisation for team automata as introduced in [2], as well as mixtures of the synchronisations defined for team automata. As a matter of fact, [1, 3] contain compositionality results not only for $\mathcal{R}^{free}$- and $\mathcal{R}^{ai}$-team automata, but also for team automata constructed according to a mixture of the *free* and *ai* synchronisations. It is important to recall, however, that the various peer-to-peer and master-slave types of synchronisation 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.

Finally, 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 obtaining the language of a team automaton via its translation into processes. Furthermore, it would be worhwhile to check thoroughly the possible axioms of the full calculus, and explicitly derive the regular expression associated to a recursive process, as well as the expressiveness of the operators of our calculus, particularly the eager synchronisation.

References

1. M.H. ter Beek. *Team Automata — A Formal Approach to the Modeling of Collaboration Between System Components.* PhD thesis, Leiden Institute of Advanced Computer Science, Leiden University, 2003.
2. M.H. ter Beek, C.A. Ellis, J. Kleijn, and G. Rozenberg. Synchronizations in Team Automata for Groupware Systems. *Computer Supported Cooperative Work — The Journal of Collaborative Computing*, 12(1):21–69, 2003.
3. M.H. ter Beek and J. Kleijn. Team Automata Satisfying Compositionality. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings of FME 2003: Formal Methods — the 12th International Symposium of Formal Methods Europe, Pisa, Italy*, volume 2805 of *Lecture Notes in Computer Science*, pages 381–400. Springer-Verlag, Berlin, 2003.
4. M.H. ter Beek and J. Kleijn. Modularity for Teams of I/O Automata. *Information Processing Letters*, 95(5):487–495, 2005.
5. M.H. ter Beek, G. Lenzini, and M. Petrocchi. Team Automata for Security – A Survey –. In R. Focardi and G. Zavattaro, editors, *Proceedings of the 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems (SecCo'04), London, UK*, volume 128 of *Electronic Notes in Theoretical Computer Science*, pages 105–119. Elsevier Science Publishers, Amsterdam, 2005.
6. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1–3):109–137, 1984.
7. J.A. Bergstra and A. Ponse. Non-regular Iterators in Process Algebra. *Theoretical Computer Science*, 269:203–229, 2001.
8. R. De Nicola and R. Segala. A Process Algebraic View of Input/Output Automata. *Theoretical Computer Science*, 138(2):391–423, 1995.
9. C.A. Ellis. Team Automata for Groupware Systems. In S.C. Hayne and W. Prinz, editors, *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge (GROUP'97), Phoenix, AZ, U.S.A.*, pages 415–424. ACM Press, New York, 1997.
10. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
11. B. Jonsson. Compositional Specification and Verification of Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, 1994.
12. J. Kleijn. Team Automata for CSCW – A Survey –. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication-Based Systems — Advances in Petri Nets*, volume 2472 of *Lecture Notes in Computer Science*, pages 295–320. Springer-Verlag, Berlin, 2003.
13. N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, California, 1996.
14. N.A. Lynch and M.R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.
15. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

16. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
17. E.W. Stark, R. Cleaveland, and S.A. Smolka. A Process-Algebraic Language for Probabilistic I/O Automata. In R.M. Amadio and D. Lugiez, editors, *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR'03), Marseille, France*, volume 2761 of *Lecture Notes in Computer Science*, pages 189–203. Springer-Verlag, Berlin, 2003.
18. F.W. Vaandrager. On the relationship between process algebra and input/output automata (extended abstract). In A.R. Meyer, editor, *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS'91), Amsterdam, The Netherlands*, pages 387–398. IEEE Computer Society Press, New York, 1991.