Modular Specifications in Process Algebra

With Curious Queues

(extended abstract)

Rob van Glabbeek and Frits Vaandrager
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

In recent years a wide variety of process algebras has been proposed in the literature. Often these process algebras are closely related: they can be viewed as homomorphic images, submodels or restrictions of each other. The aim of this paper is to show how the semantical reality, consisting of a large number of closely related process algebras, can be reflected, and even used, on the level of algebraic specifications and in process verifications. This is done by means of the notion of a module. The simplest modules are building blocks of operators and axioms, each block describing a feature of concurrency in a certain semantical setting. These modules can then be combined by means of a union operator $+$, an export operator $\Box$, allowing to forget some operators in a module, an operator H, changing semantics by taking homomorphic images, and an operator S which takes subalgebras. These operators enable us to combine modules in a subtle way, when the direct combination would be inconsistent. We show how auxiliary process algebra operators can be hidden when this is needed. Moreover it is demonstrated how new process combinators can be defined in terms of the more elementary ones in a clean way. As an illustration of our approach, a methodology is presented that can be used to specify FIFO-queues, and that facilitates verification of concurrent systems containing these queues.

Key Words & Phrases: process algebra, concurrency, modular algebraic specifications, export operator, union of modules, homomorphism operator, subalgebra operator, FIFO-queues, chaining operator, communication protocols.
1985 Mathematical Subject Classification: 68Q10, 68Q55, 68Q60.
1980 Mathematical Subject Classification: 68B10, 68C01, 68D25, 68F20.
1982 CR Categories: C.2.2, D.1.3, D.2.1, D.2.2, F.1.1, F.1.2, F.3.2.
Note: The research of the authors was supported by ESPRIT project no. 432, An Integrated Formal Approach to Industrial Software Development (METEOR). The research of the second author was also supported by RACE project no. 1046, Specification and Programming Environment for Communication Software (SPECS). A full version of this paper appeared as [23].

INTRODUCTION

During the last decade, a lot of research has been done on *process algebra*: the branch of theoretical computer science concerned with the modelling of concurrent systems as elements of an algebra. Besides the Calculus of Communicating Systems (CCS) of MILNER [31, 32], several related formalisms have been developed, such as the theory of Communicating Sequential Processes (CSP) of HOARE [17, 25], the MEIJE calculus of AUSTRY & BOUDOL [2] and the Algebra of Communicating Processes (ACP) of BERGSTRA & KLOP [11-14].

When work on process algebra started, many people hoped that it would be possible to come up, eventually, with the 'ultimate' process algebra, leading to a 'Church thesis' for concurrent computation. This process algebra, one imagined, should contain only a few fundamental operators and it should be suited to model all concurrent computational processes. Moreover there should be a calculus for this model making it possible to prove the identity of processes algebraically, thus proving correctness of implementations with respect to specifications. As far as we know, the ultimate process algebra has not yet been found, but we will not exclude that it will be discovered in the near future.

Two things however, have become clear in the meantime: (1) it is doubtful whether algebraic system verification, as envisaged in [31], will be possible in this model, and (2) even if the ultimate process algebra exists, this certainly does not mean that all other process algebras are no longer interesting. We elaborate on this below.

A central idea in process algebra is that two processes which cannot be distinguished by observation should preferably be identified: the process semantics should be fully abstract with respect to some notion of testing (see [20, 31]). This means that the choice of a suitable process algebra may depend on the tools an environment has to distinguish between certain processes. In different applications the tools of the environment may be different, and therefore different applications may require different process algebras. A large number of process semantics are not fully abstract with respect to any (reasonable) notion of testing (bisimulation semantics and partial order semantics, for instance). Still these semantics can be very interesting because they have simple definitions or correspond to some strong operational intuition. Our hypothetical ultimate process algebra will make very few identifications, because it should be resistant against all forms of testing. Therefore not many algebraic laws will be valid in this model and algebraic system verification will presumably not be possible (specification and implementation correspond to different processes in the model).

Another factor which plays a role has to do with the operators of process algebras. For theoretical purposes it is in general desirable to work with a single, small set of fundamental operators. We doubt however that such a unique optimal and minimal collection exists. What is optimal depends on the type of result one likes to prove. This becomes even more clear if we look towards practical applications. One could say that the main message of sections 4 and 5 of this paper is that chaining operators (which are not considered to be fundamental theoretically) are extremely useful for the specification of various types of queues and for the verification of properties of concurrent systems containing these queues. Some operators in process algebra can be used for a wide range of applications, but we agree with JIFENG & HOARE [26] that we may have to accept that each application will require derivation of specialised laws (and operators) to control its complexity.

Many people are embarrassed by the multitude of process algebras occurring in the literature. They should be aware of the fact that there are close relationships between the various process algebras: often one process algebra can be viewed as a homomorphic image, subalgebra or restriction of another one. The aim of this paper is to show how the semantical reality, consisting of a large number of closely related process algebras, can be reflected, and even used, on the level of algebraic specifications and in process verifications.

This paper is about process algebras, their mutual relationships, and strategies to prove that a formula is valid in a process algebra. Still, we do not present any particular process algebra in this paper. We only define classes of models of process modules. One reason for doing this is

that a detailed description of particular process algebras would make this paper too long. Another reason is that there is often no clear argument for selecting a particular process algebra. In such situations we are interested in assertions saying that a formula is valid in all algebras satisfying a certain theory. A number of times we need results stating that some formulas *cannot* be proven from a certain module. A standard way to prove this is to give a model of the module where the formulas are not true. For this reason we will often refer to particular process algebras which have been described elsewhere in the literature.

The discussion of this paper takes place in the setting of ACP. We think however that the results can be carried over to CCS, CSP, MEIJE, or any other process algebra formalism.

Modularisation.

The creation of an algebraic framework suitable to deal with realistic applications, gives rise to the construction of building blocks, or modules, of operators and axioms, each block describing a feature of concurrency in a certain semantical setting. These modules can then be combined by means of a module combinator $+$. We give some examples:

i) A kernel module, that expresses some basic features of concurrent processes, is the module ACP. For a lot of applications however, ACP does not provide enough operators. Often the use of *renaming operators* makes specifications shorter and more comprehensible. These renaming operators can be defined in a separate module RN. Now the module ACP+RN combines the specification and verification power of modules ACP and RN.

ii) The axioms of module ACP correspond to the semantical notion of bisimulation. For some applications bisimulation semantics does not make enough identifications. In these cases one would like to deal with processes on the level of, for example, failure semantics. Now one can define a module F, corresponding to the identifications made in failure semantics on top of the identifications of bisimulation semantics. The module ACP+F then corresponds to the failure model.

Once a number of modules have been defined, they can be combined in a lot of ways. Some combinations are interesting (for example the module ACP+RN+F), for other combinations no interesting applications exist (the module RN+F). Didactical aspects aside, a major advantage of the modular approach is that results which have been proved from a module M, can also be proved from a module M+N. This means that process verifications become *reusable*.

It turns out that certain pairs of modules are incompatible in a very strong sense: with the combination of two modules strange and counter-intuitive identities can be derived. In BAETEN, BERGSTRA & KLOP [6], for example, it is shown that the combination of failure semantics and the priority operator is inconsistent in the sense that an identity can be derived which says that a process that can do a b-action after it has done an a-action, equals a process that cannot do this. Another example can be found in BERGSTRA, KLOP & OLDEROG [15], where it is pointed out that the combination of failure semantics and Koomen's Fair Abstraction Rule (KFAR) is inconsistent.

In the first section of this paper we present, beside the combinator $+$, some other operators on modules. We discuss an export operator $\Box$, allowing to forget some operators in a module, an operator H, changing semantics by taking homomorphic images, and an operator S which takes subalgebras. These operators enable us to combine modules in a subtle way, when the direct combination would be inconsistent. In section 2 we describe all the basic process modules used in the rest of the paper. Section 3 contains two examples of applications of the new module operators in process algebra:

1. The axiom system ACP contains auxiliary operators $\between$ and $|$ (left-merge and communication-merge) which drastically simplify computations and have some desirable 'metamathematical' consequences (finite axiomatisability[1]; greater suitability for term

1. Recently, Faron Moller from Edinburgh showed that the merge operator cannot be finitely axiomatised without auxiliary operators.

rewriting analysis). These auxiliary operators can be defined in a large class of process algebras. However, it turns out that in a setting with the silent step τ the left-merge cannot be added consistently to all algebras (for instance not to the usual variants of failure semantics). Now one may think that this result means that someone who is doing failure semantics with τ's cannot profit from the nice properties of the left-merge. However, we will show in this paper that use of the module approach makes it possible to do failure semantics with τ's but still benefit from the left-merge in verifications. The idea is that verifications take place on two levels: the level of bisimulation semantics where the left-merge can be used, and a level of for instance failure semantics, where no left-merge is present. The failure model can be obtained from the bisimulation model by removing the auxiliary operators and taking a homomorphic image. Now we use the observation that certain formulas (the 'positive' ones without auxiliary operators) are preserved under this procedure. A consequence of this application is that even if bisimulation semantics is not considered to be an appropriate process semantics (since it is not fully abstract with respect to any reasonable notion of testing), it still can be useful as an expedient for proving formulas in failure semantics.

2. As already pointed out above, one would like to have, from a theoretical point of view, as few operators or combinators as possible. On the other hand, when dealing with applications, it is often very rewarding to introduce new operators. This paradox can be resolved if the new operators are definable in terms of the more elementary ones. In that case the new operators can be considered as notations which are useful, but do not complicate the underlying theory. A problem with defining operators in terms of other operators is that often auxiliary atomic actions are needed in the definition. These auxiliary actions can then not be used in any other place, because that would disturb the intended semantics of the operator. In the laws that can be derived for the defined operator, the auxiliary actions occur prominently. These 'side effects' are often quite unpleasant. One may think that side effects are unavoidable and that someone who really does not like them should define new operators directly in the algebras (even though this is in conflict with the desire to have as few operators as possible). However, we will show that the module approach can be used to solve also this problem: with the restriction operator we remove the auxiliary actions from the signature and then we apply the subalgebra operator in order to 'move' to algebras where the auxiliary actions are not present at all.

The concept of hiding auxiliary operators in a module in some formal way is quite familiar in the literature (see BERGSTRA, HEERING & KLINT [9] for example), but the use of module operators H and S, and their application in combining modules that would be incompatible otherwise, is, as far as we know, new. The H and S operations are in spirit related to the **abstract** operation of SANNELLA & WIRSING [38] and SANNELLA & TARLECKI [37], which also extends the model class of a module.

In previous papers on ACP, the underlying logic used in process verifications was not made explicit. The reason for this was that a long definition of the logic would distract the reader's attention from the more essential parts of the paper. It was felt that filling in the details of the logic would not be too difficult and that moreover different options were equivalent. In this paper we generalise the classical notion of a formal proof of a formula from a theory to the notion of a formal proof of a formula from a module. The definition of this last notion is parametrised by the underlying logic. What is provable from a module really depends on the logic that is used, and this makes it necessary to consider in more detail the issue of logics. In an appendix we present three alternatives: (1) Equational logic. This logic is suited for dealing with finite processes, but not strong enough for handling infinite processes; (2) Infinitary conditional equational logic. This is the logic used in the process verifications of this paper; (3) First order logic with equality.

Our investigations into the precise nature of the calculi used in process algebra, led us to alternative formulations of some of the proof principles in ACP which fit better in our formal

setup. We present a reformulation of the Recursive Specification Principle (RSP) and also an alphabet operator which returns a process instead of a set of actions.

Queues.
As an illustration of the techniques developed in sections 1 to 3, we present in section 4 an algebraic treatment of FIFO-queues. FIFO-queues play an important role in the description of languages with asynchronous message passing, the modelling of communication channels occurring in computer networks and the implementation of languages with synchronous communication. We show how the chaining operator can be used to give short specifications of various (faulty) queues and simple proofs of numerous identities, for example of the fact that the chaining of a queue with unbounded capacity and a one datum buffer is again a queue.

We give an example of an identity that holds intuitively (there is no experiment that distinguishes between the two processes) but is not valid in bisimulation semantics. We use the machinery developed in section 1-3 to extend the axiom system in a neat way (avoiding inconsistencies) so that we can prove the processes identical.

A protocol verification.
The usefulness of the proof technique for queues is illustrated in section 5, where we sketch a modular verification of a concurrent alternating bit protocol. The complete verification, which is presented in [23], takes 4 pages (or 5 if the proof of the standard facts about the queues is included) and is thereby considerably shorter than the proof of similar protocols in papers by KOYMANS & MULDER [27] and LARSEN & MILNER [28] (15 and 11 pages respectively). The verification shows that the protocol is correct if the channels behave as faulty FIFO-queues with unbounded capacity. However, a minor change in the proof is enough to show that the protocol also works if the channels behave as n-buffers, faulty n-buffers, etc. In our view the basic merit of our way of dealing with queues is that it becomes possible to use inductive arguments when dealing with the length of queues in protocol systems.

§1 MODULE LOGIC

In this paper, as in many other papers about process algebra, we use formal calculi to prove statements about concurrent systems. In this section we answer the following questions:
- Which kind of calculi do we use?
- What do we understand by a proof?

In the next sections we will apply this general setup to the setting of concurrent systems.

1.1. Statements about concurrent systems. In many theories of concurrency it is common practice to represent processes - the behaviours of concurrent systems - as elements in an *algebra*. This is a mathematical domain, on which some operators and predicates are defined. Algebras, which are suitable for the representation of processes are called *process algebras*. Thus a statement about the behaviour of concurrent systems can be regarded as a statement about the elements of a certain process algebra. Such a statement can be represented by a formula in a suitable language which is interpreted in this process algebra. Sometimes we consider several process algebras at the same time and want to formulate a statement about concurrent processes without choosing one of these algebras. In this case we represent the statement by a formula in a suitable language which has an interpretation in all these process algebras. Hence we are interested in assertions of the form: 'Formula ϕ holds in the process algebra $\mathcal{A}$', notation $\mathcal{A} \models \phi$, or 'Formula ϕ holds in the class of process algebras $\mathcal{C}$', notation $\mathcal{C} \models \phi$. Now we can formulate the goal that is pursued in the present section: to propose a method for proving assertions $\mathcal{A} \models \phi$, or $\mathcal{C} \models \phi$.

1.2. Proving formulas from theories. Classical logic gave us the notion of a formal proof of a formula ϕ from a theory T. Here a theory is a set of formulas. We write $T \vdash \phi$ if such a proof exists. The use of this notion is revealed by the following soundness theorem: *If $T \vdash \phi$ then ϕ holds in all algebras satisfying T.* Here an algebra $\mathcal{A}$ satisfies T, notation $\mathcal{A} \models T$, if all formulas of T hold in this algebra. Thus if we want to prove $\mathcal{A} \models \phi$ it suffices to prove $T \vdash \phi$ and $\mathcal{A} \models T$ for a suitable theory T. Likewise, if we want to prove $\mathcal{C} \models \phi$, with $\mathcal{C}$ a class of algebras, it suffices to prove $T \vdash \phi$ and $\mathcal{C} \models T$.

At first sight the method of proving $\mathcal{A} \models \phi$ by means of a formal proof of ϕ out of T seems very inefficient. Instead of verifying $\mathcal{A} \models \phi$, one has to verify $\mathcal{A} \models \psi$ for all $\psi \in T$, and moreover the formal proof has to be constructed. However, there are two circumstances in which this method *is* efficient, and in most applications both of them apply. First of all it might be the case that ϕ is more complicated than the formulas of T and that a direct verification of $\mathcal{A} \models \phi$ is much more work than the formal proof and all verifications $\mathcal{A} \models \psi$ together. Secondly, it might occur that a single theory T with $\mathcal{A} \models T$ is used to prove many formulas ϕ, so that many verifications $\mathcal{A} \models \phi$ are balanced against many formal proofs of ϕ out of T and a single set of verifications $\mathcal{A} \models \psi$. Especially when constructing formal proofs is considered easier then making verifications $\mathcal{A} \models \phi$, this reusability argument is very powerful. It also indicates that for a given algebra $\mathcal{A}$ we want to find a theory T from which most interesting formulas ϕ with $\mathcal{A} \models \phi$ can be proved.

Often there are reasons for representing processes in an algebra that satisfies a particular theory T, but there is no clear argument for selecting one of these algebras. In this situation we are interested in assertions $\mathcal{C} \models \phi$ with $\mathcal{C}$ the class of all algebras satisfying T. Of course assertions of this type can be conveniently proved by means of a formal proof of ϕ from T.

1.3. Proving formulas from modules. In process algebra we often want to modify the process algebra currently used to represent processes. Such a modification might be as simple as the addition of another operator, needed for the proper modelling of yet another feature of concurrency, but it can also be a more involved modification, such as factoring out a congruence, in order to identify processes that should not be distinguished in a certain application. It is our explicit concern to organise proofs of statements about concurrent systems in such a way that, whenever possible, our results carry over to modifications of the process algebra for which they were proved.

Now suppose $\mathcal{A}$ is a process algebra satisfying the theory T and a statement $\mathcal{A} \models \phi$ has been proved by means of a formal proof of ϕ out of T. Furthermore suppose that $\mathcal{B}$ is obtained from $\mathcal{A}$ by factoring out a congruence relation on $\mathcal{A}$ (so $\mathcal{B}$ is a *homomorphic image* of $\mathcal{A}$) and for a certain application $\mathcal{B}$ is considered to be a more suitable model of concurrency than $\mathcal{A}$. Then in general $\mathcal{B} \models \phi$ cannot be concluded, but if ϕ belongs to a certain class of formulas (the *positive* ones) it can. So if ϕ is positive we can use the following theorem: 'If $\mathcal{A} \models T$, $T \vdash \phi$, ϕ is positive, and $\mathcal{B}$ is a homomorphic image of $\mathcal{A}$, then $\mathcal{B} \models \phi$'. This saves us the trouble of finding another theory U, verifying that $\mathcal{B} \models U$ and proving $U \vdash \phi$ for many formulas ϕ that have been proved from T already. Another way of formulating the same idea is to introduce a module $H(T)$. We postulate that one may derive '$H(T) \vdash \phi$' from '$T \vdash \phi$' and 'ϕ is positive', and $H(T) \vdash \phi$ implies that ϕ holds in all homomorphic images of algebras satisfying T.

Thus we propose a generalisation of the notion of a formal proof. Instead of theories we use the more general notion of *modules*. Like a theory a module characterises a class $\mathcal{C}$ of algebras, but besides the class of all algebras satisfying a given set of formulas, $\mathcal{C}$ can for instance also be the class of homomorphic images or subalgebras of a class of algebras specified earlier. Now a proof in the framework of module algebra is a sequence or tree of assertions $M \vdash \phi$ such that in each step either the formula ϕ is manipulated, as in classical proofs, or the module M is manipulated. Of course we will establish a soundness theorem as before, and then an assertion $\mathcal{A} \models \phi$ can be proved by means of a module M with $\mathcal{A} \models M$ and a formal proof of ϕ out of M. We will now turn to the formal definitions.

1.4. Signatures. Let NAMES be a given set of names.
A *sort declaration* is an expression $\mathbb{S}:S$ with $S \in$ NAMES.
A *function declaration* is an expression $\mathbb{F}:f:S_1 \times \cdots \times S_n \to S$ with $f, S_1, \cdots, S_n, S \in$ NAMES.
A *predicate declaration* is an expression $\mathbb{R}:p \subseteq S_1 \times \cdots \times S_n$ with $p, S_1, \cdots, S_n \in$ NAMES.
A *signature* σ is a set of sort, function and predicate declarations, satisfying:

$$\mathbb{F}_\sigma:f:S_1 \times \cdots \times S_n \to S \;\Rightarrow\; \mathbb{S}_\sigma:S_i \; (i=1, \cdots, n) \wedge \mathbb{S}_\sigma:S$$

$$\mathbb{R}_\sigma:p \subseteq S_1 \times \cdots \times S_n \;\Rightarrow\; \mathbb{S}_\sigma:S_i \; (i=1, \cdots, n)$$

Here $\mathbb{S}_\sigma:S$ is an abbreviation for $(\mathbb{S}:S) \in \sigma$ and likewise for $\mathbb{F}_\sigma$ and $\mathbb{R}_\sigma$. A function declaration $\mathbb{F}:f:\to S$ of arity 0 is sometimes called a *constant declaration* and written as $\mathbb{F}:f \in S$.

1.5. σ-Algebras. Let σ be a signature. A *σ-algebra* $\mathfrak{A}$ is a function on σ that maps

$\mathbb{S}_\sigma:S$ to a set $S^{\mathfrak{A}}$

$\mathbb{F}_\sigma:f:S_1 \times \cdots \times S_n \to S$ to a function $f^{\mathfrak{A}}_{S_1 \times \cdots \times S_n \to S}: S_1^{\mathfrak{A}} \times \cdots \times S_n^{\mathfrak{A}} \to S^{\mathfrak{A}}$ and

$\mathbb{R}_\sigma:p \subseteq S_1 \times \cdots \times S_n$ to a predicate $p^{\mathfrak{A}}_{S_1 \times \cdots \times S_n} \subseteq S_1^{\mathfrak{A}} \times \cdots \times S_n^{\mathfrak{A}}$.

Let $\mathfrak{A}$ and $\mathfrak{B}$ be σ-algebras.

$\mathfrak{B}$ is a *subalgebra* of $\mathfrak{A}$ if $S^{\mathfrak{B}} \subseteq S^{\mathfrak{A}}$ for all $\mathbb{S}_\sigma:S$, if moreover $f^{\mathfrak{A}}_{S_1 \times \cdots \times S_n \to S}$ restricted to $S_1^{\mathfrak{B}} \times \cdots \times S_n^{\mathfrak{B}} \to S^{\mathfrak{B}}$ is just $f^{\mathfrak{B}}_{S_1 \times \cdots \times S_n \to S}$ for all $\mathbb{F}_\sigma:f:S_1 \times \cdots \times S_n \to S$, and if $p^{\mathfrak{A}}_{S_1 \times \cdots \times S_n}$ restricted to $S_1^{\mathfrak{B}} \times \cdots \times S_n^{\mathfrak{B}}$ is just $p^{\mathfrak{B}}_{S_1 \times \cdots \times S_n}$ for all $\mathbb{R}_\sigma:p \subseteq S_1 \times \cdots \times S_n$.

A *homomorphism* $h:\mathfrak{A} \to \mathfrak{B}$ consists of mappings $h_S: S^{\mathfrak{A}} \to S^{\mathfrak{B}}$ for all $\mathbb{S}_\sigma:S$, such that

$$h_S(f^{\mathfrak{A}}_{S_1 \times \cdots \times S_n \to S}(x_1, \cdots, x_n)) = f^{\mathfrak{B}}_{S_1 \times \cdots \times S_n \to S}(h_{S_1}(x_1), \cdots, h_{S_n}(x_n))$$

for all $\mathbb{F}_\sigma:f:S_1 \times \cdots \times S_n \to S$ and all $x_i \in S_i^{\mathfrak{A}} (i=1, \cdots, n)$

$$p^{\mathfrak{A}}_{S_1 \times \cdots \times S_n}(x_1, \cdots, x_n) \;\Leftrightarrow\; p^{\mathfrak{B}}_{S_1 \times \cdots \times S_n}(h_{S_1}(x_1), \cdots, h_{S_n}(x_n))$$

for all $\mathbb{R}_\sigma:p \subseteq S_1 \times \cdots \times S_n$ and all $x_i \in S_i^{\mathfrak{A}} (i=1, \cdots, n)$

$\mathfrak{B}$ is a *homomorphic image* of $\mathfrak{A}$ if there exists a surjective homomorphism $h:\mathfrak{A} \to \mathfrak{B}$.

Let $\mathfrak{A}$ be a σ-algebra. The *restriction* $\rho \square \mathfrak{A}$ of $\mathfrak{A}$ to the signature ρ is the $\rho \cap \sigma$-algebra $\mathfrak{B}$, defined by

$$S^{\mathfrak{B}} = S^{\mathfrak{A}} \text{ for all } \mathbb{S}_{\rho \cap \sigma}:S$$

$$f^{\mathfrak{B}}_{S_1 \times \cdots \times S_n \to S} = f^{\mathfrak{A}}_{S_1 \times \cdots \times S_n \to S} \text{ for all } \mathbb{F}_{\rho \cap \sigma}:f:S_1 \times \cdots \times S_n \to S$$

$$p^{\mathfrak{B}}_{S_1 \times \cdots \times S_n} = p^{\mathfrak{A}}_{S_1 \times \cdots \times S_n} \text{ for all } \mathbb{R}_{\rho \cap \sigma}:p \subseteq S_1 \times \cdots \times S_n$$

1.6. Logics. A *logic* $\mathfrak{L}$ is a complex of prescriptions, defining for any signature σ

a set $F^{\mathfrak{L}}_\sigma$ of *formulas* over σ such that $F^{\mathfrak{L}}_\sigma \cap F^{\mathfrak{L}}_\rho = F^{\mathfrak{L}}_{\sigma \cap \rho}$,

a binary relation $\models^{\mathfrak{L}}_\sigma$ on σ-algebras $\times F^{\mathfrak{L}}_\sigma$ such that for all ρ-algebras $\mathfrak{A}$ and $\phi \in F^{\mathfrak{L}}_{\sigma \cap \rho}$: $\sigma \square \mathfrak{A} \models^{\mathfrak{L}}_{\sigma \cap \rho} \phi \;\Leftrightarrow\; \mathfrak{A} \models^{\mathfrak{L}}_\rho \phi$

and a set $I^{\mathfrak{L}}_\sigma$ of *inference rules* $\dfrac{H}{\phi}$ with $H \subseteq F^{\mathfrak{L}}_\sigma$ and $\phi \in F^{\mathfrak{L}}_\sigma$.

If $\mathfrak{A} \models^{\mathfrak{L}}_\sigma \phi$ we say that the σ-algebra $\mathfrak{A}$ *satisfies* the formula ϕ, or that ϕ *holds* in $\mathfrak{A}$. A *theory* over σ is a set of formulas over σ. If T is a theory over σ and $\mathfrak{A} \models^{\mathfrak{L}}_\sigma \phi$ for all $\phi \in T$ we say that $\mathfrak{A}$ satisfies T, notation $\mathfrak{A} \models^{\mathfrak{L}}_\sigma T$. We also say that $\mathfrak{A}$ is a *model* of T.

A logic $\mathfrak{L}$ is *sound* if $\dfrac{H}{\phi} \in I^{\mathfrak{L}}_\sigma$ implies $\mathfrak{A} \models^{\mathfrak{L}}_\sigma H \;\Rightarrow\; \mathfrak{A} \models^{\mathfrak{L}}_\sigma \phi$ for any σ-algebra $\mathfrak{A}$.

A formula $\phi \in F_\sigma^{\mathfrak{L}}$ is *preserved under subalgebras* if $\mathfrak{A} \models_\sigma^{\mathfrak{L}} \phi$ implies $\mathfrak{B} \models_\sigma^{\mathfrak{L}} \phi$, for any subalgebra $\mathfrak{B}$ of $\mathfrak{A}$.
A formula $\phi \in F_\sigma^{\mathfrak{L}}$ is *preserved under homomorphisms* if $\mathfrak{A} \models_\sigma^{\mathfrak{L}} \phi$ implies $\mathfrak{B} \models_\sigma^{\mathfrak{L}} \phi$, for any homomorphic image $\mathfrak{B}$ of $\mathfrak{A}$.

Without doubt, the definition of a 'logic' as presented above is too general for most applications. However, it is suited for our purposes and anyone can substitute his/her favourite (and more restricted) definition whenever he/she likes.

In the process algebra verifications of this paper we will use infinitary conditional equational logic. The definition of this logic can be found in the appendix. For comparison, the definitions of equational logic and first order logic with equality are included too.

1.7. Classical logic.
DERIVABILITY. A σ-*proof* of a formula $\phi \in F_\sigma^{\mathfrak{L}}$ from a theory $T \subseteq F_\sigma^{\mathfrak{L}}$ using the logic $\mathfrak{L}$, is a well-founded, upwardly branching tree of which the nodes are labelled by σ-formulas, such that

- the root is labelled by ϕ
- and if ψ is the label of a node q and H is the set of labels of the nodes directly above q then
  - either $\psi \in T$ and $H = \emptyset$,
  - or $\frac{H}{\psi} \in I_\sigma^{\mathfrak{L}}$.

If a σ-proof of ϕ from T using $\mathfrak{L}$ exists, we say that ϕ is σ-*provable* from T by means of $\mathfrak{L}$, notation $T \vdash_\sigma^{\mathfrak{L}} \phi$.

TRUTH. Let $\mathcal{C}$ be a class of σ-algebras and $\phi \in F_\sigma^{\mathfrak{L}}$. Then ϕ is said to be *true* in $\mathcal{C}$, notation $\mathcal{C} \models_\sigma^{\mathfrak{L}} \phi$, if ϕ holds in all σ-algebras $\mathfrak{A} \in \mathcal{C}$. Let $Alg(\sigma, T)$ be the class of all σ-algebras satisfying T.

SOUNDNESS THEOREM: *If* $\mathfrak{L}$ *is sound then* $T \vdash_\sigma^{\mathfrak{L}} \phi$ *implies* $Alg(\sigma, T) \models_\sigma^{\mathfrak{L}} \phi$.
PROOF: Straightforward with induction. □

If no confusion is likely to result, the sub- and superscripts of $\models$ and $\vdash$ may be dropped without further warning.

1.8. Module logic. The set $\mathfrak{M}$ of modules is defined inductively as follows:

- If σ is a signature and T a theory over σ, then $(\sigma, T) \in \mathfrak{M}$,
- If M and $N \in \mathfrak{M}$ then $M + N \in \mathfrak{M}$,
- If σ is a signature and $M \in \mathfrak{M}$ then $\sigma \square M \in \mathfrak{M}$,
- If $M \in \mathfrak{M}$ then $H(M) \in \mathfrak{M}$,
- If $M \in \mathfrak{M}$ then $S(M) \in \mathfrak{M}$.

Here $+$ is the composition operator, allowing to organise specifications in a modular way, and $\square$ is the export operator, restricting the visible signature of a module, thereby hiding auxiliary items. These operators occur in some form or other frequently in the literature on software engineering. Our notation is taken from BERGSTRA, HEERING & KLINT [9] in which also additional references can be found. The homomorphism operator H and the subalgebra operator S are, as far as we know, new in the context of algebraic specifications. Of course they are well known in model theory, see for instance MONK [33].

The *visible signature* $\Sigma(M)$ of a module M is defined inductively by:

- $\Sigma(\sigma, T) = \sigma$,
- $\Sigma(M + N) = \Sigma(M) \cup \Sigma(N)$,
- $\Sigma(\sigma \square M) = \sigma \cap \Sigma(M)$,
- $\Sigma(H(M)) = \Sigma(M)$,
- $\Sigma(S(M)) = \Sigma(M)$.

TRUTH. The class $Alg(M)$ of models of a module M is defined inductively by:
- $\mathfrak{A}$ is a model of (σ,T) if it is a σ-algebra, satisfying T;
- $\mathfrak{A}$ is a model of $M+N$ if it is a $\Sigma(M+N)$-algebra, such that $\Sigma(M)\square\mathfrak{A}$ is a model of M and $\Sigma(N)\square\mathfrak{A}$ is a model of N;
- $\mathfrak{A}$ is a model of $\sigma\square M$ if it is the restriction of a model $\mathfrak{B}$ of M to the signature σ;
- $\mathfrak{A}$ is a model of $H(M)$ if it is a homomorphic image of a model $\mathfrak{B}$ of M;
- $\mathfrak{A}$ is a model of $S(M)$ if it is a subalgebra of a model $\mathfrak{B}$ of M.

Note that $Alg(M)$ is a generalisation of $Alg(\sigma,T)$ as defined earlier. All the elements of $Alg(M)$ are $\Sigma(M)$-algebras. A $\Sigma(M)$-algebra $\mathfrak{A}\in Alg(M)$ is said to *satisfy* M. A formula $\phi\in F^{\mathfrak{L}}_{\Sigma(M)}$ is *satisfied* by a module M, notation $M\models^{\mathfrak{L}}\phi$, if $Alg(M)\models^{\mathfrak{L}}_{\Sigma(M)}\phi$, thus if ϕ holds in all $\Sigma(M)$-algebras satisfying M.

DERIVABILITY. A *proof* of a formula $\phi\in F^{\mathfrak{L}}_{\Sigma(M)}$ from a module M using the logic $\mathfrak{L}$, is a well-founded, upwardly branching tree of which the nodes are labelled by assertions $N\vdash\psi$, such that
- the root is labelled by $M\vdash\phi$
- and if $N\vdash\psi$ is the label of a node q and H is the set of labels of the nodes directly above q then $\dfrac{H}{N\vdash\psi}$ is one of the inference rules of table 1.

$(\sigma,T)\vdash\phi$	if $\phi\in T$
$\dfrac{M\vdash\phi_j\ \ (j\in J)}{M\vdash\phi}$	whenever $\dfrac{\phi_j\ \ (j\in J)}{\phi}\in I^{\mathfrak{L}}_{\Sigma(M)}$
$\dfrac{M\vdash\phi}{M+N\vdash\phi}$	$\dfrac{N\vdash\phi}{M+N\vdash\phi}$
$\dfrac{M\vdash\phi}{\sigma\square M\vdash\phi}$	if $\phi\in F^{\mathfrak{L}}_{\sigma}$
$\dfrac{M\vdash\phi}{H(M)\vdash\phi}$	if ϕ is *positive*
$\dfrac{M\vdash\phi}{S(M)\vdash\phi}$	if ϕ is *universal*

TABLE 1.

Here *positive* and *universal* are syntactic criteria, to be defined for each logic $\mathfrak{L}$ separately, ensuring that a formula is preserved under homomorphisms and subalgebras respectively. We write $N\vdash\psi$ for $\dfrac{\varnothing}{N\vdash\psi}$, and omit braces in the conditions of inference rules. If a proof of ϕ from M using $\mathfrak{L}$ exists, we say that ϕ is *provable* from M by means of $\mathfrak{L}$, notation $M\vdash^{\mathfrak{L}}\phi$.

LEMMA: *If* $M\vdash^{\mathfrak{L}}\phi$ *then* $\phi\in F^{\mathfrak{L}}_{\Sigma(M)}$.
PROOF: With induction. The only nontrivial cases are the rules for $+$ and $\square$. These follow from $F^{\mathfrak{L}}_{\sigma}\subseteq F^{\mathfrak{L}}_{\sigma\cup\rho}$ and $F^{\mathfrak{L}}_{\sigma}\cap F^{\mathfrak{L}}_{\rho}\subseteq F^{\mathfrak{L}}_{\sigma\cap\rho}$ respectively. □

SOUNDNESS THEOREM: *If* $\mathfrak{L}$ *is sound then* $M\vdash^{\mathfrak{L}}\phi$ *implies* $M\models^{\mathfrak{L}}\phi$.
PROOF: With induction. Again the only nontrivial cases are the rules for $+$ and $\square$. These follow since for all ρ-algebras $\mathfrak{A}$ and $\phi\in F^{\mathfrak{L}}_{\sigma\cap\rho}$: $\sigma\square\mathfrak{A}\models\phi\Rightarrow\mathfrak{A}\models\phi$ and $\sigma\square\mathfrak{A}\models\phi\Leftarrow\mathfrak{A}\models\phi$

respectively. □

§2 Process Algebra

This is not an introductory paper on process algebra. We only give a listing of the process modules used in the rest of the paper. For an introduction to the ACP formalism we refer the reader to [11-14].

2.1. ACP_τ. In this paper a central role will be played by the module ACP_τ, the Algebra of Communicating Processes with abstraction. ACP_τ has two parameters. The first parameter is a finite set A of atomic actions. For each atomic action $a \in A$ there is a constant a in the language, representing the process, starting with an a-step and terminating after some time. Furthermore we have a special constant δ, denoting deadlock, the acknowledgement of a process that it cannot do anything anymore. We write $A_\delta = A \cup \{\delta\}$. The second parameter of ACP_τ is a binary communication function $\gamma: A_\delta \times A_\delta \to A_\delta$, which is commutative, associative and has δ as zero element:

$$\gamma(a,b) = \gamma(b,a) \quad \gamma(a,\gamma(b,c)) = \gamma(\gamma(a,b),c) \quad \gamma(a,\delta) = \delta$$

If $\gamma(a,b) = c \neq \delta$ this means that actions a and b can synchronise. The synchronous performance of a and b is then regarded as a performance of the communication action c. Formally we should add the parameters to the name of a module: $ACP_\tau(A,\gamma)$. However, in order to keep notation simple, we will always omit the parameters if this can be done without causing confusion.

In table 2 we give the signature of module ACP_τ.

Σ (ACP_τ):	$\mathbb{S}$ (sort):	P		the set of processes
	$\mathbb{F}$ (functions):	$+$:	$P \times P \to P$	alternative composition (sum)
		$\cdot$:	$P \times P \to P$	sequential composition (product)
		$\|$:	$P \times P \to P$	parallel composition (merge)
		$\lfloor$:	$P \times P \to P$	left-merge
		$\|$:	$P \times P \to P$	communication-merge
		∂_H:	$P \to P$	encapsulation, for any $H \subseteq A$
		τ_I:	$P \to P$	abstraction, for any $I \subseteq A$
		a	$\in P$	for any atomic action $a \in A$
		δ	$\in P$	deadlock
		τ	$\in P$	silent action

Table 2.

Table 3 contains the theory of the module ACP_τ. In this paper we present ACP_τ as a monolithic module. In [13, 14] however, it has been shown that ACP_τ can be viewed as the sum of a large number of sub-modules which are interesting in their own right. The module consisting of axioms A1-5 only is called BPA (from Basic Process Algebra). If we add axioms A6-7 we obtain BPA_δ, and BPA_δ plus axioms T1-3 gives $BPA_{\tau\delta}$. The module ACP consists of the axioms A1-7, CF, CM1-9 and D1-4, i.e. the left column of table 3. All axioms in table 3 are in fact axiom schemes in a, b, H and I. Here a and b range over A_δ (unless further restrictions are made in the table) and $H, I \subseteq A$. In a product $x \cdot y$ we will often omit the $\cdot$. We take $\cdot$ to be more binding than other operations and $+$ to be less binding than other operations. In case we are dealing with an associative operator, we also leave out parentheses.

ACP_τ				
	$x+y = y+x$	A1	$x\tau = x$	T1
	$x+(y+z) = (x+y)+z$	A2	$\tau x+x = \tau x$	T2
	$x+x = x$	A3	$a(\tau x+y) = a(\tau x+y)+ax$	T3
	$(x+y)z = xz+yz$	A4		
	$(xy)z = x(yz)$	A5		
	$x+\delta = x$	A6		
	$\delta x = \delta$	A7		
	$a\|b = \gamma(a,b)$	CF		
	$x\|\|y = x\lfloor\lfloor y+y\lfloor\lfloor x+x\|y$	CM1		
	$a\lfloor\lfloor x = ax$	CM2	$\tau\lfloor\lfloor x = \tau x$	TM1
	$(ax)\lfloor\lfloor y = a(x\|\|y)$	CM3	$(\tau x)\lfloor\lfloor y = \tau(x\|\|y)$	TM2
	$(x+y)\lfloor\lfloor z = x\lfloor\lfloor z+y\lfloor\lfloor z$	CM4	$\tau\|x = \delta$	TC1
	$(ax)\|b = (a\|b)x$	CM5	$x\|\tau = \delta$	TC2
	$a\|(bx) = (a\|b)x$	CM6	$(\tau x)\|y = x\|y$	TC3
	$(ax)\|(by) = (a\|b)(x\|\|y)$	CM7	$x\|(\tau y) = x\|y$	TC4
	$(x+y)\|z = x\|z+y\|z$	CM8		
	$x\|(y+z) = x\|y+x\|z$	CM9		
			$\partial_H(\tau) = \tau$	DT
			$\tau_I(\tau) = \tau$	TI1
	$\partial_H(a) = a$ if $a\notin H$	D1	$\tau_I(a) = a$ if $a\notin I$	TI2
	$\partial_H(a) = \delta$ if $a\in H$	D2	$\tau_I(a) = \tau$ if $a\in I$	TI3
	$\partial_H(x+y) = \partial_H(x)+\partial_H(y)$	D3	$\tau_I(x+y) = \tau_I(x)+\tau_I(y)$	TI4
	$\partial_H(xy) = \partial_H(x)\cdot\partial_H(y)$	D4	$\tau_I(xy) = \tau_I(x)\cdot\tau_I(y)$	TI5

TABLE 3.

2.1.1. Note. Let $n>0$. Let $D = \{d_1,\ldots,d_n\}$ be a finite set. Let $x_{d_1},\ldots,x_{d_n}$ be processes. We will use the notation $\sum_{d\in D} x_d$ for the sum $x_{d_1} + \cdots + x_{d_n}$. $\sum_{d\in\emptyset} x_d = \delta$ by definition.

2.1.2. Summand inclusion. In process verifications the summand inclusion predicate $\subseteq$ turns out to be a useful notation. It is defined by: $x\subseteq y \Leftrightarrow x+y=y$. From the ACP_τ-axioms A1, A2 and A3 respectively it follows that $\subseteq$ is antisymmetrical, transitive and reflexive, and hence a partial order.

The following proposition will play an important role in sections 4 and 5.

2.1.3. PROPOSITION: $ACP_\tau \vdash \tau x\|y = \tau(x\|y)$.
PROOF: $\tau x\|y \supseteq \tau x\lfloor\lfloor y = \tau(x\|y) = \tau x\lfloor\lfloor y = \tau\tau x\lfloor\lfloor y = \tau(\tau x\|y) \supseteq \tau x\|y$. Now use the fact that $\subseteq$ is a partial order. □

2.1.4. Monotony. Most of the operators of ACP_τ are monotonous with respect to the summand inclusion ordering. Using essentially the distributivity of the operators over $+$, one can show that if $x\subseteq y$, ACP_τ proves:
- $x+z\subseteq y+z$,
- $x\cdot z\subseteq y\cdot z$,
- $x\lfloor\lfloor z\subseteq y\lfloor\lfloor z$,
- $x|z\subseteq y|z$,

- $\partial_H(x) \subseteq \partial_H(y)$,
- $\tau_I(x) \subseteq \tau_I(y)$.

Due to branching time, in general $z \cdot x \not\subseteq z \cdot y$, $x \| z \not\subseteq y \| z$ and $z \between x \not\subseteq z \between y$. However, we do have monotony of the merge for the case were x is of the form $\tau x'$. If $\tau x' \subseteq y$, then $\mathrm{ACP}_\tau \vdash \tau x' \| z \subseteq y \| z$:

$$\tau x' \| z \overset{2.1.3}{=} \tau(x' \| z) = \tau x' \between z \subseteq y \between z \subseteq y \| z.$$

2.2. Standard Concurrency. Often we add to the module ACP_τ the following module SC of Standard Concurrency ($a \in A_\delta$), which is parametrised by A. A proof that these axioms hold for all closed recursion-free terms can be found in [12].

SC	$(x \between y) \between z = x \between (y \| z)$	SC1
	$(x \mid ay) \between z = x \mid (ay \between z)$	SC2
	$x \mid y = y \mid x$	SC3
	$x \| y = y \| x$	SC4
	$x \mid (y \mid z) = (x \mid y) \mid z$	SC5
	$x \| (y \| z) = (x \| y) \| z$	SC6

TABLE 4.

2.3. Renamings. Let $A_{\tau\delta} = A_\delta \cup \{\tau\}$. For every function $f: A_{\tau\delta} \to A_{\tau\delta}$ with the property that $f(\delta) = \delta$ and $f(\tau) = \tau$, we define an operator $\rho_f: P \to P$. Axioms for ρ_f are given in table 5 (Here $a \in A_{\tau\delta}$ and *id* is the identity). Module RN is parametrised by A.

RN	$\rho_f(a) = f(a)$	RN1
	$\rho_f(x+y) = \rho_f(x) + \rho_f(y)$	RN2
	$\rho_f(xy) = \rho_f(x) \cdot \rho_f(y)$	RN3
	$\rho_{id}(x) = x$	RN4
	$\rho_f \circ \rho_g(x) = \rho_{f \circ g}(x)$	RN5

TABLE 5.

For $t \in A_{\tau\delta}$ and $H \subseteq A$ we define mappings $r_{t,H}: A_{\tau\delta} \to A_{\tau\delta}$ as follows:

$$r_{t,H}(a) = \begin{cases} t & \text{if } a \in H \\ a & \text{otherwise} \end{cases}$$

In the following we will implicitly identify the operators ∂_H and $\rho_{r_{\delta,H}}$, and also the operators τ_I and $\rho_{r_{\tau,I}}$: encapsulation is just renaming of actions into δ, and abstraction is renaming of actions into the silent step τ.

2.4. Chaining operators. A basic situation we will encounter is one in which processes input and output values in a domain D. Often we want to 'chain' two processes in such a way that the output of the first one becomes the input of the second. In order to describe this, we define *chaining* operators $\ggg$ and $\gg$. In the process $x \ggg y$ the output of process x serves as input of process y. Operator $\gg$ is identical to operator $\ggg$, but hides in addition the communications that take place at the internal communication port. The reason for introducing two operators is a technical one: the operator $\gg$ (in which we are interested most) often leads to *unguarded recursion* (cf. sections 2.8.1 and 2.12.1). We will define the chaining operators in terms of the

operators of $ACP_\tau + RN$. In this way we obtain a simple, finite axiomatisation of the operators. The operator $\gg$ occurs (in a different notation) already in HOARE [24] and MILNER [31]. In the context of ACP the operators $\ggg$ and $\gg$ were introduced in VAANDRAGER [40].

Let for $d \in D$, $\downarrow d$ be the action of reading d, and $\uparrow d$ be the action of sending d. Furthermore let $ch(D)$ be the following set

$$ch(D) = \{\uparrow d, \downarrow d, s(d), r(d), c(d) \mid d \in D\}$$

Here $r(d)$, $s(d)$ and $c(d)$ $(d \in D)$ are auxiliary actions which play a role in the definition of the chaining operators. The module for the chaining operators is parametrised by an action alphabet A satisfying $ch(D) \subseteq A$. The module should occur in a context with a module $ACP_\tau(A, \gamma)$ where $range(\gamma) \cap \{\downarrow d, \uparrow d, s(d), r(d) \mid d \in D\} = \varnothing$ and communication on $ch(D)$ is defined by

$$\gamma(s(d), r(d)) = c(d)$$

(all other communications give δ). The renaming functions $\uparrow s$ and $\downarrow r$ are defined by

$$\uparrow s(\uparrow d) = s(d) \quad \text{and} \quad \downarrow r(\downarrow d) = r(d) \quad (d \in D)$$

and $\uparrow s(a) = \downarrow r(a) = a$ for every other $a \in A_{\tau\delta}$. Now the 'concrete' chaining of processes x and y, notation $x \ggg y$, is defined by means of the axiom ($H = \{s(d), r(d) \mid d \in D\}$):

$$\boxed{x \ggg y = \partial_H(\rho_{\uparrow s}(x) \| \rho_{\downarrow r}(y)) \quad \text{CH1}}$$

The 'abstract' chaining of processes x and y, notation $x \gg y$, is defined by means of the axiom ($I = \{c(d) \mid d \in D\}$):

$$\boxed{x \gg y = \tau_I(x \ggg y) \quad \text{CH2}}$$

The module CH^+ consists of axioms CH1 and CH2, and is parametrised by A. The '+' in CH^+ refers to the auxiliary actions in the module, which will be removed in section 3.

2.5. Recursion. A *recursive specification* E is a set of equations $\{x = t_x \mid x \in V_E\}$ with V_E a set of variables and t_x a process expression for $x \in V_E$. Only the variables of V_E may appear in t_x. A solution of E is an interpretation of the variables of V_E as processes (in a certain domain), such that the equations of E are satisfied.

Recursive specifications are used to define (or specify) infinite processes. For each recursive specification E and $x \in V_E$, the module REC introduces a constant $\langle x \mid E \rangle$, denoting the x-component of a solution of E.

In most applications the variables $X \in V_E$ in a recursive specification E will be chosen fresh, so that there is no need to repeat E in each occurrence of $\langle X \mid E \rangle$. Therefore the convention will be adopted that once a recursive specification has been declared, $\langle X \mid E \rangle$ can be abbreviated by X. If this is done, X is called a *formal variable*. Formal variables are denoted by capital letters. So after the declaration $X = aX$, a statement $X = aaX$ should be interpreted as an abbreviation of $\langle X \mid X = aX \rangle = aa\langle X \mid X = aX \rangle$.

Let $E = \{x = t_x \mid x \in V_E\}$ be a recursive specification, and t a process expression. Then $\langle t \mid E \rangle$ denotes the term t in which each free occurrence of $x \in V_E$ is replaced by $\langle x \mid E \rangle$. In a recursive language we have for each E as above and $x \in V_E$ an axiom

$$\boxed{\langle x \mid E \rangle = \langle t_x \mid E \rangle \quad \text{REC}}$$

If the above convention is used, these formulas seem to be just the equations of E. The module REC is parametrised by the signature in which the recursive equations are written. In the presence of module REC each system of recursion equations over this signature has a solution.

2.6. Projection. The operator $\pi_n : P \to P$ $(n \in \mathbb{N})$ *stops* processes after they have performed n atomic actions, with the understanding that τ-steps are transparent. The axioms for π_n are given in table 6. Module PR is parametrised by A.

PR		
	$\pi_n(\tau) = \tau$	PR1
	$\pi_0(ax) = \delta$	PR2
	$\pi_{n+1}(ax) = a \cdot \pi_n(x)$	PR3
	$\pi_n(\tau x) = \tau \cdot \pi_n(x)$	PR4
	$\pi_n(x+y) = \pi_n(x) + \pi_n(y)$	PR5

TABLE 6.

In this paper, as in other papers on process algebra, we have an infinite collection of unary projection operators. Another option, which we do not pursue here, but which might be more fruitful if one is interested in finitary process algebra proofs, is to introduce a single binary projection operator $\mathbb{F} : \pi : \mathbb{N} \times P \to P$.

2.7. Boundedness. The predicate $B_n \subseteq P$ $(n \in \mathbb{N})$ states that the nondeterminism displayed by a process before its n^{th} atomic step is bounded. If for all $n \in \mathbb{N}$: $B_n(x)$, we say x is bounded. Axioms for B_n are in table 7 $(a \in A_\delta)$. Module B is parametrised by A.

B		
	$B_0(x)$	B1
	$B_n(\tau)$	B2
	$\dfrac{B_n(x)}{B_n(\tau x)}$	B3
	$\dfrac{B_n(x)}{B_{n+1}(ax)}$	B4
	$\dfrac{B_n(x),\ B_n(y)}{B_n(x+y)}$	B5

TABLE 7.

Boundedness predicates were introduced in [22].

2.8. Approximation Induction Principle. AIP^- is a proof rule which is vital if we want to prove things about infinite processes. The rule expresses the idea that if two processes are equal to any depth, and one of them is bounded then they are equal.

$$(\mathrm{AIP}^-) \quad \frac{\forall n \in \mathbb{N} \quad \pi_n(x) = \pi_n(y),\ B_n(x)}{x = y}$$

The "$-$" in AIP^-, distinguishes the rule from a variant without predicates B_n.

2.8.1. DEFINITION. Let t be an open ACP_τ-term without abstraction operators. An occurrence of a variable X in t is *guarded* if t has a subterm of the form $a \cdot M$, with $a \in A_\delta$, and this X occurs in M. Otherwise, the occurrence is *unguarded.*

Let $E = \{x = t_x \mid x \in V_E\}$ be a recursive specification in which all t_x are ACP_τ-terms without abstraction operators. For $X, Y \in V_E$ we define:

$$X \xrightarrow{u} Y \Leftrightarrow Y \text{ occurs unguarded in } t_X$$

We call E *guarded* if relation $\xrightarrow{u}$ is well-founded (i.e. there is no infinite sequence $X \xrightarrow{u} Y \xrightarrow{u} Z \xrightarrow{u} \cdots$).

2.8.2. THEOREM *(Recursive Specification Principle* (RSP)*)*:
$\mathrm{ACP}_\tau + \mathrm{REC} + \mathrm{PR} + \mathrm{B} + \mathrm{AIP}^- \vdash$

$$\text{(RSP)} \quad \frac{E}{x = \langle x|E\rangle} \; E \text{ guarded}$$

In plain English the RSP rule says that every guarded recursive specification has at most one solution.

Example. Let $E = \{X=(a+b)\cdot X\}$ and $F = \{Y=a\cdot(a+b)\cdot Y+b\cdot Y\}$ be two recursive specifications. Since

$$\langle X|E\rangle = (a+b)\cdot\langle X|E\rangle = a\cdot\langle X|E\rangle+b\cdot\langle X|E\rangle =$$
$$= a\cdot(a+b)\cdot\langle X|E\rangle+b\cdot\langle X|E\rangle,$$

the constant $\langle X|E\rangle$ satisfies the equation of F. Because the specification F is guarded, RSP now gives that $\langle X|E\rangle = \langle Y|F\rangle$.

2.9. Koomen's Fair Abstraction Rule (KFAR). In the verification of communication protocols we often use the following rule, called Koomen's Fair Abstraction Rule ($I \subseteq A$). Module KFAR is parametrised by A.

$$\text{(KFAR)} \quad \frac{x = ix + y \quad (i \in I)}{\tau_I(x) = \tau\cdot\tau_I(y)}$$

Fair abstraction here means that $\tau_I(x)$ will eventually exit the hidden i-cycle. Below we will formulate a generalisation of KFAR, the Cluster Fair Abstraction Rule (CFAR), which can be derived from KFAR.

2.9.1. DEFINITION: Let $E = \{X=t_X \mid X \in V_E\}$ be a recursive specification, and let $I \subseteq A$. A subset C of V_E is called a *cluster (of I) in E* iff for all $X \in C$:

$$t_X = \sum_{k=1}^{m} i_k\cdot X_k + \sum_{l=1}^{n} Y_l$$

(For $m \geqslant 0$, $i_1, \ldots, i_m \in I \cup \{\tau\}$, $X_1, \ldots, X_m \in C$, $n \geqslant 0$ and $Y_1, \ldots, Y_n \in V_E - C$). Variables $X \in C$ are called *cluster variables*. For $X \in C$ and $Y \in V_E$ we say that

$$X \rightsquigarrow Y \Leftrightarrow Y \text{ occurs in } t_X$$

We define

$$e(C) = \{Y \in V_E - C \mid \exists X \in C : X \rightsquigarrow Y\}$$

Variables in $e(C)$ are called *exits*. $\rightsquigarrow^*$ is the transitive and reflexive closure of $\rightsquigarrow$. Cluster C is *conservative* iff every exit can be reached from every cluster variable via a path in the cluster:

$$\forall X \in C\, \forall Y \in e(C) : X \rightsquigarrow^* Y$$

Example. In the transition diagram of figure 1, the sets {1,2,3}, {4,5,6,7}, {8} and {1,2,3,4,5,6,7,8} are examples of conservative clusters. Cluster {1,2,3,4,5,6,7} is not conservative since exit Z cannot be reached from cluster variables 4, 5, 6 and 7.

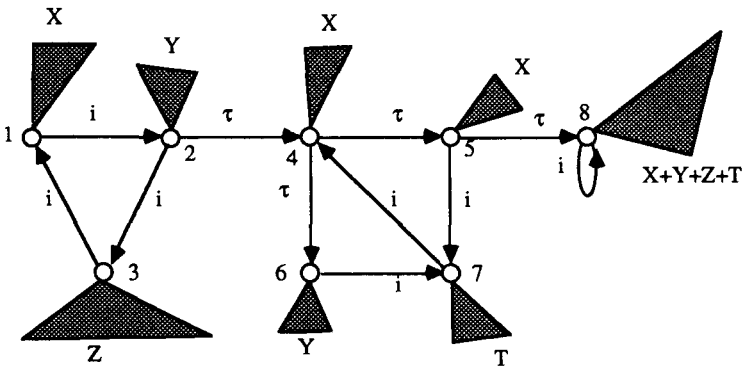

FIGURE 1.

2.9.2. DEFINITION: The *Cluster Fair Abstraction Rule (CFAR)* reads as follows:

(CFAR)	Let E be a guarded recursive specification; let $I \subseteq A$ with $\|I\| \geqslant 2$; let C be a finite conservative cluster of I in E; and let $X, X' \in C$ with $X \rightsquigarrow X'$. Then: $\tau_I(X) = \tau \cdot \sum_{Y \in e(C)} \tau_I(Y)$

2.9.3. THEOREM: $ACP_\tau + RN + REC + RSP + KFAR \vdash CFAR$.
PROOF: See [39]. □

2.10. Alphabets. Intuitively the alphabet of a process is the set of atomic actions which it can perform. This idea is formalised in [4], where an operator $\alpha: P \to 2^A$ is introduced, with axioms such as:

$$\alpha(\delta) = \varnothing$$
$$\alpha(ax) = \{a\} \cup \alpha(x)$$
$$\alpha(x+y) = \alpha(x) \cup \alpha(y)$$

In this approach the question arises what axioms should be adopted for the set-operators $\cup$, $\cap$, etc. One option, which is implicitly adopted in previous papers on process algebra, is to take the equalities which are true in set theory. This collection is unstructured and too large for our purposes. Therefore we propose a different, more algebraic solution. We view the alphabet of a process as a *process*; the alphabet operator α goes from sort P to sort P. Process $\alpha(x)$ is the alternative composition of the actions which can be performed by x. In this way we represent a set of actions by a process. A set B of actions is represented by the process expression $\boldsymbol{B} =_{def} \sum_{b \in B} b$. So the empty set is represented by δ, a singleton-set $\{a\}$ by the expression a, and a set $\{a,b\}$ by expression $a+b$. Set union corresponds to alternative composition. The process algebra axioms A1-3 and A6 correspond to similar axioms for the set union operator. The notation $\subseteq$ for summand inclusion between processes (section 2.1.2), fits with the notation for the subset predicate on sets.

The following axioms in table 8 define the alphabet of finite processes ($a \in A$). Module AB is

parametrised by A.

AB	$\alpha(\delta) = \delta$	AB1
	$\alpha(ax) = a+\alpha(x)$	AB2
	$\alpha(x+y) = \alpha(x)+\alpha(y)$	AB3
	$\alpha(\tau) = \delta$	AB4
	$\alpha(\tau x) = \alpha(x)$	AB5

TABLE 8.

In order to compute the alphabet of infinite processes, we introduce an additional module AA which is parametrised by A.

AA	$\alpha(x) \subseteq A$	AA1
	$\alpha(x \Vert y) = \alpha(x)+\alpha(y)+\alpha(x) \vert \alpha(y)$	AA2
	$\alpha \circ \rho_f(x) \subseteq \rho_f \circ \partial_H \circ \alpha(x)$ (where $H = \{a \in A \mid f(a) = \tau\}$)	AA3
	$\dfrac{\forall n \in \mathbb{N}\ \ \alpha(\pi_n(x)) \subseteq y}{\alpha(x) \subseteq y}$	AA4

TABLE 9.

It is not hard to see that the axioms of AA hold for all closed recursion-free terms.

Example. (from [4]). Let $p = \langle X \mid \{X = aX\} \rangle$, and define $q = \tau_{\{a\}}(p)$, $r = q \cdot b$ (with $b \neq a$). What is the alphabet of r? We derive:

$$\alpha(r) = \alpha(qb) = \alpha(\tau_{\{a\}}(p) \cdot b) = \alpha(\tau_{\{a\}}(p) \cdot \tau_{\{a\}}(b)) =$$
$$= \alpha(\tau_{\{a\}}(pb)) \overset{AA3}{\subseteq} \tau_{\{a\}} \circ \partial_{\{a\}} \circ \alpha(pb) \overset{RN5}{=} \partial_{\{a\}} \circ \alpha(pb).$$

Since

$$\alpha(pb) = \alpha(apb) \overset{AB2}{=} a + \alpha(pb),$$

we have that $a \subseteq \alpha(pb)$. On the other hand we derive for $n \in \mathbb{N}$:

$$\alpha(\pi_n(pb)) = \alpha(a^n \cdot \delta) \subseteq a$$

and therefore, by application of axiom AA4, $\alpha(pb) \subseteq a$. Consequently $\alpha(pb) = a$ and

$$\alpha(r) = \partial_{\{a\}} \circ \alpha(pb) = \partial_{\{a\}}(a) = \delta.$$

Information about alphabets must be available if we want to apply the following rules. These rules, which are a generalisation of the conditional axioms of [4], occur in a slightly different form also in [40]. Rules like these are an important tool in system verifications based on process algebra. Module RR is parametrised by A and γ. Observe that axioms AA1 and RR1 together imply axiom RN4 of table 5. Axiom RR2, which describes the interaction between renaming and parallel composition, looks complicated, but that is only because it is so general. The axioms RR are derivable for closed recursion-free terms.

RR	$\frac{\alpha(x)\subseteq B}{\rho_f(x)=x}\forall b\in B: f(b)=b$	RR1
	$\frac{\alpha(x)\subseteq B,\ \alpha(y)\subseteq C}{\rho_f(x\|y)=\rho_f(x\|\rho_f(y))}\forall c\in C: f(c)=f^2(c)\wedge(\forall b\in B: f\circ\gamma(b,c)=f\circ\gamma(b,f(c)))$	RR2

TABLE 10.

2.11. ACP$_\tau^\sharp$. The combination of all modules presented thus far, except for KFAR, will be called ACP$_\tau^\sharp$ (the system ACP$_\tau^\sharp$ as presented here slightly differs from a system with the same name occurring in [13]). The module is defined by:

$$\text{ACP}_\tau^\sharp = \text{ACP}_\tau + \text{SC} + \text{RN} + \text{CH}^+ + \text{REC} + \text{PR} + \text{B} + \text{AIP}^- + \text{AB} + \text{AA} + \text{RR}$$

Bisimulation semantics, as described in for instance [5], gives a model for the module ACP$_\tau^\sharp$ + KFAR. Work of BERGSTRA, KLOP & OLDEROG [15] showed that in a large number of interesting models KFAR is not valid. Therefore we have chosen not to include KFAR in the 'standard' module ACP$_\tau^\sharp$.

2.12. Generalised Recursive Specification Principle. For many applications the RSP is too restrictive. Therefore we will present below a more general version of this rule, called RSP$^+$.

2.12.1. DEFINITION: Let $\mathcal{P}$ be the set of closed expressions in the signature of ACP$_\tau^\sharp$. A process expression $p\in\mathcal{P}$ is called *guardedly specifiable* if there exists a guarded recursive specification F with $Y\in V_F$ such that

$$\text{ACP}_\tau^\sharp \vdash p=\langle Y|F\rangle.$$

We have the following theorem:

2.12.2. THEOREM *(Generalised Recursive Specification Principle* (RSP$^+$)*)*: ACP$_\tau^\sharp \vdash$

(RSP$^+$)	$\frac{E}{x=\langle x\|E\rangle}$	$\langle x\|E\rangle$ guardedly specifiable

2.12.3. Remarks. In the definition of the notion 'guardedly specifiable', it is essential that the identity $p=\langle Y|F\rangle$ is *provable*. If we would only require that $p=\langle Y|F\rangle$, then the corresponding version of RSP$^+$ would not be provable from ACP$_\tau^\sharp$, since this rule would then not be valid in the action relation model of [22]. In this model we have the identity $\langle X|\{X=X\}\rangle=\delta$.[1] Hence $\langle X|\{X=X\}\rangle=\langle Y|\{Y=\delta\}\rangle=\delta$. Since the specification $\{Y=\delta\}$ is guarded, this would mean that expression $\langle X|\{X=X\}\rangle$ is guardedly specifiable. But then RSP$^+$ gives that for arbitrary x: $x=\langle X|\{X=X\}\rangle=\delta$. This is clearly false.

We conjecture that an expression p is guardedly specifiable iff it is provably bounded, i.e. for all $n\in\mathbb{N}$: ACP$_\tau^\sharp \vdash B_n(x)$.

1. Strictly speaking, this is not correct. In [22], a recursion construct $\langle X|E\rangle$ is viewed as a kind of variable which ranges over the X-components of the solutions of E. Since any process X satisfies $X=X$, the identity $\langle X|\{X=X\}\rangle=\delta$ does not hold under this interpretation. However, if one interprets the construct $\langle X|E\rangle$ as a constant in the model of [22], then the most natural choice is to relate to $\langle X|E\rangle$ the bisimulation equivalence class of the term $\langle X|E\rangle$. Under this interpretation $\langle X|\{X=X\}\rangle=\delta$.

§3 Applications of the module approach in process algebra

3.1. The auxiliary status of the left-merge.

3.1.1. Semantics. Sometimes it happens that our 'customers' complain that they do not succeed in proving the identity of two processes in $ACP_{\tau}^{\sharp}$, whose behaviour is considered 'intuitively the same'. Often, this is because there are many intuitions possible, and $ACP_{\tau}^{\sharp}$ happens not to represent the particular intuitions of these customers. Therefore we have defined some auxiliary modules that should bridge the gaps between intuitions.

In general a user of process algebra wants that his system proves $p = q$ (here p and q are closed process expressions in the signature of $ACP_{\tau}^{\sharp}$), whenever p and q have the same interesting properties. So it depends on what properties are interesting for a particular user, whether his system should be designed to prove the equality of p and q or not. For this reason the semantical branch of process algebra research generated a variety of process algebras in which different identification strategies were pursued. In *bisimulation semantics* we find algebras that distinguish between any two processes that differ in the precise timing of internal choices; in *trace semantics* only processes are distinguished which can perform different sequences of actions; and, somewhere in between, the algebras of *failure semantics* identify processes if they have the same traces (can perform the same sequences of actions) and have the same deadlock behaviour in any context. A lot of these process algebras can be organised as homomorphic images of each other, as indicated in figure 2.

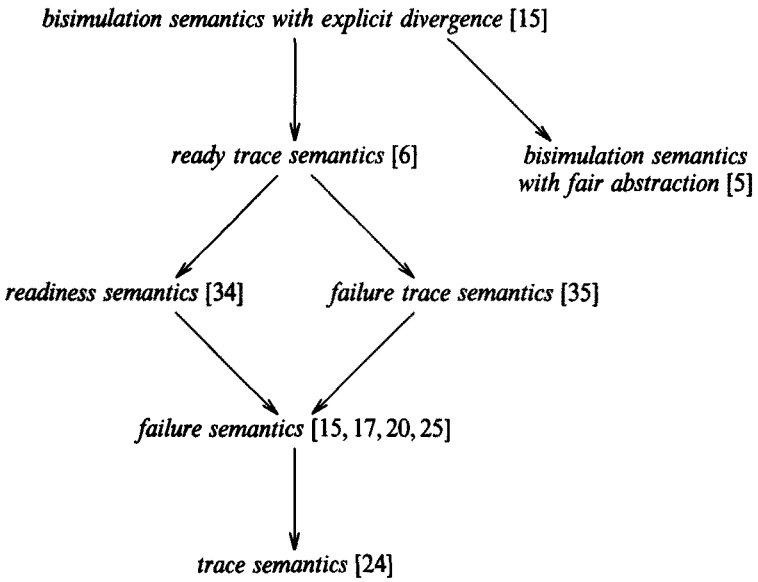

FIGURE 2. *The linear time - branching time spectrum*

If two process expressions p and q represent the same process in bisimulation semantics with explicit divergence, they have many properties in common; if they only represent the same process in trace semantics, this only guarantees that they share some of these properties; and, descending from bisimulation semantics with explicit divergence to trace semantics, less and less distinctions are made. Now a user should state exactly in which properties of processes he is interested. Suppose he is only interested in traces and deadlock behaviour, then we can tell him

that for his purposes failure semantics suffices. This means that if processes p and q are proven equal in failure semantics, this guarantees that they have the same relevant properties. If they are only identified in trace semantics (somewhere in the lattice below failure semantics) such a conclusion cannot be drawn, but if they are identified in a semantics finer than failure semantics (such as bisimulation semantics with explicit divergence), then they certainly have the same interesting properties, and probably some uninteresting ones as well. Hence a proof in bisimulation semantics with explicit divergence is just as good as one in failure semantics (or even better).

This is the reason that we do our proofs mostly in bisimulation semantics: the entire module $\mathrm{ACP}_\tau^\sharp$ is sound with respect to bisimulation semantics with explicit divergence. However, if two processes are different in bisimulation semantics, we will never succeed in proving them equal from $\mathrm{ACP}_\tau^\sharp$. In such a case we might add some axioms to the system, that represent the extra identifications made in a less discriminating semantics. If we find a proof from this enriched module, it can be used by anyone satisfied with the properties of this coarser semantics.

It is in the light of the above considerations that one should judge the appearance of the following module T4:

$$\text{T4} \quad \boxed{\tau(\tau x + y) = \tau x + y}$$

The law of this module does not hold in bisimulation semantics, but it does hold in all other semantics of figure 2. Thus any identity derived from $\mathrm{ACP}_\tau^\sharp + \text{T4}$ holds in ready trace semantics and hence also in the courser ones like failure and trace semantics, or so it seems ....

3.1.2. An inconsistency.

3.1.2.1. DEFINITION: Let M be a process module with $\Sigma(M) \supseteq \Sigma(\mathrm{BPA}_{\tau\delta})$. We call M *consistent* if for all closed expressions x and y in the signature of $\mathrm{BPA}_{\tau\delta}$ with

$$M \vdash x = y,$$

the sets of complete traces agree:

$$\mathit{trace}(x) = \mathit{trace}(y).$$

A *complete trace* is a finite sequence of actions, ending with a symbol $\surd$ or δ indicating successful resp. unsuccessful termination. A formal definition of the set $\mathit{trace}(x)$ is given in [15]. Here we only give some examples, which should make the notion sufficiently clear:

$$\mathit{trace}(abc + ad\delta + a(\tau bc + d)) = \{abc\surd, ad\delta, ad\surd\}$$

$$\mathit{trace}(\tau) = \{\surd\} \neq \{\delta, \surd\} = \mathit{trace}(\tau + \tau\delta)$$

A model $\mathcal{A}$ of M is *consistent* if for all closed expressions x and y in the signature of $\mathrm{BPA}_{\tau\delta}$ with

$$\mathcal{A} \models x = y,$$

the sets of complete traces agree. The module $\mathrm{ACP}_\tau^\sharp + \mathrm{KFAR}$ is consistent because bisimulation semantics with fair abstraction, as described in [5], gives a consistent model for this module. However, KFAR is not valid in any of the other semantics of figure 2.

3.1.2.2. PROPOSITION: $\mathrm{ACP}_\tau + \text{T4} \vdash \tau(ac + ca) + bc = \tau(\tau(ac + ca) + bc + c(\tau a + b))$.
PROOF:

$$\tau(\tau a + b) \between c = (\tau a + b) \between c = \tau(a \| c) + bc = \tau(ac + ca) + bc$$

$$\tau(\tau a + b) \between c = \tau((\tau a + b) \| c) = \tau(\tau(ac + ca) + bc + c(\tau a + b)) \quad \square$$

Proposition 3.1.2.2 shows that module $\mathrm{ACP}_\tau + \text{T4}$ is not consistent. This sudden inconsistency must be the result of a serious misunderstanding. And indeed, what's wrong is the use of ACP_τ

in the less discriminating models (say in failure semantics). It happens that, in a setting with τ, failure equivalence (or ready trace equivalence for that matter) is not a congruence for the left-merge $\lfloor\!\lfloor$, and this causes all the trouble.

3.1.3. Solution. In applications we do not use the operators $\lfloor\!\lfloor$ and $\mid$ directly. In specifications we use the merge operator $\|$, and $\lfloor\!\lfloor$ and $\mid$ are only auxiliary operators, needed to give a complete axiomatisation of the merge.

Let sacp_τ be the signature obtained from $\Sigma(\text{ACP}_\tau)$ by stripping the left-merge and communication-merge:

$$\text{sacp}_\tau = \Sigma(\text{ACP}_\tau) - \{\mathbb{F} : \lfloor\!\lfloor : P \times P \to P, \mathbb{F} : \mid : P \times P \to P\}$$

Failure equivalence as in [15], etc. are congruences for the operators of sacp_τ. However, the operators $\lfloor\!\lfloor$ and $\mid$ in ACP_τ are needed to axiomatise the $\|$-operator, and without them even the most elementary equations cannot be derived. Our solution to this problem is based on the following idea. Suppose we want to prove an equation $p = q$ in the signature sacp_τ that holds in ready trace semantics (and hence in failure semantics) but not in bisimulation semantics. Then we first prove an intermediate result from ACP_τ: one or more equations holding in bisimulation semantics (with explicit divergence) and in which no $\lfloor\!\lfloor$ and $\mid$ appear. This intermediate result is preserved after mapping the bisimulation model homomorphically on the ready trace or failure model, and can be combined consistently with the axiom T4. Thus the proof of $p = q$ can be completed. In our language of modules we can describe this as follows. The module

$$SACP_\tau = H(\text{sacp}_\tau \Box(\text{ACP}_\tau + \text{SC}))$$

does not contain the operators $\lfloor\!\lfloor$ and $\mid$ in its visible signature and since failure semantics can be obtained as a homomorphic image of bisimulation semantics, considering that $\text{ACP}_\tau + \text{SC}$ is sound w.r.t. bisimulation semantics and that the operators of sacp_τ carry over to failure semantics, we conclude that this module is sound w.r.t. failure semantics. Hence it can be combined consistently with T4, and SACP_τ is a suitable framework for proving statements in failure semantics.

We would like to stress that the use of the H-operator is essential here. The H-operator makes that from module SACP_τ only *positive* formulas are provable. The following example shows what goes wrong if we also allow non-positive formulas. From the proof of proposition 3.1.2.2 it follows that:

$$\text{sacp}_\tau \Box(\text{ACP}_\tau + \text{SC}) \vdash \frac{\tau(\tau x + y) = \tau x + y}{c(\tau a + b) \subseteq \tau(ac + ca) + bc}$$

Consequently we can prove an inconsistency if we add law T4:

$$\text{sacp}_\tau \Box(\text{ACP}_\tau + \text{SC}) + <\tau(\tau x + y) = \tau x + y> \vdash c(\tau a + b) \subseteq \tau(ac + ca) + bc$$

So although the formulas provable from module $\text{sacp}_\tau \Box(\text{ACP}_\tau + \text{SC})$ contain no left-merge, some of them (which are non-positive) cannot be combined consistently with the laws of ready trace semantics and failure semantics.

3.2. Definition of the chaining operator. ACP_τ is a universal specification formalism in the sense that in bisimulation semantics every finitely branching, effectively presented process can be specified in ACP_τ by a finite system of recursion equations (see [5, 14]). Still it often turns out that adding new operators to the theory facilitates specification and verification of concurrent systems. In general, adding new operators and laws can have far reaching consequences for the underlying mathematical theory. Often however, new operators are *definable* in terms of others operators and the axioms are *derivable* from the other axioms. In that case the new operators can be considered as notations which are useful, but do not complicate the underlying theory in any way. Examples of definable operators are the projection operators, the process creation operator of [8] and the state operators of [3].

Just like the left-merge and the communication-merge are needed in order to axiomatise the parallel composition operator, new atomic actions are often needed if we want to define a new operator in terms of more elementary operators. As an example we mention the actions $s(d)$ and $r(d)$ which we need in the definition of the chaining operators. These auxiliary atoms will never be used in process specifications. Unfortunately they have the unpleasant property that they occur in some important algebraic laws for the new operators. One of the properties of the chaining operators we use most is that they are associative under some very weak assumptions. In the model of bisimulation semantics, the following law is valid (here $H=\{s(d),r(d)|d\in D\}$):

$$\frac{\partial_H(x)=x, \partial_H(y)=y, \partial_H(z)=z}{(x\ggg y)\ggg z=x\ggg(y\ggg z)}\ \text{CC}$$

We do not have general associativity in the model. Counterexample:

$$(r(d)\ggg(s(d)+s(e)))\ggg r(e) = c(d)\cdot\delta$$

$$r(d)\ggg((s(d)+s(e))\ggg r(e)) = c(e)\cdot\delta$$

It would be much nicer if we somehow could 'hide' the auxiliary atoms, and, for the $\ggg$-operator, have associativity in general. In this section we will see how this can be accomplished by means of the module approach.

3.2.1. The associativity of the chaining operators. Although the rule CC holds in the model of bisimulation semantics, we have not been able to prove it algebraically from module ACP$_{\tau}^{\ddagger}$. However, we can prove algebraically a weaker version of rule CC if we make some additional assumptions about the alphabet. We assume that besides actions $ch(D)$, the alphabet A contains actions:

$$\overline{H} = \{\overline{s}(d),\overline{r}(d)|d\in D\} \text{ en } \underline{H} = \{\underline{s}(d),\underline{r}(d)|d\in D\}$$

One may think about these actions as special fresh atoms which are added to A only in order to prove the associativity of the chaining operators.[1] Let $H=\{r(d),s(d)|d\in D\}$ and let $\hat{H}=H\cup\overline{H}\cup\underline{H}$. We assume that actions from $\hat{H}$ do not synchronise with the other actions in the alphabet, and that $range(\gamma)\cap\hat{H}=\emptyset$. On $\hat{H}$ communication is given by ($d\in D$):

$$\gamma(\overline{s}(d),\overline{r}(d)) = \gamma(\overline{s}(d),r(d)) = \gamma(s(d),\overline{r}(d)) = \gamma(s(d),r(d)) =$$
$$= \gamma(\underline{s}(d),\underline{r}(d)) = \gamma(\underline{s}(d),r(d)) = \gamma(s(d),\underline{r}(d)) = c(d)$$

We define for $v,w\in\{\uparrow,\downarrow,s,r,\overline{s},\overline{r},\underline{s},\underline{r}\}$ the renaming function vw:

$$vw(a) = \begin{cases} w(d) & \text{if } a=v(d) \text{ for some } d\in D \\ a & \text{otherwise} \end{cases}$$

3.2.1.1. LEMMA: $SACP_{\tau} + RN + CH^{+} + AB + AA + RR \vdash$

$$\frac{\partial_{\hat{H}}(x)=x,\ \partial_{\hat{H}}(y)=y,\ \partial_{\hat{H}}(z)=z}{\partial_{\overline{H}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y))=x\ggg y=\partial_{\underline{H}}(\rho_{\uparrow\underline{s}}(x)\|\rho_{\downarrow\underline{r}}(y))}$$

PROOF: The proof of the first equality in this lemma has been spelled out in VAN GLABBEEK & VAANDRAGER [23], the full version of this paper. In this proof it is essential that

1. The *Fresh Atom Principle (FAP)* says that we can use new (or 'fresh') atomic actions in proofs. In [7], it is shown that FAP holds in bisimulation semantics. We have not included FAP in the theoretical framework of this paper. Therefore, if we need certain 'fresh' atoms in a proof, we have to assume that they were in the alphabet right from the beginning.

$$\gamma(\overline{s}(d), \overline{r}(d)) = \gamma(\overline{s}(d), r(d)) = \gamma(s(d), \overline{r}(d)) = \gamma(s(d), r(d)) = c(d)$$

The second equality then follows by symmetry. □

3.2.1.2. THEOREM: $SACP_\tau + RN + CH^+ + AB + AA + RR \vdash$

$$\frac{\partial_{\hat{H}}(x)=x,\ \partial_{\hat{H}}(y)=y,\ \partial_{\hat{H}}(z)=z}{x \ggg (y \ggg z)=(x \ggg y) \ggg z}$$

PROOF: This is essentially theorem 1.12.2 of [40]. A sketch of the proof is given in [23]. There $x \ggg (y \ggg z)$ is written as $\partial_{\overline{H}}(\rho_{\uparrow \overline{s}}(x) \| \rho_{\downarrow \overline{r}} \circ \partial_H(\rho_{\uparrow s}(y) \| \rho_{\downarrow r}(z)))$, using lemma 3.2.1.1. Now a crucial element in the proof is the observation that there is no communication possible between elements of $\overline{H}$ and $\underline{H}$. This is the reason that the sets $\overline{H}$ and $\underline{H}$ had to be introduced both. □

3.2.1.3. THEOREM: $SACP_\tau + RN + CH^+ + AB + AA + RR \vdash$

$$\frac{\partial_{\hat{H}}(x)=x,\ \partial_{\hat{H}}(y)=y,\ \partial_{\hat{H}}(z)=z}{x \gg (y \gg z) = (x \gg y) \gg z}$$

PROOF: See [23]. □

3.2.2. Removing auxiliary atoms. We will now apply the module approach to remove completely the auxiliary atoms which where used in the definition of the chaining operators and in the proofs of their associativity. Below we will employ the notation:

$$\sigma \Delta M \equiv (\Sigma(M) - \sigma) \square M.$$

Consider the module:

$$\mathrm{CH}^- = (\{\mathbf{F} : a \in P \mid a \in \hat{H}\} \cup \{\mathbf{F} : \rho_f : P \to P \mid f : A_{\pi\delta} \to A_{\pi\delta}\})$$
$$\Delta(\mathrm{SACP}_\tau + \mathrm{RN} + \mathrm{CH}^+ + \mathrm{AB} + \mathrm{AA} + \mathrm{RR}).$$

This module cannot be used to prove any formula containing atoms in $\hat{H}$. But unfortunately module CH^- still does not prove the general associativity of the chaining operators:

$$\mathrm{CH}^- \nvdash x \ggg (y \ggg z)=(x \ggg y) \ggg z$$

The reason is that the auxiliary atoms, although removed from the language, are still present in the models of module CH^-. Thus the counterexample $(r(d) \ggg (s(d)+s(e))) \ggg r(e)$ still works in the models. Let $A^- = A - \hat{H}$. We are interested in consistent models which only contain actions of A^-. The module $\mathrm{CH}^- + <\alpha(x) \subseteq A^->$ does not denote such models: all consistent models of CH^- contain the process A with $\alpha(A) = A \not\subseteq A^-$. Adding the law $\alpha(x) \subseteq A^-$ therefore throws away all consistent models. The right class of models can be denoted with the help of operator S. We consider the module

$$\mathrm{CH} = S(\mathrm{CH}^-) + <\alpha(x) \subseteq A^->.$$

Some models of module CH^- have consistent submodels which do not contain auxiliary atoms. In these models the law $\alpha(x) \subseteq A^-$ holds. Thus module CH has consistent models.

From theorems 3.2.1.2 and 3.2.1.3, together with axiom RR1, it follows that:

$$CH^- \vdash \frac{\alpha(x) \subseteq A^-,\ \alpha(y) \subseteq A^-,\ \alpha(z) \subseteq A^-}{(x \ggg y) \ggg z = x \ggg (y \ggg z)}$$

$$CH^- \vdash \frac{\alpha(x) \subseteq A^-,\ \alpha(y) \subseteq A^-,\ \alpha(z) \subseteq A^-}{(x \gg y) \gg z = x \gg (y \gg z)}$$

From this we can easily see that module CH proves the general associativity of the chaining operators:

$$\mathrm{CH} \vdash x \ggg (y \ggg z)=(x \ggg y) \ggg x \text{ and}$$

$$\mathrm{CH} \vdash x \gg (y \gg z) = (x \gg y) \gg x.$$

3.2.3. The following laws can be easily proven from module CH (here $d,e \in D$):

$$\uparrow d \cdot x \gg (\sum_{e \in D} \downarrow e \cdot y^e) = \tau \cdot (x \gg y^d) \qquad \text{L1}$$

$$\uparrow d \cdot x \gg \uparrow e \cdot y = \uparrow e \cdot (\uparrow d \cdot x \gg y) \qquad \text{L2}$$

$$(\sum_{d \in D} \downarrow d \cdot x^d) \gg (\sum_{e \in D} \downarrow e \cdot y^e) = \sum_{d \in D} \downarrow d \cdot (x^d \gg (\sum_{e \in D} \downarrow e \cdot y^e)) \qquad \text{L3}$$

$$(\sum_{d \in D} \downarrow d \cdot x^d) \gg \uparrow e \cdot y = \sum_{d \in D} \downarrow d \cdot (x^d \gg \uparrow e \cdot y) + \uparrow e \cdot ((\sum_{d \in D} \downarrow d \cdot x^d) \gg y) \qquad \text{L4}$$

The laws are equally valid when the operator $\gg$ is replaced by $\ggg$, except for law L1 where in addition the τ has to be replaced by $c(d)$.

3.3. SACP$^\sharp_\tau$. Module SACP$^\sharp_\tau$ is an 'improved' version of module ACP$^\sharp_\tau$. It is defined by:

$$\mathrm{SACP}^\sharp_\tau = \mathrm{SACP}_\tau + \mathrm{RN} + \mathrm{CH} + \mathrm{REC} + \mathrm{PR} + \mathrm{B} + \mathrm{AIP}^- + \mathrm{AB} + \mathrm{AA} + \mathrm{RR}.$$

If modules in the above equation have an alphabet as parameter, this is A^-, and if they are parametrised by a communication function this is the restriction γ^- of γ to $(A^- \cup \{\delta\}) \times (A^- \cup \{\delta\})$. All proofs in the rest of this paper, unless stated otherwise, are proofs from the module SACP$^\sharp_\tau$. The rules RSP, RSP$^+$ and CFAR can still be used in a setting with module SACP$^\sharp_\tau$. We have SACP$^\sharp_\tau \vdash$ RSP, SACP$^\sharp_\tau \vdash$ RSP$^+$ and SACP$^\sharp_\tau$ + KFAR $\vdash$ CFAR.

§4 Queues

In the specification of concurrent systems FIFO queues with unbounded capacity often play an important role. We give some examples:

- The semantical description of languages with asynchronous message passing such as CHILL (see [19]),
- The modelling of communication channels occurring in computer networks (see Larsen & Milner [28] and Vaandrager [39]),
- The implementation of languages with many-to-one synchronous communication, such as POOL (see America [1] and Vaandrager [40]).

Consequently the questions how queues can be specified, and how one can prove properties of systems containing queues, are important. For a nice sample of queue-specifications we refer to the solutions of the first problem of the STL/SERC workshop [21]. Some other references are Broy [18], Hoare [25] and Pratt [36].

4.1. Also in the setting of ACP a lot of attention has been paid to the specification of queues. Below we give an infinite specification of the process behaviour of a queue. Here D is a finite set of data, D^* is the set of finite sequences σ of elements from D, the empty sequence is ϵ. Sequence $\sigma * \sigma'$ is the concatenation of sequences σ and σ'. The sequence, only consisting of $d \in D$ is denoted by d as well.

$$\boxed{\begin{aligned} QUEUE &= Q_\epsilon = \sum_{d \in D} \downarrow d \cdot Q_d \\ Q_{\sigma * d} &= \sum_{e \in D} \downarrow e \cdot Q_{e * \sigma * d} + \uparrow d \cdot Q_\sigma \end{aligned}}$$

Note that this infinite specification uses only the signature of BPA_δ (see section 2.1). We have the following fact:

4.1.1. THEOREM: *Using read/send communication, the process QUEUE cannot be specified in ACP by finitely many recursion equations.*
PROOF: See BAETEN & BERGSTRA [3] and BERGSTRA & TIURYN [16]. □

It turns out that if one allows an arbitrary communication function, or extends the signature with an (almost) arbitrary additional operator, the process *QUEUE can* be specified by finitely many recursion equations. For some nice examples we refer to BERGSTRA & KLOP [13].

4.2. Definition of the queue by means of chaining. A problem we had with all ACP-specifications of the queue is that they are difficult to deal with in process verifications. For example, let $BUF1$ be a buffer with capacity one:

$$\boxed{\begin{aligned} BUF1 &= \sum_{d\in D} \downarrow d \cdot BUF1^d \\ BUF1^d &= \uparrow d \cdot BUF1 \end{aligned}}$$

In process verifications we need propositions like $QUEUE \gg BUF1 = QUEUE$ (in section 5 we present a protocol verification where a similar fact is actually used). However, the proof of this fact starting from the infinite specification is rather complicated. Now the following specification of a queue by means of the (abstract) chaining operator allows for a simple proof of the proposition and numerous other useful identities involving queues. This specification is also described by HOARE [25] (p. 158).

$$\boxed{Q = \sum_{d\in D} \downarrow d \cdot (Q \gg BUF1^d)}$$

The first thing we have to prove is that the process described above really is a queue.

4.2.1. THEOREM: $Q = QUEUE$.
PROOF: Define for every $n \in \mathbb{N}$ and $\sigma = d_1, \ldots, d_m \in D^*$ processes D^n_σ as follows:

$$D^n_\sigma = Q \gg BUF1 \cdots_{n \text{ times}} \gg BUF1^{d_1} \cdots \gg BUF1^{d_m}$$

So by definition $D^0_\epsilon = Q$. Using the laws of section 3.2.3, we derive the following recursion equations:

$$D^0_\epsilon = \sum_{d\in D} \downarrow d \cdot D^0_d$$

$$D^n_{\sigma * d} = \sum_{e\in D} \downarrow e \cdot D^n_{e * \sigma * d} + \uparrow d \cdot D^{n+1}_\sigma$$

In this derivation, which has been worked out in [23], we use the equation

$$BUF1^d \gg BUF1 = \tau \cdot (BUF1 \gg BUF1^d)$$

which is an instance of law L1 of section 3.2.3. Furthermore we use that $a(p \gg \tau q) = a(p \gg q)$, which follows from proposition 2.1.3 and T1. Define the process Q^0_ϵ by:

$$Q^0_\epsilon = \sum_{d\in D} \downarrow d \cdot Q^0_d$$

$$Q^n_{\sigma * d} = \sum_{e\in D} \downarrow e \cdot Q^n_{e * \sigma * d} + \uparrow d \cdot Q^{n+1}_\sigma$$

The specification of process Q^0_ϵ is clearly guarded. Applying RSP gives us on the one hand that $QUEUE = Q^0_\epsilon$, and on the other hand that $Q = D^0_\epsilon = Q^0_\epsilon$. Consequently $QUEUE = Q$. □

The proof above shows the 'view of a queue' that lies behind the specification of Q. During

execution there is a long chain of 1-datum buffers passing messages from 'the left to the right'. After the input of a new datum on the left, a new buffer is created, containing the new datum and placed at the leftmost position in the chain. Because no buffer is ever removed from the system, the number of empty buffers increases after every output of a datum.

4.2.2. LEMMA: $Q \gg BUF1 = Q$.
PROOF:

$$\begin{aligned} Q \gg BUF1 &= \sum_{d \in D} \downarrow d \cdot ((Q \gg BUF1^d) \gg BUF1) = \\ &= \sum_{d \in D} \downarrow d \cdot (Q \gg (BUF1^d \gg BUF1)) = \\ &= \sum_{d \in D} \downarrow d \cdot (Q \gg \tau \cdot (BUF1 \gg BUF1^d)) = \\ &\overset{2.1.3}{=} \sum_{d \in D} \downarrow d \cdot (Q \gg (BUF1 \gg BUF1^d)) = \\ &= \sum_{d \in D} \downarrow d \cdot ((Q \gg BUF1) \gg BUF1^d) \end{aligned}$$

Now apply RSP$^+$ (from the proof of theorem 4.2.1 it follows that Q is guardedly specifiable). □

By means of an inductive argument we can easily prove the following corollary of lemma 4.2.2.

4.2.3. COROLLARY: *Let for $\sigma \in D^*$, Q^σ be a queue with content σ:*

$$\boxed{\begin{aligned} Q^\epsilon &= Q \\ Q^{\sigma * d} &= Q^\sigma \gg BUF1^d \end{aligned}}$$

Then: $\tau \cdot (Q^\sigma \gg BUF1) = \tau \cdot Q^\sigma$.

4.2.4. PROPOSITION: $Q \gg Q = Q$.
PROOF: Like the proof of proposition 4.2.2. A new ingredient is the identity

$$BUF1^d \gg Q = BUF1 \gg (Q \gg BUF1^d)$$

which is again an instance of L1. Details can be found in [23]. □

4.2.5. COROLLARY: *Let $\sigma, \rho \in D^*$. Then:* $\tau(Q^\sigma \gg Q^\rho) = \tau Q^{\sigma * \rho}$.

4.2.6. Remark. It will be clear that the implementation which is suggested by the specification of process Q is not very efficient: at each time the number of empty storage elements equals the number of data that have left the queue. But we can do it even more inefficiently: the following queue doubles the number of empty storage elements each time a datum is written.

$$\boxed{\overline{Q} = \sum_{d \in D} \downarrow d \cdot (\overline{Q} \gg \uparrow d \cdot \overline{Q})}$$

A standard proof gives that $\overline{Q} = QUEUE$. From the point of view of process algebra this specification is very efficient. It is the shortest specification of a FIFO-queue known to the authors, except for a 5-character specification of PRATT [36]: $\downarrow\uparrow \times D^*$. A problem with Pratt's specification is that a neat axiomatisation of the orthocurrence operator $\times$ is not available. Our $\overline{Q}$-specification has the disadvantage that it does not allow for simple proofs of identities like

$\overline{Q} \gg \overline{Q} = \overline{Q}$.

4.3. Bags. In [10] a bag over data domain D is defined by:

$$BAG = \sum_{d \in D} \downarrow d \cdot (\uparrow d \| BAG)$$

In our full paper ([23]) it has been proved that $Q \gg BAG = BAG$. However, the identity $BAG \gg Q = BAG$ does not hold. The intuitive argument for this is as follows: if a bag contains an apple and an orange, and the environment wants an apple, then it can just take this apple from the bag. In the case where a system, consisting of the chaining of a bag and a queue, contains an apple and an orange, it can occur that the first element in the queue is an orange. In this situation the environment *has* to take the orange first. The argument that processes $Q \gg BAG$ and BAG are different, because in the first process the environment is not able to pick an apple that is still in the queue, does not hold. In ACP_τ we abstract from the real-time behaviour of concurrent systems. If the environment waits long enough then the apple will be in the bag.

4.4. A queue that can lose data. In the specification of communication protocols, we often encounter transmission channels that can make errors: they can lose, damage or duplicate data. All process algebra specifications of these channels we have seen thus far were lengthy and often incomprehensible. Consequently it was difficult to prove properties of systems containing these queues. Now, interestingly, the same idea that was used to specify the normal queue by means of the chaining operator, can also be used to specify the various faulty queues. One just has to replace the process $BUF1$ in the definition by a process that behaves like a buffer but can lose, damage or duplicate data.

First we describe a queue FQ that can lose every datum contained in it at every moment, without any possibilities for the environment to prevent this from happening. The basic component of this queue is the following Faulty Buffer with capacity one:

$$FBUF1 = \sum_{d \in D} \downarrow d \cdot FBUF1^d$$

$$FBUF1^d = (\uparrow d + \tau) \cdot FBUF1$$

If the faulty buffer contains a datum, then this can get lost at any moment through the occurrence of a τ-action. In the equation for $FBUF1^d$ there is no τ-action before the $\uparrow d$-action because this would make it possible for the buffer to reach a state where datum d could not get lost.

We use the above specification in the definition of the faulty queue FQ:

$$FQ = \sum_{d \in D} \downarrow d \cdot (FQ \gg FBUF1^d)$$

The idea behind this specification of the faulty queue is illustrated in figure 3.

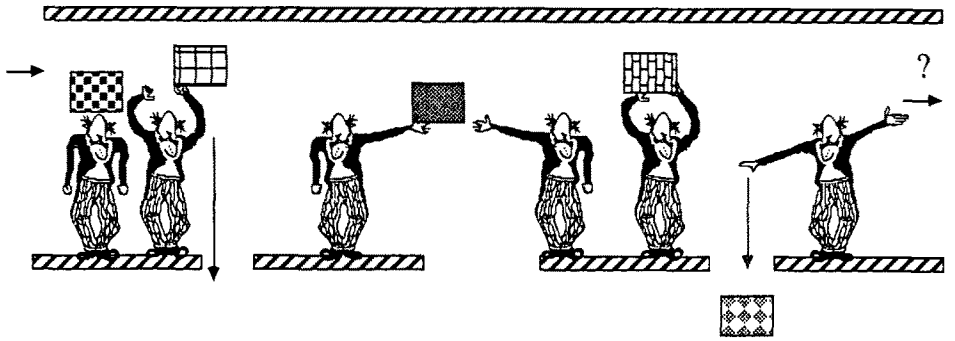

FIGURE 3. *The faulty queue*

4.4.1. LEMMA: $FBUF1^d \gg FBUF1 = \tau\cdot(FBUF1 \gg FBUF1^d)$.
PROOF:

$$FBUF1^d \gg FBUF1 = \tau\cdot(FBUF1 \gg FBUF1^d) + \tau\cdot(FBUF1 \gg FBUF1) =$$
$$= \tau\cdot(FBUF1 \gg FBUF1^d)$$

In the last step we use that: $\tau\cdot(FBUF1 \gg FBUF1) \subseteq FBUF1 \gg FBUF1^d \subseteq \tau\cdot(FBUF1 \gg FBUF1^d)$.
□

Compare the simple definition of FQ with the following $BPA_{\tau\delta}$-specification of the same process.

4.4.2. Let $\sigma,\rho \in D^*$. We write $\sigma \rightarrow \rho$ if ρ can be obtained from σ by deleting one datum. Let $R(\sigma) = \{\rho \mid \sigma \rightarrow \rho\}$ be the finite set of residues of σ after one deletion. Now $FQUEUE$ is the following process.

$$FQUEUE = FQ_\epsilon = \sum_{d\in D} \downarrow d\cdot FQ_d$$

$$FQ_{\sigma * d} = \sum_{e\in D} \downarrow e\cdot FQ_{e*\sigma*d} + \uparrow d\cdot FQ_\sigma + \sum_{\rho\in R(\sigma*d)} \tau\cdot FQ_\rho$$

4.4.3. THEOREM: $FQ = FQUEUE$.
PROOF: Analogously to the proof of theorem 4.2.1. Use lemma 4.4.1 instead of the corresponding equation for $BUF1$. □

Analogous versions of the identities we derived for the normal queue can be derived for the faulty queue in the same way.

4.4.4. PROPOSITION:
i) $FQ \gg FBUF1 = FQ$,
ii) *Let for* $\sigma\in D^*$, FQ^σ *be a faulty queue with content* σ*:*

$$\boxed{\begin{aligned} FQ^{\epsilon} &= FQ \\ FQ^{\sigma*d} &= FQ^{\sigma} \gg FBUF1^{d} \end{aligned}}$$

Then: $\tau\cdot(FQ^{\sigma} \gg FBUF1) = \tau\cdot FQ^{\sigma}$,

iii) $Q \gg FQ = FQ \gg FQ = FQ$,

iv) *Let* $\sigma,\rho \in D^*$. *Then:* $\tau\cdot(FQ^{\sigma} \gg FQ^{\rho}) = \tau\cdot FQ^{\sigma*\rho}$.

PROOF: Exactly as in section 4.2. Use lemma 4.4.1 instead of the corresponding equation for $BUF1$ and in the proof of $FQ \gg FQ = FQ$ use

$$FBUF1^{d} \gg FQ = \tau\cdot(FBUF1 \gg (FQ \gg FBUF1^{d}))$$

instead of the corresponding equation for $BUF1^{d} \gg Q$. This identity can be proved in the same way as lemma 4.4.1. □

4.5. An identity that does nót hold. In this subsection we will discuss the identity

$$FQ = Q \gg FBUF1.$$

'Intuitively' the processes FQ and $Q \gg FBUF1$ are equal since both behave like a FIFO-queue that can lose data. Furthermore, with both processes the environment cannot prevent in any way that a datum gets lost. Unlike the situation with the processes $BAG \gg Q$ and BAG which we discussed in section 4.3, we can think of no 'experiment' that distinguishes between the two processes. Still the identity cannot be proved with the axioms presented thus far.

4.5.1. THEOREM: *If parameter D of operator* $\gg$ *contains more than one element, then* $SACP^{\sharp} \nvdash FQ = Q \gg FBUF1$.

PROOF: We show that the identity is not valid in the model of process graphs modulo bisimulation congruence as presented in BAETEN, BERGSTRA & KLOP [5]. Suppose that there exists a bisimulation between processes FQ and $Q \gg FBUF1$. Suppose that process FQ reads successively two different data, starting from the initial state. Because of the bisimulation it must be possible for the process $Q \gg FBUF1$ to read the same data in such a way that the resulting state is bisimilar to the state process FQ has reached. Now process FQ executes a τ-step and forgets the second datum. We claim that process $Q \gg FBUF1$ is not capable to perform a corresponding sequence of zero or more τ-step. This is because there are only two possibilities:

1) $Q \gg FBUF1$ forgets the second datum. But this means that also the first datum is forgotten. In the resulting state $Q \gg FBUF1$ cannot output any datum (before reading one), whereas process FQ can do this.
2) $Q \gg FBUF1$ does not forget the second datum. In the resulting state $Q \gg FBUF1$ can output this datum. Process FQ cannot do that. □

The argument is illustrated in figure 4.

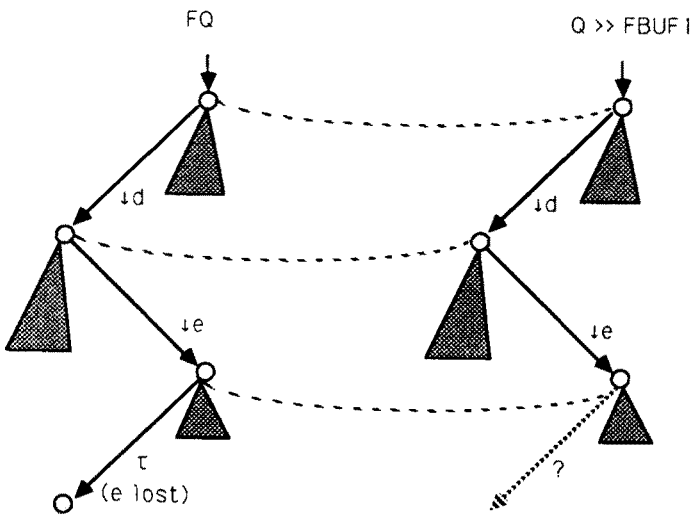

FIGURE 4.

The next theorem shows that, if we add law T4, the two faulty queues can be proven equivalent.

4.5.2. THEOREM: SACP$^{\sharp}_{\tau}$ + T4 ⊢ $FQ = Q \gg FBUF1$.
PROOF: The rather complicated proof of this theorem can be found in [23]. □

4.6. The faulty and damaging queue. In the specification of certain link layer protocols we have to deal with a communication channel that behaves like a FIFO-queue with unbounded capacity (this is of course a simplifying assumption), but has some additional properties: (1) a datum can be damaged at every moment it is in the queue; the environment cannot prevent this event, and (2) a datum can be lost at every moment it is in the queue. We give a process algebra specification of this process in two steps. First we specify the Faulty and Damaging Buffer with capacity one (FDBUF1). We assume that the domain of data D contains a special element *er*, representing a damaged datum.

$$FDBUF1 = \sum_{d \in D} \downarrow d \cdot FDBUF1^d$$

$$FDBUF1^d = \uparrow d \cdot FDBUF1 + \tau \cdot (\uparrow er + \tau) \cdot FDBUF1$$

With the help of this process we can now easily define the Faulty and Damaging Queue (FDQ):

$$FDQ = \sum_{d \in D} \downarrow d \cdot (FDQ \gg FBUF1^d)$$

4.6.1. LEMMA: $FDBUF1^d \gg FBUF1 = \tau \cdot (FDBUF1 \gg FBUF1^d)$.
PROOF: By means of T2, as in lemma 4.4.1, but more complicated. See [23]. □

Once we have lemma 4.6.1, it is standard to prove that process FDQ is guardedly specifiable. It is moreover easy to derive an analogous version of proposition 4.4.4 for FDQ.

4.6.2. Remark. One might ask if there is not a τ too many in the specification of process $FDBUF1$. Why not specify the faulty and damaging buffer simply as follows?

$$\boxed{\begin{aligned} FDB1 &= \sum_{d\in D} \downarrow\cdot FDB1^d \\ FDB1^d &= (\uparrow d + \uparrow er + \tau)\cdot FDB1 \end{aligned}}$$

A first observation we make is that if $D\neq\{er\}$:

$$\mathrm{SACP}_\tau^\sharp \nvdash FDBUF1 = FDB1$$

This is because the two processes are different in bisimulation semantics. Process $FDBUF1$ can input a datum d different from er, and then get into a state where either an output action $\uparrow er$ will be performed or no output action at all. This means that it is possible that a datum is first damaged and then lost. Process $FDB1$ does not have such a state.

For similar reasons we also have the following fact:

$$\mathrm{SACP}_\tau^\sharp \nvdash FDB1^d \gg FDB1 = \tau\cdot(FDB1 \gg FDB1^d)$$

This means that if we work with a queue defined with the help of $FDB1$, our standard technique to prove facts about queues is not applicable. Note that processes $FDB1$ and $FDBUF1$ are trivially equal if we work in a setting where the law T4 ($\tau(\tau x+y)=\tau x+y$) is valid.

4.7. The faulty and stuttering queue. This section is about a very curious queue: a FIFO-queue that can lose or duplicate any element contained in it at every moment. An infinite specification of this process can be found in LARSEN & MILNER [28]. The basic component we use in the specification of the Faulty and Stuttering Queue is a Faulty and Stuttering Buffer with capacity 1:

$$\boxed{\begin{aligned} FSBUF1 &= \sum_{d\in D} \downarrow d\cdot FSBUF1^d \\ FSBUF1^d &= \uparrow d\cdot FSBUF1^d + \tau\cdot FSBUF1 \end{aligned}}$$

$$\boxed{FSQ = \sum_{d\in D} \downarrow d\cdot(FSQ \gg FSBUF1^d)}$$

When we place two faulty and stuttering buffers in a chain, then we have the possibility of an infinite number of internal actions (the first buffer stutters and the second one loses all its input). This implies that, in the specification of the faulty and stuttering queue, we have to guard against unguarded recursion. We need a fairness assumption if we want to exclude the possibility of infinite stuttering. This explains the presence of KFAR in the following lemma.

4.7.1. LEMMA: $\mathrm{SACP}_\tau^\sharp + \mathrm{KFAR} \vdash FSBUF1^d \gg FSBUF1 = \tau\cdot(FSBUF1 \gg FSBUF1^d)$.
PROOF: See [23]. This proof is rather involved. □

From lemma 4.7.1 all the rest follows: process FSQ is guardedly specifiable and we can derive an analogous version of proposition 4.4.4.

§5 A Protocol Verification

In this section we present the specification and verification of a variant of the Alternating Bit Protocol, resembling the ones discussed in Koymans & Mulder [27] and Larsen & Milner [28]. The aim of this exercise is to illustrate the usefulness of the proof technique developed in the previous section. The architecture of the *Concurrent Alternating Bit Protocol (CABP)* is as follows:

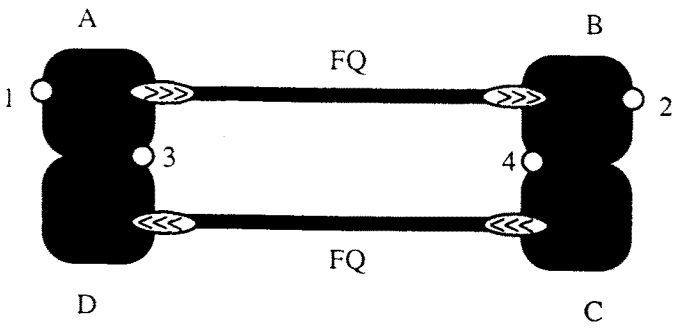

Figure 5.

Elements of a finite set of data are to be transmitted by the CABP from port 1 to port 2. Verification of the CABP amounts to a proof that (1) the protocol will eventually send at port 2 all and only data it has read at port 1, and (2) the protocol will send the data at port 2 in the same order as it has read them at port 1.

In the CABP sender and receiver send frames continuously. Since sender and receiver will have a different clock in general, the number of data that can be in the channels at a certain moment is in principle unlimited. In this section we assume that the channels behave like the process *FQ* as described in section 4.4: a FIFO-queue with unbounded capacity that can either lose frames or pass them on correctly.

In the protocol, the sender consists of two components A and D, whereas the receiver consists of components B and C. One might propose to collapse A and D into a sender process, and B plus C into a receiver process. The resulting processes would be more complicated and in the correctness proof we would have to decompose them again.

5.1. Specification. Let D be a finite set of data which have to be sent by the CABP from port 1 to port 2. Let $B = \{0,1\}$. $\mathfrak{D} = (D \times B) \cup B$ is the set of data which occur as parameter in the actions of the chaining operators. The set of ports is $\mathbb{P} = \{1,2,3,4\}$, the set of data that can be communicated at these ports is $\mathbb{D} = D \cup \{next\}$. Alphabet A and communication function γ are now defined by the standard scheme for the chaining operators, augmented with actions $ri(d)$, $si(d)$ and $ci(d)$, for which we have communications $\gamma(ri(d),si(d)) = ci(d)$ ($i \in \mathbb{P}$ and $d \in \mathbb{D}$).

We now give the specifications of processes A, B, C and D. Here b ranges over $B = \{0,1\}$ and d over D (the overloading of names B and D should cause no confusion). The specifications are standard and need no further comment.

$A = A^0$ $A^b = \sum_{d \in D} r1(d) \cdot A^{db}$ $A^{db} = \uparrow db \cdot A^{db} + r3(next) \cdot A^{1-b}$	$B = B^0$ $B^b = \sum_{d \in D} \downarrow(d, 1-b) \cdot B^b + \sum_{d \in D} \downarrow db \cdot B^{db}$ $B^{db} = s2(d) \cdot s4(next) \cdot B^{1-b}$
$D = D^0$ $D^b = \downarrow(1-b) \cdot D^b + \downarrow b \cdot s3(next) \cdot D^{1-b}$	$C = C^1$ (not C^0!) $C^b = \uparrow b \cdot C^b + r4(next) \cdot C^{1-b}$

Let H and I be the following sets of actions:

$$H = \{r3(next), s3(next), r4(next), s4(next)\}$$

$$I = \{c3(next), c4(next)\}$$

The Concurrent Alternating Bit Protocol is defined by:

$$CABP = \tau_I \circ \partial_H((A \gg FQ \gg B) \| (C \gg FQ \gg D))$$

5.2. Verification. If we do not abstract from the internal actions of the protocol, then the number of states is infinite. This means that a straightforward calculation of the state graph is not possible. A strategy which is often applied in cases like this is that one substitutes a buffer with capacity 1 for the communication channels. As a result the system is finite and can be verified automatically. Next a buffer with capacity 2 is substituted, followed by another automatic verification, etc.. The verification for the case of buffers with capacity 155 takes 23 hours CPU time. Thereafter it is decided that 'the protocol is correct'.

Of course it is not so difficult to specify a protocol that is correct for buffers with capacity less or equal than 155, but fails when the capacity is 156. The conclusion that the protocol is correct for arbitrary buffer size because it works in the cases where the buffer size is less than 156, is therefore influenced by other observations. It is for example intuitively not very plausible that the CABP works for buffer size 155, but not for buffer size 156, because the specification is so short and the only numbers which occur in it are 0 and 1.

Because intuitions can be wrong people look for formal techniques which tell in which situations induction over certain protocol parameters is allowed.

The basic merit of the results of section 4 is that they make it possible to use inductive arguments when dealing with the length of queues in protocol systems. In the verification below we show that the protocol is correct if the channels behave as faulty FIFO-queues with unbounded capacity. However, a minor change in the proof is enough to show that the protocol also works if the channels behave as n-buffers, faulty n-buffers, perfect queues, faulty and stuttering queues, etc.

The following two lemmas will be used to show that, after abstraction, the number of states of the protocol is finite. The first lemma says that if, at the head of the queue, there is a datum that will be thrown away by the receiver because it is of the wrong type, this datum can be thrown away immediately.

5.2.1. LEMMA:
i) $FBUF1^{db} \gg B^{1-b} = \tau \cdot (FBUF1 \gg B^{1-b})$;
ii) $FBUF1^{db} \gg s4(next) \cdot B^{1-b} = \tau \cdot (FBUF1 \gg s4(next) \cdot B^{1-b})$;
iii) $FBUF1^{db} \gg B^{db} = \tau \cdot (FBUF1 \gg B^{db})$.
PROOF: Straightforward with summand inclusion and T2. See [23]. □

The next lemma says that if two frames, of a type that the receiver is willing to accept, are at the head of the queue, one of these can be deleted without changing the process (modulo an initial τ).

5.2.2. LEMMA: $FBUF1^{db} \gg FBUF1^{db} \gg B^b = \tau \cdot (FBUF1 \gg FBUF1^{db} \gg B^b)$.
PROOF: Likewise; see [23]. □

5.2.3. We can now derive a transition diagram for process $A \lll FQ \gg B$. In the derivation we use lemmas 5.2.1 and 5.2.2 to keep the diagram finite. Furthermore we stop the derivation at those places where an action is performed that corresponds to the acknowledgement of a frame that has not yet arrived. In [23] the derivation is carried out in detail. The result of the calculations is presented in lemma 5.2.4, which is pictured in figure 6. The grey arcs correspond to places where we stopped the derivation.

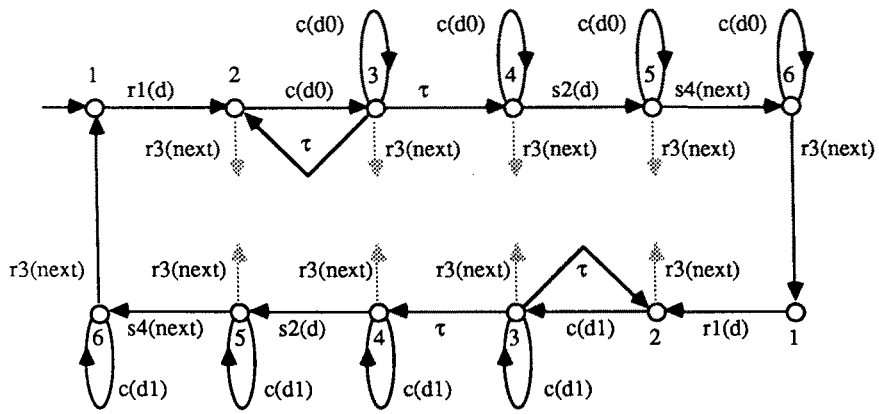

FIGURE 6. *Transition diagram of process* $A \lll FQ \gg B$

5.2.4. LEMMA: *$A \lll FQ \gg B$ satisfies the following system of recursion equations.*

$X = X_1^0$	
$X_1^b = \sum_{d \in D} r1(d) \cdot X_2^{db}$	
$X_2^{db} = c(db) \cdot X_3^{db} + Y_2^b$	$Y_2^b = r3(next) \cdot (A^{1-b} \lll FQ \gg B^b)$
$X_3^{db} = \tau \cdot X_2^{db} + c(db) \cdot X_3^{db} + \tau \cdot X_4^{db} + Y_3^{db}$	$Y_3^{db} = r3(next) \cdot (A^{1-b} \lll FQ \gg FBUF1^{db} \gg B^b)$
$X_4^{db} = c(db) \cdot X_4^{db} + s2(d) \cdot X_5^{db} + Y_4^{db}$	$Y_4^{db} = r3(next) \cdot (A^{1-b} \lll FQ \gg B^{db})$
$X_5^{db} = c(db) \cdot X_5^{db} + s4(next) \cdot X_6^{db} + Y_5^{db}$	$Y_5^{db} = r3(next) \cdot (A^{1-b} \lll FQ \gg s4(next) \cdot B^{1-b})$
$X_6^{db} = r3(next) \cdot X_1^{1-b} + c(db) \cdot X_6^{db}$	

Using CFAR immediately gives the next lemma.

5.2.5. LEMMA: *Let U be specified by:*

$U = U_1^0$	
$U_1^b = \sum_{d \in D} r1(d) \cdot U_2^{db}$	
$U_2^{db} = \tau \cdot U_4^{db} + V_2^b + V_3^{db}$	$V_2^b = r3(next) \cdot (A^{1-b} \gg FQ \gg B^b)$
	$V_3^{db} = r3(next) \cdot (A^{1-b} \gg FQ \gg FBUF1^{db} \gg B^b)$
$U_4^{db} = s2(d) \cdot U_5^{db} + V_4^{db}$	$V_4^{db} = r3(next) \cdot (A^{1-b} \gg FQ \gg B^{db})$
$U_5^{db} = s4(next) \cdot U_6^{db} + V_5^{db}$	$V_5^{db} = r3(next) \cdot (A^{1-b} \gg FQ \gg s4(next) \cdot B^{1-b})$
$U_6^{db} = r3(next) \cdot U_1^{1-b}$	

Then: $\mathrm{SACP}_\tau^\sharp + \mathrm{KFAR} \vdash U = A \gg FQ \gg B$.

In the same way we can derive similar lemmas for 'the other side' of the protocol.

5.2.6. LEMMA:
i) $FBUF1^b \gg D^{1-b} = \tau \cdot (FBUF1 \gg D^{1-b})$;
ii) $FBUF1^b \gg s3(next) \cdot D^{1-b} = \tau \cdot (FBUF1 \gg s3(next) \cdot D^{1-b})$;
iii) $FBUF1^b \gg FBUF1^b \gg D^b = \tau \cdot (FBUF1 \gg FBUF1^b \gg D^b)$.

5.2.7. LEMMA: *Let W be specified by:*

$W = W_1^1$	
$W_1^b = \tau \cdot r4(next) \cdot W_2^{1-b}$	$Z_1^b = r4(next) \cdot (C^{1-b} \gg FQ \gg D^b)$
$W_2^b = \tau \cdot W_3^b + Z_1^b + Z_2^b$	$Z_2^b = r4(next) \cdot (C^{1-b} \gg FQ \gg FBUF1^b \gg D^b)$
$W_3^b = s3(next) \cdot W_1^b + Z_3^b$	$Z_3^b = r4(next) \cdot (C^{1-b} \gg FQ \gg s3(next) \cdot D^{1-b})$

Then: $\mathrm{SACP}_\tau^\sharp + \mathrm{KFAR} \vdash C \gg FQ \gg D = W$.

The fact that CABP is a correct protocol is asserted by

5.2.8. THEOREM: $\mathrm{SACP}_\tau^\sharp + \mathrm{KFAR} \vdash CABP = \tau \cdot (\sum_{d \in D} r1(d) \cdot s2(d)) \cdot CABP$.

PROOF: Lemmas 5.2.5 and 5.2.7 together give that we can write *CABP* as:

$$CABP = \tau_I \circ \partial_H(U \| W)$$

A straightforward expansion gives:

$$\tau_I \circ \partial_H(U \| W) = \tau \cdot (\sum_{d \in D} r1(d) \cdot s2(d)) \cdot (\sum_{e \in D} r1(e) \cdot s2(e)) \cdot \tau_I \circ \partial_H(U \| W)$$

The variables V and Z vanish in the expansion, due to the fact that they only occur in situations where a receiver component sends a premature acknowledgement. An application of RSP concludes the proof of the theorem. □

5.2.9. Remark. A serious problem that has to be faced in the context of algebraic protocol verification is the fairness issue. In the verifications of this paper we used KFAR to deal with fairness. KFAR is the algebraic equivalent of the statement: 'if anything can go well infinitely often, it will go well infinitely often'. In most applications a more subtle treatment of fairness is desirable. Moreover KFAR is incompatible with lots of semantics between bisimulation and trace semantics. In [15] it is proved that failure semantics is inconsistent with the rule KFAR. In the same paper a restricted version KFAR^- of KFAR is presented which *is* consistent with the axioms of failure semantics, but this version is not powerful enough to allow for a verification of the CABP. The argument for this is simple: KFAR^- allows for the fair abstraction of *unstable divergence*. This means that a process will never stay forever in a conservative cluster of internal τ-steps if it can be exited by another internal τ-step. Since in the CABP component C can always perform an internal step, and since the protocol is finite state (after suitable abstraction), there must be a conservative cluster of internal steps which can only be exited by performing an observable action. Thus the CABP contains stable divergence.

§6 Conclusions and open problems

In this paper we presented a language making it possible to give modular specifications of process algebras. The language contains operations $+$ and $\Box$, which are standard in the theory of structured algebraic specifications, and moreover two new operators H and S. Two applications have been presented of the new operators: we showed how the left-merge operator can be hidden if this is needed and we described how the chaining operator can be defined in a clean way in terms of more elementary operators. It is clear that there are much more applications of our approach. Numerous other process combinators can be defined in terms of more elementary operators in the same way as we did with the chaining operators. Maybe also other model theoretic operations can be used in a process algebra setting (cartesian products?).

Strictly speaking we have not introduced a 'module algebra' as in [9]: we do not interpret module expressions in an algebra. However, this can be done without any problem. An interesting topic of research is to look for axioms to manipulate module expressions. Due to the presence of the operators H and S, an elimination theorem for module expressions as in [9] will probably not be achievable.

An important open problem for us is the question whether the proof system of table 1 is complete for first order logic.

In this paper the modules are parametrised by a set of actions. These actions themselves do not have any structure. The most natural way to look towards actions like $s\,1(d_0)$ however, is to see them as actions parametrised by data. We would like to include the notion of a parametrised action in our framework but it turns out that this is not trivial. Related work in this area has been done by Mauw [29] and Mauw & Veltink [30].

In order to prove the associativity of the chaining operators, we needed auxiliary actions $\overline{s}(d)$, $\overline{r}(d)$, etc. Also in other situations it often turns out to be useful to introduce auxiliary actions in verifications. At present we have to introduce these actions right at the beginning of a specification. This is embarrassing for a reader who does not know about the future use of these actions in the verification. But of course also the authors don't like to rewrite their specification all the time when they work on the verification. Therefore we would like to have a proof principle saying that it is allowed to use 'fresh' atomic actions in proofs. We think that it is possible to add a 'Fresh Atom Principle' (FAP) to our formal setting, but some work still has to be done.

In our view section 4 convincingly shows that chaining operators are useful in dealing with FIFO-queues. We think that in general it will be often the case that a new application requires new operators and laws.

In section 4.5 we presented a simple example of a realistic situation where bisimulation semantics does not work: a FIFO-queue which can loose data at every place is different from a

FIFO-queue which can only loose data at the end. Adding the law T4, which holds in ready trace semantics (and hence in failure semantics), made it possible to prove the two queues equal.

For the correctness of protocols which involve faulty queues one normally needs some fairness assumption. Koomen's Fair Abstraction Rule (KFAR) often forms an adequate, although not optimal, way to model fairness. An interesting open problem is therefore the question whether the module SACP$^\sharp_\tau$ + T4 + KFAR is consistent (conjecture: yes).

The verification of the Concurrent Alternating Bit Protocol as presented in the full version of this report takes 4 pages (or 5 if the proofs of the standard facts about the queues are included). Our proof is considerably shorter than the proof of similar protocols in [27] and [28] (15 and 11 pages respectively). But maybe this comparison is not altogether fair because the proofs in these papers were meant as an illustration of new modular proof techniques. Our proof shows that the axioms of bisimulation semantics with fair abstraction are sufficient for the modular verification of simple protocols like this. The axioms of bisimulation semantics will turn out to be not sufficient for more substantial modular verifications because bisimulation semantics is not fully abstract. We could give a shorter and simpler proof of the protocol by using the notion of redundancy in context of [41]: the grey arcs in figure 6 all correspond to summands which are redundant in the context in which they occur. Additional proof techniques will certainly be needed for the modular verification of more complex protocols.

ACKNOWLEDGEMENTS

Our thanks to Jan Bergstra for his help in the development of the *H*-operator and to Kees Middelburg for helpful comments on an earlier version.

APPENDIX: LOGICS

In this appendix equational, conditional equational and first order logic are defined. Since all these logics share the concepts of variables and terms, these will be treated first.

1. Variables and terms. Let σ be a signature. A *σ-variable* is an expression x_S with $x \in$ NAMES and $\mathbb{S}_\sigma:S$. A *valuation* of the σ-variables in a σ-algebra $\mathcal{A}$ is a function ξ that takes every σ-variable x_S into an element of $S^{\mathcal{A}}$.

For any $\mathbb{S}_\sigma:S$ the set T^τ_S of *σ-terms* of sort S is defined inductively by:
- $x_S \in T^\tau_S$ for any σ-variable x_S.
- If $\mathbb{F}_\sigma:f:S_1 \times \cdots \times S_n \to S$ and $t_i \in T^\tau_{S_i}$ for $i=1,...,n$ then $f_{S_1 \times \cdots \times S_n \to S}(t_1, \cdots, t_n) \in T^\tau_S$.

The *ξ-evaluation* $[\![t]\!]^\xi \in S^{\mathcal{A}}$ of a σ-term $t \in T^\tau_S$ in a σ-algebra $\mathcal{A}$ (with ξ a valuation) is defined by:
- $[\![x_S]\!]^\xi = \xi(x_S) \in S^{\mathcal{A}}$.
- $[\![f_{S_1 \times \cdots \times S_n \to S}(t_1, \cdots, t_n)]\!]^\xi = f^{\mathcal{A}}_{S_1 \times \cdots \times S_n \to S}([\![t_1]\!]^\xi, \cdots, [\![t_n]\!]^\xi)$.

2. Equational logic. The set F^{eql}_σ of *equations* or *equational formulas* over σ is defined by:
- If $t_i \in T^\tau_S$ for $i=1,2$ and certain $\mathbb{S}_\sigma:S$ then $(t_1 = t_2) \in F^{eql}_\sigma$.

An equation $(t_1 = t_2) \in F^{eql}_\sigma$ is *ξ-true* in a σ-algebra $\mathcal{A}$, notation $\mathcal{A}, \xi \models^{eql}_\sigma t_1 = t_2$, if $[\![t_1]\!]^\xi = [\![t_2]\!]^\xi$. Such an equation $\phi \in F^{eql}_\sigma$ is *true* in $\mathcal{A}$, notation $\mathcal{A} \models^{eql}_\sigma \phi$, if $\mathcal{A}, \xi \models^{eql}_\sigma \phi$ for all valuations ξ.

An inference system I^{eql}_σ for equational logic is displayed in table 11 below. There t, u and v are terms over σ and x is a variable. Furthermore $t[u/x]$ is the result of substituting u for all occurrences of x in t. Of course u and x should be of the same sort. Finally an inference rule $\frac{H}{\phi}$ with $H = \varnothing$ is called an *axiom* and denoted simply by ϕ.

$t=t$	$\dfrac{u=v}{v=u}$	$\dfrac{t=u,\ u=v}{t=v}$	$\dfrac{u=v}{t[u/x]=t[v/x]}$	$\dfrac{u=v}{u[t/x]=v[t/x]}$

TABLE 11.

3. Conditional equational logic. The set F_σ^{at} of *atomic formulas* over σ is defined by:
- If $t_i \in T_S^\forall$ for $i=1,2$ and certain $\mathbb{S}_\sigma : S$ then $(t_1=t_2)\in F_\sigma^{at}$.
- If $\mathbf{R}_\sigma : p \subseteq S_1 \times \cdots \times S_n$ and $t_i \in T_{S_i}^\forall$ for $i=1,\ldots,n$ then $p_{S_1\times\cdots\times S_n}(t_1,\cdots,t_n)\in F_\sigma^{at}$.

The set F_σ^{ceql} of *conditional equational formulas* over σ is defined by:
- If $C \subseteq F_\sigma^{at}$ and $\alpha \in F_\sigma^{at}$ then $(C\Rightarrow\alpha)\in F_\sigma^{ceql}$.

The *ξ-truth* of formulas $\phi \in F_\sigma^{at} \cup F_\sigma^{ceql}$ in a σ-algebra $\mathcal{A}$ is defined by:
- $\mathcal{A},\xi \models_\sigma^{ceql} t_1=t_2$ if $[\![t_1]\!]^\xi = [\![t_2]\!]^\xi$.
- $\mathcal{A},\xi \models_\sigma^{ceql} p_{S_1\times\cdots\times S_n}(t_1,\cdots,t_n)$ if $p^{\mathcal{A}}_{S_1\times\cdots\times S_n}([\![t_1]\!]^\xi,\cdots,[\![t_n]\!]^\xi)$.
- $\mathcal{A},\xi \models_\sigma^{ceql} C\Rightarrow\alpha$ if $\mathcal{A},\xi \not\models_\sigma^{ceql} \beta$ for some $\beta\in C$ or $\mathcal{A},\xi \models_\sigma^{ceql} \alpha$.

ϕ is *true* in $\mathcal{A}$, notation $\mathcal{A} \models_\sigma^{ceql} \phi$, if $\mathcal{A},\xi \models_\sigma^{ceql} \phi$ for all valuations ξ.

An inference system I_σ^{ceql} for conditional equational logic is displayed in table 12 below. There α and α_i are atomic formulas, C is a set of atomic formulas, ϕ is a conditional equational formula, t_i, t, u and v are terms over σ and x_i and x are variables. Furthermore $\alpha[u/x]$ is the result of substituting u for all occurrences of x in α. Of course u and x should be of the same sort. Likewise $\phi[t_i/x_i\ (i\in I)]$ is the result of simultaneous substitution for $i\in I$ of t_i for all occurrences of x_i in ϕ. An inference rule $\dfrac{\varnothing}{\phi}$ is again denoted by ϕ and a conditional equational formula $\varnothing\Rightarrow\alpha$ by α.

$C\Rightarrow\alpha$ if $\alpha\in C$	$\dfrac{C\Rightarrow\alpha_i\ (i\in I),\ \{\alpha_i \mid i\in I\}\Rightarrow\alpha}{C\Rightarrow\alpha}$	$\dfrac{\phi}{\phi[t_i/x_i\ (i\in I)]}$	
$t=t$ $\{u=v\}\Rightarrow(v=u)$	$\{t=u,\ u=v\}\Rightarrow(t=u)$	$\{u=v,\ \alpha[u/x]\}\Rightarrow(\alpha[v/x])$	

TABLE 12.

The logic described above is *infinitary conditional equational logic. Finitary conditional equational logic* is obtained by the extra requirement that in conditional equational formulas $C\Rightarrow\alpha$ the set of conditions C should be finite. In that case the inference rule

$$\frac{\phi}{\phi[t_i/x_i\ (i\in I)]} \quad \text{can be replaced by} \quad \frac{\phi}{\phi[t/x]}.$$

Furthermore *(in)finitary conditional logic* is obtained by omitting all reference to the equality predicate $=$.

4. First order logic. The set F_σ^{foleq} of *first order formulas with equality* over σ is defined by:
- If $t_i \in T_S^\forall$ for $i=1,2$ and certain $\mathbb{S}_\sigma : S$ then $(t_1=t_2)\in F_\sigma^{foleq}$.
- If $\mathbf{R}_\sigma : p \subseteq S_1\times\cdots\times S_n$ and $t_i\in T_{S_i}^\forall$ for $i=1,\ldots,n$ then $p_{S_1\times\cdots\times S_n}(t_1,\cdots,t_n)\in F_\sigma^{foleq}$.
- If $\phi\in F_\sigma^{foleq}$ then $\neg\phi\in F_\sigma^{foleq}$.
- If ϕ and $\psi\in F_\sigma^{foleq}$ then $(\phi\rightarrow\psi)\in F_\sigma^{foleq}$.
- If ϕ and $\psi\in F_\sigma^{foleq}$ then $(\phi\wedge\psi)\in F_\sigma^{foleq}$.
- If ϕ and $\psi\in F_\sigma^{foleq}$ then $(\phi\vee\psi)\in F_\sigma^{foleq}$.
- If ϕ and $\psi\in F_\sigma^{foleq}$ then $(\phi\leftrightarrow\psi)\in F_\sigma^{foleq}$.
- If x_S is a σ-variable and $\phi\in F_\sigma^{foleq}$ then $\forall x_S(\phi)\in F_\sigma^{foleq}$.
- If x_S is a σ-variable and $\phi\in F_\sigma^{foleq}$ then $\exists x_S(\phi)\in F_\sigma^{foleq}$.

The ξ-*truth* of a formula $\phi \in F_\sigma^{foleq}$ in a σ-algebra $\mathcal{A}$ is defined inductively by:

- $\mathcal{A},\xi \models_\sigma^{foleq} t_1 = t_2$ if $[\![t_1]\!]^\xi = [\![t_2]\!]^\xi$.
- $\mathcal{A},\xi \models_\sigma^{foleq} p_{S_1\times\cdots\times S_n}(t_1,\cdots,t_n)$ if $p^{\mathcal{A}}_{S_1\times\cdots\times S_n}([\![t_1]\!]^\xi,\cdots,[\![t_n]\!]^\xi)$.
- $\mathcal{A},\xi \models_\sigma^{foleq} \neg\phi$ if $\mathcal{A},\xi \not\models_\sigma^{foleq} \phi$.
- $\mathcal{A},\xi \models_\sigma^{foleq} \phi\to\psi$ if $\mathcal{A},\xi \not\models_\sigma^{foleq} \phi$ or $\mathcal{A},\xi \models_\sigma^{foleq} \psi$.
- $\mathcal{A},\xi \models_\sigma^{foleq} \phi\wedge\psi$ if $\mathcal{A},\xi \models_\sigma^{foleq} \phi$ and $\mathcal{A},\xi \models_\sigma^{foleq} \psi$.
- $\mathcal{A},\xi \models_\sigma^{foleq} \phi\vee\psi$ if $\mathcal{A},\xi \models_\sigma^{foleq} \phi$ or $\mathcal{A},\xi \models_\sigma^{foleq} \psi$.
- $\mathcal{A},\xi \models_\sigma^{foleq} \phi\leftrightarrow\psi$ if $\mathcal{A},\xi \models_\sigma^{foleq} \phi$ if and only if $\mathcal{A},\xi \models_\sigma^{foleq} \psi$.
- $\mathcal{A},\xi \models_\sigma^{foleq} \forall x_S(\phi)$ if $\mathcal{A},\xi' \models_\sigma^{foleq} \phi$ for all valuations ξ' with $\xi'(y_{S'})=\xi(y_{S'})$ for all variables $y_{S'}\neq x_S$.
- $\mathcal{A},\xi \models_\sigma^{foleq} \exists x_S(\phi)$ if $\mathcal{A},\xi' \models_\sigma^{foleq} \phi$ for some valuation ξ' with $\xi'(y_{S'})=\xi(y_{S'})$ for all variables $y_{S'}\neq x_S$.

ϕ is *true* is $\mathcal{A}$, notation $\mathcal{A} \models_\sigma^{foleq} \phi$, if $\mathcal{A},\xi \models_\sigma^{foleq} \phi$ for all valuations ξ.

An inference system I_σ^{foleq} for first order logic with equality is displayed in table 13 below. There ϕ, ψ and ρ are elements of F_σ^{foleq}, α is an atomic formula (constructed by means of the first two clauses in the definition of F_σ^{foleq} only), t, u and v are terms over σ and x is a variable. An occurrence of a variable x in a formula ϕ is *bound* if it occurs in a subformula $\forall x(\psi)$ or $\exists x(\psi)$ of ϕ. Otherwise it is *free*. $\phi[t/x]$ denotes the result of substituting u for all free occurrences of x in t. Of course u and x should be of the same sort. Now t *is free for* x *in* ϕ if all free occurrences of variables in t remain free in $\phi[t/x]$. As before an inference rule $\frac{H}{\phi}$ with $H=\varnothing$ is called an *axiom* and denoted simply by ϕ.

$\frac{\phi,\ \phi\to\psi}{\psi}$ *modus ponens*		$\frac{\phi}{\forall x(\phi)}$ *generalisation*
$\phi\to(\psi\to\phi)$ $\{\phi\to(\psi\to\rho)\}\to\{(\phi\to\psi)\to(\phi\to\rho)\}$ $\{\forall x(\phi\to\psi)\}\to\{\phi\to\forall x(\psi)\}$, if x does not occur free in ϕ		} *deduction axioms*
$(\neg\phi\to\phi)\to\phi$		*axiom of the excluded middle*
$\neg\phi\to(\phi\to\psi)$		*axiom of contradiction*
$\forall x(\phi)\to\phi[t/x]$, if t is free for x in ϕ		*axiom of specialisation*
$(\phi\wedge\psi)\to\phi$ $(\phi\wedge\psi)\to\psi$ $\phi\to\{\psi\to(\phi\wedge\psi)\}$	$\phi\to(\phi\vee\psi)$ $\psi\to(\phi\vee\psi)$ $(\phi\vee\psi)\to(\neg\phi\to\psi)$	$(\phi\leftrightarrow\psi)\to\{(\phi\to\psi)\wedge(\psi\to\phi)\}$ $\{(\phi\to\psi)\wedge(\psi\to\phi)\}\to(\phi\leftrightarrow\psi)$ $\exists x(\phi)\leftrightarrow\neg\forall x(\neg\phi)$
$t=t$ $(u=v)\to(v=u)$	$\{(t=u)\wedge(u=v)\}\to(t=v)$	$(u=v)\to(\alpha[u/x]\leftrightarrow\alpha[v/x])$

TABLE 13.

First order logic is obtained from first order logic with equality by omitting all reference to $=$. It is also possible to present first order logic without the connectives $\wedge$, $\vee$ and $\leftrightarrow$ and the quantifier $\exists$, and introduce them as notational abbreviations. In that case the third block of table 13 can be omitted.

5. Expressiveness. One can translate an equation $\alpha \in F_\sigma^{eql}$ by a (finitary) conditional equational formula $\varnothing\Rightarrow\alpha \in F_\sigma^{fceql}$ and a finitary conditional equational formula $\{\alpha_1,\cdots,\alpha_n\}\Rightarrow\alpha \in F_\sigma^{fceql}$ into a first order formula $(\alpha_1\wedge\cdots\wedge\alpha_n)\to\alpha \in F_\sigma^{foleq}$. Using this translation we have $F_\sigma^{eql} \subset F_\sigma^{fceql} \subset F_\sigma^{foleq}$ and furthermore $\mathcal{A} \models_\sigma^{eql} \phi \Leftrightarrow \mathcal{A} \models_\sigma^{ceql} \phi$ for $\phi \in F_\sigma^{eql}$ and $\mathcal{A} \models_\sigma^{ceql} \phi \Leftrightarrow \mathcal{A} \models_\sigma^{foleq} \phi$ for $\phi \in F_\sigma^{fceql}$. This means that first order logic with equality is more expressive then equational logic and finitary conditional equational logic is somewhere in between. However first order logic with equality and infinitary conditional equational logic are

incomparable.

6. Completeness. For all logics mentioned above the following completeness result is known to hold: $Alg(\sigma,T) \vDash_\sigma^{\mathfrak{L}} \phi \Rightarrow T \vdash_\sigma^{\mathfrak{L}} \phi$. The reverse direction also holds, since all these logics are obviously sound. As a corollary we have

$$T \vdash_\sigma^{eql} \phi \Leftrightarrow T \vdash_\sigma^{ceql} \phi \quad \text{for } \phi \in F_\sigma^{eql} \text{ and}$$

$$T \vdash_\sigma^{ceql} \phi \Leftrightarrow T \vdash_\sigma^{foleq} \phi \quad \text{for } \phi \in F_\sigma^{fceql}.$$

For this reason in a lot of process algebra papers it is not made explicit which logic is used in verifications: the space needed for stating this could be saved, since the resulting notion of provability would be the same anyway. However, the situation changes when formulas are proved from modules. Equational logic and conditional equational logic are not complete anymore and for first order logic with equality this is still an open problem (for us). Here a logic $\mathfrak{L}$ is complete if $M \vDash^{\mathfrak{L}} \phi \Rightarrow M \vdash^{\mathfrak{L}} \phi$. It is easily shown that

$$M \vdash^{eql} \phi \Rightarrow M \vdash^{ceql} \phi \quad \text{for } \phi \in F_{\Sigma(M)}^{eql} \text{ and}$$

$$M \vdash^{ceql} \phi \Rightarrow M \vdash^{foleq} \phi \quad \text{for } \phi \in F_{\Sigma(M)}^{fceql},$$

but the reverse directions do not hold. Thus we should state exactly in which logic our results are proved.

7. Notation. This paper employs infinitary conditional equational logic. However, no proof trees are constructed; proofs are given in a slightly informal way, that allows a straightforward translation into formal proofs by the reader. Furthermore all type information given in the subscripts of variables, function and predicate symbols is omitted, since confusion about the correct types is almost impossible. Outside section 1 and this appendix inference rules $\frac{H}{\phi}$ do not occur, but all conditional equational formulas $C \Rightarrow \alpha$ are written $\frac{C}{\alpha}$, as is usual. However, the suggested similarity between inference rules and conditional equational formulas is misleading: $\frac{H}{\phi}$ holds in an algebra $\mathcal{A}$ if ($\mathcal{A},\xi \vDash \psi$ for all $\psi \in H$ and all valuations ξ) implies ($\mathcal{A},\xi \vDash \phi$ for all valuations ξ), while $\frac{C}{\alpha}$ holds in $\mathcal{A}$ if for all valuations ξ: ($\mathcal{A},\xi \vDash \beta$ for all $\beta \in C$ implies $\mathcal{A},\xi \vDash \alpha$).

8. Positive and universal formulas. In equational logic all formulas are both positive and universal. In conditional equational logic all formulas are universal and the positive formulas are the atomic ones. In first order logic with equality the positive formulas are the ones without the connectives $\neg$ and $\rightarrow$ and the universal ones are the formulas without quantifiers. Model theory (see for instance [33]) teaches us that a formula ϕ is preserved under homomorphisms (respectively subalgebras) iff there is a positive (respectively universal) formula ψ with $\vdash^{foleq} \psi \leftrightarrow \phi$.

References

[1] P. America (1985): *Definition of the programming language POOL-T.* ESPRIT project 415, Doc. Nr. 91, Philips Research Laboratories, Eindhoven.

[2] D. Austry & G. Boudol (1984): *Algèbre de processus et synchronisations.* Theoretical Computer Science 30(1), pp. 91-131.

[3] J.C.M. Baeten & J.A. Bergstra (1987): *Global Renaming Operators in Concrete Process Algebra (revised version).* Report P8709, Programming Research Group, University of Amsterdam, to appear in I&C.

[4] J.C.M. Baeten, J.A. Bergstra & J.W. Klop (1987): *Conditional Axioms and α/β Calculus in Process Algebra.* In: Formal Description of Programming Concepts - III, Proceedings of

the third IFIP WG 2.2 working conference, Ebberup 1986 (M. Wirsing, ed.), North-Holland, Amsterdam, pp. 53-75.

[5] J.C.M. BAETEN, J.A. BERGSTRA & J.W. KLOP (1987): *On the Consistency of Koomen's Fair Abstraction Rule.* Theoretical Computer Science 51(1/2), pp. 129-176.

[6] J.C.M. BAETEN, J.A. BERGSTRA & J.W. KLOP (1987): *Ready trace semantics for concrete process algebra with priority operator.* The Computer Journal 30(6), pp. 498-506.

[7] J.C.M. BAETEN & R.J. VAN GLABBEEK (1987): *Merge and termination in process algebra.* In: Proceedings 7th Conference on Foundations of Software Technology & Theoretical Computer Science, Pune, India (K.V. Nori, ed.), LNCS 287, Springer-Verlag, pp. 153-172.

[8] J.A. BERGSTRA (1985): *A Process Creation Mechanism in Process Algebra.* Logic Group Preprint Series Nr. 2, CIF, State University of Utrecht, to appear in: Applications of Process Algebra, (J.C.M. Baeten, ed.), CWI Monograph, North-Holland, 1988.

[9] J.A. BERGSTRA, J. HEERING & P. KLINT (1986): *Module Algebra.* Report CS-R8617, Centrum voor Wiskunde en Informatica, Amsterdam, to appear in: Journal of the ACM.

[10] J.A. BERGSTRA & J.W. KLOP (1984): *The algebra of recursively defined processes and the algebra of regular processes.* In: Proceedings 11th ICALP, Antwerpen (J. Paredaens, ed.), LNCS 172, Springer-Verlag, pp. 82-95.

[11] J.A. BERGSTRA & J.W. KLOP (1984): *Process algebra for synchronous communication.* I&C 60(1/3), pp. 109-137.

[12] J.A. BERGSTRA & J.W. KLOP (1985): *Algebra of communicating processes with abstraction.* Theoretical Computer Science 37(1), pp. 77-121.

[13] J.A. BERGSTRA & J.W. KLOP (1986): *Process Algebra: Specification and Verification in Bisimulation Semantics.* In: Mathematics and Computer Science II, CWI Monograph 4 (M. Hazewinkel, J.K. Lenstra & L.G.L.T. Meertens, eds.), North-Holland, Amsterdam, pp. 61-94.

[14] J.A. BERGSTRA & J.W. KLOP: *ACP_τ: A Universal Axiom System for Process Specification.* This volume.

[15] J.A. BERGSTRA, J.W. KLOP & E.-R. OLDEROG (1987): *Failures without chaos: a new process semantics for fair abstraction.* In: Formal Description of Programming Concepts - III, Proceedings of the third IFIP WG 2.2 working conference, Ebberup 1986 (M. Wirsing, ed.), North-Holland, Amsterdam, pp. 77-103.

[16] J.A. BERGSTRA & J. TIURYN (1987): *Process Algebra Semantics for Queues.* Fund. Inf. X, pp. 213-224, also appeared as: MC Report IW 241, Amsterdam 1983.

[17] S.D. BROOKES & A.W. ROSCOE (1985): *An improved failures model for communicating processes.* In: Seminar on Concurrency (S.D. Brookes, A.W. Roscoe & G. Winskel, eds.), LNCS 197, Springer-Verlag, pp. 281-305.

[18] M. BROY (1987): *Views of Queues.* Report MIP-8704, Fakultät für Mathematik und Informatik, Universität Passau.

[19] CHILL (1980): *Recommendation Z.200 (CHILL Language Definition).* CCITT Study Group XI.

[20] R. DE NICOLA & M. HENNESSY (1984): *Testing equivalences for processes.* Theoretical Computer Science 34, pp. 83-134.

[21] T. DENVIR, W. HARWOOD, M. JACKSON & M. RAY (1985): *The Analysis of Concurrent Systems, Proceedings of a Tutorial and Workshop, Cambridge University 1983,* LNCS 207, Springer-Verlag.

[22] R.J. VAN GLABBEEK (1987): *Bounded Nondeterminism and the Approximation Induction Principle in Process Algebra.* In: Proceedings STACS 87 (F.J. Brandenburg, G. Vidal-Naquet & M. Wirsing, eds.), LNCS 247, Springer-Verlag, pp. 336-347.

[23] R.J. VAN GLABBEEK & F.W. VAANDRAGER (1988): *Modular Specifications in Process Algebra - With Curious Queues.* Report CS-R8821, Centrum voor Wiskunde en Informatica, Amsterdam.

[24] C.A.R. HOARE (1980): *Communicating sequential processes.* In: On the construction of

programs - an advanced course (R.M. McKeag & A.M. Macnaghten, eds.), Cambridge University Press, pp. 229-254.

[25] C.A.R. HOARE (1985): *Communicating Sequential Processes*, Prentice-Hall International.

[26] HE JIFENG & C.A.R. HOARE (1987): *Algebraic specification and proof of a distributed recovery algorithm.* Distributed Computing 2(1), pp. 1-12.

[27] C.P.J. KOYMANS & J.C. MULDER (1986): *A Modular Approach to Protocol Verification using Process Algebra.* Logic Group Preprint Series Nr. 6, CIF, State University of Utrecht, to appear in: Applications of Process Algebra, (J.C.M. Baeten, ed.), CWI Monograph, North-Holland, 1988.

[28] K.G. LARSEN & R. MILNER (1987): *A Complete Protocol Verification Using Relativized Bisimulation.* In: Proceedings 14th ICALP, Karlsruhe (Th. Ottmann, ed.), LNCS 267, Springer-Verlag, pp. 126-135.

[29] S. MAUW (1987): *An algebraic specification of process algebra, including two examples.* This volume.

[30] S. MAUW & G.J. VELTINK (1988): *A Process Specification Formalism.* Report P8814, Programming Research Group, University of Amsterdam.

[31] R. MILNER (1980): *A Calculus of Communicating Systems*, LNCS 92, Springer-Verlag.

[32] R. MILNER (1985): *Lectures on a Calculus for Communicating Systems.* In: Seminar on Concurrency (S.D. Brookes, A.W. Roscoe & G. Winskel, eds.), LNCS 197, Springer-Verlag, pp. 197-220.

[33] J.D. MONK (1976): *Mathematical Logic*, Springer-Verlag.

[34] E.-R. OLDEROG & C.A.R. HOARE (1986): *Specification-Oriented Semantics for Communicating Processes.* Acta Informatica 23, pp. 9-66.

[35] I.C.C. PHILLIPS (1987): *Refusal Testing.* Theoretical Computer Science 50, pp. 241-284.

[36] V.R. PRATT (1986): *Modelling Concurrency with Partial Orders.* International Journal of Parallel Programming 15(1), pp. 33-71.

[37] D.T. SANNELLA & A. TARLECKI (1988): *Toward Formal Development of Programs from Algebraic Specifications: Implementations Revisited.* Acta Informatica 25, pp. 233-281.

[38] D.T. SANNELLA & M. WIRSING (1983): *A kernel language for algebraic specification and implementation (extended abstract).* In: Proc. Intl. Conf. on Foundations of Computation Theory, Borgholm (M. Karpinski, ed.), LNCS 158, pp. 413-427, long version: Report CSR-131-83, Dept. of Computer Science, Univ. of Edinburgh, 1983.

[39] F.W. VAANDRAGER (1986): *Verification of Two Communication Protocols by Means of Process Algebra.* Report CS-R8608, Centrum voor Wiskunde en Informatica, Amsterdam.

[40] F.W. VAANDRAGER (1986): *Process algebra semantics of POOL.* Report CS-R8629, Centrum voor Wiskunde en Informatica, Amsterdam, to appear in: Applications of Process Algebra, (J.C.M. Baeten, ed.), CWI Monograph, North-Holland, 1988.

[41] F.W. VAANDRAGER (1988): *Some Observations on Redundancy in a Context.* Report CS-R8812, Centrum voor Wiskunde en Informatica, Amsterdam, to appear in: Applications of Process Algebra, (J.C.M. Baeten, ed.), CWI Monograph, North-Holland, 1988.