Models and logics for process algebra
van der Zwaag, M.B.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
A Short Introduction to ACP

The axiom system ACP is a collection of algebraic laws that characterize behavior, or processes. We give an introduction to ACP, starting with some general process theory. We discuss concurrency and as an example we specify a simple distributed system. We have short sections on the use of data, logic and time in process algebra, and end with some bibliographical notes.

Process Theory

A process is taken to be the behavior of some physical device, like a computer running a program, or, more abstractly, of a system, that is able to perform actions. The execution of an action may be observed by the environment of the system: an observation is a sensory perception of an output of the system, and it can also be an interaction with the system that is induced directly by the environment. We shall define the behavior of systems in terms of these external observations; we abstract from the internal operations that lead to the observations.

In process theory, there seems to be agreement that systems are modelled best in terms of transitions between states, so that the execution of an action corresponds to the transition of one state to another. A key notion then is that of a (labelled) transition system: a transition system consists of a set of states that may be connected by transitions. Each of the transitions is labelled with an action symbol; a transition

\[ s \xrightarrow{a} t \]

means that (a system in) state \( s \) can evolve into state \( t \) by the execution of action \( a \). Furthermore, some states may be identified as initial states, and some states may be identified as final, or terminating, states.

Transition systems, or at least small ones, allow a nice graphical presentation; for example, consider the system in Figure 1. Here, the black dots are the states and \( a, b \) are the transition labels; the incoming arrow at the top state indicates that it is an initial state, and the symbol \( \sqrt{\ } \) marks final states. Starting in the initial state, first the action \( a \) is executed. Then either a second action \( a \) is executed, after which nothing is possible, or the action \( b \) is executed, whereby
A Short Introduction to ACP

FIGURE 1. A transition system.

a new state is reached in which the system can first execute another action $b$ and terminate successfully after that.

We find the (execution) traces of a state, if we put together in succession the labels that we encounter on such a run through the system; in the example, $aa$ and $ab$ are traces of the initial state. An accepting trace is a trace that belongs to a run that ends in a final state, so the only accepting trace of the initial state is $abb$. The set of accepting traces of a state plays a central role in the theory of formal languages, as this set may be considered as the language that is accepted by that state (in formal language theory, a transition system would normally be called a language accepting machine).

This brings us to the notion of the equivalence of states. In formal language theory, two states are equivalent if they accept the same language. In process theory, this is only one of many notions of equivalence; this particular equivalence, accepting trace equivalence, is in fact one of the coarsest, or weakest, equivalences in the spectrum of process equivalences [42]. When looking at a transition system from a process theoretical, or behavioral, point of view, one may want to be more discriminating than is possible with language equivalence.

FIGURE 2. Nondeterminism.

An important notion is that of the branching structure of the system; this is illustrated by the basic example in Figure 2. The top state on the left accepts the same language as the top state on the right, namely the set $\{ab, ac\}$. Still, we may want to distinguish these two states: the top state on the left initially has
no options other than to execute the action $a$, thereby reaching a state where it has a choice between $b$ and $c$. On the other hand, the top state on the right has two initial options, and while these options cannot be distinguished locally, they are in fact quite different, because one leads to a state where only $b$ can be chosen and the other leads to state where only $c$ can be chosen. We say that the moment of choice between $b$ and $c$ is different, and also that the transition system on the right is nondeterministic. This difference can also be illustrated as follows: consider a computer that, after it has been turned on, offers a menu with a choice between two operating systems. Assume that the action $a$ stands for turning on the computer, action $b$ is the choice for Linux, and action $c$ is the choice for Windows. We see that the transition system on the left models this choice much better than the system on the right, where the choice for a particular operating system is made by turning on the computer—and the user cannot predict the outcome of the choice!

There are many process equivalences that take the branching structure of transition systems into account, see [42] for an excellent overview, and it is an important topic of Chapter III (where we also have silent actions).

Now, a process is usually defined as a state modulo some equivalence, that is, two states model the same process exactly if they are equivalent. (Bear in mind that we have been imprecise in our definition of a transition system, and that many variations exist, sometimes under other names.)

**Process Algebra**

When talking about process algebra we shall mean the axiom system ACP, the Algebra of Communicating Processes, introduced by Bergstra and Klop in [15]. It provides a signature, that is, a language, that allows an effective notation for processes, and a set of axioms, that are used for equational reasoning about processes. It does not provide a particular model (such as, for example, a transition system model). In this view, any model of the axiom system is a process algebra, and a process is an element of a process algebra. Or, more loosely put, a process is anything that satisfies the axioms. This may be considered more abstract than other approaches, where usually a particular model is studied. Still, proposed models for ACP have been more or less like the transition system model. Widely used is so-called structural operational semantics, which is like a term model (terms are semantical objects), where transition relations between terms are defined by induction on the syntactic structure of the terms [1].

Let us start with BPA, for Basic Process Algebra, a subsystem of ACP. We present its signature and axioms, and we give a structural operational semantics. First, the axiom system is parametrized with a set $A$ of action symbols. The action symbols, written $a, b, \ldots$, are constants: an action symbol $a$ is a process term that describes the process that executes the action $a$ and after that
Table 1. Axioms of BPA.

<table>
<thead>
<tr>
<th>Axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y = y + x$</td>
</tr>
<tr>
<td>$(x + y) + z = x + (y + z)$</td>
</tr>
<tr>
<td>$x + x = x$</td>
</tr>
<tr>
<td>$(x + y)z = xz + yz$</td>
</tr>
<tr>
<td>$(xy)z = x(yz)$</td>
</tr>
</tbody>
</table>

terminates successfully. Then, there are the two binary operations $+$ and $\cdot$, standing for alternative and sequential composition. Alternative composition describes choice: the process $x + y$ executes $x$ or $y$, but not both. This construction is used to put together possible behaviors of a system. The sequential composition $x \cdot y$ starts with the execution of $x$, and when the execution of $x$ has terminated successfully, the execution of $y$ starts. We may suppress the symbol $\cdot$ in terms, writing $xy$ for $x \cdot y$. Furthermore, we let $\cdot$ bind more strongly than $+$. For example, the processes in Figure 2 would be described by the process terms $a(b + c)$ and $ab + ac$.

The axioms of BPA are listed in Table 1. The letters $x$, $y$, $z$ occurring in the axioms are variables; we assume a countably infinite set of variables and use the rules of equational logic for derivations. The axioms express that alternative composition is commutative, associative and idempotent, that sequential composition distributes from the right over alternative composition, and that sequential composition is associative. For example, we can derive that $(a + b)c$ equals $ac + bc$, but we cannot derive that $a(b + c)$ equals $ab + ac$ (cf. the example on nondeterminism in the section on process theory). This suggests that these axioms characterize an equivalence that is stronger than language equivalence, and indeed, the equivalence axiomatized by BPA is strong bisimulation equivalence, an equivalence that respects the branching structure of processes in the extreme. Two closed BPA terms are derivably equal if and only if they represent strongly bisimilar processes.

Next, we present an operational semantics. That is, we give rules that define transition relations between closed terms. The symbol $\checkmark$ represents successful termination; it is not a process term. An action symbol describes the process that executes an action followed by termination: for all $a \in A$ we have

$$a \xrightarrow{a} \checkmark.$$ 

The rules for alternative and sequential composition are in Table 2. These rules have two parts: on the top of the bar we put the premises of the rule, and below it the conclusion. If the premises hold (for a certain instantiation of the variables, that range over closed terms), then we infer that the conclusion holds as well (for the same instantiation). Looking at these rules we see that a sequential composition starts with the actions of the first process, and that an alternative composition continues as the remainder of the process that makes
Concurrent

The initial action. Thus, the transition rules induce a transition system that has the set of closed process terms as state space. We can define strong bisimulation equivalence for this transition system and show that two terms are derivably equal exactly if they are strongly bisimilar.

We end this section with some remarks on the expression of processes. First, a deadlock state is a state that has no outgoing transitions and also does not have the option to terminate successfully; it models a system that got stuck. With the addition of the constant $\delta$ for deadlock to the signature, we can express all finite processes.\(^1\) For example, the process in Figure 1 is expressed by the process term $a(\delta + bb)$. Still, many interesting processes are infinite, and for the expression of those we can use (sets of) recursive equations. For example, the equation $x = ax$ characterizes the process that executes the action $a$ infinitely many times in succession. A more recent development is the use of so-called recursive operations for the specification of infinite processes [12, 14]. The most basic of these is the binary Kleene star operation $*$, defined by the axiom

$$x^* y = x(x^* y) + y.$$ 

For example, the term $a^* \delta$ expresses the process mentioned above (the constant $\delta$ is a zero for alternative composition). In Chapter III, we discuss the expressivity of ACP in the context of orthogonal bisimulation equivalence.

Concurrent

The primary motivation for process algebra is the description of the concurrent, or parallel, operation of processes. The term $x \parallel y$ describes the parallel execution of $x$ and $y$; that is, these processes are executed independently, but they may be able to communicate. The assumption is that the execution of an action has no duration, and that the simultaneous execution of actions is only possible if these actions are involved in a communication action. So, if we observe the execution of an action from $x \parallel y$, then this is either an action from

\(^1\)Provided that they have pure termination: final states do not have outgoing transitions (cf. Section 2 of Chapter III).
$x$, an action from $y$, or a communication between $x$ and $y$. This assumption is called the *interleaving hypothesis*; it is axiomatized by

$$x \parallel y = x \downarrow y + y \downarrow x + x \upharpoonright y,$$

where $x \downarrow y$ describes the parallel execution of $x$ and $y$ with the restriction that any initial action must be performed by $x$, and $x \upharpoonright y$ also describes the parallel execution of $x$ and $y$, but now with the restriction that any initial action must be a communication between $x$ and $y$. The operations $\downarrow$ and $\upharpoonright$ lack the natural interpretation of the other operations; they were introduced as auxiliary operations for the axiomatization of the interleaving semantics that we described. A characteristic of ACP, that is a consequence of the interleaving hypothesis, is that all operations for parallel composition can be eliminated from closed terms: terms describing concurrent processes can be rewritten into a linear form in the signature of BPA with deadlock.

As a parameter of the axiom system, we assume a communication function that defines which actions are allowed to communicate, and what the result is: with $A$ the set of action symbols, it is a partial function $\gamma : A \times A \rightarrow A$, that is associative and commutative. For example, if $\gamma(a, b) = c$, then

$$a \parallel b = ab + ba + c.$$

Assuming an operational semantics in a style as suggested above the corresponding transition system would be as depicted in Figure 3. If $\gamma(a, b)$ is undefined, then $a \upharpoonright b$ equals the deadlock process $\delta$. (Recall that $\delta$ is a zero for alternative composition.)

![Figure 3. A transition system for $a \parallel b$ with $a \upharpoonright b = c$.](image)

**Verification**

The main application of process algebra has been the verification of communication protocols. A protocol is a prescription for the behavior of the components of a distributed system, intended at the realization of a certain behavior of the system as a whole. Importantly, we distinguish between external and internal actions of the system. The communications between components are usually considered to be unobservable for an external observer, or, from a different
perspective, to be irrelevant for the interaction of the system with its environment. We use the renaming operator $\tau_I$, where $I$ is a set of actions that we consider to be internal, to hide, or to abstract from, internal activity: if $p$ is a process term, then $\tau_I(p)$ is the result of renaming all internal actions in $p$ to the special action $\tau$. The execution of the action $\tau$ is not visible, and we have several equivalences that take this special character of $\tau$ into account (cf. the introduction of Chapter III).

Now, a process algebraic verification assumes two descriptions of a distributed system: one gives an abstract or high-level view of the system in terms of its external actions—call this view the specification—while the other one gives the behavior of the parallel components, call this the implementation. The specification is usually the desired behavior of a system, and the objective of the verification is to show that the implementation complies to the specification by proving that the two descriptions are equivalent: let $Spec$ be a process expression for the system specification, and let $Impl$ be the (encapsulated, see the example below) parallel composition of the expressions for the components. Then, a process algebraic verification is a proof of the equality

$$\tau_I(Impl) = Spec,$$

where $I$ is the set of actions that we want to abstract from. By equality we mean derivable equality in the axiom system; we assume an axiomatization of our preferred abstract semantics.

As an example, we specify a simple system consisting of $n + 1$ parallel components. The components can send each other messages via numbered ports. We define for naturals $i$:

$$P_i = \sum_m r_i(m) \cdot s_{i+1}(f_i(m)).$$

The summation sign is used to describe an alternative composition: its parameter $m$ ranges over a finite set of messages. So, a process $P_i$ reads any message $m$ at port $i$ by the action $r_i(m)$. Then it applies the function $f_i$ to the received message, and proceeds to send the value $f_i(m)$ at port $i + 1$ by the action $s_{i+1}(f_i(m))$. We leave the exact status of the message terms implicit; we address this point in the next section. Send and receive actions at the same port communicate: we let $s_i \mid r_i = c_i$, and let no other communications be defined.

The implementation is the encapsulated parallel composition of the processes $P_i$ for $i = 0, \ldots, n$. The encapsulation blocks the separate execution of internal actions that are supposed to communicate. In our example, the send and receive actions at ports $1, \ldots, n$ synchronize, yielding internal communications, while $r_0$ and $s_{n+1}$ are the only external actions. Hence, we block the execution of actions in the set

$$H = \{s_i, r_i \mid i = 1, \ldots, n\},$$
by putting the parallel composition of the components in the scope of the encapsulation operator $\partial_H$; we let

$$\text{Impl} = \partial_H (P_0 || \cdots || P_n).$$

Due to the encapsulation, the only initial actions of the system are the receive actions by $P_0$: the system starts with the receiving of a message at port 0. Then the message is passed on through the system, while every process on the way updates the message with its function $f$. So, if we abstract from internal communications, then we find that we can express its external behavior as

$$\text{Spec} = \sum_{m} r_0(m) \cdot s_{n+1}(f_n(\cdots (f_0(m)) \cdots)).$$

Now, a verification would be a proof of

$$\tau_{\{c_1, \ldots, c_r\}}(\text{Impl}) = \text{Spec},$$

which is a straightforward exercise for any instance of $n$. Here, we are assuming any abstract semantics except orthogonal bisimulation equivalence, since in that semantics internal activity can be compressed, but not be hidden completely (see Chapter III).

**Data**

In the example above we assumed a data type for the messages and we used parametrization of actions with messages and summation over a data type to model the input of any datum. These are typical uses of data in applications of process algebra. However, data types are not part of ACP; usage such as in the example is informal and in the end insufficient for larger scale applications.

The axiom system $\mu\text{CRL}$ (micro Common Representation Language) [52] is an extension of ACP with equationally specified abstract data types. It offers a many-sorted signature that may be extended further by adding new data types. Data terms occur in process terms in three ways: first, actions and recursion variables may be parametrized with data; second, there is a binding construction allowing summation over possibly infinite data types; and finally there is conditional composition, where the condition is a boolean term.

For example, a buffer process transmitting natural numbers may be given by the recursive specification

$$\text{Buffer} = \sum_{n: \text{Nat}} r(n) \cdot s(n) \cdot \text{Buffer}. $$

Remember that in the example in the previous section we used summation over input values as well, but there the summation was an abbreviation for a finite alternative composition. Here, the summation binds the variable $n$ that ranges over infinitely many values (cf. [66]).
As a second example, we define a register process by

\[
\text{Register}(n : \text{Nat}) = \text{succ} \cdot \text{Register}(n + 1) \\
+ (\text{zero} \cdot \text{Register}(n) + \text{exit}) < n = 0 > \text{pred} \cdot \text{Register}(n - 1).
\]

A conditional composition \( x < b > y \) behaves like \( x \) if the boolean condition \( b \) is true, and like \( y \) if the condition is false. The register process can perform the exit action if it holds value 0; it can always do the successor action, thereby increasing its value, and it can do the predecessor action if its value is at least 1. It has a zero test action that does not change its value. (See Section 7 of Chapter III for the expression of registers in ACP using recursive operations.)

Many case studies have been performed using \( \mu \)CRL, see for example \([25, 37, 51, 78]\), and a set of tools aiding verification and analysis of systems is available, and is still under further development \([28]\).

A useful methodology for verification is the so-called cones and foci proof technique \([53]\). In Chapter IV, we present a verification of a leader election protocol using this technique, and in Chapter V we extend it to a setting with explicit timing (see also the section on time below).

**Logic**

We consider two uses of logic: first there are modal logics that are used to express properties of states in transition systems, this use falls in the domain of process theory rather than process algebra; second, logical formulas may enter process terms, if they are used as conditions in a process algebraic construction like, for example, the guarded command \( \phi ::= x \), expressing that process \( x \) can be executed under the condition that formula \( \phi \) holds.

In Chapter III, that is devoted to the introduction of orthogonal bisimulation equivalence, we encounter the first use of logic: there, we give a modal logic characterizing orthogonal bisimilarity. As an example, we present here what is probably the best-known modal characterization of a process equivalence: Hennessy-Milner logic \([56]\).

Assume a transition system with transition labels ranged over by \( a \); for simplicity of the example we do not distinguish successfully terminating states. Then formulas are defined inductively as follows: \( \top \) is a formula ('true'); if \( a \) is a transition label and \( \phi \) and \( \psi \) are formulas, then \( a\phi \), \( \neg \phi \), and \( \phi \land \psi \) are formulas. We define satisfaction of a formula \( \phi \) in a state \( s \), notation \( s \models \phi \), inductively as follows:

- \( s \models \top \),
- \( s \models \neg \psi \) if not \( s \models \psi \),
- \( s \models \psi \land \chi \) if \( s \models \psi \) and \( s \models \chi \), and
- \( s \models a\psi \) if \( s \xrightarrow{a} r \) for some state \( r \) with \( r \models \psi \).
This logic characterizes strong bisimulation equivalence: in finitely branching transition systems, it holds that two states are strongly bisimilar if and only if they satisfy the same set of formulas.

Next, we look at the second use of logic: conditionals in process algebra. A principal construction, that we encountered earlier in the section on data, is conditional composition. It is the subject of Chapter II, where it is written \( x +_\phi y \), a notation that suggests a similarity to alternative composition. Like alternative composition, conditional composition is a mechanism for summing up possible behaviors, but it has information on the nature of the choice. Unlike alternative composition, it also has an imperative interpretation: it may be read as an instruction to execute \( x \) if the condition \( \phi \) holds, and to execute \( y \) otherwise. In Chapter II we propose a four-valued logic for these conditions, that has truth values for 'overdefined' and for 'undefined'. If the condition \( \phi \) is overdefined, then \( x +_\phi y \) stands for \( x + y \); if \( \phi \) is undefined, then it stands for deadlock. Thus, conditional composition is a generalization of alternative composition.

**Time**

Until now we have not considered the timing of actions, that is, we have been able to express the order in which actions must be performed, but we have not been able to express that an action must be performed at a certain time. Still, many systems crucially depend on such timing. For example, it may be required that a system produces some output exactly at 12:15 in the afternoon, or between 27 and 44 milliseconds after some earlier event.

For the modelling of such timing-dependent systems, process algebras have been extended with timing operations in a number of ways. Two important choices to be made are that between absolute and relative timing, and that between discrete and continuous time. And then there are many more design issues, such as, to name some technical terms, urgency of execution, concurrency of simultaneous actions, (immediate) time-deadlocking, and time factorization.

In Chapter VI, an extension of the basis of \( \mu \)CRL with time-stamping of actions is presented. This exercise served as a preliminary study for the completeness proof of timed \( \mu \)CRL [77].

Timed \( \mu \)CRL [47] is a language that allows a very direct specification of timed processes: time can easily be specified as a data type; the only requirements are that the domain should be totally ordered and have a smallest element. Furthermore, the binder \( \sum \) can be used to bind time variables, and conditional composition can be used to restrict possible timings. Consider for example the following specification of a process that must perform action \( a \)
within 4 time units:
\[ \sum_{t: \text{Time}} a \cdot t < t \leq 4 \triangleright \delta \cdot 0, \]

where \( a \cdot t \) means 'action \( a \) at time \( t \)', and \( \delta \cdot 0 \) is a zero for alternative composition.

Although timed \( \mu \text{CRL} \) is adequate for the expression of timed processes, we have little experience in verifying timed systems, that is, there have been some exercises in analyzing implementations [55], but the integration of time and abstraction, and the actual verification of systems, have hardly been explored. As a step in this direction, we present a verification technique for timed systems in Chapter V. It can be used to prove timed branching bisimilarity of timed transition systems (that are the semantical objects represented by timed \( \mu \text{CRL} \) expressions). This proof technique is the timed variant of the aforementioned cones and foci technique [53].

**Bibliographical Notes**

Textbooks on ACP are [11] and [35]. Other well-known process algebras are CCS [68], and CSP [30]. See [66] for an in-depth discussion of \( \mu \text{CRL} \) (and in particular of its summation over data). The *Handbook of Process Algebra* [23] contains valuable contributions reflecting the current state of the art, and many references for further reading.