The state operator in process algebra

The State Operator in Process Algebra

Javier Oscar Blanco

The State Operator in Process Algebra

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universtiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op dinsdag 30 januari 1996 om 16:00 uur

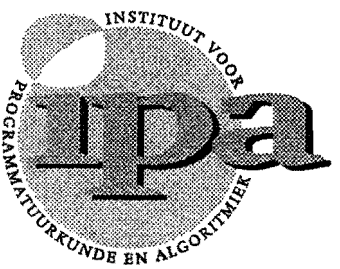
door

Javier Oscar Blanco

geboren te Miramar, Argentina

Dit proefschrift is goedgekeurd
door de promotoren
prof.dr. J.C.M. Baeten
prof.dr. J.A. Bergstra

This work has been supported by the Netherlands Organization for Scientific Research (NWO), project NFI-78.



The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

druk: Universitaire Drukkerij
TU Eindhoven

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Blanco, Javier Oscar
The state operator in process algebra / Javier Oscar Blanco. -
Eindhoven: Eindhoven University of Technology
Thesis Technische Universiteit Eindhoven. -
With index, ref. - With summary in Dutch
ISBN 90-386-0287-1
Subject headings: process algebra.

Acknowledgements

This thesis would have not been started without the interest of prof. Jos Baeten, and it would not be finished without his help and infinite patience. During the last years, he has been constantly available to discuss any of my ideas, even when they were vague and incomplete. Also, he read carefully any manuscript that I gave him having always interesting remarks. constant criticism and careful readings of everything I wrote.

Prof. Jan Bergstra gave me many ideas during my stay at the University of Amsterdam. Pedro D'Argenio has been the person with whom I worked out some ideas of this thesis. He help me to recover my interest during difficult times. Many improvements on the thesis were achieved from the accurate corrections of the readers: prof. Loe Feijs and prof. Kees Middelburg. I want to thank as well the leaders of the NFI project TRANSFER, prof. Baeten, prof. Bergstra and prof. Ollongren. Part of this thesis was written at the University of Amsterdam and the CWI thanks to the hospitality of prof. J. Bergstra and prof. J de Bakker. Special thanks to Jan Joris Vereijken for making the Dutch summary.

The Eindhoven University of Technology has been a very pleasant enviorenment to work. I thank also NFI and NWO for the financial support.

I am most indebted to Susan Doniz for her constant encouragement and love. She was always present in the hardest moments. Moreover, she read all the many manuscripts of the thesis.

During hard times I received constant encouragement from my friends, especially Almudena, Paula and Walter.

Finally I want to thank all the people whose presence during the years of writing my thesis made my life more bearable: Adriana, Alessandro, Alfredo, Aljosha, Amit, Cecilia, Cristina A, Cristina P, Dany, Elizabeth, Felpe, Gabriel, Helene, Ineke, Irene, Josefa, Leticia, Luis, Maria Jose, Marianne, Mariel, Martín, Martina, Mimma, Mirta, Mónica, Noel, Nordin, Osvaldo, Peter, Tony and Vivian.

Contents

Chapter 1

Introduction

Process algebra is the study of concurrent processes in an algebraic framework. The main algebraic technique used in process algebra is the axiomatic method, which consists of finding a set of axioms that will describe the behaviour of processes and their laws of composition. A process can be seen mainly as the behaviour of a system, where a system can be, for example, a computer system, an elementary particle, a vending machine, or a satellite communicating with the earth.

Process algebra appeared as an answer to the many problems that arose in the search for formal semantics of languages involving primitives for concurrency. One of these problems was the insufficiency of the input-output semantics, which was very successful in giving semantics to sequential languages. The growth in complexity of the problems concerning concurrent languages, led to the isolation of some basic notion of process and elementary operations on these processes. Nevertheless, as the theories grew and they were used in applications, the many features already studied in the field of sequential languages had to be reconsidered in this framework.

One application of process algebra that deserves attention is the idea of atomicity. We approach this problem using the idea that an atomic action will have some effect on its environment, as well as possibly being affected by this environment. Atomicity will mean in this context the property of this effect of being performed without interference from other components. This seems to agree with the notion of atomicity used in the field of distributed data bases.

This thesis studies some of these features in the framework of process algebra.

Chapter 2 introduces the basic notions of process algebra and the state operator, which

are fundamental in this work. The state operator used here is a generalization of the one presented in [BB88]. The main difference is that we have a more symmetrical view of the state operator, and we are not only interested in the process modified by a state but also in the set of states produced or modified by a process. This will allow us to deal with the input-output behaviour of a process, which is necessary in other chapters of this thesis. Some new non-equational principles are introduced as well as non-standard models that exemplify such principles.

Chapter 3 presents the idea of non-elementary atomic actions in process algebra. This concept has been the subject of much discussion and the source of many different models. The concept of atomicity is essential in interleaving theories of concurrency, and many models rest upon the fact that it is a primitive concept. We depart slightly from the interleaving theories in order to introduce a mechanism to prescribe that a process must be executed in an atomic way. The equality of atomic actions introduced will represent the fact that two atomic actions will act identically in any state. The concept of atomic action considered in this chapter combines ideas from [Bou89] and [BK84b] but differs from the first in the use of branching time semantics against the input-output semantics of [Bou89], and instead of syncronization as in [BK84b] it is based on multiactions. Furthermore, the concept of recoverability of an atomic action (i.e. if it does not terminates successfully, then the state of the system should be the same as it hs never been performed) is implemented using the idea that unsuccessful termination is a zero object ([BB90]) inside an atomic action. This improves over [Bou89] since it can distinguish between deadlock and livelock inside an atomic action.

Chapter 4 is a study concerning the combination of data and processes. In the literature, many different approaches were used to integrate the theories of data types and processes. Even when both theories are restricted to algebraic theories, some different combinations were used. For example in the thesis [Pon92] the data types were used as indexes for recursion equations, whereas in [AMR88] the processes were consider as a particular data type. Here, we demonstrate how certain data types can be seen as processes, in a very natural way. Thus, we can reduce the interaction between process and data to interaction between processes. Furthermore, we present a new solution for the use of data types, and the implementation of one data type by another, in a concurrent environment.

In chapter 5 the state operator is restricted to have a finite set of states in order to study whether the addition of the state operator can increase the set of processes definable by a guarded recursive specification.

Some results concerning the decidability of bisimulation in some of the classes defined

in chapter 5 are introduced in chapter 6.

Chapter 2

Preliminaries

2.1 Process Algebra

2.1.1 Introduction

In this section we present a brief description of process algebra. We refer the reader to [BW90, BVng] for further information.

Several systems will be used. The largest signature considered is the one of ACP (Algebra of Communicating Processes) with projections and renamings, to be presented in the next section, and the state operator (see section 2.6). The signature of ACP has:

constants a finite set A of atomic actions and a special constant δ indicating a deadlocked process.

unary operators given $H \subset A$ the encapsulation operator ∂_H.

binary operators $+, \cdot, \|, \lfloor\!\lfloor, |$. $+$ represents alternative composition, $\cdot$ sequential composition, and $\|$ parallel composition (merge). The auxiliary operators $\lfloor\!\lfloor$ (left merge) and $|$ (communication merge) are used to define the merge.

The sequential composition has the highest precedence followed by the merge and the auxiliary operators (left and communication merge); the alternative composition has the lowest precedence

2.1.2 Basic Process Algebra

The theory BPA has a restricted signature with only A, $+$ and $\cdot$. The axioms are given in Table 2.1.

One important characteristic of this set of axioms is the absence of the distributive law symmetric to A4. This means that the moment of choice is a distinctive characteristic of processes, and not only the set of possible traces.

A1	$x + y = y + x$
A2	$x + (y + z) = (x + y) + z$
A3	$x + x = x$
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$

Table 2.1: Axioms of BPA

The constant δ, which represents a process that cannot proceed, can be added through the axioms in Table 2.2. This constant is called *inaction*. In some works the constant δ is called *deadlock* despite the fact that Axiom A6 states that it can be avoided if the process can do something else.

The theory of BPA with the addition of the constant δ will be called BPA_δ.

A6	$x + \delta = x$
A7	$\delta \cdot x = \delta$

Table 2.2: Axioms for δ

2.1.3 Process Algebra

The signature of the theory PA_δ contains $\|$ and $\lfloor\!\lfloor$ besides the elements of the signature of BPA_δ. The $\|$ represents the free merge. The additional axioms are presented in table 2.3 (a ranges over $A \cup \delta$).

2.1.4 Algebra of Communicating Processes

The theory called ACP is presented. The theory is parametrized by a communication function γ which indicates which atomic actions communicate. This function is assumed to be commutative and associative, and to satisfies the equation $\gamma(\delta, a) = \delta$ for all $a \in A$. The axioms of table 2.1 and 2.2 should be extended with the axioms of table 2.4 ($a, b \in A \cup \{\delta\}$).

M1	$x \parallel y = x \between y + y \between x$
M2	$a \between x = a \cdot x$
M3	$a \cdot x \between y = a \cdot (x \parallel y)$
M4	$(x + y) \between z = x \between z + y \between z$

Table 2.3: Additional axioms of PA_δ

CM1	$x \parallel y = x \between y + y \between x + x \mid y$	
CM2	$a \between x = a \cdot x$	
CM3	$a \cdot x \between y = a \cdot (x \parallel y)$	
CM4	$(x + y) \between z = x \between z + y \between z$	
CF1	$a \mid b = \gamma(a, b)$	
CM5	$a \cdot x \mid b = (a \mid b) \cdot x$	
CM6	$a \mid b \cdot x = (a \mid b) \cdot x$	
CM7	$a \cdot x \mid b \cdot y = (a \mid b) \cdot (x \parallel y)$	
CM8	$(x + y) \mid z = x \mid z + y \mid z$	
CM9	$x \mid (y + z) = x \mid y + x \mid z$	
D1	$\partial_H(a) = a$	if $a \notin H$
D2	$\partial_H(a) = \delta$	if $a \in H$
D3	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	
D4	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	

Table 2.4: Additional axioms of ACP

2.1.5 Renamings

A feature that can be added to the previous algebras is the possibility of renaming atomic actions, given a function $f : A \setminus \{\delta\} \to A$. The operator ρ_f is defined in table 2.5 ($a \in A \cup \{\delta\}$).

RN0	$\rho_f(\delta) = \delta$	
RN1	$\rho_f(a) = f(a)$	if $a \neq \delta$
RN2	$\rho_f(x \cdot y) = \rho_f(x) \cdot \rho_f(y)$	
RN3	$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$	

Table 2.5: Renamings

2.1.6 Projections

Any of the signatures defined above can be extended by an infinite set of unary operators π_n with n a natural number greater than or equal to 1. The intended meaning of $\pi_n(P)$ (in some appropriate model) is the process that behaves as P but stops after executing n steps. The axioms for the projection operators are given below, in table 2.6 ($a \in A$ or $a \in A \cup \{\delta\}$ in case of a theory with δ).

PR1	$\pi_n(a) = a$
PR2	$\pi_1(a \cdot x) = a$
PR3	$\pi_{n+1}(a \cdot x) = a \cdot \pi_n(x)$
PR4	$\pi_n(x + y) = \pi_n(x) + \pi_n(y)$

Table 2.6: Projections

In [BW90] a different version of projection was also introduced. The n-th projection of a process stops after n steps but leaves an unsuccessful termination (δ) if the process has not already finished. Furthermore, this projection can be defined for any $n \geq 0$. The axioms are given in table 2.7.

2.1.7 Recursive definitions

Sometimes, the processes we will consider are defined by using a set of recursive equations, i.e., processes that are solutions of such a set in a suitable model.

Definition 2.1.7.1.

PD1	$\pi_0^\delta(x) = \delta$
PD2	$\pi_{n+1}^\delta(a) = a$
PD3	$\pi_{n+1}^\delta(a \cdot x) = a \cdot \pi_n^\delta(x)$
PD4	$\pi_n^\delta(x + y) = \pi_n^\delta(x) + \pi_n^\delta(y)$

Table 2.7: Axioms of projections with unsuccessful termination

1. A system of *recursion equations* (over a process theory, say BPA) is a finite set of the form:
$$E = \{X_i = s_i(X_0, \ldots, X_n); i = 0, \ldots, n\}$$
where the $s_i(X_0, \ldots, X_n)$ are expressions in the required signature, and the variables of s_i are among $X_0, \ldots, X_n$.

2. A countably infinite system is defined similarly, but now i ranges over the set of natural numbers:
$$E = \{X_i = s_i(\vec{X}); i \in \mathbf{N}\}$$

3. A *solution* (in a certain model) of a recursive specification is a set of processes, one for each variable, such that the equations become true statements when the variables are interpreted as the corresponding processes. We sometimes use the word solution for the process in that set which corresponds to the first variable of the specification.

4. A specification is *guarded* if every occurrence of a variable in the right hand side of an equation is, modulo the axioms of the theory, in a term of the form $a \cdot s$, for some atom a. Guardedness is a sufficient condition to guarantee uniqueness of solutions in many models.

5. We say that a process p has *head normal form* if there are n and m, natural numbers, atomic actions a_i $(i < n)$, b_j $(j < m)$ and processes p_i $(i < n)$ such that
$$p = \sum_{i<n} a_i p_i + \sum_{j<m} b_j$$
We take as a convention that $\sum_{i<0} p_i = \delta$.

6. A process is *definable* if it is a solution of a guarded recursive specification.

□

Note that for each model $\mathcal{M}$, we can consider the submodel of all definable processes, since definable processes are closed under the operations of ACP (see [BW90]).

Lemma 2.1.7.2. *Let p be a definable process. Then*

1. *p has a head normal form, i.e. we can write p as*

$$\sum_{i<n} a_i \cdot q_i + \sum_{j<m} b_j$$

and, moreover, all q_i are definable.

2. *For every n, $\pi_n(p)$ equals a closed term.*

Proof. See [BW90] □

2.1.8 Properties of process algebras

In this section we state the definitions and properties of process algebra that will be needed. A more extensive study and the proofs of these results can be found in [BW90].

Definition 2.1.8.3 (*Basic terms*). The set B of basic terms of BPA_δ is defined inductively as follows:

1. $A \subseteq B$;
2. $\delta \in B$;
3. $a \in A$ and $t \in B$ implies $a \cdot t \in B$;
4. $t, s \in B$ implies $t + s \in B$.

□

Theorem 2.1.8.4 (*Elimination*). *Given any closed term t in BPA_δ, PA_δ, or ACP, possibly with projections and renamings, there exists a basic term t' such that (in the corresponding theory) $t = t'$.*

Definition 2.1.8.5. A partial order between processes can be defined as follows:

$$p \leq q \text{ iff } q = q + p$$

which is equivalent (see [BW90]) to

$$p \leq q \text{ iff } \exists z, q = p + z$$

□

Definition 2.1.8.6 (*Alphabet*). The alphabet of a process is the subset of the set of atomic actions consisting of the actions that a process may perform. Given a process x we write $\alpha(x)$ to denote this set. The axioms in table 2.8 are taken from [BW90]($a \in A$). When definable processes are considered the axiom in table 2.9 can be used.

□

AB1	$\alpha(\delta) = \emptyset$
AB2	$\alpha(a) = \{a\}$
AB3	$\alpha(ax) = \{a\} \cup \alpha(x)$
AB4	$\alpha(x + y) = \alpha(x) \cup \alpha(y)$

Table 2.8: Alphabet

AB5	$\alpha(ax) = \bigcup_{i \in N} \alpha(\pi_i(x))$

Table 2.9: Alphabet for definable processes

2.2 Models

2.2.1 Properties of models

Some models have peculiar characteristics. One of the aims of process algebras is to develop a theory that can be used in different models of concurrency. The technique used to achieve this goal is to establish certain properties that are useful for a theory of concurrency, and then show that a number of well-know models satisfy them. Given this fact the results can be obtained modulo this property and there is no need to refer to a specific model.

A very important property that one can ask is completeness of an axiomatisation with respect to a particular model, i.e. that if the interpretation of two terms is equal in the model, then they can be proved equal using the axioms of the theory and equational reasoning. In case the equational theory of BPA is complete with respect to a model $\mathcal{M}$ we will write it as follows:

$$\mathcal{M} \models \text{COMP}$$

Completeness is a very important property as we will see in some examples, however it involves only closed terms. Models also have elements which are not represented by any closed term but they are, for example, solutions of a system of equations, or even not represented at all. Hence, some principles are introduced which can aid in dealing with models:

Definition 2.2.1.1 (*RDP*). The *Restricted Recursive Definition Principle (RDP$^-$)* says that every guarded specification has a solution. □

Definition 2.2.1.2 (*AIP*). The *Approximation Induction Principle* (AIP) states that

a process is determined by its finite projections, i.e.,

$$\frac{\forall n \geq 1.\pi_n(x) = \pi_n(y)}{x = y}$$

□

Definition 2.2.1.3 (*RSP*). The *Recursive Specification Principle (RSP)* states that every guarded recursive specification has at most one solution. This principle is a consequence of AIP (see [BW90]). □

Another concept that will be useful in the following is that of *finite projections*.

Definition 2.2.1.4. We write $\mathcal{M} \models \text{FINPROJ}$ meaning that all processes in $\mathcal{M}$ have finite projections, i.e., their projections equals a closed term. □

By Lemma 2.1.7.2, we have

Corollary 2.2.1.5. $\mathcal{M} \models \text{DEF}$ *implies that* $\mathcal{M} \models \text{FINPROJ}$.

Definition 2.2.1.6 (*FAP*). Another principle that is used implicitly in many works on process algebra is the *Fresh Atom Principle (FAP)* which says that we can use fresh atomic actions in proofs. This principle was formalized in [BG87] in the following way: Suppose we have a set of atomic actions A and a communication function (if present in the signature) γ. Given an atomic action $a \notin A$ and an extension γ^* of γ to $A \cup \{a\}$, FAP says that any equation $p = q$ over the smaller signature (with parameters A, γ) may also be proved using $A \cup \{\delta\}$ and γ^* as parameters in the proof. □

2.2.2 Standard models

In the following we introduce several models. For more details we refer to [BK84a, BW90].

Example 2.2.2.7 (*The initial algebra*). The initial algebra (say A) is defined as usual: A is the set of equivalence classes of closed BPA terms modulo provability in the equational theory. That is

$$\mathsf{A} \models s = t \iff \text{BPA} \vdash s = t$$

Analogously we can define an initial algebra for BPA+PR which will be ambiguously called A.

Example 2.2.2.8 (*The graph models*). Consider a finite set of *labels*, say $\mathbf{A}$. A graph with labels from the set $\mathbf{A}$ is a structure consisting of a set of *nodes*, whith a distinguished node called *root*, and a set of *edges* labeled with elements of $\mathbf{A}$. In general, we write $s \xrightarrow{a} t$ meaning that there is an edge with label a from the node s to the node t. If g is a graph, $\mathbf{root}(g)$ is the root of g. We call G^∞ the class of all graphs with labels belonging to $\mathbf{A}$.

Definition 2.2.2.9 (*Bisimulation*). Let $g, h \in G^\infty$. A bisimulation is a binary relation R between the nodes of g and h such that, for all $a \in \mathbf{A}$:

1. $\mathbf{root}(g)\ R\ \mathbf{root}(h)$;
2. $sRt \ \wedge\ s \xrightarrow{a} s' \ \Rightarrow\ \exists t'.t \xrightarrow{a} t' \ \wedge\ s'Rt'$;
3. $sRt \ \wedge\ t \xrightarrow{a} t' \ \Rightarrow\ \exists s'.s \xrightarrow{a} s' \ \wedge\ s'Rt'$.

If such a relation exists we say that g and h are bisimilar and we write $g \leftrightarrows h$. □

We take from [BW90] the *root unwinding map*, $\rho : G^\infty \to G^\infty$, which, given a graph, obtains a new one that is bisimilar to the original, and has no incoming edge in the root. The *(total) unwinding map*, $\mathbf{tree} : G^\infty \to G^\infty$, obtains a new graph which is bisimilar to the given one, where the root has no incoming edge, and the other nodes have at most one incoming edge.

We define an interpretation in G^∞ for the BPA+PR operations. From now on, we use superindication of the model's name in order to represent the interpretation of an operator in such a model:

1. For each $a \in \mathbf{A}$, a^{G^∞} is the graph having only two nodes with an edge between them labelled by a. The source node is the root.
2. Given $g, h \in G^\infty$, we define $g +^{G^\infty} h$ by first unwinding the roots of g and h (making $\rho(g)$ and $\rho(h)$), and then identifying the new roots.
3. $g \cdot^{G^\infty} h$ is defined by identifying every node in g having no outgoing edge with the root of h.
4. In order to define $\pi_n^{G^\infty}(g)$, first unwind g to a tree (make $\mathbf{tree}(g)$), then remove all edges leaving from a node at depth n.

Now, we have:

Theorem 2.2.2.10. $G^\infty / \leftrightarrows \ \models$ BPA + PR. *Moreover* BPA+PR *is a complete axiomatization for* $G^\infty / \leftrightarrows$.

In addition, there are other models that are completely axiomatized by BPA+PR, which are obtained by considering subsets of G^∞:

1. the model of finitely branching graphs ($G/\underline{\leftrightarrow}$),
2. the model of finite or regular graphs ($R/\underline{\leftrightarrow}$),
3. the model of finite acyclic graphs ($F/\underline{\leftrightarrow}$).

Example 2.2.2.11 (*The term model*). The set $\mathbf{P}$ of process expressions is defined by the terms in the signature of BPA+PR and a new constant $< X|E >$ for every recursive specification E and any variable X occurring in E. The expression $< t|E >$ denotes the term t with all variables in t replaced by their corresponding constants (in E). Abusing notation we sometimes write X for $< X|E >$ and t for $< t|X >$. We define the behaviour of a process in $\mathbf{P}$ according to the action relations defined by the rules in the Plotkin's style [Plo81] given in Table 2.10.

$$a \xrightarrow{a} \surd$$

$$\frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p' \quad q+p \xrightarrow{a} p' \quad pq \xrightarrow{a} p'q \quad \pi_{n+1}(p) \xrightarrow{a} \pi_n(p) \quad \pi_1(p) \xrightarrow{a} \surd}$$

$$\frac{p \xrightarrow{a} \surd}{p+q \xrightarrow{a} \surd \quad q+p \xrightarrow{a} \surd \quad pq \xrightarrow{a} q \quad \pi_n(p) \xrightarrow{a} \surd}$$

$$\frac{t_X \xrightarrow{a} p}{X \xrightarrow{a} p} \text{ if } X = t_X \in E \qquad \frac{t_X \xrightarrow{a} \surd}{X \xrightarrow{a} \surd} \text{ if } X = t_X \in E$$

Table 2.10: Operational rules for BPA+PR

A relation $\xrightarrow{a}$ holds between terms $t, s, t \xrightarrow{a} s$, if and only if it can be derived using the rules in table 2.10. Analogously for the predicate $\xrightarrow{a} \surd$ for a term $t, t \xrightarrow{a} \surd$.
Now, we define:

Definition 2.2.2.12 (*Bisimulation*). A bisimulation is a relation $R \subseteq \mathbf{P} \times \mathbf{P}$ such that, for all $a \in \mathbf{A}$, pRq implies:

1. $p \xrightarrow{a} p' \Rightarrow \exists q'.q \xrightarrow{a} q' \wedge p'Rq'$;
2. $q \xrightarrow{a} q' \Rightarrow \exists p'.p \xrightarrow{a} p' \wedge p'Rq'$;
3. $p \xrightarrow{a} \surd \iff q \xrightarrow{a} \surd$;

We say that p and q are bisimilar (notation $p \underline{\leftrightarrow} q$) if there exists a bisimulation R such that pRq. □

The set of operations on $\mathbf{P}$ is defined pointwise (see [BW90]). Then, we have:

Theorem 2.2.2.13. $\mathbf{P}/\underline{\leftrightarrow} \models \mathrm{BPA} + \mathrm{PR}$. *Moreover* BPA+PR *is a complete axiomatization for* $\mathbf{P}/\underline{\leftrightarrow}$.

Example 2.2.2.14 (*The projective models*). Let $n > 0$ be a natural number. Let $\mathbf{A}^n = \{\pi_n(x) | x \in \mathbf{A}\}$. We define equality on $\mathbf{A}^n$ as expected. Operations are interpreted as follows:

$$a^{\mathbf{A}^n} = a \qquad x +^{\mathbf{A}^n} y = \pi_n(x + y)$$
$$\pi_m^{\mathbf{A}^n}(x) = \pi_n(\pi_m(x)) \qquad x \cdot^{\mathbf{A}^n} y = \pi_n(x \cdot y)$$

Now, we have:

Theorem 2.2.2.15. *For all* $n > 0$, $\mathbf{A}^n \models \mathrm{BPA} + \mathrm{PR}$.

However, for any n, $\mathbf{A}^n \not\models \mathrm{COMP}$, since

$$(a^n)^{\mathbf{A}^n} = a^n = (a^{n+1})^{\mathbf{A}^n}$$

but $a^n \neq a^{n+1}$ in the initial algebra.

Example 2.2.2.16 (*The projective limit model*). Let $t_i \in \mathbf{A}$ $(i > 0)$. A sequence $t_1, t_2, \ldots$ of closed terms is called *projective* if for all i, $t_i = \pi_i(t_{i+1})$. Note that if $t_1, t_2, \ldots$ is a projective sequence, then $t_n \in \mathbf{A}^n$. The set $\mathbf{A}^\infty$ of all projective sequences is the *projective limit* of $\mathbf{A}^n$ $(n > 0)$. We define the operations component-wise, according to those defined in Example 2.2.2.14. Now, we have:

Theorem 2.2.2.17. $\mathbf{A}^\infty \models \mathrm{BPA} + \mathrm{PR}$. *Moreover* BPA+PR *is a complete axiomatization for* $\mathbf{A}^\infty$.

Definition 2.2.2.18. Let $\mathcal{M}$ be a model of BPA. We define a relation $\longrightarrow \subseteq \mathcal{M} \times A \times \mathcal{M}$ in the following way:

$$p \xrightarrow{a} q \iff \mathcal{M} \models p = p + a \cdot q$$

or equivalently

$$p \xrightarrow{a} q \iff \mathcal{M} \models a \cdot q \leq p$$

Analogously, we define the relation $\longrightarrow \surd \subseteq \mathcal{M} \times A$, by

$$p \xrightarrow{a} \surd \iff \mathcal{M} \models p = p + a$$

or equivalently

$$p \xrightarrow{a} \surd \iff \mathcal{M} \models a \leq p$$

□

The definition above extends the notion of transition to all BPA models, and so, the notion of bisimulation given in Definition 2.2.2.12 can also be extended to all models in the expected way. Thus, we have fact 2.2.2.22.

Definition 2.2.2.19. Given a (possibly empty) sequence of atomic actions $\sigma \in A^*$ we define the relation $\longrightarrow^*$ inductively as follows (we use ϵ for the empty sequence):

$$x \xrightarrow{\epsilon}{}^* x$$

$$x \xrightarrow{a} y \Rightarrow x \xrightarrow{a}{}^* y$$

$$x \xrightarrow{\sigma}{}^* y \vee y \xrightarrow{\tau}{}^* z \Rightarrow x \xrightarrow{\sigma\tau}{}^* z$$

□

Definition 2.2.2.20 (*Action graph*). Given a closed term t its action graph will have as nodes the set of equivalence classes (modulo provability in BPA) of closed terms $\{s | \exists \sigma \in A^*, t \xrightarrow{\sigma}{}^* s\}$ and a special termination node $\surd$. The edges will be given by the action relations. □

Proposition 2.2.2.21. *Two closed terms s, t are provably equal ($BPA \vdash s = t$) if and only if they have isomprphic action graphs.*

Fact 2.2.2.22. Let $\mathcal{M}$ be a model for BPA. Then, for all $p, q \in \mathcal{M}$, we have

$$\mathcal{M} \models p = q \Rightarrow p \leftrightarrows q$$

□

As we will see further on in this chapter, in general, it is not true that bisimilarity implies equality in a model. However, the following lemma states that it holds for the subset of closed terms in a complete model [BW90]:

Lemma 2.2.2.23. *Let $\mathcal{M}$ be a* BPA *model such that $\mathcal{M} \models$* COMP. *For all processes $p, q \in \mathcal{M}$ which can be represented by a closed* BPA *term,*

$$\mathcal{M} \models p = q \iff p \leftrightarrows q$$

2.3 Left Cancellation of Atomic Actions

A useful property, both from a theoretical and applied point of view, is the left cancellation of atomic actions. The general left cancellation property

$$x \cdot y = x \cdot z \Rightarrow z = y$$

is of course not true in the standard models, since when we take a perpetual process x (i.e. there exists no sequence of atomic actions σ such that $x \xrightarrow{\sigma}{}^* \surd$) the equation in the antecedent holds for any y, z. As we will see, the left cancellation of atomic actions is useful in defining equivalence of states in a context where the state operator is present.

2.3.1 Left cancellation of atomic actions

One property which was barely touched on in process algebra is the left cancellation of atomic actions. We express this property using the following conditional axiom:

$$\text{CANC} \quad a \cdot x = a \cdot y \Rightarrow x = y$$

This property seems to be true in most of the (interleaving) models that appeared in the literature. It is useful when one works in contexts with the state operator (see section 2.6.4).

One can state a stronger version of this property.

$$\text{CANC}^+ \quad a \cdot x \leq a \cdot y \Rightarrow x = y$$

At first sight it looks too strong, but it only means that a process of the form $a \cdot x$ can only do an a-action into x. It is an obvious fact that $\text{CANC}^+ \Rightarrow \text{CANC}$. Later we will exhibit a model that satisfies CANC but not CANC^+. Another trivial implication of CANC^+ is that

$$a \cdot p \leq a \cdot q \iff a \cdot q \leq a \cdot p$$

2.3.2 Properties

Theorem 2.3.2.1. *Let $\mathcal{M}$ be a model for* BPA + PR *satisfying* COMP,AIP *and* FINPROJ. *Then* $\mathcal{M} \models \text{CANC}^+$.

Proof. For closed terms it is straightforward, since two basic terms are equal if and only if they have the same set of summands modulo A1, A2, A3. But if

$$a \cdot p \leq a \cdot q$$

since $a \cdot q$ has only one summand, then

$$a \cdot p = a \cdot q$$

using only A1, A2, A3. It follows from the completeness of the axiomatization with respect to $\mathcal{M}$ that $p = q$.

Take two processes p and q such that

$$p \neq q$$

Then, it follows by AIP that there exists n such that

$$\pi_n(p) \neq \pi_n(q).$$

From the first part of the proof and given that $\pi_n(p)$ and $\pi_n(q)$ are closed terms, it follows that

$$a \cdot \pi_n(p) \not\leq a \cdot \pi_n(q)$$

or equivalently

$$\pi_{n+1}(a \cdot q) \neq \pi_{n+1}(a \cdot p) + \pi_{n+1}(a \cdot q)$$

or

$$\pi_{n+1}(a \cdot q) \neq \pi_{n+1}(a \cdot p + a \cdot q)$$

and then,

$$a \cdot q \neq a \cdot p + a \cdot q$$

or equivalently

$$a \cdot p \not\leq a \cdot q$$

□

Corollary 2.3.2.2. *If* $\mathcal{M} \models \mathrm{COMP}, \mathrm{AIP}, \mathrm{DEF}$ *then* $\mathcal{M} \models \mathrm{CANC}^+$

As an immediate corollary, we have that a complete model satisfying AIP and FINPROJ (or definability) also satisfies (non-strong) cancellation.

Corollary 2.3.2.3. *Let* $\mathcal{M}$ *be a model where the state operator is present. Then if*

$$\mathcal{M} \models \mathrm{COMP}, \mathrm{AIP}, \mathrm{DEF}$$

then for any pair of states s, t *it holds that*

$$s \sim t \Rightarrow s \leftrightarrows t$$

Example 2.6.4.14 shows that completeness is essential for corollary 2.3.2.3.

We know that the initial algebra A of BPA is complete and satisfies AIP, and so does the projective limite model $\mathbf{A}^\infty$. In addition, all of them satisfy FINPROJ.

The term model $\mathbf{P}/\leftrightarrows$ or the graph model $G^\infty/\leftrightarrows$, which are isomorphic, do not satisfy AIP but they satisfy CANC^+. If a graph $a \cdot g$ is bisimilar to a graph $a \cdot h$ via R it is immediate from the definition of bisimulation and of sequential composition, that R must relate the roots of g and h as well. Thus, we have that:

Theorem 2.3.2.4. *The following models satisfy* CANC *and* CANC^+*:*

- *the initial algebra* A,
- *the graph model* $G^\infty/\leftrightarrows$, *and its submodels* $G/\leftrightarrows$, $R/\leftrightarrows$ *and* $F/\leftrightarrows$
- *the term model* $\mathrm{P}/\leftrightarrows$,
- *the projective limit model* $\mathbf{A}^\infty$.

2.4 Bisimulation in an Arbitrary Model

Given a model, we use the semantic action

$$x \xrightarrow{a} y \iff x = a \cdot y + x$$

to define bisimulation equivalence. In some models it coincides with its intrinsic equality, but in others it is coarser and in some it is not even a congruence.

As in the previous section we find conditions on the model that ensure that the bisimulation equivalence coincides with the equality of the model.

2.4.1 Bisimulation and models

From Definitions 2.2.2.18 and 2.2.2.9 we can easily prove that equality in a model $\mathcal{M}$ implies bisimulation, i.e.

$$\mathcal{M} \models p = q \Rightarrow p \leftrightarrows q$$

Moreover, $=$ is a bisimulation. Nevertheless, the converse does not hold in general (see Section 5.5.7.22). If a model $\mathcal{M}$ satisfies certain properties it holds that $p \leftrightarrows q \Rightarrow \mathcal{M} \models p = q$. In order to show that we will need the following technical lemma:

Lemma 2.4.1.1. *Let p and q be two closed processes in head normal form, i.e.*

$$p = \sum_I a_i \cdot p_i + \sum_J b_j$$

$$q = \sum_K c_k \cdot q_k + \sum_L d_l$$

then $p = q$ if and only if for any $i \in I$ there exists a $k \in K$ such that $a_i \cdot p_i = c_k \cdot q_k$ and for any $j \in J$ there exists a $l \in L$ such that $b_j = d_l$ and the same with the roles of p and q exchanged. In other words, if they have the same set of summands modulo BPA equality.

Proof. In [BW90] the following rewriting system is defined.

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

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

This rewriting system is confluent modulo A1, A2, A3, i.e. two normal forms could be proved equal using these three axioms, and strongly normalizing. It is immediate by simple inspection, that the three axioms preserve the set of summands of a normal form (modulo A1, A2, A3). This means that if s is a summand of a normal form x, and y is a normal form such that $x = y$ then there is a summand t of y such that $A1, A2, A3 \vdash s = t$.

It is enough to show now that a term in head normal form has the same set of summands (modulo BPA equality) as its normal form. However, this is a consequence of the definition of the rewriting system, since $(\sum_I a_i \cdot p_i + \sum_J b_j) \longrightarrow p'$ if and only if there exist i_0, r such that $p_{i_0} \longrightarrow r$ and $p' = \sum_{I-\{i_0\}} a_i \cdot p_i + a_{i_0} \cdot r + \sum_J b_j$. This implies that all the intermediate terms in any reduction $p \longrightarrow^* p'$ have the same set of summands. □

Theorem 2.4.1.2. *Let $\mathcal{M}$ be a model of* BPA*, such that $\mathcal{M}$ is complete, satisfies* AIP *and every process in it is definable. Then two processes in $\mathcal{M}$ are equal if and only if they are bisimilar.*

Proof. By the definition of the action relation it is obvious that two equal processes are bisimilar.

Take two processes p, q such that $p \leftrightarrows q$.

If p and q are closed terms then they are bisimilar if and only if they have the same action graph, if and only if they can be proved equal by the axioms, if and only if they are equal in a complete model.

Now, suppose that p and q are definable, and assume that $p \leftrightarrows q$. We want to show that for any n, $\pi_n(p) = \pi_n(q)$. The desired result follows by AIP. Since $\pi_n(p)$ and $\pi_n(q)$ are closed terms, by the first half of the proof it is enough to prove that for any n, $\pi_n(p) \leftrightarrows \pi_n(q)$.

In order to show this, take $R : p \leftrightarrows q$ and define $S = \{(\pi_n(p), \pi_n(q)) | pRq\}$ Now, using the lemma above, it is easy to prove that S is indeed a bisimulation. □

We say that *a model $\mathcal{M}$ satisfies bisimilarity* (Notation $\mathcal{M} \models$ BISIM) iff $\mathcal{M}$ and $\mathcal{M}/\leftrightarrows$ are isomorphic. Thus, the previous theorem states a sufficient condition for a model to satisfy bisimilarity.

Hence, the models that satisfy completeness, AIP and definability, satisfy both the left cancellative property and bisimilarity as well. However, bisimulation and left-cancellation seem to be unrelated.

2.5 Non-Standard Models

In this section we introduce some models that do not satisfy some of the properties presented above. All these models are complete and some satisfy AIP as well. All the models are constructed from a BPA model that satisfies AIP, completeness and definability. Moreover, the original model can be embedded in the new one. An axiomatization for these models is also given using an extra operator.

A non-complete model that already appears in the literature is also shown which satisfies bisimulation, but not the weakest version of the cancellation property.

In all the other examples a couple of complementary operators are introduced. The operator $\uparrow$ indicates a divergence or failure. The exact interpretation of it depends on the model under consideration. The operator $\downarrow$ is the complement of the previous one, and it is supposed to represent the state of "normality" for a given process.

2.5.1 The models $\mathbf{A}^n$

These models were introduced in [BK87]. It is clear that they satisfy bisimulation, but the following example shows that they do not satisfy the cancellation property.

Example 2.5.1.1. We have that all the finite projective models A^n, which are not complete, do not satisfy CANC. Consider, for instance, the following example:

$$(a \cdot a^n)^{\mathrm{A}^n} = a \cdot a^{n-1} = (a \cdot a^{n-1})^{\mathrm{A}^n}$$

however,

$$(a^{n-1})^{\mathrm{A}^n} = a^{n-1} \neq a^n = (a^n)^{\mathrm{A}^n}$$

□

2.5.2 Processes with root divergence

Let $\mathcal{M}$ be model for BPA satisfying AIPand COMP. We define the set $\mathcal{M}_1^{\updownarrow}$ as the union $\mathcal{M}\downarrow \cup \mathcal{M}\uparrow$, where

$$\mathcal{M}\downarrow = \{p\downarrow \mid p \in \mathcal{M}\}$$
$$\mathcal{M}\uparrow = \{p\uparrow \mid p \in \mathcal{M}\}$$

The sets $\mathcal{M}\downarrow$ and $\mathcal{M}\uparrow$ are disjoint, i.e. $p\downarrow = q\downarrow \iff p\uparrow = q\uparrow \iff p = q$ while $p\downarrow \neq q\uparrow$. We take the following interpretation for the BPA operations:

$a^{\mathcal{M}_1^{\updownarrow}} = a\downarrow$
$p\downarrow +^{\mathcal{M}_1^{\updownarrow}} q\downarrow = (p +^{\mathcal{M}} q)\downarrow$
$p\downarrow +^{\mathcal{M}_1^{\updownarrow}} q\uparrow = p\uparrow +^{\mathcal{M}_1^{\updownarrow}} q\downarrow = p\uparrow +^{\mathcal{M}_1^{\updownarrow}} q\uparrow = (p +^{\mathcal{M}} q)\uparrow$
$p\downarrow \cdot^{\mathcal{M}_1^{\updownarrow}} q\downarrow = p\downarrow \cdot^{\mathcal{M}_1^{\updownarrow}} q\uparrow = (p \cdot^{\mathcal{M}} q)\downarrow$
$p\uparrow \cdot^{\mathcal{M}_1^{\updownarrow}} q\downarrow = p\uparrow \cdot^{\mathcal{M}_1^{\updownarrow}} q\uparrow = (p \cdot^{\mathcal{M}} q)\uparrow$
$\pi_n^{\mathcal{M}_1^{\updownarrow}}(p\downarrow) = \pi_n^{\mathcal{M}}(p)\downarrow$
$\pi_n^{\mathcal{M}_1^{\updownarrow}}(p\uparrow) = \pi_n^{\mathcal{M}}(p)\uparrow$

Intuitively, the symbol $\downarrow$ may be understood as "root convergence", and, conversely, $\uparrow$ may be interpreted as "root divergence". The expression $p\uparrow$ can be read as 'p may diverge in its first step'.

It is clear that the submodel $\mathcal{M}\downarrow$ of $\mathcal{M}_1^{\updownarrow}$ is isomorphic to $\mathcal{M}$.

Theorem 2.5.2.2 (*Soundness*). $\mathcal{M}_1^{\updownarrow} \models \mathrm{BPA}$.

Proof. As usual, it is proved by showing that each axiom holds for every element of $\mathcal{M}_1^{\updownarrow}$ according to the interpretation above. We only prove A4 which seems to be more difficult than the others. In order to do this we consider separately the cases of $\downarrow$ and $\uparrow$, but when they are not relevant, we will write $\updownarrow$.

$$\begin{aligned}
(p\downarrow +^{\mathcal{M}_1^{\updownarrow}} q\downarrow) \cdot^{\mathcal{M}_1^{\updownarrow}} r\updownarrow &= (p +^{\mathcal{M}} q)\downarrow \cdot^{\mathcal{M}_1^{\updownarrow}} r\updownarrow \\
&= ((p +^{\mathcal{M}} q) \cdot^{\mathcal{M}} r)\downarrow \\
&= (p \cdot^{\mathcal{M}} r +^{\mathcal{M}} q \cdot^{\mathcal{M}} r)\downarrow \qquad \text{(A4 holds on } \mathcal{M}) \\
&= (p \cdot^{\mathcal{M}} r)\downarrow +^{\mathcal{M}_1^{\updownarrow}} (q \cdot^{\mathcal{M}} r)\downarrow \\
&= p\downarrow \cdot^{\mathcal{M}_1^{\updownarrow}} r\updownarrow +^{\mathcal{M}_1^{\updownarrow}} q\downarrow \cdot^{\mathcal{M}_1^{\updownarrow}} r\updownarrow \\
(p\uparrow +^{\mathcal{M}_1^{\updownarrow}} q\updownarrow) \cdot^{\mathcal{M}_1^{\updownarrow}} r\updownarrow &= (p +^{\mathcal{M}} q)\uparrow \cdot^{\mathcal{M}_1^{\updownarrow}} r\updownarrow \\
&= ((p +^{\mathcal{M}} q) \cdot^{\mathcal{M}} r)\uparrow \\
&= (p \cdot^{\mathcal{M}} r +^{\mathcal{M}} q \cdot^{\mathcal{M}} r)\uparrow \qquad \text{(A4 holds on } \mathcal{M}) \\
&= (p \cdot^{\mathcal{M}} r)\uparrow +^{\mathcal{M}_1^{\updownarrow}} (q \cdot^{\mathcal{M}} r)\updownarrow \\
&= p\uparrow \cdot^{\mathcal{M}_1^{\updownarrow}} r\updownarrow +^{\mathcal{M}_1^{\updownarrow}} q\updownarrow \cdot^{\mathcal{M}_1^{\updownarrow}} r\updownarrow
\end{aligned}$$

The case for $p\downarrow$ and $q\uparrow$ follows quite similarly. □

Theorem 2.5.2.3 (*Completeness*). *Let s,t be two closed* BPA *terms. Then* $\mathcal{M}_1^{\updownarrow} \models s = t$ iff $\mathrm{BPA} \vdash s = t$.

Proof. Note that for closed terms s, $s^{\mathcal{M}_1^{\updownarrow}} = s^{\mathcal{M}}\downarrow$. Thus the result follows from $\mathcal{M}_1^{\updownarrow} \models p\downarrow = q\downarrow \iff \mathcal{M} \models p = q$, since $\mathcal{M} \models \mathrm{COMP}$. □

Theorem 2.5.2.4. $\mathcal{M}_1^{\updownarrow} \models \mathrm{AIP}$.

Proof. Take $p\downarrow$ and $q\downarrow$, and suppose that for all n, $\pi_n^{\mathcal{M}_1^{\updownarrow}}(p\downarrow) = \pi_n^{\mathcal{M}_1^{\updownarrow}}(q\downarrow)$. By definition of $\pi_n^{\mathcal{M}_1^{\updownarrow}}$ we have that $\pi_n^{\mathcal{M}}(p)\downarrow = \pi_n^{\mathcal{M}}(q)\downarrow$. Because $\mathcal{M} \models \mathrm{AIP}$, we have that $\pi_n^{\mathcal{M}}(p) = \pi_n^{\mathcal{M}}(q)$ implies $p = q$, and hence $p\downarrow = q\downarrow$.
If we take $p\uparrow$ and $q\uparrow$, the proof follows similarly.

For all p, q we have that $\pi_n^{\mathcal{M}_1^{\updownarrow}}(p\downarrow) = \pi_n^{\mathcal{M}}(p)\downarrow \neq \pi_n^{\mathcal{M}}(q)\uparrow = \pi_n^{\mathcal{M}_1^{\updownarrow}}(q\uparrow)$, and so it never holds that $\pi_n^{\mathcal{M}_1^{\updownarrow}}(p\downarrow) = \pi_n^{\mathcal{M}_1^{\updownarrow}}(q\uparrow)$. □

Proposition 2.5.2.5. $\mathcal{M}_1^{\updownarrow} \not\models \mathrm{CANC}$.

Proof. It can be easily proved by considering this example:

$$a\downarrow \cdot^{\mathcal{M}_1^{\updownarrow}} b\downarrow = (a \cdot^{\mathcal{M}} b)\downarrow = a\downarrow \cdot^{\mathcal{M}_1^{\updownarrow}} b\uparrow$$

but $b\downarrow \neq b\uparrow$ by definition.
In this case, we have that $b\uparrow$ is not a definable process. □

Proposition 2.5.2.6. $\mathcal{M}_1^{\updownarrow} \not\models \mathrm{BISIM}$.

Proof. We show that for any $p \in \mathcal{M}$ it holds that $p\downarrow \leftrightarrows p\uparrow$. In order to do that, we show that

$$p\downarrow \xrightarrow{a} q\updownarrow \iff p\uparrow \xrightarrow{a} q\updownarrow$$

From this fact it is immediate that $p\downarrow \leftrightarrows p\uparrow$

$$p\downarrow \xrightarrow{a} q\updownarrow \iff$$
$$a\downarrow \cdot q\updownarrow \leq p\downarrow \iff$$
$$a\cdot q\downarrow \leq p\downarrow \iff$$
$$p\downarrow = a\cdot q\downarrow + p\downarrow \iff$$
$$p\downarrow = (a\cdot q + p)\downarrow \iff$$
$$p = (a\cdot q + p)$$

and also

$$p\uparrow \xrightarrow{a} q\updownarrow \iff$$
$$a\downarrow \cdot q\updownarrow \leq p\uparrow \iff$$
$$a\cdot q\downarrow \leq p\uparrow \iff$$
$$p\uparrow = a\cdot q\downarrow + p\uparrow \iff$$
$$p\uparrow = (a\cdot q + p)\uparrow \iff$$
$$p = (a\cdot q + p)$$

□

The quotient of $\mathcal{M}_1^{\updownarrow}$ modulo bisimulation is isomorphic to $\mathcal{M}$.

We can extend BPA+PR in order to obtain a new equational theory (write BPA + PR+ $\uparrow$) which includes the unary operator $\uparrow$. In this way, terms having no $\uparrow$ are interpreted as elements having $\downarrow$ in $\mathcal{M}_1^{\updownarrow}$. Additional axioms are given in Table 2.11.

It is not difficult to prove that the related term rewriting system, where the rules are the axioms written from left to right, is strongly normalizing. Moreover, if t is a basic BPA+PR term, then t and $t\uparrow$ are basic BPA + PR+ $\uparrow$ terms. Now the following theorem can be easily proved:

Theorem 2.5.2.7. $\mathcal{M}_1^{\updownarrow} \models \mathrm{BPA} + \mathrm{PR}+\uparrow$. *Moreover,* BPA + PR+ $\uparrow$ *is a complete axiomatization for* $\mathcal{M}_1^{\updownarrow}$.

Similar equational theories can be stated for models in the following sections.

D1	$x\uparrow + y = (x + y)\uparrow$
D2	$x + y\uparrow = (x + y)\uparrow$
D3	$x\uparrow \cdot y = (x \cdot y)\uparrow$
D4	$x \cdot y\uparrow = x \cdot y$
D5	$\pi_n(x\uparrow) = \pi_n(x)\uparrow$

Table 2.11: Additional axioms for BPA + PR+$\uparrow$

2.5.3 Processes that may eventually fail

Now, we define the model $\mathcal{M}_2^{\updownarrow}$ starting from $\mathcal{M}$, a model for BPA satisfying AIPan COMP. We define this exactly as before, except that we give a new interpretation for the sequential composition:

$p\downarrow \cdot^{\mathcal{M}_2^{\updownarrow}} q\downarrow = (p \cdot^{\mathcal{M}} q)\downarrow$

$p\downarrow \cdot^{\mathcal{M}_2^{\updownarrow}} q\uparrow = p\uparrow \cdot^{\mathcal{M}_2^{\updownarrow}} q\downarrow = p\uparrow \cdot^{\mathcal{M}_2^{\updownarrow}} q\uparrow = (p \cdot^{\mathcal{M}} q)\uparrow$

Now, the symbol $\uparrow$ may be understood as "may eventually fail", and $\downarrow$, as "never fail".

Theorem 2.5.3.8 (*Soundness*). $\mathcal{M}_2^{\updownarrow} \models$ BPA.

Proof. This follows like in 2.5.2.2 □

Theorem 2.5.3.9 (*Completeness*). *Let s, t two closed* BPA *terms. Then* $\mathcal{M}_2^{\updownarrow} \models s = t$ iff BPA $\vdash s = t$.

Proof. As in the case of theorem 2.5.2.3, this follows from $\mathcal{M}_2^{\updownarrow} \models p\downarrow = q\downarrow \iff \mathcal{M} \models p = q$, since $\mathcal{M} \models$ COMP. □

Theorem 2.5.3.10. $\mathcal{M}_2^{\updownarrow} \models$ AIP.

Proof. This follows exactly as Theorem 2.5.2.4. □

Theorem 2.5.3.11. $\mathcal{M} \models$ CANC *implies* $\mathcal{M}_2^{\updownarrow} \models$ CANC.

Proof. As we know, the only possible interpretation of a in $\mathcal{M}_2^{\updownarrow}$ is $a\downarrow$. For any $p, q \in \mathcal{M}_2^{\updownarrow}$, assume $a\downarrow \cdot^{\mathcal{M}_2^{\updownarrow}} p = a\downarrow \cdot^{\mathcal{M}_2^{\updownarrow}} q$.

Now, let us consider that there is a $p' \in \mathcal{M}$ such that $p'\downarrow = p$. Hence, $a\downarrow \cdot^{\mathcal{M}_2^{\updownarrow}} p = (a \cdot^{\mathcal{M}} p')\downarrow$. Suppose that $q = q'\uparrow$ for some $q' \in \mathcal{M}$. Then, $a\downarrow \cdot^{\mathcal{M}_2^{\updownarrow}} q = (a \cdot^{\mathcal{M}} q')\uparrow$ which contradicts our assumption. So $q = q'\downarrow$ for some $q' \in \mathcal{M}$. But $(a \cdot^{\mathcal{M}} p')\downarrow = (a \cdot^{\mathcal{M}} q')\downarrow \iff (a \cdot^{\mathcal{M}} p') = (a \cdot^{\mathcal{M}} q')$. Because $\mathcal{M} \models$ CANC, $p' = q'$, which implies $p = p'\downarrow = q'\downarrow = q$.

Case of $p = p'\uparrow$ for some $p' \in \mathcal{M}$ is proved similarly. □

Lemma 2.5.3.12. $\mathcal{M}_2^{\updownarrow} \not\models \mathrm{CANC}^+$.

Proof. For any process p it holds that $p\downarrow \leq p\uparrow$, since $p\downarrow + p\uparrow = p\uparrow$. In particular $a\downarrow \cdot p\downarrow = (a \cdot p)\downarrow \leq (a \cdot p)\uparrow = a\downarrow \cdot p\uparrow$ but $p\downarrow \neq p\uparrow$. □

We can modify this model obtaining a new one that does not satisfy AIP, say $\mathcal{M}_{2\mathrm{notAIP}}^{\updownarrow}$. In order to do so, we redefine the π_n operator in the following way:

$$\pi_n^{\mathcal{M}_{2\mathrm{notAIP}}^{\updownarrow}}(p\downarrow) = \pi_n^{\mathcal{M}_{2\mathrm{notAIP}}^{\updownarrow}}(p\uparrow) = \pi_n^{\mathcal{M}}(p)\downarrow$$

Hence, $\mathcal{M}_{2\mathrm{notAIP}}^{\updownarrow}$ is a complete model for BPA that satisfies CANC but not AIP.

2.5.4 Other models

We define two other models, $\mathcal{M}_3^{\updownarrow}$ and $\mathcal{M}_4^{\updownarrow}$, such that the sequential composition is defined as in $\mathcal{M}_1^{\updownarrow}$ and $\mathcal{M}_2^{\updownarrow}$ respectively but the sum is defined for both as follows:

$p\uparrow +^{\mathcal{M}_{3,4}} q\uparrow = (p +^{\mathcal{M}} q)\uparrow$

$p\downarrow +^{\mathcal{M}_{3,4}} q\uparrow = p\uparrow +^{\mathcal{M}_{3,4}} q\downarrow = p\downarrow +^{\mathcal{M}_{3,4}} q\downarrow = (p +^{\mathcal{M}} q)\downarrow$

The intuitive interpretation is now as follows. In $\mathcal{M}_3^{\updownarrow}$, $p\uparrow$ means that p must diverge in the first step, while in $\mathcal{M}_4^{\updownarrow}$, it means that p must diverge eventually.

We study here only $\mathcal{M}_3^{\updownarrow}$.

Proposition 2.5.4.13. $\mathcal{M}_3^{\updownarrow} \not\models \mathrm{CANC}$

Proof. as in fact 2.5.2.5 □

Proposition 2.5.4.14. $\mathcal{M}_3^{\updownarrow} \not\models$ *BISIM, furthermore, bisimulation is not a congruence in* $\mathcal{M}_3^{\updownarrow}$.

Proof. First note that a divergent process cannot perform an action,

$$p\uparrow \xrightarrow{a} q \iff$$
$$a \cdot q \leq p\uparrow \iff$$
$$a \cdot q + p\uparrow = p\uparrow \iff$$
$$a\downarrow \cdot q + p\uparrow = p\uparrow \iff$$
$$(a \cdot q + p)\downarrow = p\uparrow$$

And also, that the last equality cannot be true. It follows that all divergent processes are bisimilar. If we then take $p \neq q$,

$$a \cdot p\uparrow = (a \cdot p)\downarrow$$
$$a \cdot q\uparrow = (a \cdot q)\downarrow$$

they are obviously not equal and therefore non bisimilar. □

2.6 The State Operator

2.6.1 Introduction

The state operator in process algebra is introduced as a generalization of the renaming operators. This new operator represents the fact that the execution of a process can be influenced by the environment. This is achieved by taking a set $\mathbf{S}$ whose elements will be considered as states and two functions:

$$\begin{aligned} \leftarrow &: A \times \mathbf{S} \to A \cup \{\delta\} \\ \rightarrow &: A \times \mathbf{S} \to \mathcal{P}(\mathbf{S}) \end{aligned}$$

These functions will be used to describe the interaction between the states and the atomic actions. In the original presentation of the state operator (see [BB88]) it was called λ and the functions here named *act* and *eff* respectively, but where the codomain of the second function was $\mathbf{S}$ instead of $\mathcal{P}(\mathbf{S})$. However, this first approach is not enough to describe the input-output behaviour of a process, since a nondeterministic choice could produce more than one output. The set obtained by the application of the effect function to a process and a state will consist of all states that can be reached, in at least one of the possible executions of the process, beginning in the given initial state. The change of the name of the operator is intended to reflect the more symmetrical view of the action and effect function in this thesis.

2.6.2 Axioms for the State Operator

We extend both functions to deal with processes:

$$\begin{aligned} \leftarrow &: \mathbf{P} \times \mathcal{P}(\mathbf{S}) \to \mathbf{P} \\ \rightarrow &: \mathbf{P} \times \mathcal{P}(\mathbf{S}) \to \mathcal{P}(\mathbf{S}) \end{aligned}$$

by means of the axioms in table 2.12 where $a \in A$, $s \in \mathbf{S}$ and $S, T \subseteq \mathbf{S}$. We sometimes write s for the singleton $\{s\}$.

SA1	$x\leftarrow\emptyset = \delta$	SE1	$x\rightarrow\emptyset = \emptyset$
SA2	$x\leftarrow\{s\} = x\leftarrow s$	SE2	$x\rightarrow\{s\} = x\rightarrow s$
SA3	$x\leftarrow(S \cup T) = x\leftarrow S + x\leftarrow T$	SE3	$x\rightarrow(S \cup T) = x\rightarrow S \cup x\rightarrow T$
SA4	$\delta\leftarrow s = \delta$	SE4	$\delta\rightarrow s = \emptyset$
SA5	$a \cdot x\leftarrow s = (a\leftarrow s) \cdot (x\leftarrow(a\rightarrow s))$	SE5	$a \cdot x\rightarrow s = x\rightarrow(a\rightarrow s)$
SA6	$(x + y)\leftarrow s = x\leftarrow s + y\leftarrow s$	SE6	$(x + y)\rightarrow s = x\rightarrow s \cup y\rightarrow s$

Table 2.12: Axioms for the state operator

A state I is called *inert* if for all actions $a, a \leftarrow I = a$ and $a \rightarrow I = I$. We assume the that every state space will have an inert state.

We also assume the presence in each state space of a *blocked* state called 0 such that for any atomic action $a, a \leftarrow 0 = \delta$ and $a \rightarrow 0 = 0$. This blocked state will not be of much use in BPA_δ but we introduce it here for completeness. The blocked state will be needed in contexts where δ is not present and we want to block actions using the NIL process.

An alternative way to introduce these two special states could be through the axioms in table 2.13. Since both axioms are satisfied for all definable processes, most of the models would still be consistent with these new axioms.

SA8	$x \leftarrow I = x$	SE8	$x \rightarrow I = I$
SA9	$x \leftarrow 0 = \delta$	SE9	$x \rightarrow 0 = 0$

Table 2.13: Axioms for the special states $I, 0$

In order to be able to infer input/output properties for non-closed processes we introduce the following principle. It says that a state will belong to the output set of a process if and only if a successfully terminated trace leads to it from an initial state.

Definition 2.6.2.1. We say that a process satisfies the principle of *Terminated Traces* if the following equality holds:

$$x \rightarrow s = \bigcup_{i<\omega} (\pi_i^\delta(x) \rightarrow s)$$

□

2.6.3 Properties of the state operator

Lemma 2.6.3.2. *For every definable process p, state s and $n > 0$*

$$\pi_n(p \leftarrow s) = \pi_n(p) \leftarrow s$$

Proof. Straightforward, by induction on n. □

Definition 2.6.3.3. Given a state operator we define the alphabet of a particular state by:

$$\alpha(s) = \{a \in A : a \leftarrow s \neq a \vee a \rightarrow s \neq s\}$$

□

Definition 2.6.3.4. Given a state operator, a state s and a set of atomic actions B we define:

$$\lambda_s(B) = \{b \leftarrow s : b \in B\}$$

□

Lemma 2.6.3.5. *Let s, t be two states such that the following conditions hold*

$$\alpha(s) \cap \alpha(t) = \emptyset$$

$$\lambda_s(\alpha(s)) \cap \alpha(t) = \emptyset$$

$$\lambda_t(\alpha(t)) \cap \alpha(s) = \emptyset$$

then for any definable process p

$$p \leftarrow t \leftarrow s = p \leftarrow s \leftarrow t$$

Proof. Straightforward. □

Definition 2.6.3.6. If for any pair of states s, t it holds that

$$\alpha(s) \cap \alpha(t) = \emptyset$$

$$\lambda_s(\alpha(s)) \cap \alpha(t) = \emptyset$$

and $\beta = \{s_1, \ldots, s_n\}$ is any multiset of states we can define, in view of the previous lemma,

$$p \leftarrow \beta = \ldots (p \leftarrow s_n) \ldots \leftarrow s_1$$

as the order is not important. □

Definition 2.6.3.7. Let $\leftarrow$ and $\rightarrow$ define a state operator over $\mathbf{S}$. The state operator can be extended to work over $\mathbf{S} \times \mathbf{S}$ in the following way:

$$a \leftarrow < s, t > = a \leftarrow s \leftarrow t$$

$$a \rightarrow < s, t > = < a \rightarrow s, (a \leftarrow s) \rightarrow t) >$$

□

Note that $< I, I >$ is an inert state and $< 0, 0 >$ is a blocked state. Moreover, the original state operator can be embedded into one whose state space is $\mathbf{S} \times \mathbf{S}$.

Lemma 2.6.3.8. *For all definable processes p, states s,t*

$$p \leftarrow < s, t > = p \leftarrow s \leftarrow t$$

Proof.

(i) For closed terms. This is straightforward, by structural induction. For example, if $p = a \cdot q$ and we have proven the lemma for q, then

$$\begin{aligned} p\leftarrow < s,t > &= (a \cdot q)\leftarrow < s,t > \\ &= a\leftarrow < s,t > \cdot q\leftarrow(a\rightarrow < s,t >) \\ &= a\leftarrow < s,t > \cdot q\leftarrow(< a\rightarrow s,(a\leftarrow s)\rightarrow t >) \\ &= a\leftarrow s\leftarrow t \cdot q\leftarrow(a\rightarrow s)\leftarrow((a\leftarrow s)\rightarrow t) \\ &= (a\leftarrow s \cdot q\leftarrow(a\rightarrow s))\leftarrow t \\ &= (a \cdot q)\leftarrow s\leftarrow t \\ &= p\leftarrow s\leftarrow t \end{aligned}$$

(ii) For definable processes. Using AIP, lemma 2.1.7.2 and lemma 2.6.3.2.

$$\begin{aligned} \pi_n(p\leftarrow < s,t >) &= \pi_n(p)\leftarrow < s,t > \\ &= \pi_n(p)\leftarrow s\leftarrow t \\ &= \pi_n(p\leftarrow s\leftarrow t) \end{aligned}$$

□

Definition 2.6.3.9. In a similar way, the state operator can be extended to act on finite sequences of states in the following way:

$$a\leftarrow s_1 \ldots s_n = a\leftarrow s_1\leftarrow \cdots \leftarrow s_n$$

$$a\rightarrow s_1 \ldots s_n = (a\rightarrow s_1)((a\leftarrow s_1)\rightarrow s_2)\ldots((a\leftarrow s_1 \ldots s_{n-1})\rightarrow s_n)$$

where given two sets of sequences of states T, R we define $TR = \{tr | t \in T, r \in R\}$. □

2.6.4 Equivalence of states

For some applications of the state operator we want to identify states that cannot be distinguished by any process. We define in this section two different notions that coincide in a wide class of processes.

Definition 2.6.4.10 (*State Bisimulation*). A *state bisimulation* is a relation $R \subseteq S \times S$ such that if sRt then the following two clauses hold:

- $\forall a \in A.a\leftarrow s = a\leftarrow t$
- $\forall a \in A.a\rightarrow sRa\rightarrow t$

where R extends canonically to sets of states.

We say that two states are bisimilar if there exists a state bisimulation which relates them. We write this down as

$$R : s \leftrightarrows t$$

□

Definition 2.6.4.11. Given a model $\mathcal{M}$ of process algebra we define an equivalence of states in the following way: Let $s, t \in S$. We say that s is equivalent to t if and only if for any process $p \in \mathcal{M}$ it holds that

$$p{\leftarrow}s = p{\leftarrow}t$$

and we write it as $s \sim t$.

□

The following property was studied also in section 2.3.

Definition 2.6.4.12 (*Left cancellation*)**.** A model satisfies the left cancellation property (of atomic actions) if the following conditional equation is true in such a model for any $a \in A$, x, y processes in the model:

$$a \cdot x = a \cdot y \Rightarrow x = y$$

□

Lemma 2.6.4.13. *If a model $\mathcal{M}$ satisfies the left cancellation property then two equivalent states are bisimilar as well, in other words:*

$$s \sim t \Rightarrow s \leftrightarrows t$$

Proof. We want to show that the relation $\sim$ is a state bisimulation. Take two states s, t such that $s \sim t$, and an atomic action a. We must verify that both conditions hold. The first is immediate from the definition. For the second we need the left cancellation property. We want to show that (note that we extend $\sim$ to set of states)

$$a{\rightarrow}s \sim a{\rightarrow}t$$

or equivalently

$$\forall p \in \mathcal{M}.p{\leftarrow}(a{\rightarrow}s) = p{\leftarrow}(a{\rightarrow}t)$$

We know, by definition of $\sim$, that

$$a \cdot p{\leftarrow}s = a \cdot p{\leftarrow}t$$

and this is equivalent to

$$a{\leftarrow}s \cdot (p{\leftarrow}(a{\rightarrow}s)) = a{\leftarrow}t \cdot (p{\leftarrow}(a{\rightarrow}t))$$

since, again by definition of $\sim$, $a{\leftarrow}s = a{\leftarrow}t$ we obtain the required equality applying the cancellation property. □

We will show an example of a model that does not satisfy the left cancellation property in which two states are equivalent but not bisimilar (example 2.6.4.14).

Example 2.6.4.14. For any model $\mathbf{A}^n$ we define a state operator such that there exist two states equivalent in the model that are not bisimilar. Take as a state space the following set:

$$\{s_i, t_i | i : 0 \ldots n\}$$

and define the action function $\leftarrow$ for any atomic action a and $i : 0 \ldots n-1$ as

$$a \leftarrow s_i = a \leftarrow t_i = a$$

and take given different actions c and d,

$$a \leftarrow s_n = c$$

$$a \leftarrow t_n = d$$

The effect function is defined, for every $i : 0 \ldots n-1$, as follows:

$$a \rightarrow s_i = s_{i+1}$$

$$a \rightarrow t_i = t_{i+1}$$

It is immediate that s_0 and t_0 are not bisimilar but they cannot be distinguished by any process in the model A^n, and therefore they are equivalent. □

Lemma 2.6.4.15. *In any model consisting only of definable processes and satisfying AIP, two bisimilar states are equivalent*

Proof. We need to prove it only for closed terms, since the more general result follows immediately from AIP and lemma 2.6.3.2.

We use induction on the structure of basic processes, since, using the elimination lemma, any closed term is equal to a basic term.

Let $s \leftrightarrows t$. For atomic actions and δ it is immediate that

$$a \leftarrow s = a \leftarrow t$$

$$\delta \leftarrow s = \delta \leftarrow t$$

Let p be a closed process which can be written in normal form as follows:

$$p = \sum a_i \cdot p_i + \sum b_j$$

It follows that

$$\begin{aligned} p \leftarrow s &= \sum a_i \leftarrow s \cdot (p_i \leftarrow (a_i \rightarrow s)) + \sum b_j \leftarrow s \\ &= \sum a_i \leftarrow t \cdot (p_i \leftarrow (a_i \rightarrow t)) + \sum b_j \leftarrow t \\ &= p \leftarrow t \end{aligned}$$

The second equality follows from the induction hypothesis and the fact that, by definition, $a \rightarrow s \leftrightarrows a \rightarrow t$.

□

Chapter 3

Atomic actions in process algebra

3.1 Introduction

One of the most important methods used to give semantics to a sequential program is to describe its input-output behaviour. This approach proved to be insufficient when programs were composed in parallel. An important problem of the input-output equivalence is that it is not a congruence with respect to parallel composition. This led to the introduction of *reactive systems*, which are systems that interact with the environment in a more general way than through input-output.

Nevertheless, for some problems it is still useful to identify systems with an equivalent input-output behaviour. To define such behaviour we begin by taking a set of states and interpreting the atomic actions according to their ability to modify states. This approach was studied in [Bou89, KP87a] among others, and in [BK84b] in the context of process algebra.

In this chapter we study these facts in the framework proposed by process algebra [BW90]. In addition to notions of atomic action and process, we consider multiactions and critical sections.

A *multiaction* [BB93] is a multiset of actions that can be regarded as "one step" of the execution of a process [Mil83]. We present two different ways to observe multiactions both extending step bisimulation, i.e. bisimulation in which a multiset of atomic actions can be executed in one step.

A *critical section* is an activity that cannot be interrupted while it is executing. We distinguish two important properties of such activities [Bou89]:

- *recoverability*, that is, an activity should either complete or not do anything; and
- *non-interference*: it can be observed as "locking" of the global state. An activity is not interfered with if along its execution no other action accesses the global state.

Here, we call an activity a critical section whenever it satisfies at least the recoverability

property. If a critical section is not interfered with while executing, we say that it is *atomic*.

After studying several kinds of actions, we extend the state operator [BB88] in order to deal with them. Thus, we obtain an operator for studying the input-output behaviour of a parallel system.

3.2 Critical Sections

An atomic action cannot be interrupted by a process running in parallel with it. This property is shared by *critical sections* as proposed in [BK84b]. In this section we study the equational theory and two different models.

Some improvements over [BK84b] are the following:

- A correct definition of basic terms, both for the system with and without tight multiplication.
- The addition of axioms CS3' and CS4', which are necessary to obtain the basic terms of the system without tight multiplication.
- An operational semantics with two different kind of transitions depending on whether the transition belongs to a critical section. This semantics was suggested in [BK84b] and a similar version for a related system appeared in [OL87].
- An operational semantics inspired by [Bou89] where any process can be a label for a transition.

3.2.1 AMP: Process algebra with mutual exclusion of critical sections

We define the algebra AMP parametrized by a set of constants A. The signature is given in Table 3.1. This is an algebra extending PA.

The intended behaviour of the operators (in some appropriate model) is as follows:

- atomic actions give the idea of a simple and indivisible event;

- $\langle p \rangle$ behaves like p but the execution cannot be interrupted by any other activity. That is, $\langle p \rangle$ must be seen as an *indivisible activity*;

- $\phi(p)$ makes the execution of p interruptible. In fact, ϕ is the inverse operation of $\langle \_ \rangle$;

- $\cdot, +, ||, \lfloor\!\lfloor$ as in PA.

Constants	
$a(\in A)$	atomic actions
Unary operators	
$\langle\_\rangle$	critical section
$\phi(\_)$	
Binary operators	
$\cdot$	sequential composition
$+$	alternative composition
$\|\|$	parallel composition (free merge)
$\lfloor\!\lfloor$	left merge

Table 3.1: Signature of AMP

A1	$x+y=y+x$	CS1	$\langle a\rangle = a$
A2	$x+(y+z)=(x+y)+z$	CS2	$\langle\langle x\rangle\rangle = \langle x\rangle$
A3	$x+x=x$	CS3'	$\langle\langle x\rangle \cdot y\rangle = \langle x\cdot y\rangle$
A4	$(x+y)\cdot z = x\cdot z + y\cdot z$	CS4'	$\langle x\cdot\langle y\rangle\rangle = \langle x\cdot y\rangle$
A5	$(x\cdot y)\cdot z = x\cdot(y\cdot z)$	CS5	$\langle x+y\rangle = \langle x\rangle + \langle y\rangle$
M1	$x\|\|y = x\lfloor\!\lfloor y + y\lfloor\!\lfloor x$		
M2	$a\lfloor\!\lfloor x = a\cdot x$	$\phi 1$	$\phi(a)=a$
M3	$a\cdot x\lfloor\!\lfloor y = a\cdot(x\|\|y)$	$\phi 2$	$\phi(\langle x\rangle)=\phi(x)$
LMC1	$\langle x\rangle\lfloor\!\lfloor y = \langle x\rangle\cdot y$	$\phi 3$	$\phi(x\cdot y)=\phi(x)\cdot\phi(y)$
LMC2	$\langle x\rangle\cdot y\lfloor\!\lfloor z = \langle x\rangle\cdot(y\|\|z)$	$\phi 4$	$\phi(x+y)=\phi(x)+\phi(y)$
M4	$(x+y)\lfloor\!\lfloor z = x\lfloor\!\lfloor z + y\lfloor\!\lfloor z$		

Table 3.2: Axioms of AMP

The equational theory of AMP is given in Table 3.2

Axioms LMC1 and LMC2 illustrate the property of indivisibility. In [BK84b] axioms CS3' and CS4' were missing, but they were derivable in the more general setting of AMP with tight multiplication (see Remark 3.2.4.17).

Axiom CS5 seems to be more problematic. We can consider the atomic actions as a subsort of the sort of processes (with its corresponding coercion). In this case, the right hand side of CS5 will have the wrong sort. Theoretically this is not a problem, since, after using the implicit coercion (i.e. the injection of the atomic actions into the set of processes) , we can say that the sort of the equation is process instead of atomic action.

Definition 3.2.1.1 (*Basic Term*). We define the set of basic terms in two steps, first without the critical sections, and then allowing at most one level of nesting of it. The set B_s of simple basic terms is defined inductively as follows:

1. $A \subseteq B_s$;
2. $a \in A$ and $t \in B_s$ implies $a \cdot t \in B_s$;
3. $t, s \in B_s$ implies $t + s \in B_s$.

Using B_s we define the set B of basic terms in AMP as follows

1. $A \subseteq B$;
2. $t \in B_s$ implies $\langle t \rangle \in B$.
3. $a \in A$ and $t \in B$ implies $a \cdot t \in B$;
4. $t \in B_s$ and $t' \in B$ implies $\langle t \rangle \cdot t' \in B$;
5. $t, s \in B$ implies $t + s \in B$.

□

Theorem 3.2.1.2 (*Elimination*). *Given any* AMP*-term t there exists a basic term t' such that, in the theory of* AMP*, $t = t'$ can be deduced. Moreover, for any term t there exists a simple basic term (without the critical section operator) such that $\phi(t) = t'$.*

Proof.

It is routine to prove that, when read from left to right, axioms A4-5, M1-4, LMC1-2, ϕ1-4, TM1-4, CS1-2,CS'3-4, CS5, define a convergent (modulo A1, A2, A3) rewriting system whose normal forms are basic terms.

□

3.2.2 Models

In this section we will introduce action rules for the operators of AMP. These rules can be used to define models of AMP. The first obvious model is the set of finite graphs generated by the terms. More interesting models can be obtained by the introduction of recursive specification and the standard unfolding rule for the operational semantics. This procedure is fairly standard, but some care is required to redefine notions such as guardedness for the new operators. We prefer to leave this matter for further research.

Two different sets of action rules are introduced, the first one keeps "at the same level" the activity inside or outside a critical section, whereas the second sees the critical sections as atomic actions, in the precise sense that they can be seen as labels of the action relations.

The tight-actions model

In this model we define three different relations

1. $\longrightarrow \surd \subseteq T \times A$
2. $\longrightarrow \subseteq T \times A \times T$, and
3. $\stackrel{:}{\longrightarrow} \subseteq T \times A \times T$

The meaning of these relations can be understood as follows:

1. $x \stackrel{a}{\longrightarrow} \surd$ means that x can terminate by performing an a-step
2. $x \stackrel{a}{\longrightarrow} x'$ means that x can evolve into x' by performing an a-step.
3. $x \stackrel{a:}{\longrightarrow} x'$ means that while evolving to x', the process x is executing a critical region and therefore any parallel component must wait until this section is completed.

Also, merely to shorten the presentation, we collapsed the last two relations into one by extending the set of labels to $A \cup A{:}$ were $A{:} = \{a{:} \mid a \in A\}$. The rules are given in table 3.3. We write $x \stackrel{a*}{\longrightarrow} x'$ to indicate that either $x \stackrel{a}{\longrightarrow} x'$ or $x \stackrel{a:}{\longrightarrow} x'$

Definition 3.2.2.3 (*Bisimulation*). We say that $R \subseteq T \times T$ is a *bisimulation* if, xRy implies, for all $a \in A$, $m \in A \cup A{:}$,

1. $x \stackrel{m}{\longrightarrow} x' \Longrightarrow \exists y' \in T. y \stackrel{m}{\longrightarrow} y'$ and $x'Ry'$;
2. $y \stackrel{m}{\longrightarrow} y' \Longrightarrow \exists x' \in T. x \stackrel{m}{\longrightarrow} x'$ and $x'Ry'$; and
3. $x \stackrel{a}{\longrightarrow} \surd \iff y \stackrel{a}{\longrightarrow} \surd$

We say that x and y are bisimilar (notation $x \leftrightarrows y$) if there exists a bisimulation which relates them. □

$$a \xrightarrow{a} \surd$$

$$\frac{x \xrightarrow{a*} x'}{\begin{array}{c} x+y \xrightarrow{a*} x' \\ y+x \xrightarrow{a*} x' \\ x \cdot y \xrightarrow{a*} x' \cdot y \end{array}}$$

$$\frac{x \xrightarrow{a} \surd}{\begin{array}{c} x+y \xrightarrow{a} \surd \\ y+x \xrightarrow{a} \surd \\ x \cdot y \xrightarrow{a} y \end{array}}$$

$$\frac{x \xrightarrow{a*} x'}{\begin{array}{c} \langle x \rangle \xrightarrow{a:} \langle x' \rangle \\ \phi(x) \xrightarrow{a} \phi(x') \end{array}}$$

$$\frac{x \xrightarrow{a} \surd}{\begin{array}{c} \langle x \rangle \xrightarrow{a} \surd \\ \phi(x) \xrightarrow{a} \surd \end{array}}$$

$$\frac{x \xrightarrow{a} x'}{\begin{array}{c} x||y \xrightarrow{a} x'||y \\ y||x \xrightarrow{a} y||x' \\ x \between y \xrightarrow{a} x'||y \end{array}}$$

$$\frac{x \xrightarrow{a:} x'}{\begin{array}{c} x||y \xrightarrow{a:} x' \between y \\ y||x \xrightarrow{a:} x' \between y \\ x \between y \xrightarrow{a:} x' \between y \end{array}}$$

$$\frac{x \xrightarrow{a} \surd}{\begin{array}{c} x||y \xrightarrow{a} y \\ y||x \xrightarrow{a} y \\ x \between y \xrightarrow{a} y \end{array}}$$

Table 3.3: Operational semantics for AMP($a \in A$)

The non-elementary action model

This model represents the idea that a critical section can be seen as an atomic action. Axioms CS1-4' say that it makes no sense to talk of atomicity inside an action that is already atomic. This will be reflected in the operational semantics, by the fact that we will use simple processes (i.e. without critical sections) as possible labels of the transitions. Moreover, axiom CS5 suggests that the initial choices of a critical region can be done before this section is actually entered. We choose to follow this idea in the operational semantics.

We will work in the remaining of this section within the model whose elements are closed terms modulo some appropriate equivalence. Given a set of atomic actions A, the set of non-elementary actions A_N is defined as the set of terms of the form $a \cdot x$ without any occurrence of the critical section operator. The action relations

1. $\longrightarrow \surd \subseteq T \times A_N$, and
2. $\longrightarrow \subseteq T \times A_N \times T$.

are given in table 3.4, where $a \in A$ and $z \in A_N$.

We want to identify atomic actions that have the same behaviour. In the case of this model this is easy to do since no critical section appears in the terms used as labels. We use then bisimulation equivalence to identify atomic actions. Two processes will be equal when they are bisimilar modulo equivalence of atomic actions. More precisely, when there exists a non-elementary bisimulation as defined below.

$$a \xrightarrow{a} \surd \qquad \frac{x \xrightarrow{z} x'}{\begin{array}{l} x+y \xrightarrow{z} x' \\ y+x \xrightarrow{z} x' \\ x \cdot y \xrightarrow{z} x' \cdot y \end{array}} \qquad \frac{x \xrightarrow{z} \surd}{\begin{array}{l} x+y \xrightarrow{z} \surd \\ y+x \xrightarrow{z} \surd \\ x \cdot y \xrightarrow{z} y \end{array}}$$

$$\frac{\phi(x) \xrightarrow{a} \phi(x')}{\langle x \rangle \xrightarrow{a.\phi(x')} \surd} \qquad \frac{\phi(x) \xrightarrow{a} \surd}{\langle x \rangle \xrightarrow{a} \surd} \qquad \frac{x \xrightarrow{z} x' \;\; z \xrightarrow{a} z'}{\phi(x) \xrightarrow{a} z' \cdot \phi(x')}$$

$$\frac{x \xrightarrow{z} x' \;\; z \xrightarrow{a} \surd}{\phi(x) \xrightarrow{a} \phi(x')} \qquad \frac{x \xrightarrow{z} \surd \;\; z \xrightarrow{a} z'}{\phi(x) \xrightarrow{a} z'} \qquad \frac{x \xrightarrow{z} \surd \;\; z \xrightarrow{a} \surd}{\phi(x) \xrightarrow{a} \surd}$$

$$\frac{x \xrightarrow{z} x'}{\begin{array}{l} x||y \xrightarrow{z} x'||y \\ y||x \xrightarrow{z} y||x' \\ x \between y \xrightarrow{z} x'||y \end{array}} \qquad \frac{x \xrightarrow{z} \surd}{\begin{array}{l} x||y \xrightarrow{z} y \\ y||x \xrightarrow{z} y \\ x \between y \xrightarrow{z} y \end{array}}$$

Table 3.4: Operational semantics for AMP with non-elementary actions

Definition 3.2.2.4 (*Non-elementary Bisimulation*). We say that $R \subseteq T \times T$ is a *non-elementary bisimulation* if, xRy implies, for all $z \in A_N$,

1. $x \xrightarrow{z} x' \Longrightarrow \exists y' \in T. \exists z' \in A_N. y \xrightarrow{z'} y'$ and $x'Ry'$ and $z \leftrightarrows z'$;
2. $y \xrightarrow{z} y' \Longrightarrow \exists x' \in T. \exists z' \in A_N. x \xrightarrow{z'} x'$ and $x'Ry'$ and $z \leftrightarrows z'$;
3. $x \xrightarrow{z} \surd \Longrightarrow \exists z' \in A_N. y \xrightarrow{z'} \surd$ and $z \leftrightarrows z'$; and
4. $y \xrightarrow{z} \surd \Longrightarrow \exists z' \in A_N. x \xrightarrow{z'} \surd$ and $z \leftrightarrows z'$;

In this definition, $\leftrightarrows$ represents elementary bisimulation (see definition 2.2.2.12), when only elements of A can occur as transition labels. We say that x and y are non-elementary bisimilar (notation $x \approx y$) if there exists a non-elementary bisimulation which relates them. □

3.2.3 Comparison between the different models

Many different models can be constructed from the operational semantics given above, according to the choice of a different universe of processes. In this section we establish a relation between both operational semantics.

Remark 3.2.3.5.
We write

$$TA \models x \xrightarrow{a} x'$$

to indicate that the proposition $x \xrightarrow{a} x'$ is derivable using the rules of table 3.3. Analogously, we notate with

$$NEA \models x \xrightarrow{a} x'$$

the fact that $x \xrightarrow{a} x'$ is derivable using the rules of table 3.4. □

Definition 3.2.3.6. The set of BPA-terms is defined as the set of terms that are equal to a term without any occurrence of the tight region operator.
The set of tight terms is defined as the terms equal to a term of the form $\langle t \rangle$ for any term t. □

Remark 3.2.3.7. Note that if t is a tight term, then $\langle t \rangle = t$. □

Remark 3.2.3.8. Note that the term $\langle ab \rangle \cdot \langle cd \rangle$ is neither a BPA nor a tight term. □

Lemma 3.2.3.9. *For closed terms t it holds that $\langle \phi(t) \rangle = \langle t \rangle$*

Proof. Straightforward structural induction on basic terms. □

Lemma 3.2.3.10. *If there are terms t, t' and an atomic action a such that*

$$TA \models t \xrightarrow{a:} t'$$

then there exists a term t'' and tight term u such that $t' = u \cdot t''$.

Proof. By straightforward induction on the derivation of $t \xrightarrow{a:} t'$ □

Lemma 3.2.3.11.

- *for any term $t, a \in A$*

$$NEA \models t \xrightarrow{a} t' \text{ iff } TA \models t \xrightarrow{a} t'$$

$$NEA \models t \xrightarrow{a} \surd \text{ iff } TA \models t \xrightarrow{a} \surd$$

- *For any terms t, t', tight term u and atomic action a it holds that*

$$NEA \models t \xrightarrow{a \cdot \phi(u)} t' \text{ iff } TA \models t \xrightarrow{a:} u \cdot t'$$

$$NEA \models t \xrightarrow{a \cdot \phi(u)} \surd \text{ iff } TA \models t \xrightarrow{a:} u$$

Proof. By induction on the proof of the transition. We show only the more complicated cases.

First we take as hypothesis that the transition is obtained in NEA and find the proof in TA.

- Suppose that the last rule applied was

$$\frac{\phi(t) \xrightarrow{a} \phi(u)}{\langle t \rangle \xrightarrow{a \cdot \phi(u)} \surd}$$

then, by induction hypothesis we obtain that

$$TA \models \phi(t) \xrightarrow{a} \phi(u)$$

and applying the corresponding rule we obtain that

$$TA \models \langle \phi(t) \rangle \xrightarrow{a:} \langle \phi(u) \rangle$$

or equivalently

$$TA \models \langle t \rangle \xrightarrow{a:} u$$

- Suppose that the last rule applied was

$$\frac{t \xrightarrow{a \cdot \phi(u)} t' \quad a \cdot \phi(u) \xrightarrow{a} \phi(u)}{\phi(x) \xrightarrow{a} \phi(u) \cdot \phi(t')}$$

we get by induction hypothesis that

$$TA \models t \xrightarrow{a:} u \cdot t'$$

and applying the corresponding rule we obtain that

$$TA \models \phi(t) \xrightarrow{a} \phi(u \cdot t')$$

- Suppose that the last rule applied was

$$\frac{t \xrightarrow{a \cdot \phi(u)} t'}{t||v \xrightarrow{a \cdot \phi(u)} t'||v}$$

By induction hypothesis we obtain

$$TA \models t \xrightarrow{a:} u \cdot t'$$

and applying the rule for left merge we obtain

$$TA \models t||v \xrightarrow{a:} (u \cdot t') \between v$$

and since u is a tight term, it equals (by axiom LMC1)

$$TA \models t||v \xrightarrow{a:} u \cdot (t'||v)$$

Now we suppose we have a transition in TA and we obtain the corresponding one in NEA.

- Suppose that the last rule applied was

$$\frac{t \xrightarrow{a} t'}{\langle t \rangle \xrightarrow{a:} \langle t' \rangle}$$

by induction hypothesis we have that

$$NEA \models t \xrightarrow{a} t'$$

and applying the corresponding rules we obtain

$$NEA \models \phi(t) \xrightarrow{a} (t')$$

$$NEA \models \langle t \rangle \xrightarrow{a \cdot \phi(t')} \surd$$

and this is equal to

$$NEA \models \langle t \rangle \xrightarrow{a \cdot \phi(\langle t' \rangle)} \surd$$

- Suppose that the last rule applied was

$$\frac{t \xrightarrow{a:} t'}{\langle t \rangle \xrightarrow{a:} \langle t' \rangle}$$

by induction hypothesis we have that

$$NEA \models t \xrightarrow{a \cdot \phi(t')} \surd$$

now, given the fact that $a \cdot \phi(t') \xrightarrow{a} \phi(t')$ we can apply the corresponding rule and derive

$$NEA \models \phi(t) \xrightarrow{a} \phi(t')$$

and from this

$$NEA \models \langle t \rangle \xrightarrow{a \cdot \phi(t)} \surd$$

□

Proposition 3.2.3.12. *If* $NEA \models t \xrightarrow{a \cdot \phi(v)} t'$, *then there exists a tight term* u *such that* $\phi(v) = \phi(u)$

Proof. Take, for example, $u = \langle v \rangle$. □

Fact 3.2.3.13. The following facts can be proved by simple inspection of the rules. Let $u.u'$ be tight terms, then in NEA the following properties hold:

- if $u \xrightarrow{a:} v$ then v is also a tight term.
- if u is a closed term then there exists a sequence $\sigma \in (A:)^*A$, i.e. every action except the last is tight, such that $u \xrightarrow{\sigma} \surd$.
- $u \cdot t \xrightarrow{a:} z$ if and only if $\exists u', u \xrightarrow{a:} u'$ and $z = u' \cdot t$.
- $u \cdot t \xrightarrow{a} z$ if and only if $u \xrightarrow{a} \surd$ and $z = t$.

□

Lemma 3.2.3.14. *if u, u' are tight terms, then in the model of tight actions it holds for any pair of terms t, t' that*

$$u \leftrightarrows u' \wedge t \leftrightarrows t' \Leftrightarrow u \cdot t \leftrightarrows u' \cdot t'$$

Proof. It is immediate that $u \leftrightarrows u' \wedge t \leftrightarrows t' \Rightarrow u \cdot t \leftrightarrows u' \cdot t'$.
To prove the other implication we take a bisimulation

$$R : u \cdot t \leftrightarrows u' \cdot t'$$

We first show that tRt'. Take a (tight) term v, a sequence of tight actions σ and an action a, such that

$$u \cdot t \xrightarrow{\sigma} v \cdot t \xrightarrow{a} t$$

then, there exist z, z' such that

$$u' \cdot t' \xrightarrow{\sigma} z \xrightarrow{a} z'$$

and $(v \cdot t)Rz$ and tRz'.
Applying successively the last two items of the previous fact we obtain that there exists a (tight) term w such that $z = w \cdot t'$ and $z' = t'$, and thus tRt'.
In order to show that $u \leftrightarrows u'$ we construct another bisimulation S in the following way:

$$S = \{(v, v') | (v \cdot t)R(v' \cdot t')\}$$

It is immediate that uSu'. It remains to show that S is indeed a bisimulation. Suppose that $v \xrightarrow{a:} w$, then $v \cdot t \xrightarrow{a:} w \cdot t$. There exists z such that $v' \cdot t' \xrightarrow{a:} z$ and $(w \cdot t)Rz$, since R is a bisimulation. From the previous fact it follows that there is a term w such that $z = w' \cdot t'$ and then by definition of S we obtain that wSw'.
The easier case of $v \xrightarrow{a} \surd$ is left to the reader.

□

Theorem 3.2.3.15. *Two closed terms are non-elementary bisimilar (using the system NEA) if and only if they are bisimilar using the rules of TA.*

Proof. Suppose that $R : t \leftrightarrows t'$. We construct a bisimulation in TA beginning with R in the following way: If $t \xrightarrow{a \cdot \phi(u)} s$ and $t' \xrightarrow{a \cdot \phi(u')} s'$ then take a bisimulation S such that $\phi(u) S \phi(u')$ and add the set of pairs of tight terms

$$\{(v \cdot s, v' \cdot s') | \phi(v) S \phi(v')\}$$

.

The converse is easier, since we can restrict the bisimulation to the smaller set of terms reachable with transitions generated by NEA and use the previous lemma to prove that the non-elementary labels are bisimilar.

□

3.2.4 AMP with tight multiplication

In this section we add the tight multiplication operator introduced in [BK84b]. The meaning of the new operator is a non-interruptible sequential composition; that is, $x : y$ is a process that first executes x, and upon completion of x, executes y with the additional requirement that immediately after the execution of x, one of the initial actions of y must be executed without any other interleaved action (coming from some parallel component).

The tight multiplication operator seems to be more primitive than the critical section. The reasons in favour of this claim are the fact that the critical section operator is definable in terms of the tight multiplication for closed terms, and the simplicity of the tight multiplication. Moreover, we will see in this chapter that when we add new operations to the algebras, the presence of the tight multiplication becomes more and more important in order to obtain simple axiomatizations and action rules. We choose to introduce the critical sections first because they are more intuitive and they admit, unlike tight multiplication, the model of non-elementary actions.

The axioms for AMP(:) are given in Table 3.5.

Remark 3.2.4.16. We assume in this thesis that the operators $\cdot$ and $:$ bind stronger that $|$, $||$, and $\lfloor\!\lfloor$. □

Remark 3.2.4.17. The axioms CS3' and CS4' are derivable from the axioms of table 3.5. For example:

$$\langle\langle x\rangle \cdot y\rangle = \langle\langle x\rangle\rangle : \langle y\rangle = \langle x\rangle : \langle y\rangle = \langle x \cdot y\rangle$$

□

Definition 3.2.4.18 (*Basic terms*). The set B of *basic terms* is defined inductively as follows

1. $A \subseteq B$;

A1	$x+y=y+x$	TM1	$(x:y):z=x:(y:z)$
A2	$x+(y+z)=(x+y)+z$	TM2	$(x:y)\cdot z=x:(y\cdot z)$
A3	$x+x=x$	TM3	$(x\cdot y):z=x\cdot(y:z)$
A4	$(x+y)\cdot z=x\cdot z+y\cdot z$	TM4	$(x+y):z=x:z+y:z$
A5	$(x\cdot y)\cdot z=x\cdot(y\cdot z)$		
M1	$x\|\|y=x\between y+y\between x$	CS1	$\langle a\rangle=a$
M2	$a\between x=a\cdot x$	CS2	$\langle\langle x\rangle\rangle=\langle x\rangle$
M3	$a\cdot x\between y=a\cdot(x\|\|y)$	CS3	$\langle x\cdot y\rangle=\langle x\rangle:\langle y\rangle$
LMT1	$a:x\between y=a:(x\between y)$	CS4	$\langle x:y\rangle=\langle x\rangle:\langle y\rangle$
M4	$(x+y)\between z=x\between z+y\between z$	CS5	$\langle x+y\rangle=\langle x\rangle+\langle y\rangle$
ϕ1	$\phi(a)=a$	ϕ3	$\phi(x\cdot y)=\phi(x)\cdot\phi(y)$
ϕ2	$\phi(x:y)=\phi(x)\cdot\phi(y)$	ϕ4	$\phi(x+y)=\phi(x)+\phi(y)$

Table 3.5: Axioms of AMP(:)

2. $a\in A$ and $t\in B$ implies $a\cdot t\in B$;
3. $a\in A$ and $t\in B$ implies $a:t\in B$;
4. $t,s\in B$ implies $t+s\in B$.

Two subsets of B will be defined:

- The set B_s of *simple basic terms* defined using the same set of rules as B except rule 3,
- The set B_t of *tight basic terms* defined using the same set of rules as B except rule 2.

□

Theorem 3.2.4.19 (*Elimination*). *Given any* AMP(:) *term* t *it follows that:*

1. $\exists t'\in B$ *such that* AMP(:) $\vdash t=t'$
2. $\exists t'\in B_s$ *such that* AMP(:) $\vdash \phi(t)=t'$
3. $\exists t'\in B_t$ *such that* AMP(:) $\vdash \langle t\rangle=t'$

Proof. It is routine to prove that, when read from left to right, axioms A4-5, M1-4, LMT1, ϕ1-4, TM1-4, CS1-5, define a convergent (modulo A1, A2, A3) rewriting system whose normal forms are basic terms. Furthermore, it is immediate that the normal form of $\phi(t)$ contains no tight multiplication and the normal form of $\langle t\rangle$ no sequential composition. □

The tight-actions model

The operational rules for AMP(:) are defined by the rules given in Table 3.3 plus the new rules given in Table 3.6

$$\frac{x \xrightarrow{a*} x'}{x : y \xrightarrow{a*} x' : y} \qquad \frac{x \xrightarrow{a} \surd}{x : y \xrightarrow{a:} y}$$

Table 3.6: Operational semantics for AMP(:)

Remark 3.2.4.20. Not every term of AMP(:) is equivalent to one in AMP. This fact rules out the model of non elementary actions, as the following example illustrates:

$$a : (b \cdot c + d \cdot e)$$

after action a is performed the process is in a critical region in which it executes either b or d and complete this critical activity, but the choice between them is not made until a is performed. □

3.3 Multiactions and Critical Sections

In the previous section we claimed that one can see the algebras with a critical section operator as a two sorted algebra with one sort of processes and one of atomic actions. In simpler algebras like BPA this division into two different sorts is implicit and the only operation for atomic actions is the inclusion into the set of processes.

Introducing communication in a context with critical sections seems to have some problems. It is not clear what should be the result of a communication between an atomic action and a critical section. This problem was studied in [BK84b, GMM90, Old87], where different synchronization mechanisms were used. We choose here to avoid all these problems by not using synchronous communication. We take from [Bou89] the idea that communication can be defined in terms of simpler principles such as shared variables and non-interruptible processes. It seems that the best way to describe this idea is through the notion of *multiactions*. A multiaction is a set (or, strictly speaking, a multiset) of actions that is executed "during the same period of time". The intuition behind the idea of multiaction becomes more difficult to grasp because here we cannot assume that atomic actions take no time to execute (since a complex process can be seen as an atomic action). When the atomic actions are timeless, then a multiaction can be seen as something very close to a communication, at least from an algebraic point of view. Here, we will present two different approaches, trying to keep in each one a different aspect of the timeless multiactions. The system AMMP will preserve

the idea of *step* usually associated with multiactions paying the price of not preserving totally the branching structure inside a multiaction; whereas AMBP(:) will preserve this structure but it is not possible anymore to preserve an idea of steps of multisets of timeless actions.

Both algebras will share the following signature: We consider a new sort: the sort of *multiactions* **M**. All constants belonging to the set A will be considered now as multiactions. Furthermore, the sort **M** will be considered as a subsort of the sort **P** of processes. The new operators are:

$$\begin{array}{ll} \langle\,\rangle & : \mathbf{P} \to \mathbf{M} \\ | & : \mathbf{M} \times \mathbf{M} \to \mathbf{M} \\ & \\ +, \cdot, \|, \between, | & : \mathbf{P} \times \mathbf{P} \to \mathbf{P} \\ \phi & : \mathbf{P} \to \mathbf{P} \end{array}$$

The coercion from multiactions to processes is used implicitly. The new operator | is used to define a multiaction. It will have two different interpretations in AMMP and in AMBP(:). When used between processes it will denote a parallel composition where the first action will be a multiaction.

3.3.1 Critical sections in multiactions interpreted as steps

The axioms for AMMP are axioms A1–A5, CS1–CS5 and ϕ1–ϕ4 given in Table 3.2 and those given in Table 3.7.

CM1	$x\|\|y = x \between y + y \between x + x\|y$	MA1	$\langle m\rangle = m$
LM1	$m \between x = m \cdot x$	LM3	$(x+y) \between z = x \between z + y \between z$
LM2	$m \cdot x \between y = m \cdot (x\|\|y)$		
MM1	$m\|n \cdot x = (m\|n) \cdot x$	MA1	$m\|n = n\|m$
MM2	$m \cdot x\|n = (m\|n) \cdot x$	MA2	$a\|b = \langle a \cdot b + b \cdot a\rangle$
MM3	$m \cdot x\|n \cdot y = (m\|n) \cdot (x\|\|y)$	MA3	$\langle a \cdot m\rangle\|b = \langle a \cdot (m\|b) + b \cdot a \cdot m\rangle$
MM4	$(x+y)\|z = x\|z + y\|z$	MA4	$\langle a \cdot m\rangle\|\langle b \cdot n\rangle$
MM5	$x\|(y+z) = x\|y + x\|z$		$= \langle a \cdot (m\|\langle b \cdot n\rangle) + b \cdot (\langle a \cdot m\rangle\|n)\rangle$

Table 3.7: Axioms of AMMP ($a, b \in A$, $m, n \in \mathbf{M}$)

Axiom MA2 (and MA3-4) expresses that one can identify multiactions that behave in the same way.

The theory AMMP preserves the distributivity of the multiaction merge with respect to the choice. This idea is present in most of the works on process algebra; most of them

regard a synchronization (or a multiaction) as a bag of simple actions that is also called "step" (see, for instance, [Mil83, BB93]). In this context it implies that the choices are done at the same time inside a multiaction as the following example shows:

$$a|\langle b+c\rangle = \langle a\cdot b\rangle + \langle a\cdot c\rangle + \langle b\cdot a\rangle + \langle c\cdot a\rangle$$

This means that when the process executes a the choice whether b or c will be the next (sub)action is already done.

3.3.2 Process algebra with multiactions and tight multiplication

As before, we extend AMMP with a tight multiplication operator. The axioms for AMMP(:) are given by axioms A1–A5, TM1–TM4, CS1–CS4, $\phi 1$–$\phi 4$ in Table 3.5 and the ones in Table 3.8

Some axioms require explanation. For example axiom CTM4 says that if one side of a multiaction terminates, then the other side must proceed until termination. Axiom CTM9 could be written as well as

$$a\cdot x|b\cdot y = a:b\cdot(x||y) + b:a\cdot(x||y)$$

what means that if both sides can terminate, then they can do it in any order.

M1	$x\|\|y = x\between y + y\between x + x\|y$	MA1	$\langle m\rangle = m$
		CTM1	$a\|b = a:b + b:a$
LM1	$a\between x = a\cdot x$	CTM2	$a\|b:x = a:(b:x) + b:(a\|x)$
LM2	$a\cdot x\between y = a\cdot(x\|\|y)$	CTM3	$a:x\|b = a:(x\|b) + b:(a:x)$
LMT2	$a:x\between y = a:(x\between y)$	CTM4	$a:x\|b\cdot y = a:(x\|b\cdot y) + b:(a:x\between y)$
LM3	$(x+y)\between z = x\between z + y\between z$	CTM5	$a:x\|b:y = a:(x\|b:y) + b:(a:x\|y)$
		CTM6	$a\|b\cdot x = a:(b\cdot x) + b:(a\between x)$
		CTM7	$a\cdot x\|b = a:(b\between x) + b:(a\cdot x)$
CM4	$(x+y)\|z = x\|z + y\|z$	CTM8	$a\cdot x\|b:y = a:(b:y\between x) + b:(a\cdot x\|y)$
CM5	$x\|(y+z) = x\|y + x\|z$	CTM9	$a\cdot x\|b\cdot y = a:(b\cdot y\between x) + b:(a\cdot x\between y$

Table 3.8: Axioms of AMMP(:) $(a, b \in A)$

Remark 3.3.2.1. Axioms MA2-4 are derivable in AMMP(:). For example

$$\begin{aligned}\langle a\cdot m\rangle|b &= a:\langle m\rangle|b = a:m|b = a:(m|b) + b:(a:m) = \\ &= a:\langle(m|b)\rangle + b:\langle a\cdot m\rangle = \langle a\cdot(m|b) + b\cdot a\cdot m\rangle\end{aligned}$$

□

Theorem 3.3.2.2 (*Elimination*). *Given any* AMMP(:) *term* t *it follows that:*

1. $\exists t' \in B$ *such that* AMMP(:) $\vdash t = t'$
2. $\exists t' \in B_s$ *such that* AMMP(:) $\vdash \phi(t) = t'$
3. $\exists t' \in B_t$ *such that* AMMP(:) $\vdash \langle t \rangle = t'$

with B, B_s *and* B_t *as defined in 3.2.4.18.*

3.3.3 The tight-action model

The operational rules for AMMP(:) are those given in Table 3.3 + Table 3.6 + Table 3.9.

$$\frac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{\begin{array}{l} x||y \xrightarrow{a:} b \cdot y' \between x' \\ y||x \xrightarrow{a:} b \cdot y' \between x' \\ x|y \xrightarrow{a:} b \cdot y' \between x' \\ y|x \xrightarrow{a:} b \cdot y' \between x' \end{array}} \qquad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{b:} y'}{\begin{array}{l} x||y \xrightarrow{a:} b : y' \between x' \\ y||x \xrightarrow{a:} b : y' \between x' \\ x|y \xrightarrow{a:} b : y' \between x' \\ y|x \xrightarrow{a:} b : y' \between x' \end{array}} \qquad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{b} \surd}{\begin{array}{l} x||y \xrightarrow{a:} b \between x' \\ y||x \xrightarrow{a:} b \between x' \\ x|y \xrightarrow{a:} b \between x' \\ y|x \xrightarrow{a:} b \between x' \end{array}}$$

$$\frac{x \xrightarrow{a:} x' \quad y \xrightarrow{b} y'}{\begin{array}{l} x||y \xrightarrow{a:} x'|b \cdot y' \\ y||x \xrightarrow{a:} b \cdot y'|x' \\ x|y \xrightarrow{a:} x'|b \cdot y' \\ y|x \xrightarrow{a:} b \cdot y'|x' \end{array}} \qquad \frac{x \xrightarrow{a:} x' \quad y \xrightarrow{b:} y'}{\begin{array}{l} x||y \xrightarrow{a:} x'|b : y' \\ y||x \xrightarrow{a:} b : y'|x' \\ x|y \xrightarrow{a:} x'|b : y' \\ y|x \xrightarrow{a:} b : y'|x' \end{array}} \qquad \frac{x \xrightarrow{a:} x' \quad y \xrightarrow{b} \surd}{\begin{array}{l} x||y \xrightarrow{a:} x'|b \\ y||x \xrightarrow{a:} b|x' \\ x|y \xrightarrow{a:} x'|b \\ y|x \xrightarrow{a:} b|x' \end{array}}$$

$$\frac{x \xrightarrow{a} \surd \quad y \xrightarrow{b} y'}{\begin{array}{l} x||y \xrightarrow{a:} b \cdot y' \\ y||x \xrightarrow{a:} b \cdot y' \\ x|y \xrightarrow{a:} b \cdot y' \\ y|x \xrightarrow{a:} b \cdot y' \end{array}} \qquad \frac{x \xrightarrow{a} \surd \quad y \xrightarrow{b:} y'}{\begin{array}{l} x||y \xrightarrow{a:} b : y' \\ y||x \xrightarrow{a:} b : y' \\ x|y \xrightarrow{a:} b : y' \\ y|x \xrightarrow{a:} b : y' \end{array}} \qquad \frac{x \xrightarrow{a} \surd \quad y \xrightarrow{b} \surd}{\begin{array}{l} x||y \xrightarrow{a:} b \\ y||x \xrightarrow{a:} b \\ x|y \xrightarrow{a:} b \\ y|x \xrightarrow{a:} b \end{array}}$$

Table 3.9: Operational semantics for AMMP(:)

Remark 3.3.3.3. The rule that says that:

$$\frac{x \xrightarrow{a} x' \quad y \xrightarrow{b:} y'}{x|y \xrightarrow{a:} b : y' \between x'}$$

is more difficult to present in a context without tight multiplication (this rule is still necessary in AMMP), and requires the introduction of auxiliary operators. It is remarkable

that the tight multiplication is not strictly needed in the axiomatization. This mismatch appears owing to the fact that the rules should solve the choices. Therefore, we need to know the initial actions of the other components of a multiaction. □

The parallel composition introduced here is not associative, as the following easy example shows.

Example 3.3.3.4. In AMMP it cannot be proved that the following two terms are equal

$$a||(b||c)$$

$$(a||b)||c$$

nor even the following two

$$a|(b|c)$$

$$(a|b)|c$$

We leave the proof to the reader (see also example 3.3.4.8). □

3.3.4 Another look at tight multiplication and process algebras

The system AMMP has some drawbacks which are a consequence of the fact that it does not preserve the branching structure inside the multiactions. One of the most obvious drawbacks is the complexity of the operational semantics and of the graph model (not defined here). Another problem is that it is not easy to extend AMMP with new operators, like deadlock or non-interruptible critical sections, as we will see later.

In order to solve these problems we introduce a new system in which the branching structure is preserved inside the multiactions. The price that we pay is that axioms CM4 and CM5 are no longer valid. This does not seem unreasonable if we think of the operator | as a multiaction merge. We lost the notion of step as it is intended in step bisimulation. This new axiomatization does not extend ACP of [BB93].

We add a new auxiliary operator $\lfloor\!\_$ to axiomatize the multiaction merge (it can be called left multiaction merge)

The axioms set for AMBP(:) is given by axioms A1–A5, TM1–TM4, CS1–CS4 and $\phi 1$–$\phi 4$ given in Table 3.5 and those in Table 3.10.

Theorem 3.3.4.5 (*Elimination*). *Given any* AMBP(:) *term* t *it follows that:*

1. $\exists t' \in B$ *such that* AMBP(:) $\vdash t = t'$

M1	$x\|\|y = x \between y + y \between x + x\|y$		
		CMB1	$x\|y = x \between y + y \between x$
LM1	$a \between x = a \cdot x$	CMB2	$a \between x = a : x$
LM2	$a \cdot x \between y = a \cdot (x\|\|y)$	CMB3	$a : x \between y = a : (x\|y)$
LMT2	$a : x \between y = a : (x \between y)$	CMB4	$a \cdot x \between y = a : (y \between x)$
LM3	$(x + y) \between z = x \between z + y \between z$	CMB5	$(x + y) \between z = x \between z + y \between z$

Table 3.10: Axioms of AMBP(:) $(a, b \in A)$

2. $\exists t' \in B_s$ *such that* AMBP(:) $\vdash \phi(t) = t'$

3. $\exists t' \in B_t$ *such that* AMBP(:) $\vdash \langle t \rangle = t'$

Remark 3.3.4.6. Axioms CTM1-9 are derivable in AMBP(:). For example

$$\begin{aligned} a : x|b \cdot y &= a : x \between b \cdot y + b \cdot y \between a : x \\ &= a : (x|b \cdot y) + b : (a : x \between y) \end{aligned}$$

□

Proposition 3.3.4.7. *The theories of* AMMP(:) *and* AMBP(:) *are incomparable, i.e. there is an equality valid in* AMMP(:) *but not in* AMBP(:) *and vice versa.*

Proof. The respective equalities are

$$(a + b)|c = a|c + b|c$$

and

$$(a + b)|c = a : c + b : c + c : (a + b)$$

□

The system AMBP(:) does not satisfy all the axioms of standard concurrency. In particular, the merge is not associative, as the following example demonstrates:

Example 3.3.4.8.

$$\begin{aligned} a||(b||c) &= a \between (b||c) + (b||c) \between a + a|(b||c) \\ &= a \cdot (b||c) + (b \between c) \between a + (c \between b) \between a + (b|c) \between a + a \between (b||c) + (b||c) \between a \\ &= a \cdot (b||c) + (b \cdot c) \between a + (c \cdot b) \between a + (b \between c) \between a \\ &+ (c \between b) \between a + a : (b||c) + (b \between c) \between a + (c \between b) \between a + (b|c) \between a \\ &= a \cdot (b||c) + b \cdot (c||a) + c \cdot (b||a) + b : c \cdot a \\ &+ c : b \cdot a + a : (b||c) + (b \cdot c) \between a + (c \cdot b) \between a + b : c : a + c : b : a \\ &= a \cdot (b||c) + b \cdot (c||a) + c \cdot (b||a) + b : c \cdot a \\ &+ c : b \cdot a + a : (b||c) + b : a \cdot c + c : a \cdot b + b : c : a + c : b : a \end{aligned}$$

in the same way we can deduce that

$$\begin{aligned}(a||b)||c &= c\cdot(b||a)+b\cdot(c||a)+a\cdot(b||c)+b:a\cdot c\\ &+ a:b\cdot c+c:(b||a)+b:c\cdot a+a:c\cdot b+b:a:c+a:b:c\end{aligned}$$

and it is immediate that $a:(b||c)$ is a summand of the first term but not of the second. □

3.3.5 The tight-action model

The operational rules for AMBP(:) are those given in Table 3.3 + Table 3.6 + Table 3.11.

$\dfrac{x \xrightarrow{a} x'}{\begin{array}{c} x\|\|y \xrightarrow{a:} y \between x' \\ y\|\|x \xrightarrow{a:} y \between x' \\ x\|y \xrightarrow{a:} y \between x' \\ y\|x \xrightarrow{a:} y \between x' \\ x \between y \xrightarrow{a:} y \between x' \end{array}}$	$\dfrac{x \xrightarrow{a:} x'}{\begin{array}{c} x\|\|y \xrightarrow{a:} x'\|y \\ y\|\|x \xrightarrow{a:} y\|x' \\ x\|y \xrightarrow{a:} x'\|y \\ y\|x \xrightarrow{a:} y\|x' \\ x \between y \xrightarrow{a:} x'\|y \end{array}}$	$\dfrac{x \xrightarrow{a} \surd}{\begin{array}{c} x\|\|y \xrightarrow{a:} y \\ y\|\|x \xrightarrow{a:} y \\ x\|y \xrightarrow{a:} y \\ y\|x \xrightarrow{a:} y \\ x \between y \xrightarrow{a:} y \end{array}}$

Table 3.11: Operational semantics for AMBP(:)

Both, AMBP(:) and AMMP(:) have as a drawback that the parallel composition is not associative. A possible solution for this problem is outlined in the conclusions of this chapter. The system AMBP(:) preserves completely the branching structure of the critical regions but the price that it has to pay for that is the non-distributivity of the multiaction merge with respect to the choice. This last property holds in ACP.

3.4 Multiactions, Critical Sections and Atomicity

3.4.1 Multiactions, critical sections and atomic processes

We add a new operator which makes a process atomic: [ ]. This new operator would behave in the same way as ⟨ ⟩ in a context without multiactions. This can be seen from the axioms for AMBP(:)+At presented in Table 3.12 in addition to the axioms of AMBP(:) (Table 3.10). The difference arises precisely in the absence of the axioms equivalent to CS1-5 of table 3.2 for the new operator. This means that a process of the form $[x]$ cannot be decomposed any further, even in a multiaction context. This models the idea of non-interference of an atomic section.

Definition 3.4.1.1 (*Basic Term*). The set B_s of simple basic terms is defined as in 3.2.4.18. We define the set B of basic terms in AMBP(:)+At as follows:

$$[\,] : \mathbf{P} \to \mathbf{M}$$

AP1	$[a] = a$	AP4	$[x : y] = [x \cdot y]$
AP2	$[[x]] = [x]$	AP5	$[[x] \cdot y] = [x \cdot y]$
AP3	$[x + y] = [x] + [y]$	AP6	$[x \cdot [y]] = [x \cdot y]$

Table 3.12: Additional axioms for AMBP(:)+At ($a \in A$)

1. $A \subseteq B$;
2. if $t \in B_s$ then $[t] \in B$;
3. $a \in A$, $t \in B_s$ and $t' \in B$ implies $a \cdot t', a : t', [t] \cdot t', [t] : t' \in B$;
4. $t, s \in B$ implies $t + s \in B$.

□

Remark 3.4.1.2. The atomic section operator introduces some problems when added to the theory AMMP(:). If we add its axioms to AMMP(:) we get the following equality which shows that part of the branching structure is lost:

$$\begin{aligned} a : c + b : c + c : (a + b) &= [a + b] : c + c : [a + b] \\ &= [a + b]|c \\ &= ([a] + [b])|c \\ &= [a]|c + [b]|c \\ &= a : c + c : a + b : c + c : b \end{aligned}$$

□

Remark 3.4.1.3. The difference between $\langle x \rangle$ and $[x]$ is in the fact that when composed in parallel the second does not admit any interleaving from other parallel components. For example:

$$\langle ab \rangle || c = \langle ab \rangle \between c + c \between \langle ab \rangle + \langle ab \rangle | c = \langle ab \rangle c + c\langle ab \rangle + \langle a(bc + cb) \rangle + c\langle ab \rangle$$

$$[ab] || c = [ab] \between c + c[ab][ab]|c = [ab]c + c[ab] + [ab]c + c[ab]$$

Note that in the second case, the trace acb is missing. □

3.4.2 The tight-actions model

We add a new transition relation:

$\xrightarrow{\bullet} \subseteq T \times A \times T$

$x \xrightarrow{a\bullet} x'$ means that besides evolving to x', the process x is executing an atomic action and therefore any parallel component including actions in the same "step" (that is, other actions in the same multiaction) must wait until the completion of the atomic process.

Additional rules for AMBP(:)+At are given in table 3.13.

$$\frac{x \xrightarrow{a*} x'}{[x] \xrightarrow{a\bullet} [x']} \qquad \frac{x \xrightarrow{a} \surd}{[x] \xrightarrow{a} \surd}$$

$$\frac{x \xrightarrow{a\bullet} x'}{\begin{array}{l} x||y \xrightarrow{a\bullet} x' \between y \\ y||x \xrightarrow{a\bullet} x' \between y \\ x \between y \xrightarrow{a\bullet} x' \between y \end{array}}$$

$$\frac{x \xrightarrow{a\bullet} x'}{\begin{array}{l} x||y \xrightarrow{a\bullet} x' \lfloor y \\ y||x \xrightarrow{a\bullet} x' \lfloor y \\ x|y \xrightarrow{a\bullet} x' \lfloor y \\ y|x \xrightarrow{a\bullet} x' \lfloor y \\ x \lfloor y \xrightarrow{a\bullet} x' \lfloor y \end{array}}$$

Table 3.13: Operational semantics for AMBP(:)+At

Remark 3.4.2.4. The addition of the atomic section operator to the theory AMMP(:) will introduce some equalities that do not respect the branching structure like the following:

$$\begin{aligned} a : c + b : c + c : (a + b) &= [a + b] : c + c : [a + b] \\ &= [a + b]|c \\ &= ([a] + [b])|c \\ &= [a]|c + [b]|c \\ &= a : c + b : c + c : a + c : b \end{aligned}$$

□

3.4.3 Adding the deadlock process

In order to use the state operator we need to be able to express that an action is blocked in some states. The way to do this is to add the constant δ, which represents a deadlocked process.

Axioms for δ are given in Table 3.14

Remark 3.4.3.5. Axiom TM5, TM5' are new, and TM5 is derivable from TM5', since

$$\delta : x = \langle \delta \cdot x \rangle = \langle \delta \rangle = \delta$$

A6	$x + \delta = x$	TM5	$\delta : x = \delta$
A7	$\delta \cdot x = \delta$	TM5'	$\langle \delta \rangle = \delta$

Table 3.14: Axioms for deadlock process

□

Remark 3.4.3.6. In the framework of AMBP(:) with δ the following equality holds;

$$a|\delta = a : \delta + \delta : a = a : \delta$$

Therefore, in the case that $a : \delta \neq \delta$, the constant δ does not act as a zero for the multiaction merge. □

We claimed in the introduction of this chapter that atomic actions should have the property of recoverability. In our present setting this is not yet achieved. In the next section we will introduce an idea of input-output behaviour that has this property, as already showed in [Bou89]. An interesting way to have this property in the branching semantics and at the same time solve the problem presented in the previous remark, is to take δ as a zero element with respect to tight multiplication. This can be done through the addition of the axiom in table 3.15.

TM6	$a : \delta = \delta$

Table 3.15: Axiom for deadlock process as a (:)zero element

3.4.4 Operational semantics with δ as a zero object for tight multiplication (:)

The operational semantics has to be modified in order to take axiom TM6 into account. This modification is comparable to those necessary when a zero element for sequential composition is introduced (see [BB90]). Every rule concerning tight actions should have an additional premise saying that the target of a transition is not δ. This can be done using operational rules with predicates. In our framework the predicates are not strictly needed since we have only one process that cannot perform any action, namely δ.

Thus, our objective can be achieved by replacing the rules in table 3.6 by the ones in table 3.16.

In the case of AMMP(:) there is no need to change anything except adding the rules in table 3.16. In AMBP(:) we replace the rules in table 3.11 by the ones in table 3.17, which are almost the same but for the extra premise that ensures that the other component is not δ.

$$\frac{x \xrightarrow{a*} x'}{x : y \xrightarrow{a*} x' : y} \qquad \frac{x \xrightarrow{a:} x' \;\; x' : y \xrightarrow{b*} z}{x : y \xrightarrow{a:} x' : y}$$

$$\frac{x \xrightarrow{a} \surd \;\; y \xrightarrow{b} \surd}{x : y \xrightarrow{a:} y} \qquad \frac{x \xrightarrow{a} \surd \; y \xrightarrow{b*} y'}{x : y \xrightarrow{a:} y}$$

Table 3.16: Operational semantics with δ

$$\frac{x \xrightarrow{a} x' \; y \xrightarrow{b*} y'}{\begin{array}{c} x||y \xrightarrow{a:} y \between x' \\ y||x \xrightarrow{a:} y \between x' \\ x|y \xrightarrow{a:} y \between x' \\ y|x \xrightarrow{a:} y \between x' \\ x \between y \xrightarrow{a:} y \between x' \end{array}} \qquad \frac{x \xrightarrow{a:} x' \; y \xrightarrow{b*} y'}{\begin{array}{c} x||y \xrightarrow{a:} x'|y \\ y||x \xrightarrow{a:} y|x' \\ x|y \xrightarrow{a:} x'|y \\ y|x \xrightarrow{a:} y|x' \\ x \between y \xrightarrow{a:} x'|y \end{array}} \qquad \frac{x \xrightarrow{a} \surd \; y \xrightarrow{b*} y'}{\begin{array}{c} x||y \xrightarrow{a:} y \\ y||x \xrightarrow{a:} y \\ x|y \xrightarrow{a:} y \\ y|x \xrightarrow{a:} y \\ x \between y \xrightarrow{a:} y \end{array}}$$

$$\frac{x \xrightarrow{a} x' \; y \xrightarrow{b} \surd}{\begin{array}{c} x||y \xrightarrow{a:} y \between x' \\ y||x \xrightarrow{a:} y \between x' \\ x|y \xrightarrow{a:} y \between x' \\ y|x \xrightarrow{a:} y \between x' \\ x \between y \xrightarrow{a:} y \between x' \end{array}} \qquad \frac{x \xrightarrow{a:} x' \; y \xrightarrow{b} \surd}{\begin{array}{c} x||y \xrightarrow{a:} x'|y \\ y||x \xrightarrow{a:} y|x' \\ x|y \xrightarrow{a:} x'|y \\ y|x \xrightarrow{a:} y|x' \\ x \between y \xrightarrow{a:} x'|y \end{array}} \qquad \frac{x \xrightarrow{a} \surd \; y \xrightarrow{b} \surd}{\begin{array}{c} x||y \xrightarrow{a:} y \\ y||x \xrightarrow{a:} y \\ x|y \xrightarrow{a:} y \\ y|x \xrightarrow{a:} y \\ x \between y \xrightarrow{a:} y \end{array}}$$

Table 3.17: Operational semantics for AMBP(:) with δ

3.5 The State Operator

3.5.1 Axioms for the State Operator

We add the axioms needed for the state operator to interact with the new operators in table 3.18.

SA6	$(a:x)\leftarrow s = (a\leftarrow s):(x\leftarrow(a\rightarrow s))$	SE6	$(a:x)\rightarrow s = x\rightarrow(a\rightarrow s)$
SA8	$\langle x\rangle\leftarrow s = \langle x\leftarrow s\rangle$	SE8	$\langle x\rangle\rightarrow s = x\rightarrow s$
SA9	$[x]\leftarrow s = [x\leftarrow s]$	SE9	$[x]\rightarrow s = x\rightarrow s$

Table 3.18: Axioms for the state operator with critical sections and atomicity

3.5.2 Models

Any algebra which could be used as a model of this more general state operator should have an extra sort of sets of states (with the operations of empty set and union). We will now introduce a model of atomic actions based in [Bou89] as a model of $AMBP_\delta(:)$+At with the state operator. The model is adapted to the slightly different set of operators of the algebras treated in this thesis. Given the action relation between processes defined above, we define the following relation between states called operation relation:

$$\mapsto \subseteq S \times P \times S$$

as the inductive set defined by the following rules

$$\frac{s' \in (a\rightarrow s)}{s \overset{a}{\mapsto} s'} \qquad \frac{x \overset{a}{\longrightarrow} \surd \quad s \overset{a}{\mapsto} s'}{s \overset{x}{\mapsto} s'} \qquad \frac{x \overset{a*}{\longrightarrow} y \quad s \overset{a}{\mapsto} s'' \overset{y}{\mapsto} s'}{s \overset{x}{\mapsto} s'}$$

Table 3.19: Operational semantics for the operation of processes on states

We can now define the interpretation of the operator $\rightarrow$ as

$$x\rightarrow s = \{s' | s \overset{x}{\mapsto} s'\}$$

$$x\rightarrow S = \bigcup_{s\in S} x\rightarrow s$$

This model is complete for closed terms, and moreover the set of states obtained by applying the operator to a process that can be infinite is determined by the set of of its terminated traces, in other words, this model satisfies the principle of terminated traces introduces in chapter 2.

3.5.3 Examples

Example 3.5.3.1 (*Synchronization*).

One of the applications of the theories introduced in [Bou89, BC88] was the implementation of synchronization in terms of shared variables and atomic actions. The idea is to use two boolean semaphores s and s' and Dijkstra's operations

$$P(s) = \text{when } s \text{ do } s := \mathit{false}$$

$$V(s) = \text{when } \neg s \text{ do } s := \mathit{true}$$

which are considered atomic. Using these operations we define:

$$S = \langle P(s) \cdot V(s') \rangle = P(s) : V(s')$$

$$R = \langle P(s') \cdot V(s) \rangle = P(s') : V(s)$$

We want that, in a certain state, $S \parallel R$ can only proceed through synchronization.

First, we define formally Dijkstra's operation.

Let

$$S = \{(tt, tt'), (tt, ff'), (ff, tt'), (ff, ff')\}$$

be the state space that keeps the value of both semaphores. Define (v, w booleans).

$s{\leftarrow}(ff, v) = \delta$	$s'{\leftarrow}(v, ff') = \delta$
$\bar{s}{\leftarrow}(tt, v) = \delta$	$\bar{s}'{\leftarrow}(v, tt') = \delta$
$ass_t(s){\rightarrow}(v, w) = (tt, w)$	$ass_t(s'){\rightarrow}(v, w) = (v, tt)$
$ass_f(s){\rightarrow}(v, w) = (ff, w)$	$ass_f(s'){\rightarrow}(v, w) = (v, ff)$

The atomic actions s and s' are tests for true and $\bar{s}, \bar{s}'$ are tests for false.

Now define

$$P(s) = [s \cdot ass_f(s)]$$

$$V(s) = [\bar{s} \cdot ass_t(s)]$$

and $P(s'), V(s')$ analogously.

It can be shown easily that

$$\begin{aligned}
P(s){\leftarrow}(tt, w) &= P(s) \\
P(s){\rightarrow}(tt, w) &= (ff, w) \\
P(s){\leftarrow}(ff, w) &= \delta \\
P(s'){\leftarrow}(v, tt) &= P(s') \\
P(s'){\rightarrow}(v, tt) &= (v, ff') \\
P(s'){\leftarrow}(v, ff) &= \delta
\end{aligned}$$

$$\begin{aligned} V(s)\leftarrow(ff,w) &= V(s) \\ V(s)\rightarrow(ff,w) &= (tt,w) \\ V(s)\leftarrow(tt,w) &= \delta \\ V(s')\leftarrow(v,ff') &= V(s') \\ V(s')\rightarrow(v,ff') &= (v,tt') \\ V(s')\leftarrow(v,tt') &= \delta \end{aligned}$$

When both semaphores are true neither S nor R can proceed independently. For example:

$$\begin{aligned} S\leftarrow(tt,tt') &= P(s):V(s')\leftarrow(tt,tt') \\ &= P(s):(V(s')\leftarrow(ff,tt')) \\ &= P(s):\delta \\ &= \delta \end{aligned}$$

On the other hand we have that:

$$\begin{aligned} S||R\leftarrow(tt,tt') &= P(s):V(s')||P(s'):V(s)\leftarrow(tt,tt') \\ &= (P(s):V(s') \between P(s'):V(s) + P(s'):V(s) \between P(s):V(s') \\ &+ P(s):V(s')|P(s'):V(s))\leftarrow(tt,tt') \\ &= (P(s):V(s')\cdot P(s'):V(s) + P(s'):V(s)\cdot P(s):V(s') \\ &+ P(s):V(s')|P(s'):V(s))\leftarrow(tt,tt') \\ &= P(s):V(s')|P(s'):V(s)\leftarrow(tt,tt') \end{aligned}$$

In AMBP(:)+At we have that this equals (we leave the details to the reader).

$$\begin{aligned} & P(s):(P(s'):(V(s):V(s')+V(s'):V(s))+V(s'):P(s'):V(s))\leftarrow(tt.tt') \\ + \; & P(s'):(P(s):(V(s'):V(s)+V(s):V(s'))+V(s):P(s):V(s'))\leftarrow(tt,tt') \\ = \; & P(s):(P(s'):(V(s):V(s')+V(s'):V(s)) \\ + \; & P(s'):(P(s):(V(s'):V(s)+V(s):V(s')) \end{aligned}$$

Example 3.5.3.2 (*Coupling Buffers*).
One can specify a buffer with input port 1 and output port 2 as follows (see [BW90]):

$$B^{12} = r_1 \cdot s_2 \cdot B^{12}$$

Note that we omit the data in order to reduce communication to synchronization as presented in the previous example. The general case is slightly more complicated.

We can implement a buffer with capacity 2, using two buffers with capacity one by coupling the output of one with the input of the other. Suppose we have in addition to the buffer above, another one with input port 2 and output port 3. If we ask for synchronization between s_2 and r_2 resulting in an atomic action c_2 and we disallow the independent occurrence of these two actions, then we get the required buffer (see [BW90] for the details). I.e., defining

$$X = \partial_{\{s_2,r_2\}}(B^{12}||B^{23})$$

we can derive the following equations for X:

$$X = r_1 \cdot c_2 \cdot X'$$
$$X' = s_3 \cdot X + r_1 \cdot s_3 \cdot c_2 \cdot X'$$

Now we want to implement this in our framework without a primitive for communication. We define then:

$$C = r_1 \cdot S \cdot C$$
$$D = R \cdot s_3 \cdot D$$

where R and S are as defined above. Now we calculate the following process

$$C||D{\leftarrow}(tt,tt') = C \between D{\leftarrow}(tt,tt') + D \between C{\leftarrow}(tt,tt') + C|D{\leftarrow}(tt,tt')$$

$$= C \between D{\leftarrow}(tt,tt') + D \between C{\leftarrow}(tt,tt') + C \lfloor D{\leftarrow}(tt,tt') + D \lfloor C{\leftarrow}(tt,tt')$$

First we show that the last three summands are equal to δ (we leave the last one to the reader).

$$\begin{aligned} D \between C{\leftarrow}(tt,tt') &= R \cdot (s_3 \cdot D||C){\leftarrow}(tt,tt') \\ &= R{\leftarrow}(tt,tt') \cdot (s_3 \cdot D||C{\leftarrow}(R{\rightarrow}(tt,tt'))) \\ &= \delta \cdot (s_3 \cdot D||C{\leftarrow}(R{\rightarrow}(tt,tt'))) \\ &= \delta \end{aligned}$$

$$\begin{aligned} (C \lfloor D){\leftarrow}(tt,tt') &= (r_1 \cdot S \cdot C \lfloor D){\leftarrow}(tt,tt') \\ &= (r_1 : (D \between P(s) : V(s') \cdot C)){\leftarrow}(tt,tt') \\ &= r_1 : R \cdot (s_3 \cdot D||P(s) : V(s') \cdot C){\leftarrow}(tt,tt') \\ &= r_1 : (R{\leftarrow}(tt,tt')) \cdot ((s_3 \cdot D||P(s) : V(s') \cdot C){\leftarrow}(R{\rightarrow}(tt,tt'))) \\ &= r_1 : \delta \cdot ((s_3 \cdot D||P(s) : V(s') \cdot C){\leftarrow}(R{\rightarrow}(tt,tt'))) \\ &= \delta \end{aligned}$$

We know then that:

$$C||D{\leftarrow}(tt,tt') = C \between D{\leftarrow}(tt,tt') = r_1 \cdot (S \cdot C||D{\leftarrow}(tt,tt'))$$

We calculate then

$$\begin{aligned}
S \cdot C||D{\leftarrow}(tt,tt') &= S \cdot C \between D{\leftarrow}(tt,tt') + D \between S \cdot C{\leftarrow}(tt,tt') + S \cdot C|D{\leftarrow}(tt,tt') \\
&= S \cdot (C||D){\leftarrow}(tt,tt') + R \cdot (s_3 \cdot D||S \cdot C){\leftarrow}(tt,tt') \\
&+ S \cdot C|D{\leftarrow}(tt,tt') \\
&= S \cdot C|D{\leftarrow}(tt,tt') \\
&= (P(s) : V(s') \cdot C|P(s') : V(s') \cdot s_3 \cdot D){\leftarrow}(tt,tt') \\
&= (P(s) : V(s') \cdot C \lfloor P(s') : V(s') \cdot s_3 \cdot D){\leftarrow}(tt,tt') \\
&+ (P(s') : V(s') \cdot s_3 \cdot D \lfloor P(s) : V(s') \cdot C){\leftarrow}(tt,tt')
\end{aligned}$$

We calculate now the first summand leaving the second one to the reader.

$$\begin{aligned}
& (P(s) : V(s') \cdot C \lfloor P(s') : V(s') \cdot s_3 \cdot D){\leftarrow}(tt,tt') \\
= \; & (P(s) : (V(s') \cdot C|P(s') : V(s') \cdot s_3 \cdot D)){\leftarrow}(tt,tt') \\
= \; & P(s) : ((V(s') \cdot C \lfloor P(s') : V(s') \cdot s_3 \cdot D){\leftarrow}(ff,tt') \\
+ \; & P(s') : V(s') \cdot s_3 \cdot D \lfloor V(s') \cdot C){\leftarrow}(ff,tt') \\
= \; & P(s) : (P(s') : (V(s') \cdot s_3 \cdot D|V(s') \cdot C)){\leftarrow}(ff,tt') \\
+ \; & P(s) : P(s') : ((V(s') \cdot s_3 \cdot D|V(s') \cdot C){\leftarrow}(ff,ff') \\
= \; & P(s) : P(s') : (V(s) : V(s') \cdot ((s_3 \cdot D||C){\leftarrow}(tt,tt')) \\
+ \; & V(s') : V(s) \cdot ((s_3 \cdot D||C){\leftarrow}(tt,tt'))) \\
= \; & (P(s) \lfloor P(s')) : (V(s)|V(s')) \cdot ((s_3 \cdot D||C){\leftarrow}(tt,tt'))
\end{aligned}$$

In a similar way we can calculate the equations for

$$\begin{aligned}
& (s_3 \cdot D||C){\leftarrow}(tt,tt') \\
& = s_3 \cdot (D||C){\leftarrow}(tt,tt') + r_1 \cdot s_3 \cdot (D||C){\leftarrow}(tt,tt') + (r_1|s_3) \cdot (D||C){\leftarrow}(tt,tt')
\end{aligned}$$

Using the following abbreviations:

$$\begin{aligned}
B &= (C||D){\leftarrow}(tt,tt') \\
B'' &= (S \cdot C||D){\leftarrow}(tt,tt') \\
B' &= (s_3 \cdot D||C){\leftarrow}(tt,tt')
\end{aligned}$$

we obtain the following recursive specification:

$$\begin{aligned}
B &= r_1 \cdot B'' \\
B'' &= (P(s)|P(s')) : (V(s)|V(s')) \cdot B' \\
B' &= s_3 \cdot B + r_1 \cdot s_3 \cdot B'' + (r_1|s_3) \cdot B''
\end{aligned}$$

3.6 Concluding Remarks

The systems introduced in this chapter were inspired from mainly two sources: the work concerning tight regions in [BK84b] (and the related works of [GMM90, OL87]) and the work about atomic actions in [Bou89, BC88].

The main difference between our approach and the one in [BK84b, GMM90, OL87] resides in the fact that we do not introduce synchronous communication. Instead, we borrow the idea from [Bou89] of implementing synchronous communication using shared variables represented by the state operator. On the other hand the semantics proposed in [Bou89] are input-output semantics whereas we propose branching semantics for our algebras. One of the problems of the input-output semantics appear in the version of implementation of communication presented in [Bou89]. In that work the following definition of P (and a similar one for V) was given:

$$P(s) = [\text{ while } \neg s \text{ do skip } \cdot ass_f(s)]$$

The parallel composition then of S and R defined with this version of semaphores involves a livelock, since the non-interruptible process P enters into a loop from which it cannot be taken out. But for the input-output semantics this implementation is correct, since a livelock equals a deadlock for this semantic (neither of them produce any output).

The parallel composition of [GMM90] is not associative (as is the case here). In the conclusion of that article the authors claim that it may be the case that the loss of associativity is intrinsic to the hierarchical construction. A possibility suggested in that article as well, is to have two different parallel operators, one with normal interleaving semantics and another which works as an atomic transaction manager.

One possible solution to the non-associativity could be to use complete trace equivalence to identify multiactions. This preserves the input output semantics.

The different models introduced here show that it is possible in principle to use these algebras to describe systems at different levels of abstraction, but this possibility should be further investigated. Another interesting extension would be an operator of action refinement that works in an interleaving framework.

3.7 Further work

Given the intuition behind the equivalence of multiactions which represents the fact that two multiactions will produce the same effect on any state and the property that the effect depends only on the terminated traces, it is plausible to introduce a system in which the following equality holds:

$$a : (x + y) = a : x + a : y$$

This equation contradicts the idea of branching semantics.

In such a system the inaction process δ must be a zero with respect to tight multiplication, as the following equality shows

$$a : b = a : (b + \delta) = a : b + a : \delta$$

One of the advantages of this system is that the two extensions with multiactions coincide, at least for closed terms.

Chapter 4

Data types and Processes

4.1 Introduction

In the thesis [Pon92] data types are introduced into the world of process algebra to play mainly two roles:

1. Interpreting processes: the actions of the process have certain effect on a data space. In this sense the processes are interpreted in a model defined using some presentation of data types.

2. Specifying processes: data types are used as a mathematical tool in order to define behaviours, for example as indexes of equations or as parameters of atomic actions.

In many cases the two approaches appear together. When one has a model where data types play an important role it is sometimes desirable to have a syntactic mechanism to specify them; on the other hand, using data types algebraically in the language sometimes induces an algebraic structure on the models.

In the literature about data types and processes there are many examples of how the interaction between them is achieved:

1. Data types appear as parameters of equations or actions and some programming language constructs, as conditionals, are introduced in order to deal with them. ([Pon92], lotos [Bri88], PSF [MV89], μCRL [GP90])

2. The processes run in a state space algebraically specified. The interaction is stated explicitly via functions or relations that determine the effect on such a state. ([KP87a], [Bou89], the state operator [BB88])

3. Processes are a special kind of data type ([AMR88])

4. Data types are processes. This approach has been used implicitly in many works on process algebra, including [Mil80], [BW90]

4.2 Data Types

4.2.1 Algebraic Specification

We give here a summary of the notation and concepts about partial many sorted algebras we will use. There is a wide variety of literature available on this topic, for example [Rei87].

Definition 4.2.1.1. A *signature* $\Sigma = (S, \Omega)$ is given by a set of sort names and a finite set of operations provided with their functionalities.

An *I/O signature* (input/output) $\Sigma[I] = (S, I, \Omega)$ is a signature with a distinguished subset of sort names $I \subseteq S$ called the input/output or visible sorts. □

Definition 4.2.1.2. Given a signature $\Sigma = (S, \Omega)$, a *partial* Σ*-algebra*

$$A = (\{A_s | s \in S\}, \{\sigma^A | \sigma \in \Omega\})$$

is given by a family of sets A indexed by sort names and a set of partial operations such that if $\sigma : s_1 \times \cdots \times s_n \to s$ then the domain of σ^A is a subset of $A_{s_1} \times \cdots \times A_{s_n}$.

A *partial* $\Sigma[I]$*-algebra* is just a partial Σ-algebra with some distinguished sets.

□

Definition 4.2.1.3. Given (for $\Sigma = (S, \Omega)$) two Σ-algebras (or Σ[I]-algebras) A and B, a Σ*-homomorphism* is a family

$$f = \{f_s : A_s \to B_s | s \in S\})$$

of functions indexed by S such that the following conditions hold:

1. if $(t_1, \ldots, t_n)$ is in the domain of σ^A, then $(f(t_1), \ldots, f(t_n))$ is in the domain of σ^B.
2. $f(\sigma^A(t_1, \ldots, t_n)) = \sigma^B(f(t_1), \ldots, f(t_n))$ if both sides are defined.

□

Definition 4.2.1.4. Let A, B be two $\Sigma[I]$-algebras. A *reduction* is an homomorphism $f : A \to B$ such that

1. f is surjective,
2. f is the identity on the sorts of I, and
3. if $(f(t_1), \ldots, f(t_n))$ is in the domain of σ^B, then $(t_1, \ldots, t_n)$ is in the domain of σ^A.

□

Definition 4.2.1.5. Two $\Sigma[I]$-algebras A and B are *behaviourally equivalent* if there exist a $\Sigma[I]$-algebra F and two reductions $r : F \to A$ and $r' : F \to B$. When this is the case we write:

$$A \equiv B(\bmod I)$$

□

4.2.2 Data Types as Processes

In this section we introduce a class of algebraic data types that is suitable for being studied in the framework of process algebra. Such a class was defined in [AMR88] where the name *dynamic data types*, which we are going to borrow, was introduced.

The main idea that we follow is to identify operations in the signature of the data types with atomic actions in the corresponding process. This introduces an idea of *granularity* into data types.

Definition 4.2.2.6. As a definition we say that a dynamic data type is a (partial) data type (defined in a proper theoretical way) such that there is a distinguished sort called *ds* (dynamic sort) and its operations are required to belong to one of the following three sets:

- **I** of initial elements, such that

$$\forall \sigma \in \mathbf{I}, \sigma : s_1 \times \ldots \times s_n \to ds$$

- **M** of modification operations, such that

$$\forall \sigma \in \mathbf{M}, \sigma : s_1 \times \ldots \times s_n \times ds \to ds$$

- **O** of observation operations, such that

$$\forall \sigma \in \mathbf{O}, \sigma : s_1 \times \ldots \times s_n \times ds \to s_{n+1}$$

where $\forall i : 1 \ldots n+1, s_i \neq$ ds and s_i is finite. The finiteness of the non-dynamic sorts is not essential but allows us to work in the easier world of process algebras without value passing nor infinite sums. □

Example 4.2.2.7 (*Stack*). Of course this should be the first example. We define in the framework of algebraic specifications a stack of elements over a finite set **D**. The details about the meaning of such a definition are standard and can be found in [Rei87] (we use $\circ\!\!\to$ to indicate that an operation is not total).

$$\begin{array}{rll} \text{es}: & & \rightarrow \text{St} \\ \text{push}: & \text{D} \times \text{St} & \rightarrow \text{St} \\ \text{pop}: & \text{D} \times \text{St} & \circ\!\!\rightarrow \text{St} \end{array}$$

where

$$\begin{aligned} \text{es} &= \text{es} \\ \text{pop(d,push(d,s))} &= \text{s} \end{aligned}$$

Thus, here

$$\mathbf{I} = \{\text{es}\}$$

$$\mathbf{M} = \{\text{push}, \text{pop}\}$$

The first equation means only that the empty stack is defined.

The definition of a stack in process algebra that follows is well known in the literature (with variations [BW90]). We underline the atomic actions in order to distinguish them from the operators of the algebra.

$$\begin{aligned} S_\phi &= \sum_{d \in D} \underline{\text{push}}(d) \cdot S_d \\ S_{\text{push}(d,\sigma)} &= \sum_{e \in D} \underline{\text{push}}(e) \cdot S_{\text{push}(e,\text{push}(d,\sigma))} + \underline{\text{pop}}(d) \cdot S_\sigma \end{aligned}$$

Now we draw some provisional conclusions from the previous example.

1. The definition of the stack using the a.d.t. specifications' style is not the "standard" one but the one directly related to the "standard" specification in process algebra. This is only a matter of style and nothing deep is hidden here.

2. The set of equations that define the stack in process algebra is an infinite one. To be more precise it is a variable indexed over the sets of stacks. The meaning of this is not completely clear, we assume they are elements of the carrier set of the sort of stacks in a chosen model.

Example 4.2.2.8 (*yet another stack*). In example 4.2.2.7 there are no observations. The intended meaning of observation operations is to obtain certain information about the data type without modifying it. In the tradition of a.d.t they are recognizable by the target sort. Here, we represent the intended meaning more directly in the following way:

In algebraic specifications:

$$\begin{array}{rcl} \text{es}: & & \rightarrow \text{St} \\ \text{push}: & \text{D} \times \text{St} & \rightarrow \text{St} \\ \text{pop}: & \text{D} \times \text{St} & \circ\!\!\rightarrow \text{St} \\ \text{top}: & \text{St} & \circ\!\!\rightarrow \text{D} \end{array}$$

where

$$\begin{array}{rcl} \text{es} & = & \text{es} \\ \text{pop(d,push(d,s))} & = & \text{s} \\ \text{top(push(d,s))} & = & \text{d} \end{array}$$

In process algebra:

$$\begin{aligned} S_{\text{es}} &= \sum_{d \in D} \underline{\text{push}}(d) \cdot S_d \\ S_{\text{push}(d,\sigma)} &= \sum_{e \in D} \underline{\text{push}}(e) \cdot S_{\text{push}(e,\text{push}(d,\sigma))} + \underline{\text{pop}}(d) \cdot S_\sigma + \\ &\quad \underline{\text{top}}(d) \cdot S_{\text{push}(d,\sigma)} \end{aligned}$$

Some questions still remain open after these two examples. One is: What about creation operations? In this work we assume the platonic view of a.d.t's long used in the literature: Data types exist and will exist forever, we only need to name them. Hence, the creation operations have no representation at the level of atomic actions (which one could they have?) and appear only in the indexes of the equations. By symmetry, one can think of destroying data types, and this is something that cannot be formulated in algebraic specifications. The second question is easily answered in process algebra by using successful termination of a process. The former case requires the introduction of a mechanism for process creation (as in [BV92]). We leave these issues for further research.

Definition 4.2.2.9. We are now in a position to give a general method to obtain recursive definitions from dynamic data types.

Let A be an algebra for a dynamic data type specification. We write $\sigma(t_1, \ldots, t_n)\downarrow$ iff $(t_1, \ldots, t_n)$ is in the domain of σ^A, where σ is an operation of the signature. We define a function d_σ for every $\sigma \in \mathbf{M} \cup \mathbf{O}$ by

- if $\sigma \in \mathbf{M}, \sigma : s_1 \times \cdots \times s_n \times \text{ds} \rightarrow \text{ds}$ then

$$d_\sigma : A_{\text{ds}} \rightarrow \mathcal{P}(A_{s_1} \times \cdots \times A_{s_n})$$

$$d_\sigma(x) = \{(t_1, \ldots, t_n) | \sigma(t_1, \ldots, t_n, x)\downarrow\}$$

- if $\sigma \in \mathbf{O}, \sigma : s_1 \times \cdots \times s_n \times \mathrm{ds} \to s_{n+1}$ then

$$d_\sigma : A_{\mathrm{ds}} \to \mathcal{P}(\mathbf{A}_{s_1} \times \cdots \times A_{s_{n+1}})$$

$$d_\sigma(x) = \{(t_1, \ldots, t_{n+1}) | \sigma(t_1, \ldots, t_n, x)\downarrow \wedge \sigma(t_1, \ldots, t_n, x) = t_{n+1}\}$$

Now we give a set of recursive equations indexed by the elements of the dynamic carrier of an algebra. $\forall x \in \mathbf{A}_{\mathrm{ds}}$

$$S_x = \sum_{m \in \mathbf{M}} \sum_{\vec{t} \in d_m(\vec{t},x)} \underline{m}(\vec{t}) \cdot S_{m(x)} + \sum_{o \in \mathbf{O}} \sum_{\vec{t} \in d_o(x)} \underline{o}(\vec{t}) \cdot S_x$$

We will assume, unless stated otherwise, that there is a distinguished initial state for any data type, as the empty stack in the examples above.

□

In the framework of process algebra there exist many well-known equivalences. Fortunately, a number of them are the same when we restrict ourselves to deterministic processes, which is the case for processes that are obtained from data types, as we have done above. Trace equivalence as well as bisimulation and all the ones in between can be used (see [Gla90]).

It is quite intuitive that the equivalence induced upon the data types via the translation into process algebra is coarser than just isomorphism. One would expect a kind of behavioural equivalence. Theorem 4.2.2.12 shows that this is the case.

Lemma 4.2.2.10. *A reduction, seen as a relation between elements of the dynamic sort, is a bisimulation.*

Proof. We prove the lemma for the actions that come from a modifying operation, leaving for the reader the observing operations.

Let $f : A \to B$ be a reduction.

Let a be a state, i.e. $a \in A_{ds}$, such that

$$a \xrightarrow{\sigma(d_1,\ldots,d_n)} a'$$

what means that $\sigma^A(d_1, \ldots, d_n, a)\downarrow$ and $\sigma^A(d_1, \ldots, d_n, a) = a'$. Since f is a $\Sigma[I]$-homomorphism, it follows that

$$f(\sigma^A(d_1, \ldots, d_n, a)) = \sigma^B(f(d_1), \ldots, f(d_n), f(a)) = \sigma^B(d_1, \ldots, d_n, f(a))$$

and this implies that $\sigma^B(d_1, \ldots, d_n, f(a))\downarrow$ and that $\sigma^B(d_1, \ldots, d_n, f(a)) = f(a')$ what means in another words that

$$f(a) \xrightarrow{\sigma(d_1,\ldots,d_n)} f(a')$$

.

Now, in the other direction, let $b \in B_{ds}$, such that

$$b \xrightarrow{\sigma(d_1,\ldots,d_n)} b'$$

Since f is surjective there exists $a \in A_{ds}$ such that $f(a) = b$ as $\sigma^B(d_1,\ldots,d_n,f(a))\downarrow$. Then, $\sigma^A(d_1,\ldots,d_n,a)\downarrow$ by 4.2.1.4, moreover

$$f(\sigma^A(d_1,\ldots,d_n,a)) = \sigma^B(d_1,\ldots,d_n,f(a)) = \sigma^B(d_1,\ldots,d_n,b) = b'$$

.

□

Lemma 4.2.2.11. *If two $\Sigma[I]$-algebras are bisimilar as processes, then they are behaviourally equivalent.*

Proof. Let A and B be two bisimilar $\Sigma[I]$-algebras. We define the $\Sigma[I]$-algebra G having as elements the bisimulation classes and the operations defined in the obvious way. The facts that this is actually a $\Sigma[I]$-algebra and that there exist reductions $r : A \to G$ and $r' : B \to G$ are a consequence of the definition of bisimulation. Consequently, we can take the pullback of both reductions and using corollary 5.1.6. of [Rei87] the lemma is proven. The pullback will be in this case the algebra P whose elements are pairs $(a,b), a \in A, b \in B$ such that $a \leftrightarrows b$, with the projections onto A and B as reductions. □

The immediate consequence of the two lemmas above is the following theorem which demonstrates the adequacy of the notions of bisimulation and behavioural equivalence for dynamic data types.

Theorem 4.2.2.12. *Two dynamic data types are behaviourally equivalent if and only if they are bisimilar.*

4.2.3 Data types and the State Operator

One of the most direct ways to introduce data types in process algebra is via the state operator. In this approach the state space is specified algebraically and the actions will have an effect that can be expressed using the operations of the algebra. Some restrictions over the functions $\leftarrow$ and $\rightarrow$ should be sometimes imposed in order to disallow trivial answers to the problems that may arise.

In [KP87b] a notion of process specification is introduced. It consists of a data specification which is a standard algebraic specification together with a specification of the effect of atomic actions over the data type. When we take an algebra as model of the data type specification, we can use its carrier set (or the main one, in the case of a

many-sorted algebra) as a state space and interpret the effect specification as the effect function of the state operator. We illustrate this with an example adapted from [KP87b]

Example 4.2.3.13 (*Stack*). As usual, a stack with elements from a (finite) set D can be defined as some model of an algebraic specification as before. We take the underlying set of a model of the algebraic specification as a state space, and defin the the effect function for the atomic actions. We assume that the action function is trivial.

$$\begin{aligned} \underline{\text{push(d)}} \rightarrow s &= \text{push(d,}s\text{)} \\ \underline{\text{pop}} \rightarrow \text{es} &= \emptyset \\ \underline{\text{pop}} \rightarrow \text{push(d,}s\text{)} &= s \end{aligned}$$

We can now use the state operator defined in this way to describe processes that can interact with a stack.

4.3 Implementing data types in a concurrent environment

In this section different notions of implementation of data types are described. We consider first implementations of a state operator by another. This notion captures the essential ideas present in [KP87a]. This notion is defined following the tradition of algebraic data types as a suitable morphism. We present afterwards the completely different idea of using finite control (in the form of a regular process) and some mechanism for interaction (state operator or communication) and abstraction in order to implement one data type (seen as a process) by another.

4.3.1 Implementations as morphisms between state operators

Given two state operator definitions $(S, \rightarrow), (S', \rightarrow')$ (we do not consider the action part here, in order to follow as closely as possibly the example presented in [KP87a]), an implementation of the first in terms of the second is a pair of functions:

$$\phi_s : S \longrightarrow S'$$

$$\phi_a : A \longrightarrow M$$

such that for any state $s \in S$, any atomic action $a \in A$ and any finite multiaction $m \in M$ the following equation holds:

$$\phi_a(m) \rightarrow' \phi_s(s) = \phi_s(m \rightarrow s)$$

where we extend ϕ_s to sets and ϕ_a to multiactions in the obvious way:

$$\phi_a(m|n) = \phi_a(m)|\phi_a(n)$$

$$\phi_s(S) = \{\phi s | s \in S\}$$

This equation requires that the implementation is preserved by any atomic action and that no undesirable interleavings occur which can lead to an inconsistent state. We illustrate this with the following example rewritten from a similar one in [KP87a].

Example 4.3.1.1. One of the standard implementations of a stack is obtained using an array and a pointer. This implementation is correct from a sequential point of view but it must be refined in a concurrent environment, as this example will demonstrate.

The state space will consist of a pair formed by an array with elements taken from the set D and a pointer. An array is a function from the natural numbers into D. A distinguished element $\perp$ is added to D in order to represent an empty cell. An algebraic specification of an array can be given as follows:

$$\begin{array}{rcl} \mathrm{ea}: & & \to \mathrm{Ar} \\ \_[\_]: & \mathrm{Ar} \times \mathrm{Nat} & \to \mathrm{D} \\ \_[\_] := \_: & \mathrm{Ar} \times \mathrm{Nat} \times \mathrm{D} & \circ\!\!\to \mathrm{Ar} \end{array}$$

where

$$\begin{array}{rcl} \mathrm{ea} & = & \mathrm{ea} \\ \mathrm{ea}[n] & = & \perp \\ (a[n] := \mathrm{d})[n] & = & \mathrm{d} \\ (a[n] := \mathrm{d})[m] & = & a[m] \text{ if } n \neq m \end{array}$$

We describe first the state operator.

$$\begin{array}{rcl} \underline{\mathrm{ass(d)}} \to (a,n) & = & (a[n] := \mathrm{d}, n) \\ \underline{\mathrm{inc}} \to (a,n) & = & (a, n+1) \\ \underline{\mathrm{dec}} \to (a,n) & = & \text{if } n = 0 \text{ then } \emptyset \text{ else } (a[n-1] := \perp, n-1) \end{array}$$

Note that decrementing the pointer has the effect of deleting the last element of the array as well. We will not need to do that when we use behavioural equivalence for the states. This example tries to follow faithfully the one in [KP87a].

The function ϕ_s is obvious: a stack will be represented by an array with the same elements arranged in the same way. The value of the pointer is one more than the value of the cell that contains the top of the stack. The rest of the array will be empty.

The function ϕ_a is defined as follows:

$$\begin{aligned} \phi_a(\underline{\text{push(d)}}) &= \langle \underline{\text{inc}} \cdot \underline{\text{ass(d)}} \rangle \\ \phi_a(\underline{\text{pop}}) &= \underline{\text{dec}} \end{aligned}$$

This implementation works correctly in the sequential case, but in a concurrent environment it can produce inconsistent states, and this implies that it will not satisfy our definition of implementation.

The following equations are satisfied:

$$\phi_a(\underline{\text{push(d)}}) \rightarrow' \phi_s(s) = \phi_s(\underline{\text{push(d)}} \rightarrow s)$$

$$\phi_a(\underline{\text{pop}}) \rightarrow' \phi_s(s) = \phi_s(\underline{\text{pop}} \rightarrow s)$$

However, the implementation is not correct, as the following example shows (we leave the details to the reader):

$$\begin{aligned} (\phi_a(\underline{\text{push(d)}} | \underline{\text{push(e)}}) \rightarrow' \phi_s(\text{es}) &= \langle \underline{\text{inc}} \cdot \underline{\text{ass(d)}} \rangle | \langle \underline{\text{inc}} \cdot \underline{\text{ass(e)}} \rangle \rightarrow' (\text{ea}, 0) \\ &= \langle \underline{\text{inc}} \cdot \underline{\text{ass(d)}} \cdot \underline{\text{inc}} \cdot \underline{\text{ass(e)}} \rangle \rightarrow' (\text{ea}, 0) \\ &+ \langle \underline{\text{inc}} \cdot \underline{\text{inc}} \cdot \underline{\text{ass(d)}} \cdot \underline{\text{ass(e)}} \rangle \rightarrow' (\text{ea}, 0) \\ &+ \langle \underline{\text{inc}} \cdot \underline{\text{inc}} \cdot \underline{\text{ass(e)}} \cdot \underline{\text{ass(d)}} \rangle \rightarrow' (\text{ea}, 0) \\ &+ \langle \underline{\text{inc}} \cdot \underline{\text{ass(e)}} \cdot \underline{\text{inc}} \cdot \underline{\text{ass(d)}} \rangle \rightarrow' (\text{ea}, 0) \\ &= \{([d, e], 2), ([\bot, e], 2), ([\bot, d], 2), ([e, d], 2)\} \end{aligned}$$

It is clear that the two states which have a bottom as first element of the array cannot be the representation of any stack.

□

A solution to this problem proposed in [KP87a] consist of the addition of a semaphore that, after an increment of the counter is performed, blocks any new increment until a value is assigned.

We present only the definition and leave the details to the reader. We will study a similar implementation in our next example when we use equivalence of states instead of equality.

Definition 4.3.1.2.

$$\begin{aligned}
\underline{\text{ass(d)}}\rightarrow(a,n,b) &= (a[n] := d, n, tt) \\
\underline{\text{inc}}\rightarrow(a,n,tt) &= (a, n+1, ff) \\
\underline{\text{inc}}\rightarrow(a,n,ff) &= \emptyset \\
\underline{\text{dec}}\rightarrow(a,n,b) &= \text{if } n = 0 \text{ then } \emptyset \text{ else } (a[n-1] := \bot, n-1, b)
\end{aligned}$$

□

4.3.2 Observational implementations

The intuition behind behavioural equivalence is that two states of a data type will be considered equal when there are no observation operations that could distinguish them. Since in our framework we use the atomic actions to represent the possible operations over a data type and the states of the data type as our state space, we will use the equivalence of states as criteria for behaviour equivalence.

The equivalence of states depends on the set of atomic actions considered. In our case, since we intend to use the data type only through the operations of the specification, we use this set operation to define the equivalence of the states of the implementation. In order to do that we need first to define the effect of these actions over the data type of the implementation, however this is done through the implementation of the atomic actions.

Thus, the effect function for any atomic action a belonging to the signature of the specification and a state s in the implementation is defined as

$$a{\rightarrow}s = \phi_a(a){\rightarrow}s$$

Definition 4.3.2.3.

Given two state operator definitions $(S, \leftarrow, \rightarrow), (S', \leftarrow', \rightarrow')$ a behavioural implementation of the first in terms of the second is a pair of functions

$$\phi_s : S \longrightarrow S'$$

$$\phi_a : A \longrightarrow M$$

such that for any state $s \in S$, any atomic action $a \in A$ and any finite multiaction $m \in M$ the following equivalence holds:

$$\phi_a(m){\rightarrow}'\phi_s(s) \sim \phi_s(m{\rightarrow}s)$$

where the equivalence is extended to sets of states in the obvious way.

□

Example 4.3.2.4. We present here another version of the implementation of a stack using an array, a pointer and a semaphore. Since we want to show an implementation where different states will be behaviourally equivalent, we need to have some atomic actions that observe the current state of the data type.

We add then the atomic action $\underline{\text{top(d)}}$ with the following effect function:

$$\begin{aligned} \underline{\text{top(d)}} \rightarrow \text{es} &= \emptyset \\ \underline{\text{top(d)}} \rightarrow \text{push(d,}s\text{)} &= \text{push(d,}s\text{)} \\ \underline{\text{top(d)}} \rightarrow \text{push(e,}s\text{)} &= \emptyset \text{ if } d \neq e \end{aligned}$$

In order to implement this operation we need to add an observation operation to the array, which, for simplicity, will have the same behaviour as the top:

$$\begin{aligned} \underline{\text{val(d)}} \rightarrow (ea, 0) &= \emptyset \\ \underline{\text{val(d)}} \rightarrow (a, n+1) &= (a, n+1) \text{ if } a[n+1] = \text{d} \\ \underline{\text{val(d)}} \rightarrow (a, n+1) &= \emptyset \text{ if } a[n+1] \neq \text{d} \end{aligned}$$

Now the implementation is obvious

$$\phi_a(\underline{\text{top(d)}}) = \underline{\text{val(d)}}$$

Now, it is clear that the relation that relates two arrays if they are equal at least for the elements whose position is less than the pointer is a state bisimulation. Thus, the implementation of the $\underline{\text{pop}}$ can be simplified, in particular, we can simplify the implementation of the decrement function for the array:

$$\underline{\text{dec}} \rightarrow (a, n) = \text{if } n = 0 \text{ then } \emptyset \text{ else } (a, n-1)$$

□

4.4 Implementing dynamic data types

4.4.1 Abstract process algebra

In the following, we will need to use abstraction in process algebra. The traditional mechanism to introduce abstraction in process algebra is through the use of the so called *silent step* τ. The idea is that τ represents an internal action that cannot be observed by the environment.

There exist many different theories that introduce this constant, and we refer the reader to [BW90] for an introduction to abstraction in process algebra. Here we present

only the axioms for τ in so-called rooted branching bisimulation semantics, and the abstraction operator τ_I that hides (makes silent) the atomic actions belonging to the set I.

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

Table 4.1: Silent action

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

Table 4.2: Abstraction

4.4.2 Implementations using the state operator

In order to define a different notion of implementation for dynamic data types we use the fact that they can be considered either as a space state or as processes also. The idea is to use the carrier set of the implementing dynamic data type as state space and, with the help of a regular process running inside a state operator, to produce the behaviour of the process that represents the data type we want to implement.

We impose certain restrictions on the action and effect function that define the state operator in question. The aim of these restrictions is to preserve the level of granularity of the data type avoiding trivial answers for the problems that arise. Furthermore, we will assume that every data type has a distinguished initial element.

Definition 4.4.2.1. As in 2.4, given any model of process algebra, a semantic action relation can be defined as follows:

$$p \xrightarrow{a} q \text{ iff } p = p + aq$$

$$p \xrightarrow{a} \surd \text{ iff } p = p + a$$

where the equality is valid in the model under consideration.

A process p is *regular* if there are only a finite number of processes q_i such that there are sequences σ_i with $p \xrightarrow{\sigma_i}{}^{*} q_i$. In [BW90] it is shown that the regular processes are exactly the processes obtained as solutions of linear recursive equations. □

Definition 4.4.2.2. Let P and Q be two data types. An implementation of P in terms of Q is a 4-tuple

$$K = (\leftarrow, \rightarrow, I, R)$$

where $\leftarrow$and $\rightarrow$are the functions used to define a state operator, $I \subseteq A$ and R is a regular process.

we write

$$Q \xrightarrow{K} P$$

or even

$$K(Q) = P$$

if and only if the following equality holds:

$$\tau \cdot P = \tau \cdot \tau_I(R \leftarrow q)$$

where q is the initial element of Q.

We impose the following restriction on the definition of the state operator involved. The only action and effect function considered as valid are defined in one of the following ways for any $a \in A$:

- the action a is inert, this corresponds to an independent action of the finite control involved, or
- given an operation σ

  $$a \rightarrow s = \sigma(\vec{d}, s)$$

  and

  $$a \leftarrow s = \begin{cases} a_{\sigma,\vec{d}} & \text{if } \sigma(\vec{d}, s)\downarrow \\ \delta & \text{otherwise} \end{cases}$$

Where $a_{\sigma,\vec{d}}$ is independent of the state, it may depend only on σ and $\vec{d}$.

□

The intuitive meaning of such a restriction is that a state of the data type can be accessed only through the operations explicitly provided and in the way specified by them. The restriction over the $\leftarrow$ function enforces the first point and the one over $\rightarrow$ enforces the second point.

Example 4.4.2.3. As an example of this notion of implementation we implement a stack over a set of four elements in terms of a stack over a set of two. The idea behind such an implementation is to use two items of the stack over the smaller domain to represent one in the other.

Let **4** be the set $\{0, 1, 2, 3\}$ and **2** the set $\{0, 1\}$. The following recursive specification defines a stack with elements in the set **4**.

$$
\begin{aligned}
S_{\mathrm{es}} &= \sum_{d \in \mathbf{4}} \underline{\mathrm{push}}(d) \cdot S_{\mathrm{push}(d,\mathrm{es})} + \underline{\mathrm{empty}} \cdot S_{\mathrm{es}} \\
S_{\mathrm{push}(d,\sigma)} &= \sum_{e \in \mathbf{4}} \underline{\mathrm{push}}(e) \cdot S_{\mathrm{push}(e,\mathrm{push}(d,\sigma))} + \underline{\mathrm{pop}}(d) \cdot S_{\sigma} + \\
&\quad \underline{\text{non-empty}} \cdot S_{\mathrm{push}(d,\sigma)}
\end{aligned}
$$

Moreover we specify algebraically a stack with elements in the set **2** as follows (**1** is the type with only one element)

$$
\begin{array}{rccc}
\mathrm{es}: & & \rightarrow & \mathrm{St} \\
\mathrm{push}: & \mathbf{2} \times \mathrm{St} & \rightarrow & \mathrm{St} \\
\mathrm{pop}: & \mathbf{2} \times \mathrm{St} & \circ\!\!\rightarrow & \mathrm{St} \\
\text{empty}: & \mathrm{St} & \circ\!\!\rightarrow & \mathbf{1} \\
\text{non-empty}: & \mathrm{St} & \circ\!\!\rightarrow & \mathbf{1}
\end{array}
$$

where

$$
\begin{aligned}
\mathrm{es} &= \mathrm{es} \\
\mathrm{pop}(d,\mathrm{push}(d,s)) &= s \\
\text{empty}(\mathrm{es}) &= \text{empty}(\mathrm{es}) \\
\text{non-empty}(\mathrm{push}(d,s)) &= \text{non-empty}(\mathrm{push}(d,s))
\end{aligned}
$$

Now we want an implementation of P in terms of Q. The regular process involved will be

$$
\begin{aligned}
R &= (\underline{\mathrm{push}}(0) \cdot ph(0) \cdot ph(0) + \underline{\mathrm{push}}(1) \cdot ph(1) \cdot ph(0) + \\
&\quad \underline{\mathrm{push}}(2) \cdot ph(0) \cdot ph(1) + \underline{\mathrm{push}}(3) \cdot ph(1) \cdot ph(1)) \cdot S + \\
&\quad \underline{\text{empty}} \cdot R \\
S &= e \cdot R + ne \cdot S' \\
S' &= pp(0) \cdot (pp(0) \cdot S_0 + pp(1) \cdot S_1) + pp(1) \cdot (pp(0) \cdot S_2 + pp(1) \cdot S_3) \\
S_0 &= (\underline{\mathrm{pop}}(0) + \underline{\mathrm{push}}(0) \cdot ph(0) \cdot ph(0) + \underline{\mathrm{push}}(1) \cdot ph(1) \cdot ph(0) + \\
&\quad \underline{\mathrm{push}}(2) \cdot ph(0) \cdot ph(1) + (\underline{\mathrm{push}}(3) \cdot ph(1) \cdot ph(1)) \cdot S + \\
&\quad \underline{\text{non-empty}} \cdot S
\end{aligned}
$$

and S_1, S_2, S_3 similar to S_0

The action and effect functions are defined when they are not trivial as follows (where i represents an internal action):

$$
ph(d) \leftarrow s = i
$$

$$\begin{aligned} ph(d)\rightarrow s &= \text{push}(d,s) \\ e\leftarrow s &= i, \text{ iff } \text{empty}(s)\downarrow \\ e\rightarrow s &= s \\ ne\leftarrow s &= i, \text{ iff non-empty}(s)\downarrow \\ ne\rightarrow s &= s \\ pp(d)\leftarrow s &= i, \text{ iff } \text{pop}(d,s)\downarrow \\ pp(d)\rightarrow s &= \text{pop}(d,s) \end{aligned}$$

moreover the internal action is hidden

$$I = \{i\}$$

It is routine to check that the following equality holds in the theory:

$$\tau \cdot S_{\text{es}} = \tau \cdot \tau_I(R\leftarrow\text{es})$$

□

4.4.3 Implementations using the communication function

In the previous section the finite control interacts with a data type through a state operator. Here we achieve the interaction by using communication. Now we look at both members of the implementation relation as processes and implement a data type using some finite control that communicates with the other data types. Actually we do not need to restrict ourselves to data types since any process can be put in the relation. We will show that when we use data types as processes this notion and the previous one coincide in a sense that we make clear later. As a result, we can define a notion of composition of implementation in the definition with the state operator.

Definition 4.4.3.4. Let P and Q be two processes. An implementation of P in terms of Q is a 4-tuple

$$K = (\gamma, H, I, R)$$

where γ is a communication function, $H, I \subseteq A, \alpha(Q) \subseteq H, \alpha(Q \mid R) \cap H = \emptyset$ and R is a regular process. We also assume that for any atomic action $a \in \alpha(R)$ there is at most one $b \in \alpha(Q)$ such that $\gamma(a,b)\downarrow$.

We write

$$Q \xrightarrow{K} P$$

or even

$$K(Q) = P$$

if and only if the following equality holds:

$$\tau \cdot P = \tau \cdot \tau_I \circ \partial_H(Q \parallel R)$$

Although the communication function γ does not appear explicitly it is used in the definition of the merge. □

Example 4.4.3.5. We give the same example as before for this notion of implementation.

A stack over a set with two elements will be defined as

$$\begin{aligned} T_{\text{es}} &= \sum_{d \in \mathbf{2}} \underline{\text{push2}}(d) \cdot T_{\text{push}(d,\text{es})} + \underline{\text{empty2}} \cdot T_{\text{es}} \\ T_{\text{push}(d,\sigma)} &= \sum_{e \in \mathbf{2}} \underline{\text{push2}}(e) \cdot T_{\text{push}(e,\text{push}(d,\sigma))} + \underline{\text{pop2}}(d) \cdot T_\sigma + \\ &\quad \underline{\text{non-empty2}} \cdot T_{\text{push}(d,\sigma)} \end{aligned}$$

Now take as the process R the same as in example 4.4.2.3, the function γ is defined as

$$\begin{aligned} \gamma(ph(d), \underline{\text{push2}}(d)) &= i \\ \gamma(pp(d), \underline{\text{pop2}}(d)) &= i \\ \gamma(e, \underline{\text{empty2}}) &= i \\ \gamma(ne, \underline{\text{non-empty2}}) &= i \end{aligned}$$

and δ otherwise.

The set H and I are defined as

$$H = \{\underline{\text{push2}}(d), \underline{\text{pop2}}(d), \underline{\text{empty2}}, \underline{\text{non-empty2}}, ph(d), pp(d), e, ne | d \in \mathbf{2}\}$$

$$I = \{i\}$$

It is again routine to prove that:

$$\tau \cdot S_{\text{es}} = \tau \cdot \tau_I \circ \partial_H(T_{\text{es}} \parallel R)$$

□

4.4.4 Relating different notions of implementation

One of the desirable properties of implementation is that one can compose them. In this framework only the so-called vertical composition is meaningful. There is no immediate

way to compose implementations since there could be a clash of names of the internal actions. One solution is to choose new names in order to avoid confusion of the internal names of a data type, but if one wants the composition to be associative, then these names should be chosen in a canonical way. It seems that this presents no theoretical problems but increases the bulk of definitions needed.

Now we want to compare both notions of implementation defined above. The result is hardly surprising given that the intuition behind both definitions is the same.

Theorem 4.4.4.6. *Let P, Q be two data types. There is a bijective correspondence between implementations of P in terms of Q using the state operator and using regular processes.*

Proof.

We define two mappings, S that given an implementation using regular processes gives one that uses the state operator and T in the opposite direction.

$$T(\leftarrow, \rightarrow, I, R) = (\gamma, H, I, R)$$

where

$$\gamma(a, \sigma(\vec{d})) = a_{\sigma,\vec{d}}$$

$$H = \{a \in \alpha(R) | a \text{ is not inert}\} \cup \alpha(Q)$$

The mapping S is defined by

$$S(\gamma, H, I, R) = (\leftarrow, \rightarrow, I, R)$$

where

$$a{\leftarrow}q = \begin{cases} a \mid \pi_1(Q_q) & \text{if } a \in H \\ a & \text{if } a \notin H \end{cases}$$

The function *act* is well-defined because any atomic action of R communicate at most with one of Q.

$$a{\rightarrow}q = \begin{cases} \sigma(\vec{d}, q) & \text{if } a \in H \text{ and } \gamma(a, \sigma(\vec{d}))\downarrow \\ q & \text{if } a \notin H \end{cases}$$

It is easy to see that these two maps are the inverse of each other. What remains to be demonstrated is that they preserve implementations, i.e. for any processes P, Q and implementation K,

$$K(Q) = P \Rightarrow S(K)(Q) = P$$

and

$$K(Q) = P \Rightarrow T(K)(Q) = P$$

or, in other words,

$$S(K)(Q) = K(Q)$$

and

$$T(K)(Q) = K(Q)$$

We can prove the last two equations by using AIP. (We only show the inductive case of the first equality, the second one is similar.)

$$\begin{aligned}
\pi_{n+1}(S(K)(Q)) &= \pi_{n+1}(R_i \leftarrow q) \\
&= \pi_{n+1}(\Sigma a_{ij} \leftarrow q \cdot R_j \leftarrow (a_{ij} \rightarrow q)) \\
&= \Sigma a_{ij} \leftarrow q \cdot \pi_n(R_j \leftarrow (a_{ij} \rightarrow q)) \\
\text{(by I.H.)} \quad &= \Sigma a_{ij} \leftarrow q \cdot \pi_n(\partial_H(Q_{a_{ij} \rightarrow q} \parallel R_j)) \\
\text{(by def. of } a_{ij} \leftarrow q) \quad &= \Sigma_{a \in H}(a \mid b_q) \cdot \pi_n(\partial_H(Q_{a \rightarrow q} \parallel R_j)) + \Sigma_{a \notin H} a \cdot \pi_n(\delta_H(Q_{a \rightarrow q} \parallel R_j)) \\
&= \pi_{n+1}(\partial_H(Q_q \mid R_i)) + \pi_{n+1}(\partial_H(Q_q \lfloor\!\lfloor R_i)) \\
&= \pi_{n+1}(\partial_H(Q \parallel R))
\end{aligned}$$

□

Chapter 5

A taxonomy of process algebra

5.1 Introduction

In this chapter the state operator is studied from the point of view of its defining power. Given the definition of the operator based on two functions over a set of states and a set of atomic actions, some restriction should be imposed in order to avoid a trivial answer to the question of whether a process is definable or not. For example, if every computable function is allowed then all processes with uniformly bounded nondeterminism, i.e. with a constant bound for the outdegree of each node in a representing graph, are definable. The restriction we make is a strong one, we take only finite sets of atomic actions and states.

A slighty more general version of this operator is also studied, the so-called generalized state operator, which already appeared in [BB88]

We study the two systems that already appear in [BB91a]. The first consists of processes defined only by sequential and alternative composition that run in a (global) context. This class of processes is closely related to the graphs of pushdown automata [Cau90b]. The second class allows the occurrence of the state operator inside the recursive specification and constitutes an interesting and stable subclass (as we will show) of the class of processes definable in ACP. For example, it is closed under sequential and parallel composition. In [BB91a] some examples are presented, for instance a bag and a queue. In this system the power of the state operator is also used to encode any recursive specification as a linear specification.

We introduce two other systems similar to the previous two but in which we use the generalized state operator of [BB88] instead of the state operator. The first of them, i.e. when the generalized state operator cannot be used inside a recursive specification, is shown to be equivalent to the corresponding one with only the state operator. However, when the generalized state operator is introduced inside the recursive specification we will see that some non-uniformly finitely branching processes are definable, and this implies that the system is more powerful than the corresponding one with the (regular)

state operator.

We study also the system BPP (see [Chr93]) of basic parallel processes and we introduce BPPA as the smallest class of processes containing BPP and BPA and closed under sequential and parallel composition.

5.2 Process algebra with a NIL process

5.2.1 Introduction

Since most of the results of this chapter are simpler in a process algebra with only one mode of termination we introduce here process algebra with a NIL process. We claim that all results to be presented (unless explicitely stated) also hold in a setting with two modes of termination (viz. successful and unsuccessful termination).

The constant NIL does not appear traditionally in process algebra. Its introduction complicates the axiom systems, therefore in most of the works it is avoided. We use it here in order to be able to work with algebras with only one mode of termination, and this will simplify some results in this chapter. The theory of process algebras with a NIL process was developed in [Mol89, BV89].

The signature will be the same as for ACP with the exception that δ will be replaced by NIL. In this section we present the axioms for the systems studied in the previous one with the new signature.

5.2.2 Basic Process Algebra

The theory $\mathrm{BPA}_{\mathrm{NIL}}$ has a restricted signature with only A, NIL, + and $\cdot$. The axioms are given in table 5.1. We claim that all results to be presented also hold in a setting with both modes of termination (viz. successful and unsuccessful termination).

Axiom A4 of BPA has been replaced by the weaker A4' (from [BV89], which is essentially equivalent to the following conditional axiom:

$$x \neq \mathrm{NIL}, y \neq \mathrm{NIL} \Rightarrow (x + y) \cdot z = x \cdot z + y \cdot z$$

In the absence of the precondition we obtain the following counterintuitive equality:

$$a \cdot b = (a + \mathrm{NIL}) \cdot b = a \cdot b + \mathrm{NIL} \cdot b = a \cdot b + b$$

5.2.3 Process Algebra

The signature of the theory $\mathrm{PA}_{\mathrm{NIL}}$ contains $\|$ and $\lfloor\!\lfloor$ besides the elements of the signature of $\mathrm{BPA}_{\mathrm{NIL}}$. The $\|$ represents the free merge. The additional axioms are presented in table 5.2 (a ranges over A).

A1	$x + y = y + x$
A2	$x + (y + z) = (x + y) + z$
A3	$x + x = x$
A4'	$(a \cdot x + b \cdot y + z)w = a \cdot x \cdot w + (b \cdot y + z)w$
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
A6'	$x + \mathrm{NIL} = x$
A8	$\mathrm{NIL} \cdot x = x$
A9	$x \cdot \mathrm{NIL} = x$

Table 5.1: Axioms of $\mathrm{BPA}_{\mathrm{NIL}}$

CM1	$x \parallel y = x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x$
CM2'	$\mathrm{NIL} \mathbin{\lfloor\!\lfloor} x = \mathrm{NIL}$
CM3	$a \cdot x \mathbin{\lfloor\!\lfloor} y = a \cdot (x \parallel y)$
CM4	$(x + y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z$

Table 5.2: Additional axioms of $\mathrm{PA}_{\mathrm{NIL}}$

5.2.4 Algebra of Communicating Processes

The theory called ACP is presented. The theory is parametrized by a partial communication function γ that indicates which atomic actions communicate. This function is assumed to be commutative and associative. The axioms of table 5.1 should be extended with the axioms of table 5.3 ($a, b \in A$).

5.2.5 Renamings

A feature that can be added to the previous algebras is the possibility of renaming atomic actions, given a function $f : A \to A \cup \{\mathrm{NIL}\}$. The operator ρ_f is defined in table 5.4 ($a \in A$).

5.2.6 Projections

Any of the signatures defined above can be extended by an infinite set of unary operators π_n with n a natural number greater than or equal to 0. The intended meaning of $\pi_n(P)$ (in some appropriate model) is the process that behaves as P but stops after executing n steps. The axioms for the projection operators are given below in table 5.5 ($a \in A$).

CM1	$x \parallel y = x \between y + y \between x + x \mid y$	
CM2'	$\mathrm{NIL} \between x = \mathrm{NIL}$	
CM3	$a \cdot x \between y = a \cdot (x \parallel y)$	
CM4	$(x + y) \between z = x \between z + y \between z$	
CF1	$a \mid b = \gamma(a, b)$ if $\gamma(a, b) \downarrow$	
CF1'	$a \mid b = \mathrm{NIL}$ if $\gamma(a, b) \uparrow$	
CF1"	$\mathrm{NIL} \mid x = \mathrm{NIL}$	
CM5'	$a \cdot x \mid b \cdot y = (a \mid b) \cdot (x \parallel y)$ if $\gamma(a, b) \downarrow$	
CM5"	$a \cdot x \mid b \cdot y = \mathrm{NIL}$ if $\gamma(a, b) \uparrow$	
CM8	$(x + y) \mid z = x \mid z + y \mid z$	
CM9	$x \mid (y + z) = x \mid y + x \mid z$	
D1'	$\partial_H(\mathrm{NIL}) = \mathrm{NIL}$	
D1	$\partial_H(a \cdot x) = \mathrm{NIL}$	if $a \in H$
D2	$\partial_H(a \cdot x) = a \cdot \partial_H(x)$	if $a \notin H$
D3	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	
D4	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	

Table 5.3: Additional axioms of $\mathrm{ACP}_{\mathrm{NIL}}$

RN0'	$\rho_f(\mathrm{NIL}) = \mathrm{NIL}$
RN2'	$\rho_f(a \cdot y) = f(a) \cdot \rho_f(y)$ if $f(a) \neq \mathrm{NIL}$
RN2"	$\rho_f(a \cdot y) = \mathrm{NIL}$ if $f(a) = \mathrm{NIL}$
RN3	$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$

Table 5.4: Renamings

PR1'	$\pi_n(\mathrm{NIL}) = \mathrm{NIL}$
PR2'	$\pi_0(x) = \mathrm{NIL}$
PR3	$\pi_{n+1}(a \cdot x) = a \cdot \pi_n(x)$
PR4	$\pi_n(x + y) = \pi_n(x) + \pi_n(y)$

Table 5.5: Projections

5.2.7 State Operator with a NIL Process

In a context with the process NIL instead of δ we can adapt the definition of the state operator in the following way:

SA1	$x\leftarrow\emptyset = \text{NIL}$	SE1	$x\rightarrow\emptyset = \emptyset$
SA2	$x\leftarrow\{s\} = x\leftarrow s$	SE2	$x\rightarrow\{s\} = x\rightarrow s$
SA3	$x\leftarrow(S \cup T) = (x\leftarrow S) + (a\leftarrow T)$	SE3	$x\rightarrow(S \cup T) = x\rightarrow S \cup x\rightarrow T$
SA4'	$\text{NIL}\leftarrow s = \text{NIL}$	SE4'	$\text{NIL}\rightarrow s = s$
SA5	$a \cdot x\leftarrow s = (a\leftarrow s) \cdot (x\leftarrow(a\rightarrow s))$	SE5	$a \cdot x\rightarrow s = x\rightarrow(a\rightarrow s)$
SA7	$(x + y)\leftarrow s = x\leftarrow s + y\leftarrow s$	SE7	$(x + y)\rightarrow s = x\rightarrow s \cup y\rightarrow s$

Table 5.6: Axioms for the state operator with NIL

We introduce an inert state I as before, and a blocked state 0 such that for any atomic action $a, a\leftarrow 0 = \text{NIL}$ and $a\rightarrow 0 = 0$. Furthermore we require that if $a\leftarrow s = \text{NIL}$ then $a\rightarrow s = 0$.

In this chapter we restrict ourselves to the case where the effect function $\rightarrow$ can produce at most one state, and use the convention that if $a\rightarrow s = \emptyset$ then $a\leftarrow s = \text{NIL}$.

We can axiomatize the behaviour of these two states with the axioms in table 5.7. As before, both axioms are satisfied for all definable processes.

SA8	$x\leftarrow I = x$	SE8	$x\rightarrow I = I$
SA9'	$x\leftarrow 0 = \text{NIL}$	SE9	$x\rightarrow 0 = 0$

Table 5.7: Axioms for the special states $I, 0$

5.3 Preliminaries

5.3.1 The generalized state operator

For some applications (see for example [BB88, Vaa90]) the state operator as previously defined is, if not insufficient, inappropriate. A slight generalization of this operator is given below. We keep the original asymmetric notation since in this chapter we are interested only in the process obtained by an application of the operator and not in the set of states reachable from a certain one.

Definition 5.3.1.1. Given a finite set of states S, and two functions:

$$Act : A \times S \longrightarrow \wp(A)$$

$$Eff : A \times A \times S \longrightarrow S$$

We assume always the presence of an inert state I and a blocked state 0 with obvious definitions. The generalized state operator Λ is defined by the axioms in table 5.8. As a convention we will assume

$$\sum_{i \in \emptyset} x = \text{NIL}$$

□

GSO1'	$\Lambda_s(\text{NIL}) = \text{NIL}$
GSO2	$\Lambda_s(a \cdot x) = \sum_{b \in Act(a,s)} b \cdot \Lambda_{Eff(a,b,s)}(x)$
GSO3	$\Lambda_s(x + y) = \Lambda_s(x) + \Lambda_s(y)$
GSO4	$\Lambda_0(x) = \text{NIL}$
GSO5	$\Lambda_I(x) = x$

Table 5.8: Axioms for the generalized state operator

It is immediate that if for any a, s, it holds that $|Act(a, s)| \leq 1$, then the generalized state operator can be replaced by a normal state operator. The case when $|Act(a, s)| = 0$ corresponds to the case in which action a is blocked in state s.

5.4 The systems

5.4.1 Basic parallel processes

The BPP specifications use alternative composition, parallel composition and action prefix. The general sequential composition is not allowed. In [Chr93] a normal form was found for specifications in BPP. Throughout this section we will assume BPP specification E with variables $\text{Var}(E) = \{X_1 \ldots X_n\}$, such that each variable is defined as

$$X_i = \sum_{j=1}^{m_i} a_{ij} \cdot \alpha_{ij}$$

where α_{ij} is a parallel composition of variables which will be identified with a multiset (possibly empty for the case of NIL), when it is convenient. We also assume that every variable appears in at least one multiset reachable from the root (otherwise it can be removed from the specification).

5.4.2 Recursive definitions with the state operator

In this section some results of [BB91a] and [Bla92] are revised and new results are obtained. We keep the λ in the names of the classes to indicate the presence of the state operator.

The concepts defined in definition 2.1.7.1 can be used also in a context with the state operator. However, it may not be immediate how to extend the property of linearity to this more general framework. If the definition above is used without modification then the state operator would not appear at all in a linear specification. The most obvious extension, the one we will adopt is to require equations of the form

$$X = \sum a_i \cdot X_i \leftarrow \sigma_i$$

Sometimes we will make explicit the terminating actions as

$$X = \sum a_i \cdot X_i \leftarrow \sigma_i + \sum b_j$$

This includes the form given in definition 2.1.7.1 due to the possibility of having an inert state.

The classes of processes considered here are:

- $\mathrm{BPA}_{\mathrm{NIL}}$rec ($\mathrm{PA}_{\mathrm{NIL}}$rec, ACPrec, BPPrec,): processes defined by a guarded recursive specification over $\mathrm{BPA}_{\mathrm{NIL}}$ ($\mathrm{PA}_{\mathrm{NIL}}$, ACP,BPP).
- $\lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec})$: processes obtained by an application of the state operator to a process in $\mathrm{BPA}_{\mathrm{NIL}}$rec.
- $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)$lin: processes defined by a linear recursive specification over $\mathrm{BPA}_{\mathrm{NIL}} + \lambda$.
- $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)$rec: processes defined by a guarded recursive specification over $\mathrm{BPA}_{\mathrm{NIL}} + \lambda$.
- $\Lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec})$: processes obtained by an application of Λ to a process in $\mathrm{BPA}_{\mathrm{NIL}}$rec.
- $(\mathrm{BPA}_{\mathrm{NIL}} + \Lambda)$lin: processes defined by a linear recursive specification over $\mathrm{BPA}_{\mathrm{NIL}} + \Lambda$.
- $(\mathrm{BPA}_{\mathrm{NIL}} + \Lambda)$rec: processes defined by a guarded recursive specification over $\mathrm{BPA}_{\mathrm{NIL}} + \Lambda$.

5.4.3 An inductive class of processes

In this section we define a new class of processes that includes both BPArec and BPPrec but is strictly contained in PArec and also, as we shall see, in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)$lin.

One difference with the classes defined above is that this one is not defined as the processes that are solutions of certain recursive specifications, but in an inductive way.

Definition 5.4.3.1. Given a model of process algebra we define the subclass BPPArec by the following inductive definition:

1. $\text{BPA}_{\text{NIL}}\text{rec} \subseteq \text{BPPArec}$
2. $\text{BPPrec} \subseteq \text{BPPArec}$
3. if $p, q \in \text{BPPArec}$, then also $p \cdot q$ and $p \parallel q$ are in BPPArec

□

This class will be, by definition, the smallest class containing both BPA and BPP and closed under parallel and sequential composition.

5.5 Definability with the state operator

5.5.1 $\Lambda(\text{BPA}_{\text{NIL}}\text{rec}) = \lambda(\text{BPA}_{\text{NIL}}\text{rec})$

In this section we will show that by applying either the state operator or the generalized state operator to a BPA specification one obtains the same set of processes.

Theorem 5.5.1.1. *Any process definable in* $\Lambda(\text{BPA}_{\text{NIL}}\text{rec})$ *is definable in* $\lambda(\text{BPA}_{\text{NIL}}\text{rec})$ *as well (the property FAP is required).*

Proof. Let

$$X_i = \sum_{j=1}^{k_i} a_{ij} \cdot t_{ij}$$

be a specification in BPA_{NIL}, here we use the head normal form property introduced in 2.1.7.1. Let S, Act and Eff be the set of states and the functions defining the generalized state operator. For any $a \in A, s \in S$ enumerate $Act(a, s)$ as follows $\{a_s^1, \ldots a_s^{n_s^a}\}$, where $n_s^a = |Act(a, s)|$.

Now, define the following recursive specification in which we enlarge the set of atomic actions by adding the actions $< a, b, s >$, with $a \in A, b \in A, s \in S$.

$$X_i' = \sum_{j,s} \sum_{b \in Act(a_{ij}, s)} < a_{ij}, b, s > \cdot t_{ij}'$$

where $t_{ij}' = t_{ij}[X_i := X_i']$.

We define the action and effect functions in the following way;

$$< a, b, s > \leftarrow s = b$$

$< a, b, s > \leftarrow s'$ is blocked if $s \neq s'$

$$< a, b, s > \rightarrow s = Eff(a, b, s)$$

From this definition it follows that

$$X' \leftarrow s = \Lambda_s(X)$$

since

$$X' \leftarrow s = \sum_j \sum_{b \in Act(a_{ij}, s)} b \cdot t'_{ij} \leftarrow Eff(a_{ij}, b, s)$$

which is precisely the equation for $\Lambda_s(X)$, then, by RSP, they define the same process. □

5.5.2 $\lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec}) \subset (\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$

The main result of this section is that $\lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec}) \subset (\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$. As a corollary we have that $\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec} \subset (\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$. A similar result (but for BPA_δ instead of $\mathrm{BPA}_{\mathrm{NIL}}$) appeared in [Bla92].

This result is also a consequence of the more general one in section 5.5.4.

Lemma 5.5.2.2. *Let X be a process in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$, then for every state s, $X \leftarrow s$ is a process in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$.*

Proof. By linearity, we can write the specification for X in the form

$$X = \sum_{i<n} a_i \cdot (X_i \leftarrow s_i)$$

where the X_i are defined also by a linear specification over $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$, with $s_i \in S$ for all $i < n$.

We can take now $S \times S$ as state space (as in 2.6.3.7) and define Y as:

$$Y = \sum_{i<n} a_i \leftarrow s \cdot X_i \leftarrow < s_i, s >$$

It is immediate to see that $X \leftarrow s$ is a solution of Y. □

Using the notation of the beginning of this section we can paraphrase this lemma as $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin} = \lambda((\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin})$.

The following lemma appeared in [BB91b] but the proof was not correct. This lemma is a consequence of lemma 5.5.4.18 as well, but the proof presented here is less complicated.

Lemma 5.5.2.3. *Let X be a process in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{rec}$. Then X is in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$ (using FAP).*

Proof. Let a recursive specification E over $\mathrm{BPA}_{\mathrm{NIL}}$ be given that uses variables from $X_1, \ldots, X_n$ and has X_1 as solution. Assume the specification of each X_i is in restricted GNF, i.e.

$$X_i = \sum_j a_{ij} \cdot X_{p_{ij}} \cdot X_{q_{ij}} + \sum_k b_{ik} \cdot X_{r_{ik}} + \sum_l c_{il}$$

Define (using another set of atomic actions)

$$X = \sum_{i,j} < a_{ij}, i > \cdot (X \leftarrow p_{ij} \leftarrow q_{ij}) + \sum_{i,k} < b_{ik}, i > \cdot (X \leftarrow r_{ik}) + \sum_{i,l} < c_{il}, i > \cdot X$$

The state space S is $\{init, 1, \ldots, n\}$. The functions *act* and *eff* are trivial except in the following cases:

$$\begin{aligned} &< a, i > \rightarrow m = \emptyset \text{ if } i \neq m \\ &< a, i > \rightarrow init = \emptyset \\ &< a, i > \leftarrow i = a \\ &< a, i > \rightarrow i = I \end{aligned}$$

Now we claim that for each sequence $n_1, \ldots, n_m$ the following equality holds

$$X_{n_1} \cdots X_{n_m} = X \leftarrow n_1 \leftarrow \cdots \leftarrow n_m \leftarrow init$$

The intuition underlying this construction is that the state operators indicate what the process still has to do

We prove the claim by showing that both sides of the equality satisfy the same (infinite) recursive specification. We abbreviate $X \leftarrow s_1 \cdots \leftarrow s_n$ by $X \leftarrow s_1 \ldots s_n$ Let σ be a sequence of states.

$$\begin{aligned} X \leftarrow i \leftarrow \sigma \leftarrow init &= \sum_j a_{ij} \cdot X \leftarrow p_{ij} \leftarrow q_{ij} \leftarrow I \leftarrow \sigma \leftarrow init + \\ &\quad + \sum_k b_{ik} \cdot X \leftarrow r_{ik} \leftarrow I \leftarrow \sigma \leftarrow init + \sum_l c_{il} \cdot X \leftarrow I \leftarrow \sigma \leftarrow init \\ &= \sum_j a_{ij} \cdot X \leftarrow p_{ij} \leftarrow q_{ij} \leftarrow \sigma \leftarrow init + \\ &\quad + \sum_k b_{ik} \cdot X \leftarrow r_{ik} \leftarrow \sigma \leftarrow init + \sum_l c_{il} \cdot X \leftarrow \sigma \leftarrow init \end{aligned}$$

(note that the process terminates when it can do a c-action into $X \leftarrow init$ which equals NIL)

It follows that if we replace $X \leftarrow n_1 \leftarrow \cdots \leftarrow n_m \leftarrow init$ by $X_{n_1} \cdots X_{n_m}$ the equations above are valid. Note that when σ is the empty string, then $X \leftarrow init = \mathrm{NIL}$.

Note that for this proof, reduction to *restricted* GNF was not necessary. A similar construction works for any specification in GNF. □

We are now able to prove the main result of this section.

Theorem 5.5.2.4. *Let X be a process in $\lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec})$. Then X is in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$.*

Proof. Immediate from lemmas 5.5.2.2 and 5.5.2.3. □

5.5.3 BPP $\subset$ (BPA$_{\text{NIL}}$ + λ)lin

In this section we show that the so called basic parallel processes can be defined by a linear specification using the state operator.

Theorem 5.5.3.5. *Let p be the solution of a recursive specification in* BPP, *then* $p \in$ (BPA$_{\text{NIL}}$ + λ)lin *(FAP used).*

Proof. Let

$$X_i = \sum a_{ij} \cdot \alpha_{ij}$$

with $0 < i < n$ be a specification in BPP(so $\forall i, j, \alpha_{ij}$ is a parallel composition of recursion variables). We define the following specification in which we look at α_{ij} as a multiset. If α is a multiset of recursion variables, we denote by α' the multiset of indexes of the variables in α. We take this set of indexes (plus an initial state *init*) as state space, and use the notation introduced in 2.6.3.6. The set of atomic actions is extended with actions $< a, i >, a \in A, i < n$.

$$X = \sum_{i,j} < a_{ij}, i > \cdot X \leftarrow \alpha'_{ij}$$

where

$$< a, i > \leftarrow i = a$$

$$< a, i > \leftarrow j = < a, i > \text{ if } i \neq j$$

$$< a, i > \leftarrow init \text{ is blocked}$$

$$< a, i > \rightarrow i = I$$

The definition is correct since for any state i it holds that

$$\alpha(i) = \{< a, i > | a \in A\}$$

and they are disjoint for different states i, j, furthermore

$$\lambda_i(\alpha(i)) = A$$

and A is disjoint with any $\alpha(i)$.

Now we claim that

$$X \leftarrow i \leftarrow init = X_i$$

In order to prove that, we show that for any multiset α it holds that

$$X \leftarrow \alpha' \leftarrow init = \alpha$$

First we note that for any multiset (or equivalently, for any parallel composition) α of recursion variables the following equality holds:

$$\alpha = \sum_{X_i \in \alpha} a_{ij} \cdot (\alpha - \{X_i\} \cup \alpha_{ij})$$

This is the case since a parallel composition can perform a step if and only if one of its components can do it.

Moreover,

$$\begin{aligned} X{\leftarrow}\alpha'{\leftarrow}init &= (\sum_{i\in\alpha'}(< a_{ij}, i > {\leftarrow}\alpha') \cdot (X{\leftarrow}(< a_{ij}, i > {\rightarrow}\alpha')){\leftarrow}init \\ &= \sum_{i\in\alpha'} a_{ij} \cdot X{\leftarrow}((\alpha' - \{i\}) \cup \{I\} \cup \alpha'_{ij}){\leftarrow}init \end{aligned}$$

It follows that both are solutions of the same (infinite) recursive specification, and then they are equal by RSP.

□

5.5.4 $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{rec} = (\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$

The result of this section shows that the class of processes $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$ is stable, in the sense that the obvious generalization of admitting guarded recursion does not increase it. All the complexity of general sequential composition in the recursive specifications can be encoded in the state operator. This also allows us to work in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$ any time we want to do it in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{rec}$ with the obvious adventage in simplicity.

The following lemma shortens some of the proofs.

Lemma 5.5.4.6. *Let E be a specification of a process in* $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{rec}$ *with main variable X. We can construct another specification with only one recursion variable Y such that there exist a state s for which*

$$X = Y{\leftarrow}s$$

(FAP needed).

Proof. Let

$$E = \{X_i = \sum a_{ij} \cdot t_{ij} | 1 \leq i \leq n\}$$

Define (with an extended set of atomic actions)

$$X = \sum_{i,j} < a_{ij}, i > \cdot t'_{ij}$$

where

$$t'_{ij} = t_{ij}[X_i := X{\leftarrow}i]$$

($t[X := t']$ means syntactic substitution of variable X by t' in term t).

The state space will be extended with states $\{1, \ldots, n\}$. The state operator is defined on this set by

$$< a, i > {\leftarrow}i = a$$

$$< a, i > \rightarrow j = \emptyset \text{ if } i \neq j$$
$$< a, i > \rightarrow i = I$$

Now, we claim that $\{X \leftarrow i | 1 \leq i \leq n\}$ is a solution of E.

$$X \leftarrow i = \sum(< a_{kj}, k > \leftarrow i) \cdot (t'_{ij} \leftarrow I)$$
$$= \sum a_{ij} \cdot (t'_{kj} \leftarrow I)$$
$$= \sum a_{ij} \cdot t'_{ij}$$

but then our claim follows by definition of the t'_{ij}. □

Definition 5.5.4.7.

1. A term is in *product normal form (pnf)* if it is constructed only with recursion variables, sequential composition $(\cdot)$ and the state operator $(\leftarrow)$.
2. A guarded recursive specification

$$E = \{X_i = \sum_j a_{ij} \cdot t_{ij} | 1 \leq i \leq n\}$$

is in pnf if for all i, j, t_{ij} is in pnf.

□

Lemma 5.5.4.8. *Let p be a solution of a recursive specification in* $(\text{BPA}_{\text{NIL}} + \lambda)$rec. *Then, there is a specification in pnf of the same process.*

Proof. Let E be a specification in $(\text{BPA}_{\text{NIL}} + \lambda)$rec. The right hand sides of E can be rewritten using axiom A4 as a rewriting rule from left to right. For simplicity we can assume by the previous lemma that E has only one recursive equation, say

$$X = \sum a_i \cdot p_i$$

For any i, if p_i is not in pnf then it can be written without loss of generality as

$$p_i = q_i \cdot (p + p')$$

where the q_i is in pnf, since the state operator distributes with respect to the $+$ and q_i can possibly be NIL. Take then a new variable X' and replace p_i by $q_i \cdot X'$ and add a new equation

$$X' = p + p'$$

By the previous lemma, this new specification can be rewritten again in a new one with only one equation and then repeat the procedure for the new specification. Since in p and in p' there are strictly less occurrences of the operator $+$ the procedure will terminate. □

Definition 5.5.4.9. Given a specification in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)$rec with main variable X we define the set T of terms inductively as follows:

- $\mathrm{NIL} \in T$,
- if $\beta \in T, \beta \neq \mathrm{NIL}$ then for any state $s, \beta \leftarrow s \in T$, and
- if $\beta \in T$ then for any state $s, (\beta \cdot X \leftarrow s) \in T$

□

The following definition generalizes the Greibach Normal Form of BPA for the case where the state operator can occur inside a specification.

Definition 5.5.4.10. A specification in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)$rec is in *GNF* if it has the following form:

$$X = \sum_i a_i \cdot p_i$$

where for all i, it holds that $p_i \in T$. Note that lemma 5.5.4.6 allows us to restrict this definition to the case where there is only one variable without losing generality.

□

Theorem 5.5.4.11. *Given a specification in* $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)$rec *another specification can be written in GNF which specifies the same process.*

Proof. Take a specification in pnf

$$X = \sum a_i \cdot p_i$$

The general form of the p_i will be

$$p_i = (\cdots((X \leftarrow s_i \cdot p_i^1) \leftarrow s^1 \cdot p_i^2) \leftarrow s^2 \cdots p_i^n) \leftarrow s^n$$

This form is general enough given that a finite sequence of states can be encoded as a state as in 2.6.3.9 without losing the condition of finiteness of the state space.

Now, for any i, j if p_i^j is not a variable possibly modified by some states, then replace it by a new variable X_{ij}. Add then a new equation

$$X_{ij} = p_i^j$$

for all the new variables. We write the new equation for X as

$$X = \sum a_i \cdot p_i'$$

This new specification is then compressed into a specification with only one variable and the process is repeated. The process will terminate since

$$p_i^j = (\cdots((X \leftarrow t \cdot q_i^1) \leftarrow t^1 \cdot q_i^2) \leftarrow t^2 \cdots q_i^m) \leftarrow t^m$$

and after unfolding all the occurrences of the state operator it is equal to the following term:

$$\sum a_i \leftarrow t, t^1, \ldots, t^n \cdot (\cdots((p_i' \leftarrow (a \rightarrow t) \cdot q_i^1) \leftarrow ((a \leftarrow t) \rightarrow t^1)) \cdots q_i^m) \leftarrow ((a \leftarrow t, t^1, \ldots t^{m-1}) \rightarrow t^m)$$

where all the q_i are strictly simpler than the p_i. □

A specification in GNF can easily be encoded in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$ since it has the property that every occurence of the state operator in it has one of the following two properties:

- It modifies only one occurrence of the main variable as in $X \leftarrow s$, or
- it modifies all the previous occurrences of the main variable, as s in $(X \leftarrow t \cdot X \leftarrow u) \leftarrow s$.

A state operator will never modify more than one but not all of the previous occurences of the main variable like s in

$$X \leftarrow t \cdot (X \leftarrow u \cdot X \leftarrow v) \leftarrow s$$

The intuition behind our encoding is that any occurrence of a state will be understood as modifying all variables occurring before it, and the case when it only modifies one variable will be solved by a new set of states.

Definition 5.5.4.12. Given the state space S it can be extended with a disjoint copy of itself $\underline{S}$ defined in the following way:

$$\underline{S} = \{\underline{s} | s \in S\}$$

□

Definition 5.5.4.13. We define a function

$$[\![\_]\!] : T \longrightarrow S_X^*$$

inductively over T

$$\begin{aligned} [\![\mathrm{NIL}]\!] &= \epsilon \text{ (the empty sequence)} \\ [\![\beta \leftarrow s]\!] &= [\![\beta]\!] s \\ [\![\beta \cdot X \leftarrow s]\!] &= [\![\beta]\!] \underline{s} \end{aligned}$$

□

Remark 5.5.4.14. Note that a non empty sequence in $Im([\![\_]\!])$ (the image of $[\![\_]\!]$) will always begin with a state of the form $\underline{s}$. □

Definition 5.5.4.15. We define a function $\Theta : \underline{S}(S \cup \underline{S})^* \longrightarrow T$ in the following way:

$$\begin{aligned} \Theta(\epsilon) &= \text{NIL} \\ \Theta(\sigma s) &= \Theta(\sigma) \leftarrow s \\ \Theta(\sigma \underline{s}) &= \Theta(\sigma) \cdot X \leftarrow s \end{aligned}$$

□

Lemma 5.5.4.16. *The function* Θ *is the inverse of* $[\![\_]\!]$.

Proof. Straightforward infuction on T and on the length of the sequences over $\underline{S}(S \cup \underline{S})^*$. □

Definition 5.5.4.17. Given a specification

$$X = \sum_i a_i \cdot p_i$$

in GNF, we define the following specification: We define the following specification

$$Y = \sum \underline{a_i} \cdot Y \leftarrow [\![p_i]\!]$$

The state space is the set $S \cup \underline{S}$ and the action and effect function are extended in the following way:

$$\underline{a} \leftarrow \underline{s} = a \leftarrow s$$

$$\underline{a} \rightarrow \underline{s} = a \rightarrow s$$

$$\underline{a} \rightarrow s = \emptyset$$

$$a \leftarrow \underline{s} = a$$

$$a \rightarrow \underline{s} = s$$

□

Theorem 5.5.4.18. *The class of processes* $(\text{BPA}_{\text{NIL}} + \lambda)\text{rec}$ *is included in* $(\text{BPA}_{\text{NIL}} + \lambda)\text{lin}$.

Proof. Let X and Y be as defined in 5.5.4.17. Obviously, $Y \in (\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$. Then, the theorem follows once we have shown that for any state $s \in S$ it holds that

$$X\leftarrow s = Y\leftarrow \underline{s}$$

In order to show that we construct an infinite specification and show that a solution of either side of the equation would satisfy this specification. The result then follows by RSP.

We define the system Z_σ where $\sigma \in \underline{S}(S \cup \underline{S})$.

$$\begin{aligned} Z_\epsilon &= \mathrm{NIL} \\ Z_{\underline{s}\sigma} &= \sum_{i\in I} a_i\leftarrow s\sigma \cdot Z_{[\![p_i]\!](a_i\rightarrow s\sigma)} \end{aligned}$$

First, we will show that

$$Y\leftarrow\sigma = Z_\sigma$$

for any $\sigma \in Im([\![\_]\!])$.

When σ is empty both sides of the equation equal NIL.

$$\begin{aligned} Y\leftarrow\underline{s}\sigma &= \sum \underline{a_i}\leftarrow\underline{s}\sigma \cdot Y\leftarrow[\![p_i]\!]\leftarrow(\underline{a_i}\rightarrow\underline{s}\sigma) \\ &= \sum a_i\leftarrow s\sigma \cdot Y\leftarrow[\![p_i]\!]\leftarrow(a_i\rightarrow s\sigma) \end{aligned}$$

Now, we will show that for any $\sigma \in \underline{S}(S \cup \underline{S})$ the process $\Theta(\sigma)$ is a solution of Z_σ as well. In particular we will obtain that

$$X\leftarrow s = Z_{\underline{s}} = Y\leftarrow\underline{s}$$

We show by induction that $\Theta(\sigma)$ satisfy the specification for Z_σ.

- $\Theta(\epsilon) = \mathrm{NIL}$
- $$\begin{aligned} \Theta(\sigma s) &= \Theta(\sigma)\leftarrow s \\ &= \sum[a_i\leftarrow\sigma \cdot \Theta([\![p_i]\!](a_i\rightarrow\sigma)]\leftarrow s \\ &= \sum a_i\leftarrow\sigma s \cdot \Theta([\![p_i]\!])\leftarrow(a_i\rightarrow\sigma))\leftarrow(a_i\leftarrow\sigma) \\ &= \sum a_i\leftarrow\sigma s \cdot \Theta([\![p_i]\!])\leftarrow(a_i\rightarrow\sigma)(a_i\leftarrow\sigma)) \\ &= \sum a_i\leftarrow\sigma s \cdot \Theta([\![p_i]\!])\leftarrow(a_i\rightarrow\sigma s)) \end{aligned}$$
- $\sigma = \sigma' s$ Left to the reader

- $\sigma = \sigma' \underline{s} t$ where $\sigma' \neq \mathrm{NIL}$, say $\sigma' = x_u \sigma''$.

$$\begin{aligned}
\Theta(\sigma) &= \Theta(\sigma' \underline{s} t) \\
&= (\Theta(\sigma') \cdot X \leftarrow s) \leftarrow t \\
&= (\Theta(x_u \sigma'') \cdot X \leftarrow s) \leftarrow t \\
\text{(IH)} &= (\textstyle\sum a_i \leftarrow u\sigma'' \cdot (\Theta([\![p_i]\!] a_i \rightarrow u\sigma'') \cdot X \leftarrow s)) \leftarrow t \\
&= \textstyle\sum a_i \leftarrow u\sigma'' t \cdot (\Theta([\![p_i]\!] a_i \rightarrow u\sigma'') X \leftarrow s) \leftarrow (a \leftarrow u\sigma'') \rightarrow t \\
&= \textstyle\sum a_i \leftarrow u\sigma'' t \cdot \Theta([\![p_i]\!] (a_i \rightarrow u\sigma'') \underline{s} ((a \leftarrow u\sigma'') \rightarrow t) \\
&= \textstyle\sum a_i \leftarrow u\sigma'' t \cdot \Theta([\![p_i]\!] (a_i \rightarrow u\sigma'' \underline{s} t)
\end{aligned}$$

□

5.5.5 BPPA $\subset$ (BPA$_{\mathrm{NIL}}$ + λ)lin

An immediate corollary of the results of the previous section is the fact that the system (BPA$_{\mathrm{NIL}}$ + λ)lin is closed under sequential composition. Hence, if we can prove that it is closed under parallel composition we obtain the result announced in the title of this section, since we already know that the set of processes definable in BPA and BPP is contained in (BPA$_{\mathrm{NIL}}$ + λ)lin, and BPPA is the smallest set containing it.

Theorem 5.5.5.19. (BPA$_{\mathrm{NIL}}$ + λ)lin *is closed under parallel composition (FAP used).*

Proof. Let $X \leftarrow s$ and $Y \leftarrow t$ be two processes in (BPA$_{\mathrm{NIL}}$ + λ)lin where X and Y are solutions of the following recursive specifications:

$$X = \textstyle\sum a_i \cdot (X \leftarrow s_i)$$

$$Y = \textstyle\sum b_j \cdot (Y \leftarrow t_j)$$

We define the following specification

$$Z = \textstyle\sum_i a_i^X \cdot (Z \leftarrow s_i^X) + \sum_j b_j^Y \cdot (Z \leftarrow t_j^Y)$$

with the state operator defined over the set formed by two disjoints copies of S and an initial element:

$$S^{XY} = S^X \cup S^Y \cup \{init\}$$

where

$$S^X = \{s^X | s \in S\}$$

$$S^Y = \{s^Y | s \in S\}$$

by:

$$\begin{aligned} a^X{\leftarrow}s^X &= (a{\leftarrow}s)^X \\ a^X{\rightarrow}s^X &= (a{\rightarrow}s)^X \\ b^Y{\leftarrow}t^Y &= (b{\leftarrow}t)^Y \\ b^Y{\rightarrow}t^Y &= (b{\rightarrow}t)^Y \\ a^X{\leftarrow}init &= a \\ b^Y{\leftarrow}init &= b \end{aligned}$$

We want to prove that for any sequence $\sigma \in (S^{XY})^*$ the following equation holds:

$$Z{\leftarrow}\sigma{\leftarrow}init = X{\leftarrow}\sigma_X \| Y{\leftarrow}\sigma_Y$$

where σ_X is sequence in S^* defined as

$$\begin{aligned} \epsilon_X &= \epsilon \\ (s^Y\sigma')_X &= \sigma'_X \\ (s^X\sigma')_X &= s\sigma'_X \end{aligned}$$

whose elements are in S^X component of the elements and analogously for σ^Y.

First we state the following proposition (and its symmetric version): For any sequence $\sigma \in (S \times S)^*$ and atomic action a

$$\begin{aligned} a^X{\leftarrow}\sigma &= (a{\leftarrow}(\sigma_X))^X \\ a^X{\rightarrow}\sigma &= \tau \end{aligned}$$

where

$$\begin{aligned} \tau_X &= a{\rightarrow}\sigma_X \\ \tau_Y &= \sigma_Y \end{aligned}$$

The proof is a straightforward induction on the length of σ.

Now we show that for any sequence σ the following two processes

$$Z{\leftarrow}\sigma{\leftarrow}init$$

$$X{\leftarrow}\sigma_X \| Y{\leftarrow}\sigma_Y$$

satisfy the same recursive specification.

$$\begin{aligned} Z{\leftarrow}\sigma{\leftarrow}init &= [(\sum(a_i^X{\leftarrow}\sigma)\cdot(Z{\leftarrow}s_i^X{\leftarrow}(a_i^X{\rightarrow}\sigma)) \\ &+ \sum(b_j^Y{\leftarrow}\sigma)\cdot(Z{\leftarrow}t_j^Y{\leftarrow}(b_j^Y{\rightarrow}\sigma))]{\leftarrow}init \\ &= (\sum(a_i{\leftarrow}\sigma_X)^X\cdot(Z{\leftarrow}s_i^X(a_i^X{\rightarrow}\sigma)) \\ &+ \sum(b_j{\leftarrow}\sigma_Y)\cdot(Z{\leftarrow}t_j(b_j^Y{\rightarrow}\sigma)){\leftarrow}init \\ &= \sum a_i{\leftarrow}\sigma_X\cdot Z{\leftarrow}s_i^X(a_i^X{\rightarrow}\sigma){\leftarrow}init \\ &+ \sum b_j{\leftarrow}\sigma_X\cdot Z{\leftarrow}t_j(b_j^Y{\rightarrow}\sigma){\leftarrow}init \end{aligned}$$

$$\begin{aligned} X{\leftarrow}\sigma_X || Y{\leftarrow}\sigma_Y &= X\sigma_X \between Y\sigma_Y + Y\sigma_X \between X\sigma_Y \\ &= \sum(a_i{\leftarrow}\sigma_X) \cdot [(X{\leftarrow}s_i{\leftarrow}(a_i{\rightarrow}\sigma_X))||(Y{\leftarrow}\sigma_Y)] \\ &+ \sum(b_j{\leftarrow}\sigma_Y) \cdot [(Y{\leftarrow}t_j{\leftarrow}(b_j{\rightarrow}\sigma_Y))||(X{\leftarrow}\sigma_X)] \end{aligned}$$

The desired results follows since:

$$(s_i^X(a_i^X{\rightarrow}\sigma))_X = s_i{\leftarrow}(a_i{\rightarrow}\sigma_X)$$

$$(s_i^X(a_i^X{\rightarrow}\sigma))_Y = (a_i^X{\rightarrow}\sigma)_Y = \sigma_Y$$

□

5.5.6 $(\mathrm{BPA}_{\mathrm{NIL}} + \Lambda)\mathrm{lin} \neq (\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$

The following example shows that there exist processes definable in $(\mathrm{BPA}_{\mathrm{NIL}} + \Lambda)\mathrm{lin}$ that are not uniformly finitely branching. In [BB91a] it is shown that all processes in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$ are uniformly finitely branching. Given the fact already proven that $\Lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec}) = \lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec})$, this result comes as a surprise.

Example 5.5.6.20.

$$X = a \cdot \Lambda_2(X) + b \cdot \Lambda_1(X) + c_1 \cdot X + c_2 \cdot X$$

where the generalized state operator is defined by $(i = 1, 2)$:

$$Act(c_i, i) = \{d_i\}$$

$$Act(c_i, 2-i) = \emptyset$$

$$Eff(c_i, i, d_i) = I$$

$$Act(a, 1) = Act(b, 1) = \{a, b\}$$

$$Eff(a, 1, a) = Eff(b, 1, a) = 1$$

$$Eff(a, 1, b) = Eff(b, 1, b) = 2$$

□

Lemma 5.5.6.21.

1. *Let $\sigma, \tau \in \{1, 2\}^*$. If $\sigma \neq \tau$, then $\Lambda_\sigma(X) \neq \Lambda_\tau(X)$.*
2. *For any n the process $\Lambda_{1^n}(X)$ has outdegree greater or equal than 2^n.*

Proof.

1. It is immediate that for any $\sigma \in \{1,2\}^*$ there exist a trace in $\{d_1, d_2\}^*$ that distinguishes $\Lambda_\sigma(X)$.

2. We show by induction on n that

$$\Lambda_{1^n}(X) = \sum_{\sigma \in \{1,2\}^n} (a \cdot \Lambda_{1\sigma}(X) + b \cdot \Lambda_{2\sigma}(X)) + d_1 \cdot \Lambda_{1^{(n-2)}}(X)$$

When $n = 2$

$$\Lambda_1(X) = \sum_{d \in Act(a,1)} d \cdot \Lambda_{Eff(a,1,d)}(\Lambda_1(X)) + \sum_{d \in Act(b,1)} d \cdot \Lambda_{Eff(b,1,d)}(\Lambda_2(X)) + d_1 \cdot X$$

$$= a \cdot \Lambda_{11}(X) + b \cdot \Lambda_{21}(X) + a \cdot \Lambda_{12}(X) + b \cdot \Lambda_{22}(X) + d_1 \cdot X$$

When $n = m + 2$

$$\Lambda_1(\Lambda_{1^m}(X)) = \Lambda_1(\sum_{\sigma \in \{1,2\}^m} (a \cdot \Lambda_{1\sigma}(X) + b \cdot \Lambda_{2\sigma}(X)) + d_1 \cdot \Lambda_{1^{(m-2)}}(X))$$

$$= \sum_{d \in Act(a,1)} d \cdot \Lambda_{Eff(a,1,d)}(\Lambda_{1\sigma}(X)) + \sum_{d \in Act(b,1)} d \cdot \Lambda_{Eff(f,1,d)}(\Lambda_{2\sigma}(X)) + d_1 \cdot \Lambda_{1^m}(X)$$

$$= \sum_{\sigma \in \{1,2\}^m} (a \cdot \Lambda_{11\sigma}(X) + b \cdot \Lambda_{21\sigma}(X) + a \cdot \Lambda_{12\sigma}(X) + b \cdot \Lambda_{22\sigma}(X)) + d_1 \cdot \Lambda_{1^m}(X)$$

$$= \sum_{\sigma \in \{1,2\}^n} (a \cdot \Lambda_{1\sigma}(X) + b \cdot \Lambda_{2\sigma}(X)) + d_1 \cdot \Lambda_{1^n}(X)$$

□

5.5.7 $\lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec}) \not\subset \mathrm{PA}_{\mathrm{NIL}}\mathbf{rec}$

The next and last result of this section shows that the application of a state operator outside of a $\mathrm{BPA}_{\mathrm{NIL}}$ specification adds to the expressive power of $\mathrm{PA}_{\mathrm{NIL}}$. The converse of this, i.e. that there exist processes definable in $\mathrm{PA}_{\mathrm{NIL}}$ and not in $\lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec})$ (even not in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$), was already answered affirmatively in [BB91a].

Example 5.5.7.22. Consider the following (infinite) recursive specification ($i \geq 0$).

$$\begin{aligned} X_0 &= p \cdot X_1 + a \\ X_{i+1} &= p \cdot X_{i+2} + m \cdot X_i + a \cdot D_i \\ D_0 &= d \\ D_{i+1} &= d \cdot D_i \end{aligned}$$

The intuitive meaning of this process is a counter that once in its existence can do an a-action and then do as many d-actions as the current value of the counter.A similar example appears in [Cau90b]. Note that if the a-actions and the D_i are removed then the usual counter is obtained.

We see that this process is definable in $\lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec})$ as follows.

Let $\mathbf{S} = \{1, 2\}$ and

$$\begin{aligned} a{\leftarrow}1 &= d \\ p{\leftarrow}1, m{\leftarrow}1 & \quad \text{are blocked.} \\ a{\rightarrow}2 &= 1 \end{aligned}$$

and trivial otherwise. Now

$$X_0 = C{\leftarrow}2$$

where

$$\begin{aligned} C &= p \cdot T \cdot C + a \\ T &= p \cdot T \cdot T + m + a \end{aligned}$$

If the a-action is removed in both equations, then C defines a counter.

We show that $C{\leftarrow}2 = X_0$ by RSP.

$D_n = (T^n \cdot C){\leftarrow}1$:

$$\begin{aligned} C{\leftarrow}1 &= (p \cdot T \cdot C){\leftarrow}1 + a{\leftarrow}1 \\ &= \mathrm{NIL} + d \\ &= d \end{aligned}$$

$$\begin{aligned} (T^{n+1} \cdot C){\leftarrow}1 &= (p \cdot T \cdot T^{n+1} \cdot C){\leftarrow}1 + \\ &\quad (m \cdot T^n \cdot C){\leftarrow}1 + (a \cdot T^n \cdot C){\leftarrow}1 \\ &= \mathrm{NIL} + \mathrm{NIL} + d \cdot (T^n \cdot C){\leftarrow}1 \\ &= d \cdot (T^n \cdot C){\leftarrow}1 \end{aligned}$$

$X_n = (T^n \cdot C)\leftarrow 2$:

$$\begin{aligned} C\leftarrow 2 &= (p \cdot T \cdot C)\leftarrow 2 + a\leftarrow 2 \\ &= p \cdot (T \cdot C)\leftarrow 2 + a \end{aligned}$$

$$\begin{aligned} (T^{n+1} \cdot C)\leftarrow 2 &= (p \cdot T \cdot T^{n+1} \cdot C)\leftarrow 2 + \\ &\quad (m \cdot T^n \cdot C)\leftarrow 2 + (a \cdot T^n \cdot C)\leftarrow 2 \\ &= p \cdot (T^{n+2} \cdot C)\leftarrow 2 + m \cdot (T^n \cdot C)\leftarrow 2 + a \cdot (T^n \cdot C)\leftarrow 1 \\ &= p \cdot (T^{n+2} \cdot C)\leftarrow 2 + m \cdot (T^n \cdot C)\leftarrow 2 + a \cdot D_n \end{aligned}$$

□

Lemma 5.5.7.23. *The counter of 5.5.7.22 cannot be finitely defined in* $\mathrm{PA}_{\mathrm{NIL}}$.

Proof. We will now prove that this process is not definable in $\mathrm{PA}_{\mathrm{NIL}}$ looking at the term model (or isomorphically at the graph model, see [BW90]).
The following definitions are adaptations of some appearing in [Vaa91].

Definition 5.5.7.24.

1. Let s be a term in PA_δ. An occurrence of a subterm t of s is *sleeping* in s if it occurs in r in a subterm of s of the form $r' \between r$ or $r' \cdot r$.
2. A subterm t of s is sleeping in s if every occurrence of t in s is sleeping.
3. A subterm that is not sleeping is called *awake*.
4. A subterm t of s is *dead* if for every s' such that $s \xrightarrow{\sigma} s'$ t is sleeping in s'
5. Let E be a recursive specification. A term s is dead in E if it is dead in every right hand side of E.

□

Fact 5.5.7.25. If a term t is dead in a recursive specification E with root X, then X is bisimilar to X', where X' is the root of the specification E' which equals E except that every occurrence of t is replaced for t' (t' any term). □

Fact 5.5.7.26. If $p \parallel q = d^n, d \in \mathbf{A}$ then $\exists m, p$ such that $p = d^m, q = d^p$ and $n = p+m$
□

As a consequence of fact 5.5.7.26, if $p \parallel q = d^n$ then $p \parallel q = \bar{p} \cdot \bar{q}$ where $\bar{x}$ is x where all $\parallel$ and $\between$ are replaced by $\cdot$.

Proposition 5.5.7.27. *Let E be a specification over PA_δ with root X. Then there exists a specification E' with root X' such that $X = X'$ and all occurrences of $\between$ appear inside the scope of a $\|$.*

Proof. Rewrite every $p \between q$ that is outside the scope of a $\|$ into

$$\sum a_i \cdot (t_i \| q) + \sum b_j \cdot q$$

where $\sum a_i \cdot t_i + \sum b_j$ is the head normal form of p. □

Theorem 5.5.7.28. *Let E be a specification with root X of the process X_0 of example 5.5.7.22 over $\mathrm{PA}_{\mathrm{NIL}}$, then there exists a specification E' of the same process over BPA_δ.*

Proof. Let E' be E where all $\|$ and $\between$ are replaced by $\cdot$.

Let $p \| q$ be a subterm of (a right hand side of) E not in the scope of a $\|$ or a $\between$. We will show that $p \| q$ is dead or equal to d^n for some n.

Suppose $X \xrightarrow{\sigma} t$ and $p \| q$ is a subterm of t that is awake and not equal to d^n for any n. Since t cannot be d^n for some n, t should be able to perform an a-action.

As $p \| q$ is awake then

$$t = (p \| q) \cdot r + s$$

where possibly r is missing or $s = \delta$. Suppose that $p \xrightarrow{a} p'$ or $p \xrightarrow{a} \surd$. Then $t \xrightarrow{a} p' \| q$ or $t \xrightarrow{a} q$, in any case q should be able to do a d-action. Since in this t would be able to perform a d-action followed an a-action, which is impossible, we conclude that the a-action is done by s. However, in this case p is able to do a p-action or a m-action. In both cases $t \xrightarrow{p} p' \| q \cdot r$ or $t \xrightarrow{p} q \cdot r$. Now, $p' \| q$ or q should be able to do an a-action and the same argument as before applies. □

Now, Lemma 5.5.7.23 follows since the process of 5.5.7.22 is not definable in BPA_δ (see [Cau90b]).

□

As an immediate consequence we have the following theorem:

Theorem 5.5.7.29. *There is a process in $\lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec})$ that is not definable in $\mathrm{PA}_{\mathrm{NIL}}$.*

5.5.8 $(\mathrm{BPA}_{\mathrm{NIL}} + \Lambda)\mathrm{lin} \subset \mathrm{ACP}_{\mathrm{NIL}}$

The main result of this section is the fact that all processes of $(\mathrm{BPA}_{\mathrm{NIL}} + \Lambda)\mathrm{lin}$ are definable in $\mathrm{ACP}_{\mathrm{NIL}}$ with renamings.

Definition 5.5.8.30. Let A be the set of atomic actions parametrizing the signature of ACP. Let p be a process definable in such a theory. The alphabet of p is the set of atomic actions that P can perform (FAP used). □

Definition 5.5.8.31. Let S, *Act, Eff* be the parameters for a generalized state operator and let $s \in S$. Let A_1 be the set of atomic actions. and γ_1 a communication function over A_1. Then we define an operator κ_s over ACP_{NIL} with a larger set of atomic actions A (i.e. $A_1 \subset A$) and using a communication function γ that extends γ_1 by

$$\kappa_s(X) = \rho_f \circ \partial_H(X \parallel p_s)$$

Here

- $A = \{< a, b >_s | a \in A_1, s \in S, b \in Act(a, s)\} \cup \{a' | a \in A_1\} \cup A_1$
- $\gamma(a, < a, b >_s) = b'$
- $f(a') = a$
- $H = \{< a, b >_s | a \in A_1, s \in S, b \in Act(a, s)\} \cup A_1$
- $p_s = \sum_{a \in A} \sum_{b \in Act(a,s)} < a, b >_s \cdot P_{Eff(a,b,s)}$

□

Note that given a process p definable in ACP_{NIL} with a set A of atomic actions and a state operator, then the alphabet of $\kappa_s(p)$ will still be a subset of A.

Lemma 5.5.8.32. *Let p be a definable process over* ACP_{NIL}, *then*

$$\kappa_s(p) = \Lambda_s(p)$$

Proof.

(i) For closed terms.

$p = \text{NIL}$

$$\begin{aligned} \kappa_s(\text{NIL}) &= \rho_f \circ \partial_H(\text{NIL} \parallel p_s) \\ &= \rho_f \circ \partial_H(p_s) \\ &= \text{NIL} \\ &= \Lambda_s(\text{NIL}) \end{aligned}$$

$p = a \cdot q$

$$\begin{aligned}
\kappa_s(a \cdot q) &= \rho_f \circ \partial_H(a \cdot q \parallel p_s) \\
&= \rho_f \circ \partial_H(a \cdot q \mid p_s) \text{ since } a \in A_1 \\
&= \sum_{b \in Act(a,s)} b \cdot \rho_f \circ \partial_H(q \parallel p_{Eff(a,b,s)}) \\
&= \sum_{b \in Act(a,s)} b \cdot \Lambda_{Eff(a,b,s)}(q) \\
&= \Lambda_s(a \cdot q)
\end{aligned}$$

$p = q + r$ straightforward.

(ii) For definable processes. Let q be a definable process. q has a head normal form $\sum a_i \cdot q_i$. We prove by induction over n that

$$\pi_n(\kappa_s(q)) = \pi_n(\Lambda_s(q))$$

Now we use lemma 2.1.7.2.

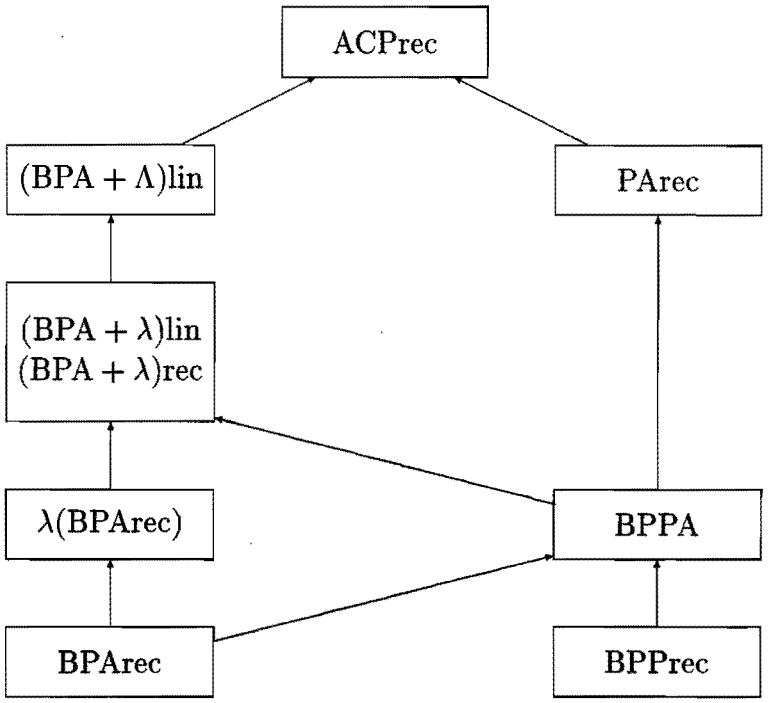
$$\begin{aligned}
\pi_1(\kappa_s(q)) &= \sum_{b \in Act(a_i,s)} b \\
&= \pi_1(\Lambda_s(q))
\end{aligned}$$

$$\begin{aligned}
\pi_{n+1}(\kappa_s(q)) &= \sum_{b \in Act(a_i,s)} b \cdot \pi_n(\kappa_s(q_i)) \\
&= \sum_{b \in Act(a_i,s)} b \cdot \pi_n(\Lambda_s(q_i)) \\
&= \pi_{n+1}(\sum_{b \in Act(a_i,s)} b \cdot \Lambda_s(q_i)) \\
&= \pi_{n+1}(\Lambda_s(q))
\end{aligned}$$

□

5.6 Summary

The next picture illustrates all the (proper) inclusions among the different systems.


ACPrec
(BPA + Λ)lin
PArec
(BPA + λ)lin
(BPA + λ)rec
λ(BPArec)
BPPA
BPArec
BPPrec

Chapter 6

Some results on decidability of bisimulation

6.1 Introduction

In this chapter we use some techniques introduced in the previous one, in order to solve some of the decidability problems of bisimulation. Since the publication of the work [BBK87b], which showed that for an important subset of the context free process bisimulation equivalence is decidable, many extensions have appeared. When a new class of processes is presented it is now a natural question whether in this class bisimulation is still decidable or not. The results in this chapter are far from complete: we show that bisimulation is decidable in the union of (normed) BPA and BPP, and undecidable in the class $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$.

6.2 Decidability results

6.2.1 Introduction

There exist algorithms for deciding bisimulation equivalence in BPP (Basic Parallel Processes) and in BPA (Basic Process Algebra) (see [BBK93, Cau90a, Chr93, HS91]). An interesting question is whether one can decide if a BPP process is bisimilar to a BPA process.

In both cases an important subset for which the problem of deciding bisimulation is easier, is the set of normed processes. A process graph is normed if from any node, there exists a finite sequence of steps to a termination state. In this chapter the previously mentioned problem for normed processes is solved. The strategy to solve the problem is the following: given a normed BPP process we can decide if it is also a BPA process, in which case we can construct a BPA specification for it and therefore use the algorithm to decide bisimulation in BPA.

Also, given a normed BPP process, we can decide whether it is a λ(BPA) process.
We list now some results about λ(BPA) graphs from [Cau90b, MS85].

Definition 6.2.1.1. The *norm* of a process p is the length of the shortest sequence σ such that $p \xrightarrow{\sigma} \surd$ (if there is such a sequence). Analogously, given a process graph the norm of a vertex of the graph is the length of the shortest path from it to a termination node. □

Definition 6.2.1.2. Let P be a normed λ(BPA) graph. For any vertex v with norm $|v|$ we define $P(v)$ as the connected component containing v in the subgraph of P whose vertices are the vertices of P with norm greater or equal than $|v|$ and the edges are those of P whose source and target have norm greater that or equal to $|v|$.
The vertices of $P(v)$ with norm equal to $|v|$ are called *frontier points* □

Definition 6.2.1.3. An *end-isomorphism* is a label preserving graph isomorphism that maps frontier points onto frontier points □

Theorem 6.2.1.4. *Let P be a normed λ(BPA) process. The set*

$$\{P(v) | v \text{ is a vertex of } P\}$$

has only finitely many isomorphism classes under end-isomorphism.

Corollary 6.2.1.5. *Given a normed $\lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec})$ process, for any n, the number of frontier points of norm n is bounded.*

6.2.2 Basic Parallel Processes

We reformulate here the definition of BPP presented in the previous chapter. The BPP specifications use alternative composition, parallel composition and action prefix. The general sequential composition is not allowed. In [Chr93] a normal form was found for specifications in BPP. Throughout this section we will assume BPP specification E with variables $\mathrm{Var}(E) = \{X_1 \ldots X_n\}$, such that each variable is defined as

$$X_i = \sum_{j=1}^{m_i} a_{ij} \cdot \alpha_{ij}$$

where α_{ij} is a parallel composition of variables which will be identified with a multiset (eventually empty), when it is convenient. The graph associated to a BPP specification will have multisets of variables as labels on the nodes. We also assume that every variable appears in at least one multiset reachable from the root (otherwise it can be removed from the specification).

Definition 6.2.2.6. The following relation on $\mathrm{Var}(E)$ is defined:

$$X_i \rightarrow X_k \text{ iff there exists an } \alpha_{ij} \text{ with } X_k \in \alpha_{ij}$$

Furthermore, we define the set

$$R(X) = \{Y \in \mathrm{Var}(E) : X \rightarrow^+ Y\}$$

□

Definition 6.2.2.7.

- A variable X is *bounded* if there exists a number n such that in any state of the graph of E the multiplicity of X is at most n.

- a variable is *unbounded* if it is not bounded.

□

Fact 6.2.2.8. An infinite state BPP process has at least one unbounded variable. □

Example 6.2.2.9. A normed counter can be defined in BPP as follows:

$$C = m + p \cdot C||C$$

This process has an infinite number of states ($C^{||n}$ for any n),

□

The following theorem will be useful in finding a necessary condition for a BPP process to belong to λ(BPA).

Proposition 6.2.2.10. *Let Y be an unbounded variable. If $X \in R(Y)$, then X is also unbounded*

Proof. If Y is unbounded, then for any n there is a state that contains $Y^{||n}$ (in particular, since every variable is normed, the state $Y^{||n}$ belongs to the graph). Since $X \in R(Y)$, then there is a sequence σ such that $Y^{||n} \xrightarrow{\sigma} X^{||n}$, then, for any n, $X^{||n}$ is also a state of the graph. □

Remark 6.2.2.11. We use $Y^{||k}$ to indicate $Y \parallel Y \parallel \ldots \parallel Y$ (k- times) and Y^k for $Y \cdot Y \cdots Y$. □

Lemma 6.2.2.12. *Given a specification E, a variable Y is unbounded if and only if at least one of the following properties holds:*

1. *there is another unbounded variable* X *such that* $Y \in R(X)$;
2. $Y \xrightarrow{\sigma}{}^{*} Y^n, n \geq 2$;
3. *there exist a variable* X *and a path* $X \xrightarrow{\sigma}{}^{*} X \parallel Y$.

Proof. It is clear that if one of the conditions holds then Y is unbounded. Now, suppose that Y is unbounded and that the first two conditions are not true. Since Y is unbounded, there is an infinite path

$$\alpha_1 \xrightarrow{\sigma_1}{}^{*} \alpha_2 \parallel Y^{\parallel k_1} \xrightarrow{\sigma_2}{}^{*} \cdots \xrightarrow{\sigma_n}{}^{*} \alpha_n \parallel Y^{\parallel k_{n-1}} \xrightarrow{\sigma_{n+1}}{}^{*} \cdots$$

where for all $i, k_i < k_{i+1}$ and $Y \notin \alpha_i$.

Moreover, since condition 1 is not true we can assume that for all i, all variables in α_i are bounded. Otherwise we can take another infinite path where we take out all the unbounded variables from the α_i. Note that we can do this because every variable is normed.

For cardinality reasons we know then that there exist α, σ such that $\alpha \parallel Y^{\parallel k} \xrightarrow{\sigma}{}^{*} \alpha \parallel Y^{\parallel k+r}, r > 0$. We can assume that $r = 1$ since Y is normed. From the fact that condition 2 is not true it follows that there exists σ such that $\alpha \xrightarrow{\sigma}{}^{*} \alpha \parallel Y$, because Y cannot reduce to a variable in α since they are all normed. Again we can assume that all variables of α are used.

Assume that for any $X \in \alpha, X \xrightarrow{\sigma_i}{}^{*} \beta_i$, such that $\bigcup \beta_i = \alpha \cup \{Y\}$. This can be done because one can split the computation σ according to the variable in α from which each step originates. If some of the β_i is empty, then we can remove the variable from α (again because it is normed). If one of the $\beta_i = \{Y\}$, say $X \xrightarrow{\sigma_i}{}^{*} Y$ then there is another variable X' such that $X' \xrightarrow{\sigma_{i'}}{}^{*} X$. Then we can get rid of the X. In this way we can reduce the α until we have a variable X such that $X \xrightarrow{\sigma_i}{}^{*} X' \parallel Y$. Since all other variables reduce through their correspondent σ_j to exactly one variable in α, then it follows that $X \in R(X')$, since X must be also in α, and in consequence condition 3 holds. □

6.2.3 BPP ∩ λ(BPA)

Definition 6.2.3.13. The canonical graph of a specification is the graph in which two nodes are bisimilar if and only if they are the same (see [BW90]). □

Theorem 6.2.3.14. *Let* E *be a canonical BPP specification and* X, Y *two variables from* E *such that in the graph defined by* E *there is an infinite number of* p*'s and of* q*'s such that* $X^{\parallel p} \parallel Y^{\parallel q}$ *is the label of a node. Then there is no specification in* $\lambda(BPA)$ *of the same process.*

Proof. Assume that the norms of X and Y are given by

$$|X| = m$$

$$|Y| = n$$

then, it is immediate that for any p, q

$$|X^{\|p} \parallel Y^{\|q}| = pm + qn$$

Since all variables are normed then $X^{\|p} \parallel Y^{\|q}$ is the label of a node for *any* p, q. Moreover there is always a path from $X^{\|(p+p1)} \parallel Y^{\|(q+q1)}$ to $X^{\|p} \parallel Y^{\|q}$ for any $p1, q1$. This implies that if we take only the nodes which norm is greater than a constant, then two nodes of this form will belong to the same connected component. Now we will show that the number of frontier points is unbounded.

For any number k the number of nodes with norm equal to kmn is at least $k + 1$, and $1 < i < n$

$$\begin{aligned} |X^{\|kn}| &= |X^{\|(k-1)n} \parallel Y^{\|m}| \\ &= |X^{\|(k-i)n} \parallel Y^{\|im}| = \\ &= |Y^{\|km}| = \\ &= kmn \end{aligned}$$

□

Finally, the following theorem illustrates that this condition is not only necessary, but sufficient as well.

Theorem 6.2.3.15. *A BPP process that does not have a pair of unbounded variables as in theorem 6.2.3.14 is also a λ(BPA) process.*

Proof. We prove it for the case where there is only one unbounded variable. The general case is similar. Let

$$E = \{X_i = \sum a_{ij} \cdot \alpha_{ij} : 1 \leq i \leq n\} \cup \{Y = \sum b_j \cdot \beta_j\}$$

Note that β_j can only contain the variable Y, since this is the only unbounded variable. Define $\#_X(\alpha)$ as the multiplicity of X in the multiset α. Assume Y is the unbound variable and for all i let m_i be the bound corresponding to X_i. Now define

$$\begin{aligned} B = &\sum_i \sum_j (i, a_{ij}) \cdot B'^{\#_Y(\alpha_{ij})} \cdot B + \sum_{\#_Y(\beta_j) \geq 1} b_j \cdot B'^{\#_Y(\beta_j)-1} \cdot B + \\ &\sum_{\#_Y(\beta_j)=0} (b'_j \cdot C + b''_j) \end{aligned}$$

$$B' = \sum_i \sum_j (i, a_{ij}) \cdot B'^{\#_Y(\alpha_{ij})+1} + \sum_{\#_Y(\beta_j)\geq 1} b_j \cdot B'^{\#_Y(\beta_j)} + \sum_{\#_Y(\beta_j)=0} + b_j$$

$$C = \sum \sum_{\#_Y(\alpha_{ij})\geq 1} (i, a_{ij}) \cdot B'^{\#_Y(\alpha_{ij})-1} \cdot B + \sum_{\alpha_{ij}\neq\emptyset} \sum_{\#_Y(\alpha_{ij})=0} (i, a_{ij})' \cdot C + \sum \sum_{\alpha_{ij}=\emptyset} (i, a_{ij})'' \cdot C + \sum \sum_{\alpha_{ij}=\emptyset} (i, a_{ij})'''$$

The state space is defined as

$$S = (k_1, \ldots, k_n) | k_i \leq m_i\}$$

The intuitive idea behind this construction is that the state $(k_1, \ldots, k_n)$ will represent the multiplicity of the bounded variables and the recursion variables defined above will represent the multiplicity of the unbounded variable.

The functions *act* and *eff* are defined as follows:

- $(i, a_{ij}) \leftarrow (k_1, \ldots, k_n) = \delta$, if $k_i = 0$.
- $(i, a_{ij}) \leftarrow (k_1, \ldots, k_n) = a_{ij}$, if $k_i > 0$.
- $(i, a_{ij})' \leftarrow (k_1, \ldots, k_n) = \delta$, if $k_i = 0$
- $(i, a_{ij})'' \leftarrow (k_1, \ldots, k_n) = \delta$, if $k_i = 1$ and for all $j \neq i, k_j = 0$. It is allowed only when it is not the last action before termination.
- $(i, a_{ij})''' \leftarrow (k_1, \ldots, k_n) = \delta$, if $k_i \neq 1$ or there exist a $j \neq i$ such that $k_j \geq 0$. It is allowed only when it is the last action before termination.
- $(i, a_{ij})' \leftarrow (k_1, \ldots, k_n) = a_{ij}$, if $k_i > 0$
- $(i, a_{ij})'' \leftarrow (k_1, \ldots, k_n) = a_{ij}$, if $k_i > 1$ or when both $k_i = 1$ and there exist a $j \neq i$ such that $k_j \geq 0$.

- $(i, a_{ij})'''\leftarrow(k_1, \ldots, k_n) = a_{ij}$, if $k_i = 1$ and for all $j \neq i, k_j = 0$.

- $b'_j\leftarrow(k_1, \ldots, k_n) = \delta$, if for all $j \neq i, k_j = 0$. It is allowed only in a state with some bounded variable.

- $b''_j\leftarrow(k_1, \ldots, k_n) = \delta$, if there exist a $j \neq i$ such that $k_j \geq 0$. It is allowed only in states where there are not unbounded variables.

- $b'_j\leftarrow(k_1, \ldots, k_n) = b_j$, if there exist a i such that $k_i \geq 0$.

- $b''_j\leftarrow(k_1, \ldots, k_n) = b_j$, if for all $i, k_i = 0$.

- $(i, a_{ij})\rightarrow(k_1, \ldots, k_n) = (k_1+(\#_{X_1}(\alpha_{ij})), \ldots, (k_i+(\#_{X_i}(\alpha_{ij})-1), \ldots, (k_n+(\#_{X_n}(\alpha_{ij})))$
- $(i, a_{ij})'\rightarrow(k_1, \ldots, k_n) = (k_1+(\#_{X_1}(\alpha_{ij})), \ldots, (k_i+(\#_{X_i}(\alpha_{ij})-1), \ldots, (k_n+(\#_{X_n}(\alpha_{ij})))$
- $(i, a_{ij})''\rightarrow(k_1, \ldots, k_n) = (k_1+(\#_{X_1}(\alpha_{ij})), \ldots, (k_i+(\#_{X_i}(\alpha_{ij})-1), \ldots, (k_n+(\#_{X_n}(\alpha_{ij})))$

Accordingly, using RSP for infinite sets of equations, it is easy to prove that

$$C\leftarrow(k_1, \ldots, k_n) = X_1^{\|k_1} \| \cdots \| X_n^{\|k_n}$$

and

$$B^{\prime r} \cdot B\leftarrow(k_1, \ldots, k_n) = X_1^{\|k_1} \| \cdots \| X_n^{k_n} \| Y^{\|r+1}$$

For example the first equality when it holds that $\sum_i k_i > 1$, i.e. when there are at least two recursion variables in the multiset.

$$\begin{aligned}
C\leftarrow(k_1, \ldots, k_n) \;=\; & \Big(\sum_i \sum_{\#_Y(\alpha_{ij})\geq 1} (i, a_{ij}) \cdot B'^{\#_Y(\alpha_{ij})-1} \cdot B + \\
& \sum_i \sum_{\#_Y(\alpha_{ij})=0} (i, a_{ij})' \cdot C + \sum_i \sum_{\#_Y(\alpha_{ij})=0} (i, a_{ij})''\Big)\leftarrow(k_1, \ldots, k_n) \\
= \; & \sum_{k_i>0} \sum_{\#_Y(\alpha_{ij})\geq 1} a_{ij} \cdot (B'^{\#_Y(\alpha_{ij})-1} \cdot B) \\
& \leftarrow(k_1 + (\#_{X_1}(\alpha_{ij})), \ldots, (k_i + (\#_{X_i}(\alpha_{ij}) - 1), \ldots, (k_n + (\#_{X_n}(\alpha_{ij}))) + \\
& \sum_{k_i>0} \sum_{\#_Y(\alpha_{ij})=0} i, a_{ij} \cdot C \\
& \leftarrow(k_1 + (\#_{X_1}(\alpha_{ij})), \ldots, (k_i + (\#_{X_i}(\alpha_{ij}) - 1), \ldots, (k_n + (\#_{X_n}(\alpha_{ij}))) + \\
& \sum_{k_i>0} \sum_{\alpha_{ij})=\emptyset} a_{ij} \cdot C\leftarrow(k_1, \ldots, k_i - 1, \ldots, k_n)
\end{aligned}$$

$$
\begin{aligned}
X_1^{\|k_1} \| \cdots \| X_n^{\|k_n} &= \sum_{k_i \neq 0} a_{ij} \cdot X_1^{\|k_1} \| \cdots \| X_i^{\|k_i - 1} \| \cdots \| X_n^{\|k_n} || \alpha_{ij} \\
&= \sum_{k_i \neq 0} a_{ij} \cdot X_1^{\|k_1 + (\#_{X_1}(\alpha_{ij}))} \| \cdots || \\
&\quad X_i^{\|k_i + (\#_{X_i}(\alpha_{ij})) - 1} || \cdots \| X_n^{\|k_n + (\#_{X_n}(\alpha_{ij}))} || Y^{(\#_Y(\alpha_{ij}))} \\
&= \sum_{k_i \neq 0, \#_Y(\alpha_{ij}) = 0} a_{ij} \cdot (X_1^{\|k_1 + (\#_{X_1}(\alpha_{ij}))} \| \cdots || \\
&\quad X_i^{\|k_i + (\#_{X_i}(\alpha_{ij})) - 1} || \cdots \| X_n^{\|k_n + (\#_{X_n}(\alpha_{ij}))} + \\
&\quad \sum_{k_i \neq 0, \#_Y(\alpha_{ij}) \geq 1} a_{ij} \cdot (X_1^{\|k_1 + (\#_{X_1}(\alpha_{ij}))} \| \cdots || \\
&\quad X_i^{\|k_i + (\#_{X_i}(\alpha_{ij})) - 1} || \cdots \| X_n^{\|k_n + (\#_{X_n}(\alpha_{ij}))} || Y^{\#_Y(\alpha_{ij})}
\end{aligned}
$$

The result then follows dividing the first summatory in the last term of the equation in the two possible cases (α_{ij} empty or not).

□

6.2.4 BPA

Some results concerning BPA processes are recalled in this section.

Definition 6.2.4.16. Given a graph of a process, a vertex s is a multiple start for a vertex t, if there exist two paths from s to t such that their only common vertices are s and t. □

The following proposition is taken from [Cau90b].

Proposition 6.2.4.17. *The set of multiple starts for any state of a BPA graph is finite.*

Theorem 6.2.4.18. *Let E specify a process in BPP ∩ BPA with an unbounded variable Y. Then Y cannot appear an infinite number of times in a state with another variable.*

Proof. Suppose that Y appears an infinite number of times beside X. Given that the process is normed we know then that $X \| Y^{\|p}$ is a state for any natural number p. There exists a path

$$X \xrightarrow{a_1} \alpha_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} Y^{\|r}$$

where $r \geq 0$ and all α_i are different than $Y^{\|s}$ for any s. (This is easy, take the first state of the form $Y^{\|r}$ in a path from X to the termination state or take $r = 0$ if no Y is encountered) Now the set

$$\{X \| Y^{\|j} : j > r\}$$

is a set of multiple starts for Y^r. We can take the path from $X \parallel Y^{\parallel j}$ to X and then from there to $Y^{\parallel r}$ or take the path from $X \parallel Y^{\parallel j}$ to $Y^{\parallel j+r}$ and then reduce it in j steps to $Y^{\parallel r}$. It is clear that the paths satisfy the properties of a multiple start. (Note that because Y cannot do a step to another variable and it is normed, then it must be able to terminate in just one step.) □

Corollary 6.2.4.19. *Given an infinite process in BPP ∩ BPA with an unbound variable Y, there is no variable X (different than Y) such that $X \xrightarrow{\sigma}{}^* X \parallel Y$.*

Corollary 6.2.4.20. *An unbounded variable cannot appear with a different variable in a state at all.*

Proof. From corollary 6.2.4.19, the only possibility for Y to be unbound is that it can do a step to $Y^{\parallel k}$ with $k \geq 2$. Yet, if there is a state of the form $X \parallel Y$ then there is also a $X \parallel Y^{\parallel j}$ for any j. □

Finally, all this gives us an algorithm to decide whether a BPP process is bisimilar to a BPA process.

Theorem 6.2.4.21. *Given a BPP specification we can decide if the process defined belongs to BPA, in which case we can construct a BPA specification for it.*

Proof. Using the algorithm in [Chr93] for normed processes based on unique decomposition, we can find a specification of the canonical graph of that process. One can easily calculate whether a variable is unbounded using lemma 6.2.2.12. If more than one unbounded variable appears together, then the process does not belong to BPA. Otherwise we know that it already belongs to λ(BPA). If it is the case that for every unbounded variable Y no other variable X can perform a sequence of actions to $X \parallel Y$, then the process is in BPA. To obtain a specification in BPA for this process, first find a linear specification (as in [BW90]) for the regular process obtained by removing all occurrences of the unbounded variables. For every unbounded variable Y, if a variable in the original BPP specification has a summand of the form $a \cdot Y^{\parallel r}$, then add a summand $a \cdot Y^{\parallel r}$ to the corresponding variable in the new specification. Finally, add an equation for every unbouded variable where all $\parallel$ are replaced by $\cdot$. □

Example 6.2.4.22. The BPP specification

$$X = a \cdot (Z \parallel W) + b \cdot (Y \parallel Y \parallel Y) + b \cdot Y'$$

$$Y = c \cdot (Y \parallel Y) + d$$

$$Y' = c' \cdot (Y' \parallel Y' \parallel Y') + d'$$

$$Z = e \cdot Z + f$$

$$W = g$$

is transformed into

$$A = a \cdot B + b \cdot Y \cdot Y + b \cdot Y'$$
$$B = e \cdot B + f \cdot D + g \cdot C$$
$$C = e \cdot C + f$$
$$D = g$$
$$Y = c \cdot Y \cdot Y + d$$
$$Y' = c' \cdot (Y' \cdot Y' \cdot Y') + d'$$

□

6.3 Undecidable classes

6.3.1 Bisimulation is undecidable in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$

The problem of whether bisimulation equivalence is decidable for certain classes of processes has received many partial answers. It is well known that bisimulation is decidable for BPA (see [BBK87a, Cau90b, SCS92]). On the other hand, bisimulation is undecidable for ACP (see [BK84a]) and therefore for all systems containing it. For PA it is still an open problem (some work in that direction is [Chr92]). One of the most interesting open problems, that is also related with the decidability of deterministic languages, is whether bisimulation is decidable for processes in $\lambda(\mathrm{BPA}_{\mathrm{NIL}}\mathrm{rec})$. Theorem 6.3.1.1 shows that if we move to $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$ then the problem of decidability of bisimulation has a negative answer. The construction is based on one in [BK84a].

Theorem 6.3.1.1. *Given two processes in* $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)\mathrm{lin}$ *there is no algorithm which decides if they are bisimilar.*

Proof. Let K be a r.e. set that is not recursive. In [HU79] it is proven that K can be recognized by a three counter machine. Such a machine has three counters a, b, c (ranged over by a metavariable α) and a sequence of instructions. The instructions have one of the following forms,

i $\alpha := \alpha + 1$; goto j

ii $\alpha := \alpha \dot{-} 1$; goto j

iii if $\alpha = 0$ then goto j else goto j'

iv stop

The meaning of each instruction is obvious ($\dot{-}$ means subtract except if it is zero) The sequence of instructions is numbered and it is assumed that the labels of the goto instructions are among these numbers.

Now, let P be a program that recognizes K, i.e. $P(n,0,0) \longrightarrow$stop iff $n \in K$, where $P(x,y,z)$ means that x, y, and z are the initial values of the counters. We will give a recursive specification with main variable X_1 in $(\mathrm{BPA}_{\mathrm{NIL}} + \lambda)$lin such that $X_1 \leftarrow a^n \leftarrow s$ can do an infinite number of actions if and only if the machine P does not finish.

The state space will be

$$S = \{a, b, c, s\}$$

First we translate each instruction I_i into a linear equation in the following way (following the pattern above)

i $X_i = b \cdot X_j \leftarrow \alpha$

ii $X_i = (0_\alpha + p_\alpha) \cdot X_j$

iii $X_i = 0_\alpha \cdot X_j + p_\alpha \cdot X_{j'} \leftarrow \alpha$

iv $X_i = \delta$

The state operator will model the state of the counters of the machine. An expression of the form $X_i \leftarrow \sigma \leftarrow s$ where $\sigma \in \{a, b, c, I\}^*$ means that the machine is executing instruction i and that counter α has value $\#_\alpha(\sigma)$, that is the number of occurrences of α in σ.

The action and effect functions are trivial except in the following cases

- $0_\alpha \leftarrow \alpha = \mathrm{NIL}$ (the counter has a positive value)
- $0_\alpha \leftarrow s = b$ (the counter is empty)
- $p_\alpha \leftarrow \alpha = b$ (the counter has a positive value)
- $p_\alpha \rightarrow \alpha = I$ (decrease the counter)
- $p_\alpha \leftarrow s = \mathrm{NIL}$ (don't decrease an empty counter)

Given B defined by $B = b \cdot B$, then B is bisimilar to $X_1 \leftarrow \sigma \leftarrow s$ if and only if $P(n,0,0)$ diverges. □

Chapter 7

Concluding Remarks

In this thesis it has been shown that the state operator is a powerful tool for process algebra, both from a theoretical and applied point of view. It has been used in an essential way to introduce different classes of processes definable by recursive specifications. Since the operator is defined axiomatically this classification is not restricted to a specific model. For applied research, this classification is useful as well since the different classes have different properties, such as decidability of bisimulation equivalence, uniformly finite branching, etc, which can be used to simplify the analysis of a process that belong to a some of these classes.

In previous works on the state operator (e.g. [BB88, BB91a]), it has been considered essentially as an operator that modifies processes, but little or no attention has been paid to how processes modify the states in their turn. We extended slightly this operator in order to cope with this idea, giving more symmetric roles to states and processes.

A new notion of atomicity for complex processes was introduced, where communication has been replaced by (complex) multiactions. The state operator is of crucial importance here to implement patterns of communication. Furthermore, it has been used to introduce input-output semantics in a concurrent environment, in order to define a new version of implementation of data types.

The introduction of new classes of processes opens new decidability problems. One of the problems that has been open for a long time is the decidability of bisimulation for the class PArec. The class BPPA introduces in this thesis is contained in PArec and contains BPPrec and BPArec. This properties make it an interesting candidate for the decidability problem. Its main difference with PArec is that all its processes have uniformly finite branching.

Bibliography

[AMR88] E. Astesiano, G.F. Mascari, and G. Reggio. Data in a concurrent environment. volume 335 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[BB88] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78(3):205–245, 1988.

[BB90] J.C.M. Baeten and J.A. Bergstra. Process algebra with a zero object. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 83–98. Springer-Verlag, 1990.

[BB91a] J.C.M. Baeten and J.A. Bergstra. Recursive process definitions with the state operator. *Theoretical Computer Science*, 82:285–302, 1991.

[BB91b] J.C.M. Baeten and J.A. Bergstra. A survey of axiom systems for process algebras. Report P9111, Programming Research Group, University of Amsterdam, 1991.

[BB93] J.C.M. Baeten and J.A. Bergstra. Non interleaving process algebras. In E. Best, editor, *Proceedings of CONCUR '93, 4th. International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.

[BBK87a] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *Proceedings PARLE conference,* Eindhoven, *Vol. II (Parallel Languages)*, volume 259 of *Lecture Notes in Computer Science*, pages 94–113. Springer-Verlag, 1987.

[BBK87b] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Ready-trace semantics for concrete process algebra with the priority operator. *Computer Journal*, 30(6):498–506, 1987.

[BBK93] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *Journal of the ACM*, 40(3):653–682, 1993.

[BC88] G. Boudol and I. Castellani. A non-interleaving semantics for CCS based on proved transitions. *Fundamenta Informaticae*, 11(4):433–452, 1988.

[BG87] J.C.M. Baeten and R.J. van Glabbeek. Merge and termination in process algebra. In K.V. Nori, editor, *Proceedings 7th Conference on Foundations of Software Technology and Theoretical Computer Science,* Pune, India, volume 287 of *Lecture Notes in Computer Science*, pages 153–172. Springer-Verlag, 1987.

[BK84a] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Proceedings 11th ICALP,* Antwerp, volume 172 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 1984.

[BK84b] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[BK87] J.A. Bergstra and J.W. Klop. A convergence theorem in process algebra. Report CS-R8733, CWI, Amsterdam, 1987.

[Bla92] J. Blanco. Definability with the state operator in process algebra. Report P9221, University of Amsterdam, Amsterdam, 1992.

[Bou89] G. Boudol. Atomic actions (note). *Bulletin of the European Association for Theoretical Computer Science*, 38:136–144, 1989.

[Bri88] E. Brinksma. *On the design of Extended LOTOS – a specification language for open distributed systems.* PhD thesis, Department of Computer Science, University of Twente, 1988.

[BV89] J.C.M. Baeten and F.W. Vaandrager. An algebra for process creation. Report CS-R8907, CWI, Amsterdam, 1989. Under revision for *Acta Informatica.*

[BV92] J.C.M. Baeten and F.W. Vaandrager. An algebra for process creation. *Acta Informatica*, 29:303–334, 1992.

[BVng] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, Dov M. Gabay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4. Oxford University Press, Forthcoming.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[Cau90a] D. Caucal. Graphes canoniques de graphes algébriques. *Theoretical Informatics and Applications*, 24(4):339–352, 1990.

[Cau90b] D. Caucal. On the transition graphs of automata and grammars. Report 1318, INRIA, 1990.

[Chr92] S. Christensen. Distributed bisimilarity is decidable for a class of infinite state-space processes. In W.R. Cleaveland, editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 148–161, 1992.

[Chr93] S. Christensen. *Decidability and Decomposition in Process Algebra.* PhD thesis, Department of Computer Science, University of Edinburgh, Edinburgh, 1993.

[Gla90] R.J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 1990.

[GMM90] R. Gorrieri, S. Marchetti, and U. Montanari. A^2CCS: Atomic actions for CCS. *Theoretical Computer Science*, 1(72):203–223, 1990.

[GP90] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Report CS-R9076, CWI, Amsterdam, 1990.

[HS91] H. Hüttel and C. Stirling. Actions speak louder than words: Proving bisimilarity for context-free processes. In *Proceedings* 6^{th} *Annual Symposium on Logic in Computer Science,* Amsterdam, The Netherlands, pages 376–386. IEEE Computer Society Press, 1991.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[KP87a] S. Kaplan and A. Pnuelli. Specification and implemetation of concurrently accessed data structures: an abstract data type approach. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings STACS 87,* volume 247 of *Lecture Notes in Computer Science*, pages 220–244. Springer-Verlag, 1987.

[KP87b] S. Katz and D. Peled. Interleaving set temporal logic. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing,* Vancouver, Canada, pages 178–190, 1987. See also Technical Report #505, Technion – Israel Institute of Technology, Department of Computer Science, Haifa, Israel, March 1988.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mol89] F. Moller. *Axioms for concurrency.* PhD thesis, Report CST-59-89, Department of Computer Science, University of Edinburgh, 1989.

[MS85] D.E. Muller and P.E. Schupp. The theory of ends, pushdown automata, and second order logic. *Theoretical Computer Science*, 37:51–75, 1985.

[MV89] S. Mauw and G.J. Veltink. An introduction to PSF_d. In J. Díaz and F. Orejas, editors, *Proceedings TAPSOFT 89, Vol. 2*, volume 352 of *Lecture Notes in Computer Science*, pages 272–285. Springer-Verlag, 1989.

[OL87] A. Obaid and L. Logrippo. An atomic calculus of communicating systems. In *7th Conf. on Protocol Specification, Testing and Verification*, volume 104. IFIP WG 6.1, 1987.

[Old87] E.-R. Olderog. Operational Petri net semantics for CCSP. In G. Rozenberg, editor, *Advances in Petri Nets 1987*, volume 266 of *Lecture Notes in Computer Science*, pages 196–223. Springer-Verlag, 1987.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[Pon92] A. Ponse. *Process Algebra with Data.* PhD thesis, University of Amsterdam, 1992.

[Rei87] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* International Series od Monograph on Computer Science, No 2. Oxford, 1987.

[SCS92] H. Hüttel S. Christensen and C. Stirling. Bisimulation is decidable for all context-free processes. In W.R. Cleaveland, editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 138–147, 1992.

[Vaa90] F.W. Vaandrager. Process algebra semantics of POOL. In J.C.M. Baeten, editor, *Applications of process algebra*, Cambridge Tracts in Theoretical Computer Science 17, pages 173–236. Cambridge University Press, 1990.

[Vaa91] F.W. Vaandrager. On the relationship between process algebra and I/O automata. In *Proceedings LICS 91*, pages 387–398. Proceedings of the IEEE, 1991.

Samenvatting

Procesalgebra is de studie van concurrente processen op algebaïsche wijze. De belangrijkste algebraïsche techniek die in de procesalgebra wordt gebruikt is de axiomatische methode. Deze methode behelst het vormen van een verzameling axioma's die het gedrag van processen en hun compositiewetten beschrijft. Een proces kan voornamelijk beschouwd worden als het gedrag van een systeem, waarbij een systeem bijvoorbeeld een computer systeem, een elementair deeltje, een verkoopautomaat, of een satelliet die communiceert met de aarde kan zijn.

Procesalgebra verscheen als een antwoord op de vele problemen die opdoken bij de zoektocht naar formele semantieken van talen met primitieven voor concurrente processen. Een van deze problemen was het tekort schieten van invoer/uitvoer semantieken, die zeer succesvol waren gebleken bij het geven van semantiek aan sequentieële talen. De complexiteitsgroei van de problemen aangaande concurrente talen leidde tot het vaststellen van een zeker basisbegrip "proces" en de elementaire operaties op deze processen. Desalniettemin moesten, naarmate de theorieën groeiden en gebruikt werden in toepassingen, de vele eigenschappen die reeds bestudeerd waren op het gebied van sequentieële talen ook binnen dit kader beschouwd worden.

Een toepassing van procesalgebra die de aandacht verdient is het begrip "atomairiteit". We benaderen dit probleem vanuit de gedachte dat een atomaire actie een zeker effect zal hebben op zijn omgeving, en wellicht ook door deze omgeving beïnvloed kan worden. Atomairiteit behelst in deze de eigenschap van uitgevoerd worden zonder wisselwerking van andere componenten. Dit lijkt overeen te komen met het atomairiteitsbegrip dat gebruik wordt in het vakgebied van gedistribueerde databases.

Dit proefschrift behandelt enkele van deze eigenschappen, in een procesalgebraïsch kader.

In Hoofdstuk 2 worden de basisbegrippen van de procesalgebra en de toestandsoperator geïntroduceerd, die ten grondslag liggen aan dit werk. De toestandsoperator die hier gebruikt wordt is een generalisatie van die die behandeld wordt in [BB88]. Het belangrijkste verschil is dat we de toestandsoperator meer symmetrisch beschouwen, en dat we niet alleen geïnteresseerd zijn in het proces dat gewijzigd wordt door een toestand, maar ook in de verzameling toestanden die voortgebracht of gewijzigd wordt door een proces. Hierdoor zijn we in staat om te gaan met het invoer-uitvoer gedrag van een proces, hetgeen noodzakelijk is in de overige hoofdstukken van dit proefschrift. We introduceren enkele niet-equationele principes, alsmede enkele niet-standaard modellen om deze principes te verduidelijken.

Hoofdstuk 3 behandelt het principe van niet-elementaire atomaire acties in de procesalgebra. Dit concept is het onderwerp geweest van veel discussie, en de bron van veel verschillende modellen. Het concept atomairiteit is essentieel bij interleaving theorieën van concurrency, en veel modellen berusten op het feit dat dit een primitief concept is. We wijken enigszins af van interleaving theorieën teneinde een mechanisme te introduc-

eren om een proces te beschrijven dat op atomaire wijze uitgevoerd dient te worden. De gelijkheid op atomaire acties die we invoeren zal weergeven dat twee atomaire acties zich identiek zullen gedragen in iedere toestand. Het concept van atomaire actie dat in dit hoofdstuk beschouwd wordt combineert ideeën van [Bou89] en [BK84b], maar wijkt af van eerstegenoemde in het gebruik van branching time semantiek in tegenstelling tot de invoer-uitvoer semantiek van [Bou89], en is in plaats van synchronisatie als in [BK84b] gebaseerd op multi-acties. Verder is het principe van herstelbaarheid van een atomaire actie (d.w.z. als de actie niet succesvol termineert, dan behoort de toestand van het systeem hetzelfde blijven, alsof de actie nooit was uitgevoerd) geïmplementeerd op grond van het idee dat een onsuccesvolle terminatie een nul-object ([BB90]) binnen een atomaire actie is. Dit is een verbetering ten opzichte van [Bou89], aangezien er onderscheid gemaakt kan worden tussen deadlock en livelock binnen een atomaire actie.

Hoofdstuk 4 behandelt het combineren van data en processen. In de literatuur zijn veel verschillende aanpakken gebruikt om de theorieën van datatypes en processen te integreren. Zelfs wanneer beide theorieën beperkt werden tot algebraïsche theorieën, werden verschillende combinaties gebruikt. In het proefschrift van Ponse [Pon92] bijvoorbeeld, worden datatypes gebruikt als indices voor recursie-vergelijkingen, terwijl in [AMR88] de processen zelf als een datatype worden beschouwd. Hier laten we zien hoe bepaalde datatypes op een hele natuurlijk manier gezien kunnen worden als proces. Op die manier kunnen we interactie tussen processen en data vereenvoudigen tot interactie tussen processen onderling. Bovendien geven we een nieuwe oplossing voor het gebruik van datatypes, en de implementatie van een datatype door middel van een ander datatype, in een concurrente omgeving.

In Hoofdstuk 5 beperken we de toestandsoperator tot een eindige verzameling van toestanden, om te bestuderen of de toevoeging van een dergelijke toestandsoperator al dan niet de verzameling van met een guarded recursieve specificatie definieerbare processen kan vergroten.

Enkele resultaten betreffende de beslisbaarheid van bisimulatie op enkele klasses gedefinieer in Hoofdstuk 5 worden geïntroduceerd in Hoofdstuk 6.

Stellingen

behorende bij het proefschrift

The State Operator in Process Algebra

van

Javier Oscar Blanco

1. One of the advantages of process algebra that is often cited (for example [BW90]) is the ability to obtain results that are independent of the choice of a particular model. However, some properties that are interesting both from a theoretical and applied point of view (e.g left-cancellation property of atomic actions, agreement between the equality in the model and bisimulation) are not valid in many reasonable models. Given any class of models a new class can be constructed for which these properties are valid, viz. the class of submodels of definable processes of the models of the original class. These models only contain processes that are solutions of guarded systems of recursive equations.

2. Depending on the operations used, the set of definable processes of a given model can vary. This provides the means to classify a set of processes according to a number of notions of complexity without explicit reference to the model. In this thesis, some results are obtained concerning the inclusions and overlappings among the different classes. The procedure that was implicitly used in most of these results was to convert a recursive specification which satisfies certain restrictions into another that satisfies a different set of restrictions, with a different set of operations in some cases. Further could be done by making all these algorithms explicit and using them as part of a transformational approach to the design of concurrent systems.

3. The tight sequential composition (:) introduced in [BK84] and elaborated in this thesis, has the property of non-interruption. For example the process $(a : b)||x$ must perform b immediately after a is performed, even if x is able to perform some action. Furthermore, it satisfies a property of recoverability that can be expressed with the equality $a : \delta = \delta$, which means that if a process reaches a deadlock while trying to complete a tight sequential composition, it must recover the state of the system before this tight sequential composition is started. An interesting question is whether the following recursive definition must have a unique solution

$$X = a : X$$

   Intuitively, a solution to this equation is a process with a livelock, but the deadlocked process δ is a solution to this equation as well.

4. In many works on data types and processes in which the processes act on the data types, there was some redundancy in the sense that the data type is defined in an algebraic way, which implies that there is a fixed set of operations for the data type. Moreover, a subset of the set of atomic actions was chosen to represent these operations at the level of the processes. The state operator suggests the simpler natural way to define data types as a set of atomic actions and a set of states where the definition of the operations of the data type is given as the state function of the state operator. Also, there is no need for new concepts to deal with the case of partial data types, since any atomic action could be blocked (=undefined) in a given state.

5. Scientific activity, at least in computing science, has become a self-referent activity. However, it is hard to find serious reflections on its own methods and goals. I think the only hope to achieve a living science lies beyond the self-imposed boundaries of the scientific status quo, consisting in a movement towards an understanding of the fact that science is meaningful from a social and political point of view and as a creative and pleasurable activity.

6. Bertolt Brecht was confronted with an Aristotelian theatre whose methods had become sterile. After Stanislavski, whose work is nonetheless admirable, it seemed that there was no other choice for the actor than to incarnate the character. He had to work on his own feelings in order to achieve the "magic" that was, and sometimes still is, equated with quality in theater. Brecht's epic theatre however, proposed plays that were not directed to the empathy of the spectators but to their critical capacity. The actor showed the contradictions between himself and his character, thus producing an effect of alienation from the social conditions in which everyone lives. Epic theatre also has a didactic effect, mainly showing that history is made by men and it is therefore changeable by men.

7. Some words which are used to objectively describe some facts are often used in a derogatory way. An example that is impressively widespread is the use of words related to language like, for example, *dialect* and *accent*. In languages that have many different dialects spoken in one area, the word dialect is used too frequently to denote a dialect different from the official dialect. Analogously, some people claim that they speak without an accent (which is technically impossible) meaning that they speak with the accent some particular region or social class. Even if this phenomenon is not new, it is more anachronic than ever nowadays, considering the advances in linguistic knowledge achieved during the last century.

8. Some contemporary thinkers have remarked, and correctly in my view, that in contradiction with a common prejudice, the modern forms of power are not mainly of a repressive nature. Rather, their most evident mechanisms are the production of discourses, even certain forms of knowledge. Perhaps one of the strongest forms of oppression today is the obligation to say, to define.

References

[BK84] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Proceedings* 11^{th} *ICALP*, Antwerp, volume 172 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 1984.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.