Computing with Actions and Communications

Wan Fokkink[1,2] and Jan Willem Klop[1,2,3]

[1] Vrije Universiteit Amsterdam, Department of Computer Science
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
[2] CWI, Department of Software Engineering
PO Box 94079, 1090 GB Amsterdam, The Netherlands
[3] Radboud Universiteit, Institute for Computing and Information Sciences
PO Box 9010, 6500 GL Nijmegen
wanf@cs.vu.nl,jwk@cs.vu.nl

Abstract. The paradigm of computer science is nowadays shifting from computation to communication. In this paper we aim to give an impression of the algebra of actions and communications as it has been developed during the past quarter of a century.

We present specifically the system ACP, Algebra of Communicating Processes. Here $a, b, c, \ldots$ denote atomic events (or actions, or steps). For example, $a{\cdot}a{\cdot}(b + c)$ is the process able to perform two a-steps followed by a b-step or a c-step. This differs from the process $a{\cdot}a{\cdot}b + a{\cdot}a{\cdot}c$, due to the different timing of the choice between a b- and a c-step. Process x can operate in parallel with process y, notation $x\|y$, where some actions in x may happen simultaneously with some actions in y; the resulting actions are called communication actions. Such actions are 'half actions', needing their counterpart for the execution as a full action. In daily life a handshake is an example of a pair of half actions. We even have ternary communication actions on the keyboard of a PC: control-alt-delete. In music we find also quaternary actions, etc. The results of such internal communications we want to abstract away. This is formalised with the invisible action τ ('silent move'), facilitating the composition of modular hierarchies of process systems. The classical triple specification-implementation-verification then takes a straightforward form. The specification of the desired external behaviour is a simple process SPEC. The implementation is a complicated process IMP. The verification that the implementation is indeed correct, can be a purely algebraic, equational computation that a suitable abstraction of IMP yields SPEC: ABS(IMP) = SPEC.

Next to an overview of the process algebraic framework, we present a soundness and completeness proof for the axiom system underlying this framework, with respect to a fundamental semantics called rooted branching bisimulation. Thus a derivation of ABS(IMP) = SPEC implies that the implementation and its specification satisfy the same behavioural properties (soundness), while on the other hand the absence of such a derivation implies that the implementation and its specification are not rooted branching bisimulation equivalent (completeness).

Computing with actions and communications, process algebra, is logically and mathematically interesting. On top of that, process algebra

contributes to security and comfort: current process algebra tools are used to prove correctness of railway emplacements, helicopter software, and communication protocols in the remote control of a television set. This paper has grown out of an invited lecture of the second author at the Joint Mathematical BeNeLuxFra Conference in Ghent, May 2005. Slides of that lecture can be found at `www.cs.vu.nl/~jwk`.

1 History

A few words only on the origins of the present subject, seen from the Dutch perspective of the authors. The founding fathers of process algebra were around 1980 the Turing Award recipients Hoare (CSP) and Milner (CCS). Process algebra, then also named 'concurrency', was introduced in the Netherlands by de Bakker in 1980, via metric topology. In 1982-1990 an algebraic version named ACP, Algebra of Communicating Processes, was developed by a group of people, led by Bergstra and including Baeten and Klop. The subject was elaborated and various applications were developed by van Glabbeek, Vaandrager, Groote, Fokkink, van de Pol, Rutten, Kok and many others. Since 1985 there have been at least seventy PhD theses in the Netherlands in this research area.

2 Basic Process Algebra

Consider the following situation. On the five-by-five grid in Figure 1 we want to transport the black dot to the grey square, using elementary actions r, l, u, d. Several transport processes are possible, that we can specify as process expressions or terms using the operators '$+$' (choice) and '$\cdot$' (sequencing) as follows: $r^3 \cdot u^3$, or $(u \cdot r)^3$, or $r^3 \cdot u^3 + (u \cdot r)^3$, or $u^3 \cdot r^3 + r \cdot (r^2 \cdot u^3 + u^3 \cdot r^2)$. The last process is one that first chooses between reaching the goal via three up steps and next three right steps; or first a right step followed by the process $r^2 \cdot u^3 + u^3 \cdot r^2$ involving again a choice. It is at present of no concern who actually makes these choices, 'we', or the environment.

At this moment we already have a simple algebra of actions and processes. The processes originate by composing actions like u, r using the operators '$+$' and '$\cdot$'. This Basic Process Algebra (BPA for short) is axiomatised by the axioms in Table 1. According to the classical rules of equational logic, the axioms yield after instantiating equations like $(u+r) \cdot u = u \cdot u + r \cdot u$, equations that are indeed true in the situation of this example. Later on we will be able to represent the process of Figure 1 in a quite different and more economic way, namely as $r^3 \| u^3$, the parallel execution or 'merge', also named 'interleaving', of r^3 and u^3.

Note the absence in BPA of the distributivity law $z \cdot (x + y) = z \cdot x + z \cdot y$. This is crucial – but at this stage of our presentation the compelling reason is not immediately to be grasped. Adoption of this 'wrong' distributivity would invalidate our whole endeavour; the insight in the choice structure of a process would be lost, and so would our understanding whether complex systems may deadlock or may function correctly.

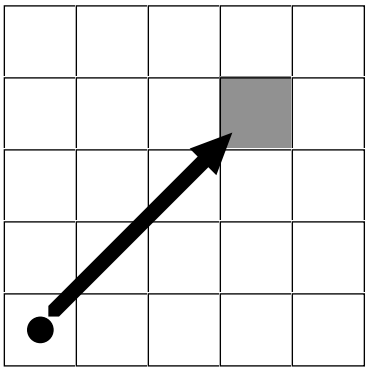

Fig. 1. Move dot to square.

$$x + y = y + x$$

$$x + (y + z) = (x + y) + z$$

$$x + x = x$$

$$(x + y)\cdot z = x\cdot z + y\cdot z$$

$$(x\cdot y)\cdot z = x\cdot(y\cdot z)$$

Table 1. Basic Process Algebra.

3 Recursion

The simple axiom system BPA is already very interesting – at least, if we add the construction principle to form infinite processes, namely *recursion*. This fascinating and useful principle for constructing infinite objects is illustrated by the well-known Droste nurse. In logic, computer science as well as in non well-founded set theory, we usually have more general situations in which the nurse carries two or more boxes of cacao, or where several nurses are operational with trays displaying several boxes, each picturing (part of) the situation.

Figure 2 displays the recursively defined process $X = a\cdot X + b$ (with a, b actions and recursion variable X) in the form of a transition diagram, or *process graph*. It is a 'finite-state' process with only two states, the nodes of the graph, the entrance node (root node) with label X and the empty termination node. The terminating paths ('finite traces') of the process X can be succinctly described by the regular expression a^*b, with the Kleene-star '*' denoting an iteration

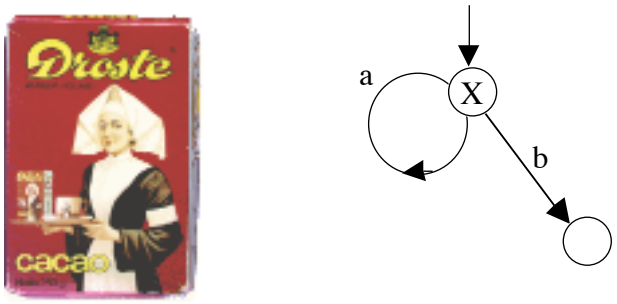

Fig. 2. Recursion.

of zero or more times. The equation $X = a{\cdot}X + b$ is called *linear*, because it does not contain products of recursion variables $X, Y, \ldots$. Linear systems of equations correspond to finite state processes, which often suffice for purposes of verification as discussed below. More interesting, from a theoretical perspective, are the non-linear systems of equations, denoting infinite state processes. Here we find a nice link to the theory of formal languages, in particular context-free languages. (Type 2 in the Chomsky hierarchy; the systems of linear equations correspond to type 3 languages, the regular ones.)

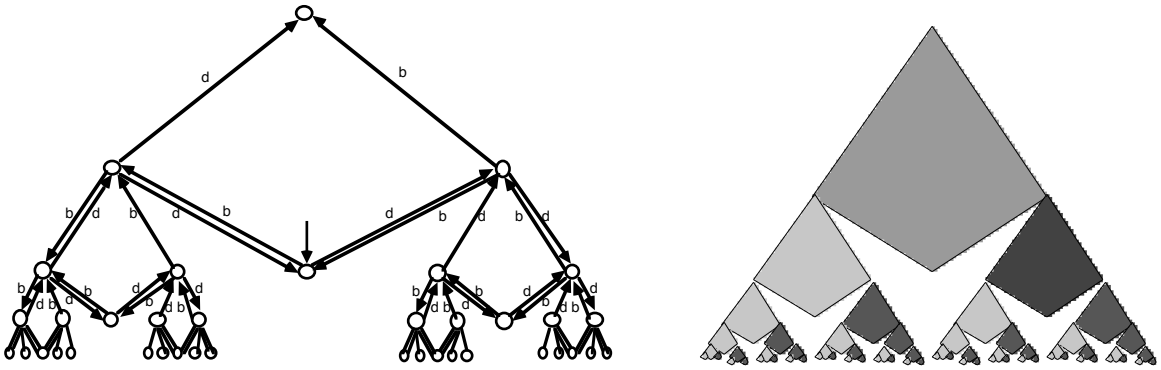

Fig. 3. Periodical structure of BPA processes with non-linear recursion.

Figure 3 (left) shows the process graph of the system of recursion equations $E = \{X = d{\cdot}Y + b{\cdot}Z, Y = b + b{\cdot}X + d{\cdot}Y{\cdot}Y, Z = d + d{\cdot}X + b{\cdot}Z{\cdot}Z\}$. This system is non-linear, due to the products of recursion variables $Y{\cdot}Y$ and $Z{\cdot}Z$. The process $\langle X|E\rangle$, which denotes the behaviour of X in the specification E, determines a context-free language, containing the words obtained by traveling from the root

node (marked by the short arrow) to the termination node on top. These are just all words containing equal numbers of b's and d's, e.g. $bdddbb$. The process graph is periodical in the sense that it is built from finitely many, namely three, different graph fragments, as suggested in Figure 3 (right). But note that it is not regular, i.e. it is infinite state.

An important consequence of this periodicity is that equality of such non-linear recursive BPA-processes is decidable, at least when we understand equality in the sense of process equality, given by bisimulation, a notion that we will discuss in Section 5. This decidability under bisimulation equality is in marked contrast with the classical theorem from formal language theory stating that equality of context-free grammars is undecidable. There, equality refers to the equality of the corresponding languages (the sets of finite completed traces), and that semantics is coarser than bisimulation semantics.

4 Parallel processes

To describe parallelism and communication of processes, BPA is extended with the parallel operator '$\|$', called *parallel composition* or *merge*. The idea is that with $\|$ we are able to let two processes run in parallel, such that their actions are interleaved. Figure 4 shows the merge of processes $a{\cdot}b$ and $b{\cdot}a$ (a and b are actions). Here the original process $a{\cdot}b\|b{\cdot}a$ is 'pealed off' by doing steps left and right as soon as they are possible. The process tree that arises, stops when all actions are done. The working of the operator $\|$ may be clear with this verbal exposition, but our real goal is to find an algebra that determines $\|$. Such an algebra is given in Tables 1 and 2; the latter table contains four axioms for $\|$, using an auxiliary operator '$\lfloor\!\lfloor$', called *left merge*. A rather deep theorem says that without such an auxiliary operator a finite axiomatisation is not possible.

$$x\|y = x\lfloor\!\lfloor y + y\lfloor\!\lfloor x$$

$$a\lfloor\!\lfloor x = a{\cdot}x$$

$$a{\cdot}x\lfloor\!\lfloor y = a{\cdot}(x\|y)$$

$$(x+y)\lfloor\!\lfloor z = x\lfloor\!\lfloor z + y\lfloor\!\lfloor z$$

Table 2. Axioms for the merge.

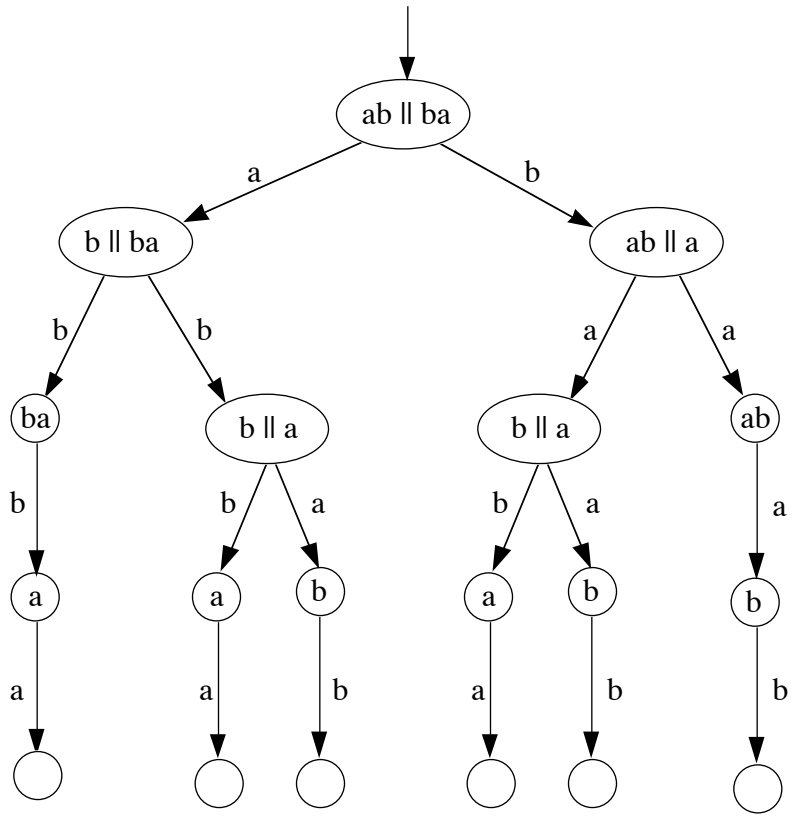

Fig. 4. Interleaving or merge of processes $a{\cdot}b$ and $b{\cdot}a$.

Now the procedure that gave us the process tree in Figure 4 in an informal way, takes the following formal algebraic form:

$$
\begin{array}{ll}
 & a{\cdot}b \| b{\cdot}a \\
= & a{\cdot}b \between b{\cdot}a & + b{\cdot}a \between a{\cdot}b \\
= & a{\cdot}(b \| b{\cdot}a) & + b{\cdot}(a \| a{\cdot}b) \\
= & a{\cdot}(b \between b{\cdot}a + b{\cdot}a \between b) & + b{\cdot}(a \between a{\cdot}b + a{\cdot}b \between a) \\
= & a(b{\cdot}b{\cdot}a + b{\cdot}(a \| b)) & + b{\cdot}(a{\cdot}a{\cdot}b + a{\cdot}(b \| a)) \\
= & a{\cdot}(b{\cdot}b{\cdot}a + b{\cdot}(a \between b + b \between a)) & + b{\cdot}(a{\cdot}a{\cdot}b + a{\cdot}(b \between a + a \between b)) \\
= & a{\cdot}(b{\cdot}b{\cdot}a + b{\cdot}(a{\cdot}b + b{\cdot}a)) & + b{\cdot}(a{\cdot}a{\cdot}b + a{\cdot}(b{\cdot}a + a{\cdot}b)).
\end{array}
$$

The final term

$$a{\cdot}(b{\cdot}b{\cdot}a + b{\cdot}(a{\cdot}b + b{\cdot}a)) + b{\cdot}(a{\cdot}a{\cdot}b + a{\cdot}(b{\cdot}a + a{\cdot}b))$$

is called a *basic term*: it does not contain parallel operators $\|$ and $\between$ anymore, but only the basic operators $+$ and $\cdot$. This basic term is the term corresponding to the process tree in Figure 4.

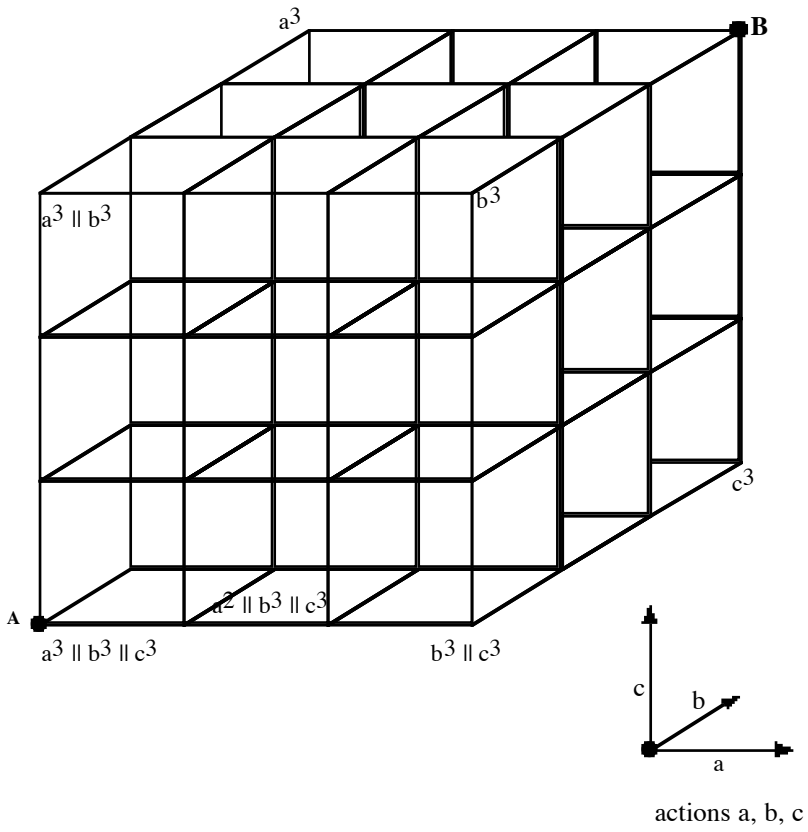

Fig. 5. The process $a^3 \| b^3 \| c^3$.

Figure 5 displays the example process $a^3 \| b^3 \| c^3$, with steps a, b, c as in Figure 5 right-below. This process describes the total of possibilities and choices that an ant encounters, walking from starting-point A to end-point B along the edges of the 3 x 3 x 3 cubes. When we compute this process $a^3 \| b^3 \| c^3$ with the axioms in Table 2 into a basic term, not containing the operators $\|$ and $\between$ anymore, just as we did for the example above for $a \cdot b \| b \cdot a$, then we find a term of several pages long, containing 1680 traces (i.e. different ways from A to B in Figure 5). The message of this example is both positive and negative: positive, because we can describe a process whose basic term is pages long, as a short process term $(a^3 \| b^3 \| c^3)$; negative, because apparently there soon arises a *state space explosion*. This phenomenon of enormous state spaces is a major concern for a large part of the efforts in formal software verification.

At Eindhoven University of Technology, a special visualisation tool for very large state spaces has been developed, in which clusters of states can be collapsed [3]. In Figure 6, a computer generated visualisation of the state space of a

protocol underlying the movement of robot arms for the construction of chips is depicted. Under this 'microscope', the protocol turned out to contain a deadlock, which manifests itself as a small dot.

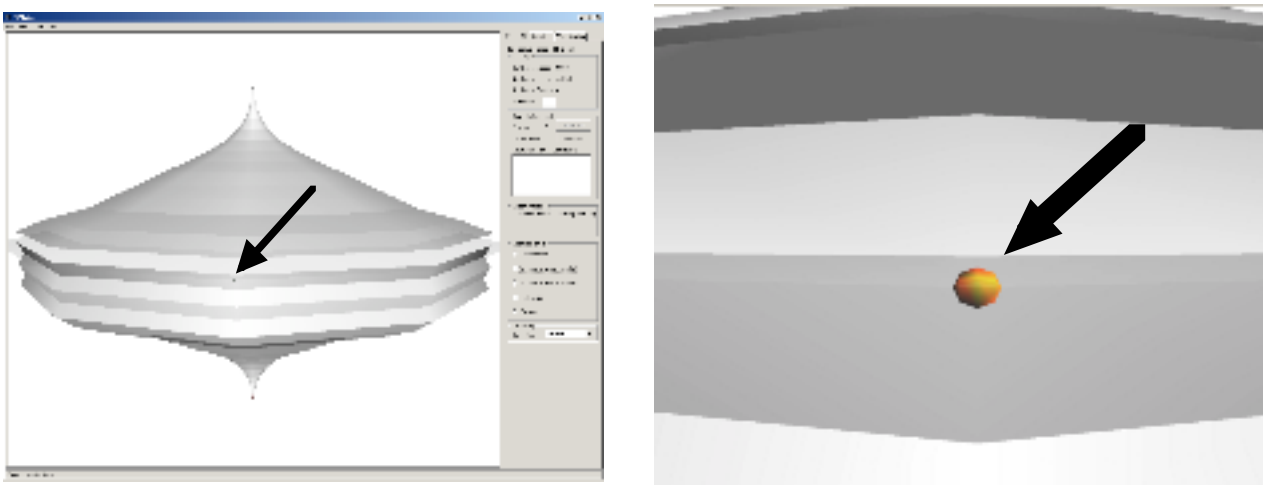

Fig. 6. Visualisation of a very large state space.

5 Semantics of processes

Up to this point we have only treated the algebra of processes. There is also the matter of semantics of processes – how do we interpret the process terms in a model, in other words, what is the meaning of process terms. There are several models, and in fact there is a whole model theory, also called 'comparative concurrency semantics'. There are models obtained via metric topology (due to de Bakker), models constructed with projective limits or ultraproducts, but to our taste the simplest and most straightforward models are the graph models, constructed from process graphs, modulo a fundamental equivalence called *bisimulation* [5]. Figure 7 illustrates this notion. The upper part of this figure shows a bisimulation relation, with the spaghetti-like strings, between a finite, cyclic process and the result of unwinding the cycle, once; the lower part of Figure 7 shows the same process in bisimulation with its infinite tree unwinding. (Now pairs of matching nodes are indicated by equal number labels.) The matching relation is a bisimulation if (1) the root nodes are related, and (2) in related nodes the same steps are possible, after which the end points of such steps must again be related. This is a so-called coinductive definition. Processes are bisimulation equivalent if there exists a bisimulation between these processes.

Bisimulation is the finest, most discriminating notion of process equality that exists; trace equality is the coarsest, least discriminating notion of process equality. A foundational underpinning of these process semantical notions was given in the last two decades with the help of a new set theory, the non well-founded set theory, mainly developed by Peter Aczel.

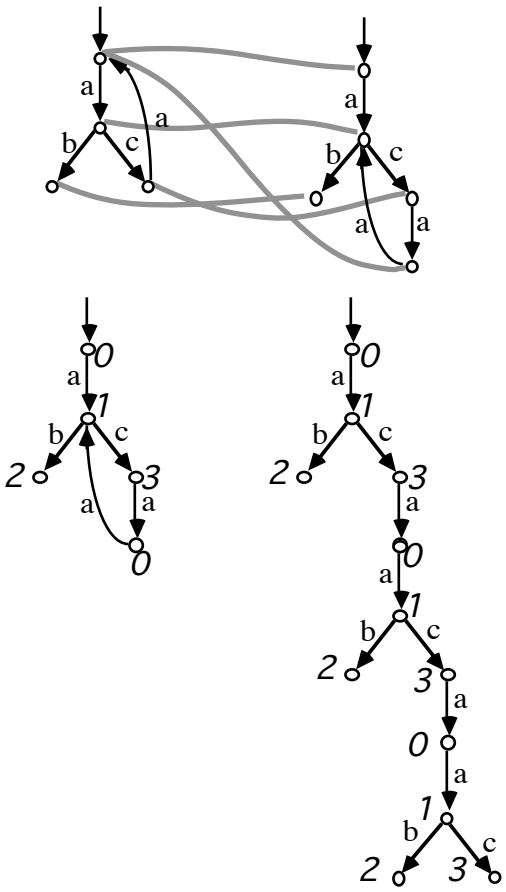

Fig. 7. Bisimulation of processes, after 'unwinding', once, and totally.

6 Communication

Thus far, we have developed an algebraic framework of processes that can be executed in parallel, but still independent from each other. We now turn our attention to the representation of communication. We approach this problem

from the negative side, namely starting from the phenomenon of *deadlock*, that arises when processes *want* to communicate, but cannot.

Imagine a ring of five processes $P_1, \ldots, P_5$, copies of each other, able to perform a cycle of the five steps a, b, c, d, e, in that order. So $P_i = a_i \cdot b_i \cdot c_i \cdot d_i \cdot e_i \cdot P_i$, for $i = 1, \ldots, 5$; see Figure 8 (left).

Independent execution of these five processes is captured by the process term $P_1 \| P_2 \| P_3 \| P_4 \| P_5$. But we want more than independent execution. Suppose there is a communication regime requiring that steps connected by a grey bar must be performed simultaneously. So b_1 forms a pair with e_2, and c_2 with a_3, and d_3 with b_4, and e_4 with c_5, and a_5 with d_1. We write $b_1|e_2$ for the simultaneous action b_1 with e_2; likewise $c_2|a_3$, $d_3|b_4$, ... The total process T can still execute without deadlock, cyclically, in many ways, of which one infinite cyclic execution path is given by:

$$p = a_1 \cdot (b_1|e_2) \cdot a_4 \cdot a_2 \cdot b_2 \cdot c_1 \cdot (c_2|a_3) \cdot d_2 \cdot b_3 \cdot c_3 \cdot (d_3|b_4) \cdot e_3 \cdot d_3 \cdot (e_4|c_5) \cdot a_4 \cdot d_5 \cdot e_5 \cdot (a_5|d_1) \cdot p.$$

We now modify two communication links, as shown in Figure 8 (right). Then the process will halt after some steps: deadlock! One can still easily see by mental inspection that this must be the case. But again our goal is to turn these verbal and intuitive descriptions into algebraic equations.

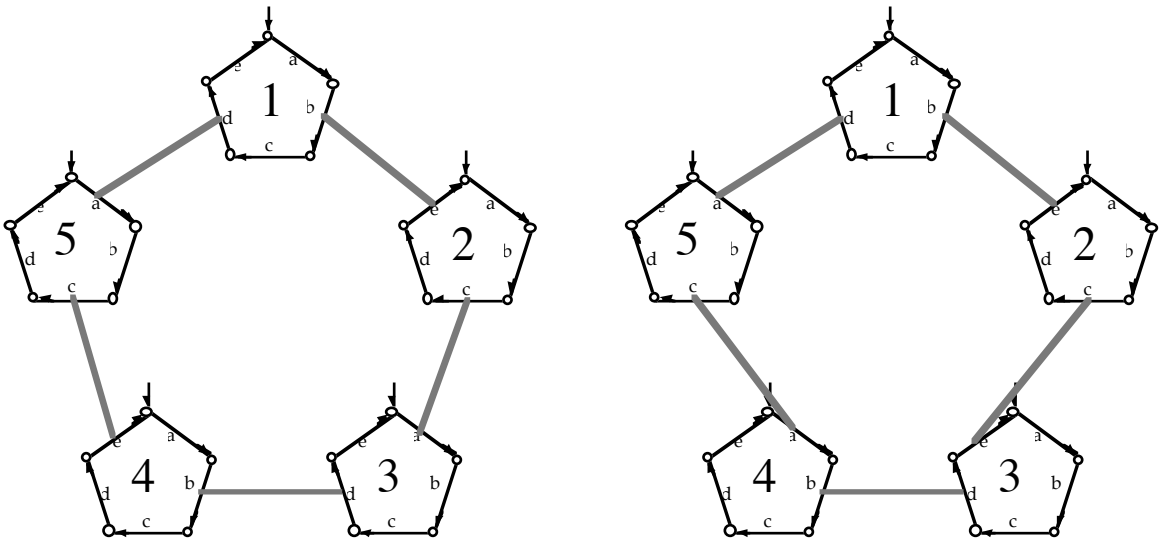

Fig. 8. Ring of communicating processes without deadlock, and with deadlock.

The system in Figure 8 (right) has a state space as depicted in Figure 9, which was generated by computer. In fact this process graph corresponds to the process term $(a_1||a_2 \cdot b_2||a_3 \cdot b_3 \cdot c_3) \cdot \delta$. This yields 2 x 3 x 4 = 24 nodes (states), and

all maximal paths (traces) are six steps long. In this case we have a small state space, with steps that are separately visible; in realistic cases with (hundreds of) millions of steps, the network of steps is so fine that the steps are not separately visible or only after zooming in; then one has to resort to graphic visualisations such as in Figure 6 (right).

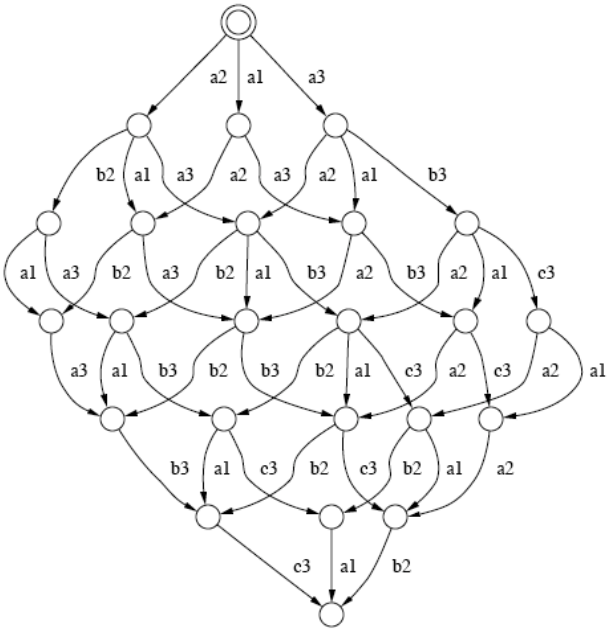

Fig. 9. Computer-generated analysis of deadlocking ring of processes.

Formally deadlock is represented by the constant symbol δ, a special 'step' or rather a 'non-step', the impossibility of a step. For δ we have the two laws in Table 3; note that we do not have $x{\cdot}\delta = \delta$. For, a process terminating with δ is in fact displaying deadlock, like the example above. Only in a choice the δ disappears.

$x + \delta = x$
$\delta{\cdot}x = \delta$

Table 3. Axioms for the deadlock δ.

Communication is modeled in process algebra by means of actions that have to be simultaneously executed, as in the example of the ring of five cyclic processes before.

$$
\begin{aligned}
x\|y &= (x\between y + y\between x) + x|y \\
a\between y &= a\cdot y \\
(a\cdot x)\between y &= a\cdot(x\|y) \\
(x+y)\between z &= x\between z + y\between z \\
a|b &= \gamma(a,b) \\
a|(b\cdot y) &= \gamma(a,b)\cdot y \\
(a\cdot x)|b &= \gamma(a,b)\cdot x \\
(a\cdot x)|(b\cdot y) &= \gamma(a,b)\cdot(x\|y) \\
(x+y)|z &= x|z + y|z \\
x|(y+z) &= x|y + x|z \\
a \notin H \qquad \partial_H(a) &= a \\
a \in H \qquad \partial_H(a) &= \delta \\
\partial_H(x+y) &= \partial_H(x) + \partial_H(y) \\
\partial_H(x\cdot y) &= \partial_H(x)\cdot\partial_H(y)
\end{aligned}
$$

Table 4. ACP, Algebra of Communicating Processes.

Table 4 contains all axioms necessary to describe communication and *encapsulation* of processes. γ is the communication function stipulating which atoms should communicate and with which result. The encapsulation operator ∂_H removes from its argument the remainders of failed communications, by renaming them into δ; H is the set of atoms involved in a communication (the 'half actions').

The axiom system in Tables 1, 2, 3 and 4 is called *ACP*, Algebra of Communicating Processes [1]. At this point, we can make precise, why in setting up the system initially, we discarded the distributivity law $z\cdot(x+y) = z\cdot x + z\cdot y$. Suppose we have communications $a|a = \hat{a}$ and $b|b = \hat{b}$. The set of half actions

is $H = \{a, b, c\}$. Now we compute, using the axioms of ACP, on the one hand, $\partial_H(a{\cdot}(b+c)\|a{\cdot}b) = \hat{a}{\cdot}\hat{b}$, and on the other hand, $\partial_H((a{\cdot}b + a{\cdot}c)||a{\cdot}b] = \hat{a}{\cdot}\hat{b} + \hat{a}{\cdot}\delta$. The last process does have a deadlock possibility, as the summand $\hat{a}{\cdot}\delta$ witnesses. So if we had adopted $z{\cdot}(x+y) = z{\cdot}x + z{\cdot}y$ as one of our axioms, then ACP would not be able to detect deadlock behaviour, and lose most of its value.

7 Abstraction

Now we have almost the capacity to calculate with communicating processes in a very precise way, and also to verify their behaviour in an automated fashion. But there is one vital element still missing: the possibility of *abstraction*. The discovery of Robin Milner [4] was to facilitate this by the introduction of the silent action τ, signifying an invisible state transition in a process, a step without observable effect. It is also referred to as a 'silent move', or 'hidden move'. The τ-step is as before subjected to axioms, namely the two τ-laws displayed in Table 5. Their effect is that τ can be eliminated, contracted as it were, provided the process does not lose options after the τ-step. Figure 10 (left) demonstrates such a situation; the τ-step there connects two states having the same 'potential'. In Figure 10 (right) there is after the τ-step a difference in potential, since the a-option is lost; and therefore the latter τ cannot be eliminated.

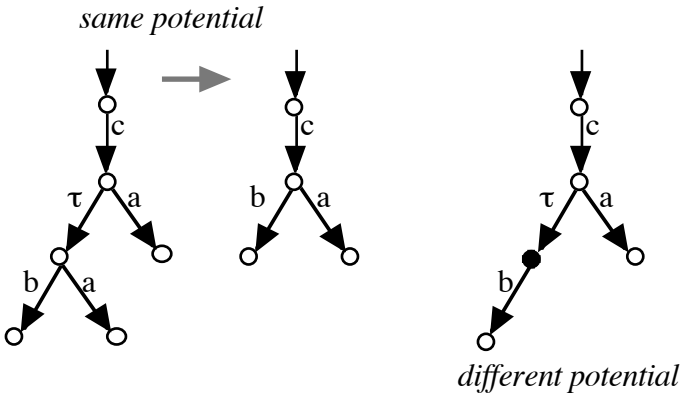

Fig. 10. Examples of the silent step

The *hiding* operator τ_I, for action sets I, renames in its argument all actions from I into τ. Its axioms are given in Table 6. The hiding operator makes it possible to abstract away from the internal communication actions of a system.

In the presence of the silent move, we consider process graphs modulo *branching bisimulation*. That is, let processes p and q be related and $p \xrightarrow{a} p'$. The no-

B1	$x \cdot \tau = x$
B2	$x \cdot (y + \tau \cdot (y + z)) = x \cdot (y + z)$

Table 5. Axioms for the silent step τ.

TI1	$a \notin I$	$\tau_I(a) = a$
TI2	$a \in I$	$\tau_I(a) = \tau$
TI3		$\tau_I(\delta) = \delta$
TI4		$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$
TI5		$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$

Table 6. Axioms for the hiding operator

tion of bisimulation from Section 5 would impose that $q \xrightarrow{a} q'$ for some process q' where p' and q' are related. In branching bisimulation, it is allowed that q first performs some τ-transitions: $q \xrightarrow{\tau} q_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_k \xrightarrow{a} q'$, where p must be related to q_k. Moreover, if $a = \tau$, then branching bisimulation allows that p' is itself related to q, in which case q does not need to mimic the τ-transition of p at all. The motivation for branching bisimulation is that, like bisimulation, it leaves the branching structure of processes intact, so that for instance deadlock behaviour is preserved. A rootedness condition on top of branching bisimulation, saying that initial τ-transitions must always be mimicked instantly, produces an equivalence that is preserved by the process algebraic operators. For instance, if p_i is rooted branching bisimulation equivalent to q_i for $i = 1, 2$, then $p_1 + p_2$ is rooted branching bisimulation equivalent to $q_1 + q_2$. Such a *congruence* property is crucial to capture the equivalence in an axiomatic framework.

A linear recursive specification is called *guarded* if it does not give rise to τ-cycles. A typical example of an unguarded linear recursive specification is $X = \tau \cdot X$. Note that in the setting of rooted branching bisimulation, this recursive specification has multiple solutions; one could e.g. substitute $\tau \cdot a$ and $\tau \cdot b$ for X.

RDP and *RSP* are axiom schemes for recursion. RDP expresses that given a linear recursive specification E, the process terms $\langle X_1 | E \rangle, \ldots, \langle X_n | E \rangle$ form a solution for E. RSP expresses that if E is guarded, then this is the only solution for E, modulo rooted branching bisimulation.

8 Cluster fair abstraction

Central in process algebra are so-called *soundness* and *completeness* results, for a given axiom system with respect to a given process semantics. An axiom system is *sound* if for each equation $p = q$ that can be derived from it, p and q are semantically equivalent. Vice versa, an axiom system is *complete* if for each pair p, q of semantically equivalent terms, the equation $p = q$ can be derived. Here we give a soundness and completeness proof for the process algebra and axiom system presented in the previous sections, with respect to rooted branching bisimulation (see also [2]).

We first present one more axiom scheme, to eliminate a cluster of τ-transitions, so that only the exits of such a cluster remains.

Let E be a guarded linear recursive specification, and $I \subseteq A$. Two recursion variables X and Y in E are in the same *cluster* for I if and only if there exist sequences of transitions $\langle X|E\rangle \xrightarrow{b_1} \cdots \xrightarrow{b_m} \langle Y|E\rangle$ and $\langle Y|E\rangle \xrightarrow{c_1} \cdots \xrightarrow{c_n} \langle X|E\rangle$ with $b_1, \ldots, b_m, c_1, \ldots, c_n \in I \cup \{\tau\}$. a or aX is an *exit* for the cluster C if:

1. a or aX is a summand at the right-hand side of the recursion equation for a recursion variable in C; and
2. in the case of aX, either $a \notin I \cup \{\tau\}$ or $X \notin C$.

Table 7 presents an axiom scheme called *cluster fair abstraction rule* (CFAR) [6] for guarded linear recursive specifications. CFAR allows one to abstract away from a cluster of actions that are renamed into τ, after which only the exits of this cluster remain. In Table 7, E is a guarded linear recursive specification.

CFAR If X is in a cluster for I with exits $\{a_1{\cdot}Y_1, \ldots, a_m{\cdot}Y_m, b_1, \ldots, b_n\}$, then
$\tau{\cdot}\tau_I(\langle X\|E\rangle) \;=\; \tau{\cdot}\tau_I(a_1{\cdot}\langle Y_1\|E\rangle + \cdots + a_m{\cdot}\langle Y_m\|E\rangle + b_1 + \cdots + b_n)$

Table 7. Cluster fair abstraction rule

Theorem 1. *The axiom CFAR is sound modulo rooted branching bisimulation equivalence.*

Proof. Let X be in a cluster for I with exits $\{a_1{\cdot}Y_1, \ldots, a_m{\cdot}Y_m, b_1, \ldots, b_n\}$. Then $\langle X|E\rangle$ can execute a string of actions from $I \cup \{\tau\}$ inside the cluster of X, followed by an exit $a_i{\cdot}\langle Y_i|E\rangle$ (for some $i \in \{1, \ldots, m\}$) or b_j (for some $j \in \{1, \ldots, n\}$). Hence, $\tau_I(\langle X|E\rangle)$ can execute a string of τ's inside the cluster of X, followed by an exit $\tau_I(a_i\langle Y_i|E\rangle)$ (for some $i \in \{1, \ldots, m\}$) or $\tau_I(b_j)$ (for some $j \in \{1, \ldots, n\}$). The execution of τ's inside the cluster does not lose the possibility to execute

any of the exits. Moreover, in the process graph of $\tau{\cdot}\tau_I(\langle X|E\rangle)$ these τ's are non-initial, owing to the initial τ-transition, so they are truly silent. This means that modulo rooted branching bisimulation equivalence only the exits of the cluster of X remain, i.e., $\tau{\cdot}\tau_I(\langle X|E\rangle)$ is rooted branching bisimulation equivalent to

$$\tau{\cdot}\tau_I(a_1{\cdot}\langle Y_1|E\rangle + \cdots + a_m{\cdot}\langle Y_m|E\rangle + b_1 + \cdots + b_n).$$

So CFAR is sound modulo rooted branching bisimulation equivalence. □

For example, let E denote the guarded linear recursive specification

$$X \;=\; \mathit{heads}{\cdot}X + \mathit{tails}.$$

The process term $\langle X|E\rangle$ represents tossing a fair coin until the result is tails. We abstract away from throwing heads, expressed by $\tau_{\{\mathit{heads}\}}(\langle X|E\rangle)$.

$\{X\}$ is the only cluster for $\{\mathit{heads}\}$, and the only exit of this cluster is the action *tails*. So

$$\tau{\cdot}\tau_{\{\mathit{heads}\}}(\langle X|E\rangle) \overset{\text{CFAR}}{=} \tau{\cdot}\tau_{\{\mathit{heads}\}}(\mathit{tails}) \overset{\text{TI1}}{=} \tau{\cdot}\mathit{tails}. \tag{1}$$

Hence,

$$\begin{aligned}\tau_{\{\mathit{heads}\}}(\langle X|E\rangle) &\overset{\text{RDP}}{=} \tau_{\{\mathit{heads}\}}(\mathit{heads}{\cdot}\langle X|E\rangle + \mathit{tails})\\ &\overset{\text{TI1,2,4,5}}{=} \tau{\cdot}\tau_{\{\mathit{heads}\}}(\langle X|E\rangle) + \mathit{tails}\\ &\overset{(1)}{=} \tau{\cdot}\mathit{tails} + \mathit{tails}.\end{aligned}$$

In other words, fair abstraction implies that tossing a fair coin infinitely many times will eventually produce the result tails.

Theorem 2. $\mathcal{E}_{\mathrm{ACP}_\tau}$ + RDP, RSP, CFAR *is complete for* ACP_τ *with guarded linear recursion, modulo rooted branching bisimulation equivalence.*

Proof sketch. It suffices to prove that each process term t in ACP_τ with guarded linear recursion is provably equal to a process term $\langle X|E\rangle$ with E a guarded linear recursive specification. Namely, then the desired completeness result follows immediately from the known fact that if $\langle X_1|E_1\rangle$ is rooted branching bisimulation equivalent to $\langle Y_1|E_2\rangle$ for guarded linear recursive specifications E_1 and E_2, then $\langle X_1|E_1\rangle = \langle Y_1|E_2\rangle$ can be derived from $\mathcal{E}_{\mathrm{ACP}} + \mathrm{B1}, 2 + \mathrm{RDP}, \mathrm{RSP}$.

We apply structural induction with respect to the size of t. It can be shown that each process term in ACP with silent step and guarded linear recursion is provably equal to a process term $\langle X|E\rangle$ with E a guarded linear recursive specification. So the only case that remains to be covered is when $t \equiv \tau_I(s)$. By induction it may be assumed that $s = \langle X|E\rangle$ with E a guarded linear recursive specification, so $t = \tau_I(\langle X|E\rangle)$. We divide the collection of recursion variables in E into its clusters $C_1, \ldots, C_N$ for I. For $i \in \{1, \ldots, N\}$, let

$$a_{i1}{\cdot}Y_{i1} + \cdots + a_{im_i}{\cdot}Y_{im_i} + b_{i1} + \cdots + b_{in_i}$$

be the alternative composition of exits for the cluster C_i. Furthermore, for actions $a \in A \cup \{\tau\}$ we define

$$\hat{a} = \begin{cases} \tau \text{ if } a \in I \\ a \text{ otherwise.} \end{cases}$$

Finally, for $Z \in C_i$ ($i \in \{1, \ldots, N\}$) we define

$$s_Z \;=\; \hat{a}_{i1}{\cdot}\tau_I(\langle Y_{i1}|E\rangle) + \cdots + \hat{a}_{im_i}{\cdot}\tau_I(\langle Y_{im_i}|E\rangle) + \hat{b}_{i1} + \cdots + \hat{b}_{in_i}. \tag{2}$$

For $Z \in C_i$ and $a \in A \cup \{\tau\}$,

$$\begin{aligned} a{\cdot}\tau_I(\langle Z|E\rangle) &\overset{\text{CFAR}}{=} a{\cdot}\tau_I(a_{i1}{\cdot}\langle Y_{i1}|E\rangle + \cdots + a_{im_i}{\cdot}\langle Y_{im_i}|E\rangle + b_{i1} + \cdots + b_{in_i}) \\ &\overset{\text{TI1-5}}{=} a{\cdot}s_Z. \end{aligned} \tag{3}$$

Let the linear recursive specification F contain the same recursion variables as E, where for each $Z \in C_i$ the recursion equation in F is

$$Z \;=\; \hat{a}_{i1}{\cdot}Y_{i1} + \cdots + \hat{a}_{im_i}{\cdot}Y_{im_i} + \hat{b}_{i1} + \cdots + \hat{b}_{in_i}.$$

We show that there is no sequence of one or more τ-transitions from $\langle Z|F\rangle$ to itself. Suppose $\hat{a}_{ij} \equiv \tau$ for some $j \in \{1, \ldots, m_i\}$. Then the fact that $a_{ij}{\cdot}Y_{ij}$ is an exit for the cluster C_i ensures that $Y_{ij} \notin C_i$, so there cannot exist a sequence of transitions $\langle Y_{ij}|E\rangle \xrightarrow{d_1} \cdots \xrightarrow{d_\ell} \langle Z|E\rangle$ with $d_1, \ldots, d_\ell \in I \cup \{\tau\}$. Then by the definition of F there cannot exist a sequence of transitions $\langle Y_{ij}|F\rangle \xrightarrow{\tau} \cdots \xrightarrow{\tau} \langle Z|F\rangle$. Hence, F is guarded.

For each recursion variable $Z \in C_i$ ($i \in \{1, \ldots, N\}$),

$$s_Z \overset{(2),(3)}{=} \hat{a}_{i1}{\cdot}s_{Y_{i1}} + \cdots + \hat{a}_{im_i}{\cdot}s_{Y_{im_i}} + \hat{b}_{i1} + \cdots + \hat{b}_{in_i}.$$

This means that substituting s_Z for recursion variables Z in F is a solution for F. Hence, by RSP, $s_Z = \langle Z|F\rangle$ for recursion variables Z in F. So for $a \in A \cup \{\tau\}$ and recursion variables Z in F,

$$a{\cdot}\tau_I(\langle Z|E\rangle) \overset{(3)}{=} a{\cdot}s_Z = a{\cdot}\langle Z|F\rangle. \tag{4}$$

Recall that $t = \tau_I(\langle X|E\rangle)$. Let the linear recursion equation for X in E be

$$X \;=\; c_1{\cdot}Z_1 + \cdots + c_k{\cdot}Z_k + d_1 + \cdots + d_\ell.$$

Let the linear recursive specification G consist of F extended with a fresh recursion variable W and the recursion equation

$$W \;=\; \hat{c}_1{\cdot}Z_1 + \cdots + \hat{c}_k{\cdot}Z_k + \hat{d}_1 + \cdots + \hat{d}_\ell.$$

Since F is guarded, it is clear that G is also guarded.

$$\begin{aligned} \tau_I(\langle X|E\rangle) &\overset{\text{RDP}}{=} \tau_I(c_1{\cdot}\langle Z_1|E\rangle + \cdots + c_k{\cdot}\langle Z_k|E\rangle + d_1 + \cdots + d_\ell) \\ &\overset{\text{TI1-5}}{=} \hat{c}_1{\cdot}\tau_I(\langle Z_1|E\rangle) + \cdots + \hat{c}_k{\cdot}\tau_I(\langle Z_k|E\rangle) + \hat{d}_1 + \cdots + \hat{d}_\ell \\ &\overset{(4)}{=} \hat{c}_1{\cdot}\langle Z_1|F\rangle + \cdots + \hat{c}_k{\cdot}\langle Z_k|F\rangle + \hat{d}_1 + \cdots + \hat{d}_\ell. \end{aligned}$$

Furthermore, for $Z \in C_i$ $(i \in \{1, \ldots, N\})$,

$$\langle Z|F\rangle \stackrel{\text{RDP}}{=} \hat{a}_{i1}{\cdot}\langle Y_{i1}|F\rangle + \cdots + \hat{a}_{im_i}{\cdot}\langle Y_{im_i}|F\rangle + \hat{b}_{i1} + \cdots + \hat{b}_{in_i}.$$

Hence, substituting $\tau_I(\langle X|E\rangle)$ for W and $\langle Z|F\rangle$ for all other recursion variables Z in G is a solution for G. So RSP yields

$$\tau_I(\langle X|E\rangle) \;=\; \langle W|G\rangle. \quad \Box$$

Acknowledgements We thank Prof.Dr. Jan-Friso Groote for assistance with the automatic verification of some processes, in particular those in Figures 6 and 9. For a beautiful gallery of similar visualisations of large state spaces, see his homepage `www.win.tue.nl/~jfg`, and the one of Dr. Frank van Ham: `www.win.tue.nl/~fvham/fsm/`.

References

1. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.
2. W.J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science, An EATCS Series. Springer, 2000.
3. J.F. Groote and F.J.J. van Ham. Interactive visualization of large state spaces. *International Journal on Software Tools for Technology Transfer*, 8:77–91, 2006.
4. R. Milner. *A Calculus of Communicating Systems*. Volume 92 of Lecture Notes in Computer Science. Springer, 1980.
5. D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, ed., *Proceedings 5th GI (Gesellschaft für Informatik) Conference*, Karlsruhe, Volume 104 of Lecture Notes in Computer Science, pp. 167–183. Springer, 1981.
6. F.W. Vaandrager. Verification of two communication protocols by means of process algebra. Report CS-R8608, CWI, Amsterdam, 1986.