Modular Specification of Process Algebras

Rob van Glabbeek

Institut für Informatik der Technischen Universität
Postfach 20 24 20, D-8000 München 2, Germany

Frits Vaandrager

Centrum voor Wiskunde en Informatica
Postbus 4079, 1009 AB Amsterdam, The Netherlands

This paper proposes a modular approach to the algebraic specification of process algebras. 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 $\square$, 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 give a presentation of equational logic, infinitary conditional equational logic - of which we also proof the completeness - and first order logic and show how the notion of a formal proof of a formula from a theory can be generalized to that of a proof of a formula from a module. This module logic is then applied in process algebra. We show how auxiliary process algebra operators can be hidden when this is needed. Moreover we demonstrate how new process combinators can be defined in terms of more elementary ones in a clean way. As an illustration of our approach we specify some FIFO-queues and verify several of their properties.

Notes. This is a substantial revision of part of our paper *Modular specifications in process algebra - with curious queues*, Report CS-R8821, Centrum voor Wiskunde en Informatica, Amsterdam 1988, which appeared partly in [46] and entirely in [24]. An extended abstract of that paper has been published as [25]. This revision intends to be an improvement; however, for more applications of our module approach, e.g. to the specification of more curious queues and to the verification of the concurrent alternating bit protocol, as well as for the treatment of many-sorted module logic, we refer to the old version.
The research of the authors was supported by ESPRIT project no. 432, An Integrated Formal Approach to Industrial Software Development (METEOR), and by Sonderforschungsbereich 342 of the Technical University of Munich. The research of the second author was also supported by RACE project no. 1046, Specification and Programming Environment for Communication Software (SPECS).
Most of the work on this revision was done during a visit from the second author at the Technical University of Munich. The current affiliations of the authors are:
R.J. van Glabbeek, Computer Science Department, Stanford University, Stanford, California 94305, USA. rvg@cs.stanford.edu
F.W. Vaandrager, Ecole des Mines, Centre de Mathematiques Appliquees, Sophia Antipolis, 06565 Valbonne Cedex, France. frits@cma.cma.fr

TABLE OF CONTENTS

INTRODUCTION

During the last decade, a lot of research has been done on *process algebra*: the branch of theoretical computer science concerned with the modeling of concurrent systems as elements of an algebra. Besides the Calculus of Communicating Systems (CCS) of MILNER [34, 35], a large number of related formalisms have been developed, such as the theory of Communicating Sequential Processes (CSP) of HOARE [29], the MEIJE calculus of AUSTRY & BOUDOL [3] and the Algebra of Communicating Processes (ACP) of BERGSTRA & KLOP [10, 12, 13]. There are a number of factors that help explain why so many different proposals occur in the literature.

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 [21, 34]. 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.

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 a unique optimal and minimal collection exists. What is optimal depends on the type of results one likes to prove. This becomes even more clear if we look towards practical applications. Some operators in process algebra can be used for a wide range of applications, but we agree with JIFENG & HOARE [30] that we may have to accept that each application will require derivation of specialized 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 here. We only define classes of models of process modules. One reason for doing this is that the semantical notions we refer to are well documented in the literature [10, 27, 29, 35] and a detailed description of all the particular process algebras we use would make this paper needlessly 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. Finally, we would like to stress that the verifications in this paper are completely model independent, and there is no better way of doing that than by presenting no models at all. However, 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 sometimes refer to particular process algebras which have been documented 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.

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 $+$ in a lot of ways. Some combinations are interesting, for other combinations no interesting applications exist. 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 (so-called *trace inconsistencies*, which do not hold in any known process algebra). In BAETEN, BERGSTRA & KLOP [8], for example, it is shown that the combination of failure semantics and the priority operator is trace inconsistent. Another example can be found in BERGSTRA, KLOP & OLDEROG [14], where it is pointed out that the combination of failure semantics and Koomen's Fair Abstraction Rule (KFAR) leads to unwanted identifications.

In the first section we present, besides 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 a large number of process modules which play a role in the ACP framework. Section 3 contains two examples of applications of the new module operators in process algebra:

1. In a setting with internal actions, the left-merge operator of ACP cannot be combined in a trace consistent manner with e.g. the usual laws of failure semantics. 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 a verification takes place by writing the proof on two pages. On one page the left-merge may be used; on the other one the laws of failure semantics can be applied. Then there are some rules that say which intermediate results may be moved from one page to the other.
2. 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. However, we will show that also this problem can be solved in a clean way via the module approach.

The concept of hiding auxiliary operators in a module is quite familiar in the literature (see BERGSTRA, HEERING & KLINT [11]), 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 [42] and SANNELLA & TARLECKI [41], 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 generalize 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 parametrized 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 this paper 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 most process verifications in the ACP framework until now; we take the opportunity to prove its completeness; (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.

As an illustration of the techniques developed in Sections 1-3, we present in Section 4 some examples dealing with FIFO-queues. Amongst others, 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 Sections 1-3 to extend the axiom system in a neat way (avoiding trace inconsistencies) so that we can prove the processes identical.

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 concurrent

systems.

1.1. Statements about concurrent systems. In many theories of concurrency it is common practice to represent processes - the behaviors 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 behavior 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 may 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 may 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 one often wants 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 modeling 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 organize 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 generalization of the notion of a formal proof. Instead of theories we use the more general notion of *module*. Like a theory a module characterizes 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 logic is a sequence or tree of assertions $M \vdash \phi$ such that in each step either the formula ϕ is manipulated, as in a classical proof, 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 *function declaration* is an expression $\mathbb{F}(f,n)$ with $f \in$ NAMES and $n \in \mathbb{N}$.
A function declaration $\mathbb{F}(f, 0)$ of arity 0 is sometimes called a *constant declaration.*
A *predicate declaration* is an expression $\mathbb{R}(p,n)$ with $p \in$ NAMES and $n \in \mathbb{N}$.
A *signature* σ is a set of function and predicate declarations.

1.5. σ-Algebras. Let σ be a signature. A *σ-algebra* $\mathcal{A}$ is a pair of a set $D_{\mathcal{A}}$ and a function on σ that maps

$\mathbb{F}(f,n) \in \sigma$ to a function $f^{\mathcal{A}}: D_{\mathcal{A}}^n \to D_{\mathcal{A}}$ and

$\mathbb{R}(p,n) \in \sigma$ to a predicate $p^{\mathcal{A}} \subseteq D_{\mathcal{A}}^n$.

Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras. $\mathcal{B}$ is a *subalgebra* of $\mathcal{A}$ if $D_{\mathcal{B}} \subseteq D_{\mathcal{A}}$ and if $f^{\mathcal{A}}$ restricted to $D_{\mathcal{B}}^n$ is just $f^{\mathcal{B}}$ for all function and predicate declarations $\mathbb{F}(f,n)$ and $\mathbb{R}(f,n)$ in σ.

A *homomorphism* $h: \mathcal{A} \to \mathcal{B}$ is a mapping $h: D_{\mathcal{A}} \to D_{\mathcal{B}}$ such that

$$h(f^{\mathcal{A}}(x_1, \ldots, x_n)) = f^{\mathcal{B}}(h(x_1), \ldots, h(x_n))$$

for all $\mathbb{F}(f,n) \in \sigma$ and all $x_i \in D_{\mathcal{A}}$ $(i = 1, \ldots, n)$,

$$p^{\mathcal{A}}(x_1, \ldots, x_n) \Leftrightarrow p^{\mathcal{B}}(h(x_1), \ldots, h(x_n))$$

for all $\mathbb{R}(p,n) \in \sigma$ and all $x_i \in D_{\mathcal{A}}$ $(i = 1, \ldots, n)$.

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

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

$$D_{\mathcal{B}} = D_{\mathcal{A}},$$

$f^{\mathfrak{B}} = f^{\mathfrak{A}}$ for all $\mathbb{F}(f,n) \in \rho \cap \sigma$,

$p^{\mathfrak{B}} = p^{\mathfrak{A}}$ for all $\mathbb{R}(p,n) \in \rho \cap \sigma$.

A *congruence* on a σ-algebra $\mathfrak{A}$ is an equivalence relation $\equiv$ on $D_{\mathfrak{A}}$ satisfying, for $x_i, y_i \in D_{\mathfrak{A}}$, the following congruence properties:
- $\forall \mathbb{F}(f,n) \in \sigma$: if $x_i \equiv y_i$ $(i = 1,\ldots,n)$ then $f(x_1,\ldots,x_n) \equiv f(y_1,\ldots,y_n)$,
- $\forall \mathbb{R}(p,n) \in \sigma$: if $x_i \equiv y_i$ $(i = 1,\ldots,n)$ then $p(x_1,\ldots,x_n) \Leftrightarrow p(y_1,\ldots,y_n)$.

(Sometimes (weak) congruences appear in the literature, that are not required to satisfy the congruence properties for predicates.) For $x \in D_{\mathfrak{A}}$, the *congruence class* $(x)_{\equiv}$ of x is the set of all $y \in D_{\mathfrak{A}}$ with $y \equiv x$.
For $\mathfrak{A}$ a σ-algebra and $\equiv$ a congruence on $\mathfrak{A}$, the σ-algebra $\mathfrak{A}/_{\equiv}$, called $\mathfrak{A}$ *modulo* $\equiv$, is defined by

$$D_{\mathfrak{A}/_{\equiv}} = \{(x)_{\equiv} \mid x \in D_{\mathfrak{A}}\},$$

$$f^{\mathfrak{A}/\equiv}((x_1)_{\equiv}, \ldots, (x_n)_{\equiv}) = (f^{\mathfrak{A}}(x_1, \ldots, x_n))_{\equiv},$$

$$((x_1)_{\equiv}, \ldots, (x_n)_{\equiv}) \in p^{\mathfrak{A}/\equiv} \Leftrightarrow (x_1, \ldots, x_n) \in p^{\mathfrak{A}}.$$

Due to the congruence properties this definition is independent of the choice of the representing $x_i \in (x_i)_{\equiv}$.

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* $\dfrac{P}{\phi}$ with $P \subseteq F^{\mathfrak{L}}_{\sigma}$ 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. An inference rule $\dfrac{P}{\phi}$ with $P = \varnothing$ is called an *axiom* and will be denoted simply by ϕ.

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

A formula $\phi \in F^{\mathfrak{L}}_{\sigma}$ is *preserved under subalgebras* if $\mathfrak{A} \models^{\mathfrak{L}}_{\sigma} \phi$ implies $\mathfrak{B} \models^{\mathfrak{L}}_{\sigma} \phi$, for any subalgebra $\mathfrak{B}$ of $\mathfrak{A}$.
A formula $\phi \in F^{\mathfrak{L}}_{\sigma}$ is *preserved under homomorphisms* if $\mathfrak{A} \models^{\mathfrak{L}}_{\sigma} \phi$ implies $\mathfrak{B} \models^{\mathfrak{L}}_{\sigma} \phi$, 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 favorite (and more restricted) definition whenever he/she likes.

In the process algebra verifications of this paper we will use equational logic and sometimes infinitary conditional equational logic. The definition of these logics can be found below. At some places we will refer to first order logic with equality, and therefore a definition of this logic is included in Appendix I.

1.6.1. Variables and terms. Since all logics announced above share the concepts of variables and terms, these will be treated first.
Let $\mathcal{V}$ be a given infinite set of *variables*, disjoint from NAMES.

Let σ be a signature. The set $\mathbb{T}(\sigma)$ of *σ-terms* is defined inductively by:
- $x \in \mathbb{T}(\sigma)$ for any variable x;
- if $\mathbb{F}(f,n)$ is in σ and $t_i \in \mathbb{T}(\sigma)$ for $i = 1,...,n$ then $f(t_1,...,t_n) \in \mathbb{T}(\sigma)$.

A σ-term that contains no variables is called *closed*. We use $T(\sigma)$ to denote the set of closed σ-terms.

A *σ-substitution* (or just *substitution*) is a mapping $\zeta: \mathcal{V} \to \mathbb{T}(\sigma)$. With $t[\zeta]$ we denote the result of simultaneous substitution for $x \in \mathcal{V}$ of $\zeta(x)$ for all occurrences of x in t. With u/x we denote the substitution which maps variable x to u and all other variables to themselves. So $t[u/x]$ is the result of substituting u for all occurrences of x in t.

A *valuation* in a σ-algebra $\mathcal{A}$ is a function ξ that takes every variable x into an element of $D_{\mathcal{A}}$. The *ξ-evaluation* $[\![t]\!]^{\xi} \in D_{\mathcal{A}}$ of a σ-term t in $\mathcal{A}$ is defined by:
- $[\![x]\!]^{\xi} = \xi(x)$,
- $[\![f(t_1,...,t_n)]\!]^{\xi} = f^{\mathcal{A}}([\![t_1]\!]^{\xi},...,[\![t_n]\!]^{\xi})$.

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

Let ϕ be an equation $(t_1 = t_2) \in F_{\sigma}^{eql}$. We say that ϕ is *ξ-true* in a σ-algebra $\mathcal{A}$, notation $\mathcal{A}, \xi \models_{\sigma}^{eql} \phi$, if $[\![t_1]\!]^{\xi} = [\![t_2]\!]^{\xi}$.
ϕ is *true* in $\mathcal{A}$, notation $\mathcal{A} \models_{\sigma}^{eql} \phi$, if $\mathcal{A}, \xi \models_{\sigma}^{eql} \phi$ for all valuations ξ.

An inference system I_{σ}^{eql} for equational logic is displayed in Table 1 below. There t, u and v are terms over σ and x is a variable.

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

TABLE 1

1.6.3. Conditional equational logic. The set F_σ^{at} of *atomic formulas* over σ is defined by:
- $F_\sigma^{eql} \subseteq F_\sigma^{at}$;
- if $\mathbb{R}(p,n)$ is in σ and $t_i \in \mathbb{T}(\sigma)$ for $i=1,...,n$ then $p(t_1,...,t_n) \in F_\sigma^{at}$.

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 $(C \Rightarrow \alpha) \in F_\sigma^{ceql}$.

Let ζ be a substitution. For $\alpha \in F_\sigma^{at}$, $C \subseteq F_\sigma^{at}$ and $\phi \in F_\sigma^{ceql}$, $\alpha[\zeta]$, $C[\zeta]$ and $\phi[\zeta]$ are defined as the result of applying ζ to all terms in α, C, resp. ϕ.

The ξ-*truth* of formulas $\phi \in F_\sigma^{at} \cup F_\sigma^{ceql}$ in a σ-algebra $\mathcal{A}$ is defined by:
- $\mathcal{A},\xi \models_\sigma^{ceql} \phi$ if $\phi \in F_\sigma^{eql}$ and $\mathcal{A},\xi \models_\sigma^{eql} \phi$;
- $\mathcal{A},\xi \models_\sigma^{ceql} p(t_1,...,t_n)$ if $([\![t_1]\!]^\xi,...,[\![t_n]\!]^\xi) \in p^{\mathcal{A}}$;
- $\mathcal{A},\xi \models_\sigma^{ceql} C \Rightarrow \alpha$ 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 2 below. There α and α_i are atomic formulas, C is a set of atomic formulas, ϕ is a conditional equational formula, ζ is a substitution, t, u and v are terms over σ and x is a variable. A conditional equational formula $\varnothing \Rightarrow \alpha$ is denoted by α.

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

TABLE 2

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

$$\frac{\phi}{\phi[\zeta]} \quad \text{can be replaced by} \quad \frac{\phi}{\phi[t/x]}.$$

Furthermore *(in)finitary conditional logic* is obtained by omitting all reference to the equality predicate $=$. Note that all these logics and also the first order logic of Appendix I satisfy the general requirements for logics set out above.

1.6.4. Expressiveness. One can translate an equation $\alpha \in F_\sigma^{eql}$ by a (finitary) conditional equational formula $\varnothing \Rightarrow \alpha$ and a finitary conditional equational formula $\{\alpha_1,...,\alpha_n\} \Rightarrow \alpha$ into a first order formula $(\alpha_1 \wedge \cdots \wedge \alpha_n) \rightarrow \alpha$ (see Appendix I). 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 have incomparable expressive power.

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 labeled by σ-formulas, such that

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

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 $\mathcal{A} \in \mathcal{C}$. With $Alg(\sigma, T)$ we denote 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 by induction. □

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

1.8. Equational logic. This section recalls the soundness and completeness theorem for equational logic, that was first established by BIRKHOFF [16, Theorem 10], and presents an alternative notion of proof that is tailored to equational logic and more convenient in the applications of this paper. We establish that the new concept of proof induces the same notion of provability as the proof concept of Section 1.7.

1.8.1. THEOREM: $T \vdash_\sigma^{eql} \phi \Leftrightarrow Alg(\sigma,T) \models_\sigma^{eql} \phi$.

1.8.2. An *instance* of an equation $(u=v)\in F_\sigma^{eql}$ is an equation $t[u/x][\zeta] = t[v/x][\zeta]$. An *equational σ-proof* of an equation $(u=v)\in F_\sigma^{eql}$ is a string $t_0=t_1=\cdots=t_n$ $(n\in\mathbb{N})$ of σ-terms t_i with $t_0=u$ and $t_n=v$, such that, for $i=1,...,n$, $(t_{i-1}=t_i)$ or $(t_i=t_{i-1})$ is an instance of an equation from T. Write $u \overset{T}{\underset{\sigma}{=}} v$ if such a proof exists.

1.8.3. THEOREM: $u \overset{T}{\underset{\sigma}{=}} v \Leftrightarrow T \vdash_\sigma^{eql} (u=v)$.
PROOF. Straightforward by induction on the size of the proofs. □

1.9. Conditional equational logic. This section presents an alternative notion of proof that is tailored to infinitary conditional equational logic and more convenient in the applications of this paper. We establish that this new concept of proof induces the same notion of provability as the proof concept of Section 1.7, and simultaneously we give a short and elegant proof of the (soundness and) completeness of infinitary conditional equational logic. The completeness of finitary conditional equational logic without predicates has already been proven in SELMAN [43] and in ROBINSON [40].

1.9.1. A *substitution instance* of a conditional equational formula ϕ is a conditional equational formula of the form $\phi[\zeta]$. A *conditional equational σ-proof* of a formula $(C\Rightarrow\alpha)\in F_\sigma^{ceql}$ from a theory $T\subseteq F_\sigma^{ceql}$ is a well-founded, upwardly branching tree of which the nodes are labeled by atomic σ-formulas, such that

- the root is labeled by α, and
- if β is the label of a node q and D is the set of labels of the nodes directly above q then
  - either $\beta\in C$ and $D=\varnothing$, or
  - $D\Rightarrow\beta$ is a substitution instance of a conditional equational formula ϕ from T or the bottom part of Table 2.

An example of a (fragment of) a conditional equational proof is given in

Section 3.1.1.4. Write $C \overset{T}{\underset{\sigma}{\Rightarrow}} \alpha$ if such a proof exists.

1.9.2. THEOREM: *The following three statements are equivalent:*
1. $C \overset{T}{\underset{\sigma}{\Rightarrow}} \alpha$,
2. $T \vdash_\sigma^{ceql} C \Rightarrow \alpha$,
3. $Alg(\sigma, T) \models_\sigma^{ceql} C \Rightarrow \alpha$.

PROOF. The implications (1) $\Rightarrow$ (2) and (2) $\Rightarrow$ (3) can be proved by straightforward induction on the size of proofs. Here we only present the hard part of the proof which is the implication (3) $\Rightarrow$ (1).

Suppose $Alg(\sigma, T) \models_\sigma^{ceql} C \Rightarrow \alpha$. Let $\mathcal{C}$ be the σ-algebra with

$$D_{\mathcal{C}} = \mathbb{T}(\sigma),$$

$$f^{\mathcal{C}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n) \text{ for } \mathbb{F}(f,n) \in \sigma, \text{ and}$$

$$(t_1, \ldots, t_n) \in p^{\mathcal{C}} \text{ iff } (C \overset{T}{\underset{\sigma}{\Rightarrow}} p(t_1, \ldots, t_n)) \text{ for } \mathbb{R}(p,n) \in \sigma.$$

Define the relation $\equiv$ on $D_{\mathcal{C}}$ by: $t \equiv u$ iff $(C \overset{T}{\underset{\sigma}{\Rightarrow}} (t = u))$. By definition of $\overset{T}{\underset{\sigma}{\Rightarrow}}$, $\equiv$ is a congruence, so $\mathcal{C}/_\equiv$ is again a σ-algebra.

Any function $\xi: \mathcal{V} \to \mathbb{T}(\sigma)$ is both a valuation in $\mathcal{C}$ and a σ-substitution. Let $\xi/_\equiv$ be the valuation in $\mathcal{C}/_\equiv$ defined by $\xi/_\equiv(x) = (\xi(x))_\equiv$. Any valuation ξ' in $\mathcal{C}/_\equiv$ is of the form $\xi/_\equiv$ for certain ξ: take $\xi(x) \in \xi'(x)$ for $x \in \mathcal{V}$. By induction one easily establishes that for any $t \in \mathbb{T}(\sigma)$: $[\![t]\!]^{\xi/\equiv} = ([\![t]\!]^{\xi})_\equiv = (t[\xi])_\equiv$.

CLAIM: *For any* $\beta \in F_\sigma^{at}$ *we have:* $(\mathcal{C}/_\equiv, \xi/_\equiv \models \beta) \Leftrightarrow (C \overset{T}{\underset{\sigma}{\Rightarrow}} \beta[\xi])$.

PROOF: Let β be $p(t_1, \ldots, t_n)$ for certain $\mathbb{R}(p,n) \in \sigma$ and $t_1, \ldots, t_n \in \mathbb{T}(\sigma)$. Then

$$\begin{aligned}
\mathcal{C}/_\equiv, \xi/_\equiv \models p(t_1, \ldots, t_n) &\Leftrightarrow ([\![t_1]\!]^{\xi/\equiv}, \ldots, [\![t_n]\!]^{\xi/\equiv}) \in p^{\mathcal{C}/\equiv} \Leftrightarrow \\
&\Leftrightarrow ((t_1[\xi])_\equiv, \ldots, (t_n[\xi])_\equiv) \in p^{\mathcal{C}/\equiv} \Leftrightarrow \\
&\Leftrightarrow (t_1[\xi], \ldots, t_n[\xi]) \in p^{\mathcal{C}} \Leftrightarrow \\
&\Leftrightarrow (C \overset{T}{\underset{\sigma}{\Rightarrow}} p(t_1[\xi], \ldots, t_n[\xi])) \Leftrightarrow \\
&\Leftrightarrow (C \overset{T}{\underset{\sigma}{\Rightarrow}} p(t_1, \ldots, t_n)[\xi]).
\end{aligned}$$

Now let β be $(t_1 = t_2)$ for certain $t_1, t_2 \in \mathbb{T}(\sigma)$. Then:

$$\begin{aligned}
\mathcal{C}/_\equiv, \xi/_\equiv \models (t_1 = t_2) &\Leftrightarrow [\![t_1]\!]^{\xi/\equiv} = [\![t_2]\!]^{\xi/\equiv} \Leftrightarrow \\
&\Leftrightarrow (t_1[\xi])_\equiv = (t_2[\xi])_\equiv \Leftrightarrow t_1[\xi] \equiv t_2[\xi] \Leftrightarrow \\
&\Leftrightarrow (C \overset{T}{\underset{\sigma}{\Rightarrow}} p(t_1[\xi] = t_2[\xi])) \Leftrightarrow
\end{aligned}$$

$$\Leftrightarrow \ (C \overset{T}{\underset{\sigma}{\Rightarrow}} p(t_1 = t_2)[\xi]). \ \square$$

Using this claim we prove that $\mathcal{C}/_{\equiv}$ is a model of T. Suppose $(D \Rightarrow \gamma) \in T$ and $\xi/_{\equiv}$ is a valuation in $\mathcal{C}/_{\equiv}$ with $\mathcal{C}/_{\equiv}, \xi/_{\equiv} \models \beta$ for all $\beta \in D$. Then $C \overset{T}{\underset{\sigma}{\Rightarrow}} \beta[\xi]$ for all $\beta \in D$. The corresponding proof trees (one for each $\beta \in D$) can be combined into one big proof tree, proving $C \overset{T}{\underset{\sigma}{\Rightarrow}} \gamma[\xi]$, by applying the substitution instance $(D \Rightarrow \gamma)[\xi]$ of the conditional equational formula $(D \Rightarrow \gamma) \in T$. Hence $\mathcal{C}/_{\equiv}, \xi/_{\equiv} \models \gamma$, which had to be proved.

Thus $\mathcal{C}/_{\equiv} \in Alg(\sigma, T)$, so we have $\mathcal{C}/_{\equiv}, \xi' \models (C \Rightarrow \alpha)$ for any valuation ξ' in $\mathcal{C}/_{\equiv}$. In particular this holds for $\xi' = id/\equiv$, with id the identity valuation in $\mathcal{C}$, defined by $id(x) = x$ for $x \in \mathcal{V}$. By means of a trivial proof (a one-node-only proof tree) one establishes that $C \overset{T}{\underset{\sigma}{\Rightarrow}} \beta[id]$ for any $\beta \in C$. Hence, by the claim, $\mathcal{C}/_{\equiv}, id/_{\equiv} \models \beta$ for $\beta \in C$. Thus, by the definition of truth in conditional equational logic, we must have $C/_{\equiv}, id/_{\equiv} \models \alpha$. Applying the claim once more gives $C \overset{T}{\underset{\sigma}{\Rightarrow}} \alpha$. $\square$

1.10. Module logic.

1.10.1. The set $\mathfrak{M}$ of *module expressions* is defined inductively by:

- If σ is a signature and T a theory over σ, then $(\sigma, T) \in \mathfrak{M}$,
- If $M \in \mathfrak{M}$ 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 organize 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 [11], where 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 [36].

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

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

- $\Sigma(S(M)) = \Sigma(M)$.

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

Note that $Alg(M)$ is a generalization of $Alg(\sigma, T)$ as defined earlier. All the elements of $Alg(M)$ are $\Sigma(M)$-algebras. A $\Sigma(M)$-algebra $\mathcal{A}\in Alg(M)$ is said to *satisfy* M. A formula $\phi\in F^{\mathfrak{L}}_{\Sigma(M)}$ is *satisfied* by a module expression 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. A theory $T \subseteq F^{\mathfrak{L}}_{\Sigma(M)}$ is *satisfied* by a module expression M, $M \models^{\mathfrak{L}} T$, if $M \models^{\mathfrak{L}} \phi$ for all $\phi\in T$.

1.10.4. DERIVABILITY. A *proof* of a formula $\phi\in F^{\mathfrak{L}}_{\Sigma(M)}$ from a module expression M using the logic $\mathfrak{L}$, is a well-founded, upwardly branching tree of which the nodes are labeled by assertions $N \vdash \psi$, such that
- the root is labeled by $M \vdash \phi$, and
- if $N \vdash \psi$ is the label of a node q and P is the set of labels of the nodes directly above q, then $\dfrac{P}{N \vdash \psi}$ is one of the inference rules of Table 3.

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. 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. 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 $M \vdash^{\mathfrak{L}} \phi$.

$(\sigma,T)\vdash\phi$	if $\phi\in T$
$\dfrac{M\vdash\phi_j\ (j\in J)}{M\vdash\phi}$	whenever $\dfrac{\phi_j\ (j\in J)}{\phi}\in I^{\mathfrak{L}}_{\Sigma(M)}$
$\dfrac{M\vdash\phi}{M+N\vdash\phi}$	$\dfrac{N\vdash\phi}{M+N\vdash\phi}$
$\dfrac{M\vdash\phi}{\sigma\Box M\vdash\phi}$	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 3

1.10.5. LEMMA. *If* $M\vdash^{\mathfrak{L}}\phi$ *then* $\phi\in F^{\mathfrak{L}}_{\Sigma(M)}$.
PROOF. By induction. The only nontrivial cases are the rules for $+$ and $\Box$. 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. □

1.10.6. SOUNDNESS THEOREM. *If* $\mathfrak{L}$ *is sound then* $M\vdash^{\mathfrak{L}}\phi$ *implies* $M\models^{\mathfrak{L}}\phi$.
PROOF. By induction to the size of proof trees. Suppose $M\vdash^{\mathfrak{L}}\phi$ by means of a proof tree whose last step is an application of an inference rule $\dfrac{P}{M\vdash\phi}$ from Table 3. By induction we may assume that for all assertions $N\vdash\psi$ in P, being the roots of smaller proof trees, we have $N\models^{\mathfrak{L}}\psi$. It has to be established that $M\models^{\mathfrak{L}}\phi$, i.e. that for all models $\mathfrak{A}$ of M we have $\mathfrak{A}\models^{\mathfrak{L}}_{\Sigma(M)}\phi$. Now six cases can be distinguished.

1. $P=\varnothing$, $M=(\sigma,T)$ and $\phi\in T$.
   Let $\mathfrak{A}$ be a model of $M=(\sigma,T)$. Then $\mathfrak{A}\models^{\mathfrak{L}}_{\Sigma(M)}\phi'$ for all $\phi'\in T$, so in particular $\mathfrak{A}\models^{\mathfrak{L}}_{\Sigma(M)}\phi$.
2. $P=\{M\vdash\phi_j\mid j\in J\}$ and $\dfrac{\phi_j\ (j\in J)}{\phi}\in I^{\mathfrak{L}}_{\Sigma(M)}$.
   Let $\mathfrak{A}$ be a model of M. By induction $\mathfrak{A}\models^{\mathfrak{L}}_{\Sigma(M)}\phi_j$ for $j\in J$. Thus $\mathfrak{A}\models^{\mathfrak{L}}_{\Sigma(M)}\phi$, since $\mathfrak{L}$ is sound.
3. $M=N_1+N_2$ and $P=\{N_i\vdash\phi\}$ $(i=1$ or $2)$.
   By Lemma 1.10.5, $\phi\in F^{\mathfrak{L}}_{\Sigma(N_i)}=F^{\mathfrak{L}}_{\Sigma(N_i)\cap\Sigma(M)}$. Let $\mathfrak{A}$ be a model of

$M=\{N_1+N_2\}$. Then $\Sigma(N_i)\Box\mathcal{A}$ is a model of N_i, hence, by induction, $\Sigma(N_i)\Box\mathcal{A} \models^{\mathcal{L}}_{\Sigma(N_i)\cap\Sigma(M)} \phi$. Taking $\sigma=\Sigma(N_i)$ and $\rho=\Sigma(M)$ in the requirement of a logic that for all ρ-algebras $\mathcal{A}$ and $\phi\in F^{\mathcal{L}}_{\sigma\cap\rho}$: $\sigma\Box\mathcal{A} \models^{\mathcal{L}}_{\sigma\cap\rho} \phi \Rightarrow \mathcal{A} \models^{\mathcal{L}}_{\rho} \phi$, we find $\mathcal{A} \models^{\mathcal{L}}_{\Sigma(M)} \phi$.

4. $M=\sigma\Box N$, $P=\{N \vdash \phi\}$ and $\phi\in F^{\mathcal{L}}_{\sigma}$.
By Lemma 1.10.5, $\phi\in F^{\mathcal{L}}_{\Sigma(N)}$, so $\phi\in F^{\mathcal{L}}_{\sigma}\cap F^{\mathcal{L}}_{\Sigma(N)}=F^{\mathcal{L}}_{\sigma\cap\Sigma(N)}$. Let $\mathcal{A}$ be a model of $M=\sigma\Box N$. Then there is a model $\mathcal{B}$ of N with $\mathcal{A}=\sigma\Box\mathcal{B}$. By induction, $\mathcal{B} \models^{\mathcal{L}}_{\Sigma(N)} \phi$. Taking $\rho=\Sigma(N)$ in the requirement of a logic that for all ρ-algebras $\mathcal{A}$ and $\phi\in F^{\mathcal{L}}_{\sigma\cap\rho}$: $\sigma\Box\mathcal{A} \models^{\mathcal{L}}_{\sigma\cap\rho} \phi \Leftarrow \mathcal{A} \models^{\mathcal{L}}_{\rho} \phi$, we find $\sigma\Box\mathcal{B} \models^{\mathcal{L}}_{\sigma\cap\Sigma(N)} \phi$, i.e. $\mathcal{A} \models^{\mathcal{L}}_{\Sigma(M)} \phi$.

5. $M=H(N)$, $P=\{N \vdash \phi\}$ and ϕ is positive.
Let $\mathcal{A}$ be a model of $M=H(N)$, i.e. $\mathcal{A}$ is the homomorphic image of a model $\mathcal{B}$ of N. By induction $\mathcal{B} \models^{\mathcal{L}}_{\Sigma(M)} \phi$ and since ϕ is positive $\mathcal{A} \models^{\mathcal{L}}_{\Sigma(M)} \phi$.

6. $M=S(N)$, $P=\{N \vdash \phi\}$ and ϕ is universal.
Let $\mathcal{A}$ be a model of $M=S(N)$, i.e. $\mathcal{A}$ is a subalgebra of a model $\mathcal{B}$ of N. By induction, $\mathcal{B} \models^{\mathcal{L}}_{\Sigma(M)} \phi$ and since ϕ is universal $\mathcal{A} \models^{\mathcal{L}}_{\Sigma(M)} \phi$. □

1.10.7 Modules. This paper deals with module expressions, which are syntactic objects, rather than *modules*, in the sense of semantic objects which are denoted by module expressions. Nevertheless it may be helpful to have an idea what these modules could be. Here several options are available (see for instance BERGSTRA, HEERING & KLINT [11]). One possibility is that a module expression denotes the class of its models, or the class of its countable models. Another possibility is that it denotes the set of all formulas which are derivable from it. In each of these cases the operators of the module language can be easily interpreted (see the Subsections 1.10.3 and 1.10.4 above). The simplest and least abstract interpretation however, which may be considered to be the one employed in this paper, is the one in which a module is an annotated and structured collection of axiom systems: a module expression simply denotes itself.

1.11. Completeness. All logics mentioned in Section 1.6 and Appendix I are sound and complete:

$$Alg(\sigma,T) \models^{\mathcal{L}}_{\sigma} \phi \Leftrightarrow T \vdash^{\mathcal{L}}_{\sigma} \phi. \qquad (1)$$

As a corollary, using 1.6.4, we have

$$T \vdash^{eql}_{\sigma} \phi \Leftrightarrow T \vdash^{ceql}_{\sigma} \phi \qquad \text{for } \phi\in F^{eql}_{\sigma} \text{ and} \qquad (2)$$

$$T \vdash_\sigma^{ceql} \phi \Leftrightarrow T \vdash_\sigma^{foleq} \phi \quad \text{for } \phi \in F_\sigma^{fceql}. \tag{3}$$

For this reason in most 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 module expressions. Equational logic and conditional equational logic are not complete anymore (for counterexamples see Sections 3.1.1.5 resp. 3.2.4) and for first order logic with equality this is still an open problem (as far as we know). Here a logic $\mathfrak{L}$ is complete if $M \models^{\mathfrak{L}} \phi \Rightarrow M \vdash^{\mathfrak{L}} \phi$. It is easily proved that

$$M \vdash^{eql} \phi \Rightarrow M \vdash^{ceql} \phi \quad \text{for } \phi \in F_{\Sigma(M)}^{eql} \text{ and} \tag{4}$$

$$M \vdash^{ceql} \phi \Rightarrow M \vdash^{foleq} \phi \quad \text{for } \phi \in F_{\Sigma(M)}^{fceql}, \tag{5}$$

but the reverse directions do not hold (as is also shown by the counterexamples of Sections 3.1.1.5 and 3.2.4). Thus we should state exactly in which logic our results are proved.

1.12. Towards applications. This paper employs infinitary conditional equational module logic. However, proofs in the sense of Section 1.10 will hardly occur. In this section we show that in most applications we may use simpler proof methods.

1.12.1. LEMMA: *If $M \vdash^{\mathfrak{L}} \phi$ has been established before, then $M \vdash \phi$ may occur as a leaf in a proof tree. Likewise, if $T \vdash_\sigma^{\mathfrak{L}} \phi$ has been established before, then ϕ may be used in the construction of a proof as if it were a member of T.*

PROOF. The figurative leaf can be expanded into a proof tree in order to make the proof correct. □

1.12.2. LEMMA: $(\sigma_0, T_0) + \cdots + (\sigma_n, T_n) \vdash^{\mathfrak{L}} \phi \Leftrightarrow (\bigcup_{i=0}^{n} \sigma_i, \bigcup_{i=0}^{n} T_i) \vdash^{\mathfrak{L}} \phi$.

PROOF. "⇒": Replace all modules that appear on the left of turn-styles in a proof of ϕ from $(\sigma_0, T_0) + \cdots + (\sigma_n, T_n)$ by $(\bigcup_{i=0}^{n} \sigma_i, \bigcup_{i=0}^{n} T_i)$ and remove the applications of the inference rules for $+$.

"⇐": Also straightforward. □

1.12.3. LEMMA: $(\sigma,T) \vdash^{\mathcal{L}} \phi \Leftrightarrow T \vdash^{\mathcal{L}}_{\sigma} \phi$.
PROOF. Trivial. □

These three lemmas say that (parts of) proofs from modules that do not contain the operators $\Box$, H or S may be given in classical logic instead of module logic. In case of conditional equational logic, a conditional equational proof can then be given. Alternatively, using Lemma 1.12.1 again and observation 1.11.2, any part of a conditional equational proof that deals with equations only, may be given in equational logic instead. For these parts - which are most numerous in this paper - the equational proof method of Section 1.8 applies.

1.13. Notation. In this paper a module (σ,T) will sometimes be introduced by mentioning only T. In such cases σ is understood to be the signature of all functions and predicates appearing in T. Outside this Section 1 and Appendix I inference rules $\frac{P}{\phi}$ 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: $\frac{P}{\phi}$ holds in an algebra $\mathcal{A}$ if ($\mathcal{A},\xi \models \psi$ for all $\psi \in P$ and all valuations ξ) implies ($\mathcal{A},\xi \models \phi$ for all valuations ξ), while $\frac{C}{\alpha}$ holds in $\mathcal{A}$ if for all valuations ξ: ($\mathcal{A},\xi \models \beta$ for all $\beta \in C$ implies $\mathcal{A},\xi \models \alpha$).

2. PROCESS ALGEBRA

This is not an introductory paper on process algebra. Our only aim here is to show how the ACP formalism can be presented in a precise and structured way using our notion of a module. For a comprehensive presentation of the ACP formalism we refer the reader to BAETEN & WEIJLAND [10]. We will use as much as possible the notations and terminology from this reference.

2.1. ACP$^{\tau}$. In this paper a central role will be played by the module ACP$^{\tau}$, the Algebra of Communicating Processes with abstraction. A first parameter of ACP$^{\tau}$ is a finite set A of *actions*. For each action a in A there is a constant a in the language, representing the process which starts with an a-action and terminates (successfully) after some time.

The first two composition operations we consider are $\cdot$, denoting

sequential composition, and $+$ for *alternative composition*. If x and y are two processes, then $x \cdot y$ is the process that starts execution of y after successful completion of x, and $x+y$ is the process that either behaves like x or like y. We do not specify whether the choice between x and y is made by the process itself, or by the environment.

We have a special constant δ, denoting *deadlock, inaction,* a process that does nothing at all. In particular δ does not terminate successfully. We write $A_\delta = A \cup \{\delta\}$.

Next we have a *parallel composition* operator $\|$. $x\|y$ denotes the process corresponding to the parallel execution of x and y. Execution of $x\|y$ either starts with a step from x, or with a step from y, or with a *synchronization* of an action from x and an action from y. Synchronization of actions is described by the second parameter of ACP^τ, which is a partial *communication function* $\gamma: A \times A \to A$ which is commutative and associative:

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

In the above equations we also imply that one side of an equation is defined exactly when the other side is. With $\gamma(a,b)\downarrow$ we indicate that $\gamma(a,b)$ is defined. If, for some $a,b,c \in A$, $\gamma(a,b)=c$ this means that actions a and b can synchronize. 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: $\mathrm{ACP}^\tau(A,\gamma)$. However, in order to keep notation simple, we will always omit the parameters if this can be done without causing confusion. In order to axiomatize the $\|$-operator we use two auxiliary operators $\lfloor\!\lfloor$ (*left-merge*) and $|$ (*communication merge*). $x\lfloor\!\lfloor y$ is $x\|y$, but with the restriction that the first step comes from x, and $x|y$ is $x\|y$ but with a synchronization action as the first step.

Next we have for each $H \subseteq A$ an *encapsulation* operator ∂_H. The operator ∂_H blocks actions from H. The operator is used to encapsulate a process, i.e. to block synchronization with the environment.

When describing concurrent systems and reasoning about their behavior, it is often useful to have a distinguished action that cannot synchronize with any other action. Such an action is denoted by the constant $\tau \notin A_\delta$. We write A_δ^τ for $A \cup \{\delta,\tau\}$. The fact that τ cannot synchronize makes that in some sense this action is not observable. Therefore it is often called the *silent* action. For each $I \subseteq A$ the language contains an *abstraction* or *hiding* operator τ_I. This operator hides actions in I by renaming them into τ, thus expressing that certain actions in a system

behavior cannot be observed.

In Table 4 we summarize the signature of module ACP$^\tau$.

binary operators	$+$	alternative composition (sum)
	$\cdot$	sequential composition (product)
	$\|\|$	parallel composition (merge)
	$\lfloor\!\lfloor$	left-merge
	$\|$	communication merge
unary operators	∂_H	encapsulation, for any $H \subseteq A$
	τ_I	abstraction, for any $I \subseteq A$
constants	a	for any atomic action $a \in A$
	δ	inaction, deadlock
	τ	silent action

TABLE 4

Table 5 contains the theory of the module ACP$^\tau$.

$x+y = y+x$	A1	$x\tau = x$	B1
$x+(y+z) = (x+y)+z$	A2	$x(\tau(y+z)+y) = x(y+z)$	B2
$x+x = x$	A3		
$(x+y)z = xz+yz$	A4		
$(xy)z = x(yz)$	A5		
$x+\delta = x$	A6	$a\|b = \gamma(a,b)$ if $\gamma(a,b)\downarrow$	CF1
$\delta x = \delta$	A7	$a\|b = \delta$ otherwise	CF2
$x\|\|y = x\lfloor\!\lfloor y + y\lfloor\!\lfloor x + x\|y$	CM1	$(ax)\|b = (a\|b)x$	CM5
$a\lfloor\!\lfloor x = ax$	CM2	$a\|(bx) = (a\|b)x$	CM6
$(ax)\lfloor\!\lfloor y = a(x\|\|y)$	CM3	$(ax)\|(by) = (a\|b)(x\|\|y)$	CM7
$(x+y)\lfloor\!\lfloor z = x\lfloor\!\lfloor z + y\lfloor\!\lfloor z$	CM4	$(x+y)\|z = x\|z + y\|z$	CM8
		$x\|(y+z) = x\|y + x\|z$	CM9
$\partial_H(a) = a$ if $a \notin H$	D1	$\tau_I(a) = a$ if $a \notin I$	TI1
$\partial_H(a) = \delta$ if $a \in H$	D2	$\tau_I(a) = \tau$ if $a \in I$	TI2
$\partial_H(x+y) = \partial_H(x)+\partial_H(y)$	D3	$\tau_I(x+y) = \tau_I(x)+\tau_I(y)$	TI3
$\partial_H(xy) = \partial_H(x)\cdot\partial_H(y)$	D4	$\tau_I(xy) = \tau_I(x)\cdot\tau_I(y)$	TI4

TABLE 5

All axioms in Table 5 are in fact axiom schemes in a, b, H and I. Here a

and b range over A_δ^τ (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. In this paper we present ACP$^\tau$ as a monolithic module. In [10] however, it is shown that ACP$^\tau$ 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$_\delta$, and BPA$_\delta$ plus axioms B1-2 gives BPA$_\delta^\tau$. The module ACP consists of the axioms A1-7, CF1-2, CM1-9 and D1-4.

2.1.1. Note. Let $D = \{d_1,...,d_n\}$ be a finite set and let $t_{d_1},...,t_{d_n}$ be process expressions. We use the notation $\sum_{d\in D} t_d$ for the expression $t_{d_1} + \cdots + t_{d_n}$. $\sum_{d\in\varnothing} t_d = \delta$ by definition.

2.1.2. PROPOSITION: ACP$^\tau \vdash a(\tau x \| y) = a(x \| y)$.

PROOF. $a(\tau x \| y) \overset{B1}{=} a\tau(\tau x \| y) \overset{CM3}{=} a(\tau\tau x \mathbin{\llcorner} y) \overset{B1}{=} a(\tau x \mathbin{\llcorner} y) \overset{CM3}{=} a\tau(x \| y) \overset{B1}{=} a(x \| y)$. □

2.1.3. Summand inclusion. In process verifications the summand inclusion predicate $\subseteq$ turns out to be a useful notation. We write $x \subseteq y$ as an abbreviation for $x+y=y$. From the ACP$^\tau$-axioms A1, A2 and A3 respectively it follows that $\subseteq$ is antisymmetrical, transitive and reflexive, and hence a partial order.

Most of the operators of ACP$^\tau$ are monotonic with respect to the summand inclusion ordering. For instance, if f is a unary ACP$^\tau$ operator, then we can prove the following conditional formula:

$$\frac{x \subseteq y}{f(x) \subseteq f(y)}.$$

For reasons to be explained in Section 3, we prefer to state properties of ACP$^\tau$-processes in an equational rather than a conditional form. Therefore we write the above monotonicity property as

$$f(x) \subseteq f(x+y).$$

The reader may check that both formulas are equivalent in a setting with the laws A1-3. Using essentially the distributivity of the operators over $+$, one can prove from ACP$^\tau$:

- $x+z \subseteq x+y+z$,

- $x \cdot z \subseteq (x+y) \cdot z$,
- $x \between z \subseteq (x+y) \between z$,
- $x \mid z \subseteq (x+y) \mid z$,
- $\partial_H(x) \subseteq \partial_H(x+y)$,
- $\tau_I(x) \subseteq \tau_I(x+y)$.

Due to branching time (see [23]), we cannot prove $z \cdot x \subseteq z \cdot (x+y)$, $x \| z \subseteq (x+y) \| z$ or $z \between x \subseteq z \between (x+y)$. However, we do have the following weak form of monotonicity for the merge operator:

- $\tau(x \| z) \subseteq (\tau x + y) \| z$.

The last formula formula follows since

$$\tau(x \| z) = \tau x \between z \subseteq (\tau x + y) \between z \subseteq (\tau x + y) \| z.$$

2.2. Standard Concurrency. Often one adds to ACP^τ the following module SC of Standard Concurrency. These axioms are not included in ACP^τ because they can be derived for all closed ACP^τ-terms.

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

TABLE 6

2.2.1. PROPOSITION. $ACP^\tau + SC \vdash$
(i) $x \| y = y \| x$,
(ii) $x \| (y \| z) = (x \| y) \| z$.
PROOF. See [10]. □

2.3. Renamings. For every function $f: A_\delta^\tau \to A_\delta^\tau$ with the property that $f(\delta) = \delta$ and $f(\tau) = \tau$, the module RN introduces a unary operator ρ_f. Axioms for ρ_f are presented in Table 7 (here a ranges over A_δ^τ and id is the identity function). Module RN is parametrized 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 7

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

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

In the rest of this paper 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 the possibility of an infinite sequence of internal actions corresponding to hidden synchronizations between the two arguments of the operator (a form of *unguarded recursion*, cf. Section 2.9.1). In order to deal with such behaviors, it is useful to view $\gg$ as the composition of two operators: the $\ggg$ operator and an abstraction operator that hides the internal communications. We will define the chaining operators in terms of the operators of $ACP^\tau + RN$. In this way we obtain a simple, finite axiomatization of the operators. The operator $\gg$ occurs (in different notations) already in HOARE [28] and MILNER [34].

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 of actions:

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

Here $r(d)$, $s(d)$ and $c(d)$ $(d \in D)$ are auxiliary actions used in the definition of the chaining operators. The module for the chaining operators is parametrized by an action alphabet A satisfying $ch(D) \subseteq A$. The module should be used in a context with modules $\mathrm{ACP}^{\tau}(A, \gamma)$ and $\mathrm{RN}(A)$ 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 on or with $ch(D)$ are undefined). The renaming functions $\uparrow s$ and $\downarrow r$ are given 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 $a \in A_{\delta}^{\tau}$. Now the 'concrete' chaining of processes x and y, notation $x \ggg y$, is defined by means of the axiom $(H = \{s(d), r(d) \mid d \in D\})$:

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

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

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

The module CH^{+} consists of axioms CH1 and CH2, and is parametrized by A. The '+' in CH^{+} refers to the auxiliary actions in the module, which will be removed in Section 3.2.

2.4.1. The following laws can be easily proven from module ACP^{τ} + SC + RN + CH^{+}. Here $d0$ and $e0$ are elements of D and d and e are variables ranging over D. L5 follows using Proposition 2.1.2.

$$\uparrow d0 \cdot x \gg (\sum_{e \in D} \downarrow e \cdot y^{e}) = \tau \cdot (x \gg y^{d0}) \qquad \text{L1}$$

$$\uparrow d0 \cdot x \gg \uparrow e0 \cdot y = \uparrow e0 \cdot (\uparrow d0 \cdot x \gg y) \qquad \text{L2}$$

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

$$(\sum_{d \in D} \downarrow d \cdot x^{d}) \gg \uparrow e0 \cdot y = \sum_{d \in D} \downarrow d \cdot (x^{d} \gg \uparrow e0 \cdot y) + \uparrow e0 \cdot ((\sum_{d \in D} \downarrow d \cdot x^{d}) \gg y) \qquad \text{L4}$$

$a(\tau x \gg y) = a(x \gg \tau y) = a(x \gg y)$ L5

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 $c(d0)$.

2.4.2. EXAMPLE. Let $D=\{0,1\}$. Process AND reads two bits and then outputs 1 if both are 1, and 0 otherwise:

$$AND = \downarrow 0 \cdot (\downarrow 0 \cdot \uparrow 0 + \downarrow 1 \cdot \uparrow 0) + \downarrow 1 \cdot (\downarrow 0 \cdot \uparrow 0 + \downarrow 1 \cdot \uparrow 1).$$

Process OR reads two bits, outputs 0 if both are 0, and 1 otherwise:

$$OR = \downarrow 0 \cdot (\downarrow 0 \cdot \uparrow 0 + \downarrow 1 \cdot \uparrow 1) + \downarrow 1 \cdot (\downarrow 0 \cdot \uparrow 1 + \downarrow 1 \cdot \uparrow 1).$$

Process NEG reads a bit b and outputs $1-b$:

$$NEG = \downarrow 0 \cdot \uparrow 1 + \downarrow 1 \cdot \uparrow 0.$$

These processes can be composed using chaining operators. It is not too hard to prove from $\mathrm{ACP}^{\tau} + \mathrm{RN} + \mathrm{CH}^{+}$:

$$(NEG \cdot NEG \gg AND) \gg NEG = OR$$

Note however that we do not have

$$(NEG \cdot NEG \ggg AND) \ggg NEG = OR$$

since in the LHS process internal computation steps are still visible.

2.5. Projection. The unary operators π_n ($n \in \mathbb{N}$) *stop* processes after they have performed n atomic actions, with the understanding that τ-steps are transparent. The axioms for π_n are presented in Table 8, where a ranges over A_δ. Module PR is parametrized 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 8

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 proofs, is to introduce a single binary projection operator $\mathbb{F}:\pi:\mathbb{N}\times P\to P$ (here P is the sort of processes).

2.6. Alphabets. Intuitively, the alphabet of a process is the set of atomic actions which it can perform. This idea is formalized in [6], where an operator α from processes to sets of actions 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 again: α is an operator from processes to processes; $\alpha(x)$ 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 $B=_{def}\sum_{b\in B}b$. 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 2.1.3), fits with the notation for the subset predicate on sets.

The following axioms in Table 9 can be used to compute the alphabet of finite processes. In the table a ranges over the set A of actions which occurs as a parameter of module AB.

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 9

In order to compute the alphabet of infinite processes, we introduce an

additional module AA which is also parametrized by A.

AA		
	$\alpha(x)\subseteq A$	AA1
	$\alpha(x\|\|y)=\alpha(x)+\alpha(y)+\alpha(x)\|\alpha(y)$	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 10

It is not hard to see that the axioms of AA are derivable for all closed BPA_δ^τ-terms.

2.6.1. EXAMPLE (from [6]). Let P,Q,R be processes satisfying $P=a\cdot P$, $Q=\tau_{\{a\}}(P)$ and $R=Q\cdot b$ (with $b\neq a$). We derive the alphabet of R.

$$\begin{aligned}\alpha(R) &= \alpha(Qb) = \alpha(\tau_{\{a\}}(P)\cdot b) = \alpha(\tau_{\{a\}}(P)\cdot\tau_{\{a\}}(b)) = \\ &= \alpha(\tau_{\{a\}}(Pb)) \overset{AA3}{\subseteq} \tau_{\{a\}}\circ\partial_{\{a\}}\circ\alpha(Pb) \overset{RN5}{=} \partial_{\{a\}}\circ\alpha(Pb).\end{aligned}$$

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 generalization of the conditional axioms of [6], occur in a slightly different form also in [45]. Rules like these are an important tool in system verifications based on process algebra. Module RR is parametrized by A and γ. Observe that axioms AA1 and RR1 together imply axiom RN4 of Table 7. Axiom RR2, which

$$\frac{\alpha(x)\subseteq \boldsymbol{B}}{\rho_f(x)=x}\forall b\in B: f(b)=b \qquad \text{RR1}$$

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

TABLE 11

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 terms.

2.7. 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. 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 $<x\mid E>$, denoting the x-component of a solution of E. Here 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.

In most applications the variables $X\in V_E$ in a recursive specification E will be chosen freshly, so that there is no need to repeat E in each occurrence of $<X\mid E>$. Therefore the convention will be adopted that once a recursive specification has been declared, $<X\mid E>$ 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 $<X\mid X=aX> = aa<X\mid X=aX>$.

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

$$<x\mid E> = <t_x\mid E> \qquad \text{REC}$$

If the above convention is used, these formulas seem to be just the equations of E. The module REC is parametrized 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.

2.8. Boundedness. The unary predicates B_n ($n \in \mathbb{N}$) state that the non-determinism displayed by a process before its n^{th} atomic steps is bounded. If for all $n \in \mathbb{N}$: $B_n(x)$, we say x is bounded. Axioms for B_n are in Table 12. Module B is parametrized by A. In the table a ranges over $a \in A_\delta$.

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 12

Boundedness predicates were introduced in [22].

2.9. Approximation Induction Principle. AIP$^-$ is a proof rule which is vital if we want to prove statements 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.

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

The '$-$' in AIP$^-$, distinguishes the rule from a variant without predicates B_n (see [22]).

2.9.1. DEFINITION. Let t be an open BPA$^\tau_\delta$-term. An occurrence of a variable x in t is *guarded* if t has a subterm of the form $a \cdot t'$, with $a \in A$, and this x occurs in t'. 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 BPA$^\tau_\delta$-terms. 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 $x \xrightarrow{u} y \xrightarrow{u} z \xrightarrow{u} \cdots$).

2.9.2. THEOREM *(Recursive Specification Principle* (RSP)*): Let E be a guarded recursive specification over the signature of BPA_δ^τ and let $x \in V_E$. Then:* $\mathrm{BPA}_\delta^\tau + \mathrm{REC} + \mathrm{PR} + \mathrm{B} + \mathrm{AIP}^- \vdash$

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

In earlier papers on process algebra, RSP referred to the assumption, stated in plain English, that a guarded recursive specification has at most one solution. The RSP we present here says exactly the same in the language of infinitary conditional equational logic. Because suppose we have two collections of processes $\{p_y | y \in V_E\}$ and $\{q_y | y \in V_E\}$ which both are a solution of the equations of E, that is, if we substitute for each occurrence of a variable y in E the corresponding processes p_y resp. q_y, then we obtain valid statements. In such a situation RSP allows us to conclude that $p_x = \langle x|E\rangle$ and also that $q_x = \langle x|E\rangle$. Here it is important to note that $\langle x|E\rangle$ is a constant and that we are not allowed to substitute for variables occurring in it. Consequently we have for each $y \in V_E$: $p_y = q_y$. This means that E has a unique solution.

2.9.3. EXAMPLE. Let

$$E = \{X = (a+b)\cdot X\} \quad \text{and} \quad F = \{Y = a\cdot(a+b)\cdot Y + b\cdot Y\}$$

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 $\langle X|E\rangle = \langle Y|F\rangle$.

2.10. $\mathrm{ACP}_\sharp^\tau$. The combination of all modules presented thus far will be called $\mathrm{ACP}_\sharp^\tau$. Formally, the module is defined by:

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

Branching bisimulation semantics, as described in for instance [10], gives

a model for the module $\mathrm{ACP}_{\sharp}^{\tau}$.

2.11. RSP^{+}. For many verifications RSP is not really practical and one would like to use a more powerful principle. Therefore, we will present below a more general version of RSP, which we will call RSP^{+}. Section 4 contains a number of examples which illustrate the use of RSP and RSP^{+}.

2.11.1. DEFINITION. A process expression $t \in T(\Sigma(\mathrm{ACP}_{\sharp}^{\tau}))$ is called *guardedly specifiable* if there exists a guarded recursive specification E with $x \in V_E$ such that

$$\mathrm{ACP}_{\sharp}^{\tau} \vdash t = \langle x \mid E \rangle.$$

2.11.2. THEOREM *(Generalized Recursive Specification Principle* (RSP^{+})*): Let* $\langle x \mid E \rangle$ *be a guardedly specifiable process expression. Then:* $\mathrm{ACP}_{\sharp}^{\tau} \vdash$

$$(\mathrm{RSP}^{+}) \quad \frac{E}{x = \langle x \mid E \rangle}$$

PROOF. Suppose $\langle x \mid E \rangle$ is a guardedly specifiable process expression. Then there exists a guarded recursive specification F with $y \in V_F$ such that

$$\mathrm{ACP}_{\sharp}^{\tau} \vdash \langle x \mid E \rangle = \langle y \mid F \rangle.$$

Let p be a conditional equational proof of $\langle x \mid E \rangle = \langle y \mid F \rangle$ from the theory $\mathrm{ACP}_{\sharp}^{\tau}$. Let p' be the node labeled tree obtained by replacing, for each $z \in V_E$, all occurrences of $\langle z \mid E \rangle$ in p by z. We claim that p' is a conditional equational proof of $E \Rightarrow x = \langle y \mid F \rangle$. The proof goes by a straightforward induction on the size of p. The important case is where p consists of a single node only, with a label of the form $\langle x \mid E \rangle = \langle t_x \mid E \rangle$, obtained as an instance of axiom REC. Via the substitution this is turned into the equation $x = t_x$, which we are allowed to use as a leaf in p' since it is an equation from E. Using the the rules from Table 2 for symmetry and associativity of '=', it is easy to combine the proofs p and p' to a conditional equational proof of $E \Rightarrow x = \langle x \mid E \rangle$. □

2.11.3. REMARK. In the definition of the notion 'guardedly specifiable', it is essential that the identity $t=\langle x|E\rangle$ is *provable*. If we would only require $t=\langle x|E\rangle$, then the corresponding version of RSP^+ would not be provable from $\mathrm{ACP}_\sharp^\tau$, since this rule would then not be valid in the action relation model of [22]. Strictly speaking, 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 $\langle x|\{x=x\}\rangle=\delta$. 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 z: $z=\langle x|\{x=x\}\rangle=\delta$. This is clearly false.

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

3. APPLICATIONS OF MODULE LOGIC IN PROCESS ALGEBRA

In the previous section we have illustrated how the $+$-construct for modules can be used to present a large number of operators and axioms for processes in a structured way. In this section we will present some less trivial applications of module logic in process algebra which involve use of the module constructs H and S.

3.1. The H-construct. In general a user of a process algebra module wants that this module proves the equality $p=q$ of two closed process expressions p and q, whenever p and q 'have the same interesting properties'. So it depends on what properties are interesting for a particular user, whether the module (s)he uses should be designed to prove $p=q$ or not. For this reason the semantical branch of process algebra research generated a variety of process algebras in which different identification strategies are pursued. In *branching bisimulation semantics with explicit divergence* [26], for instance, a distinction will be made between any two processes that differ in the precise timing of choices or divergencies (infinite τ-sequences); in *trace semantics* on the other hand only processes are distinguished which can perform different sequences of actions; and, somewhere in between, *failure semantics* identifies processes if they have the same traces (can perform the same sequences of actions)

and have the same deadlock behavior in any context. A lot of the process algebras which have been proposed in the literature can be organized as homomorphic images of each other, as indicated in Figure 1. (In fact, the standard branching bisimulation semantics of [26] cannot be mapped homomorphically to failure semantics; see [14]).

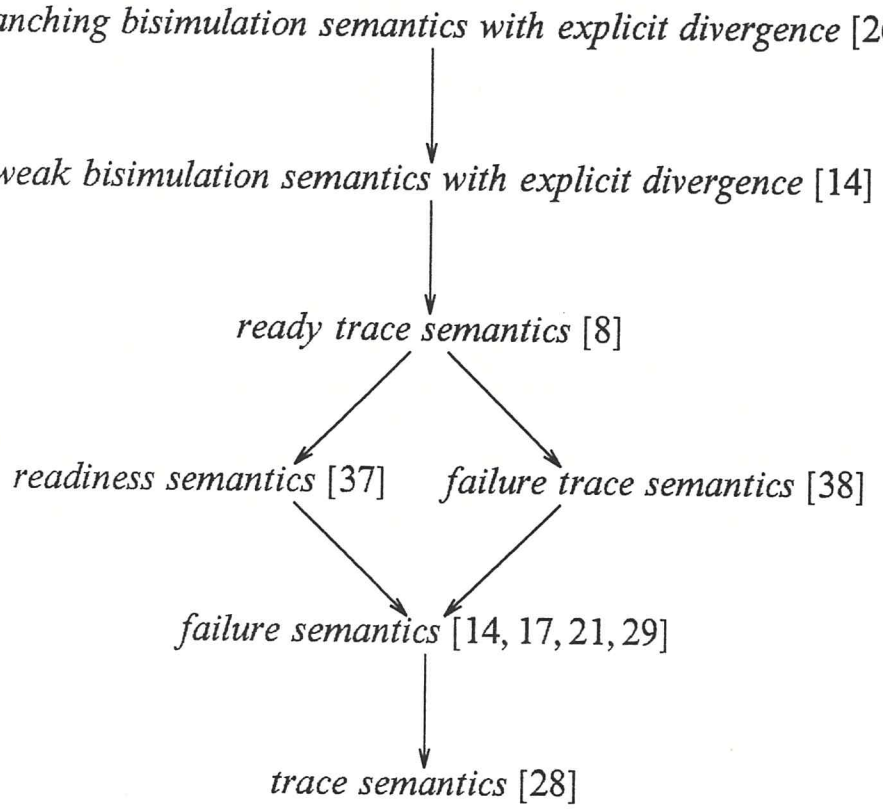


FIGURE 1. *The linear time - branching time spectrum*

For concrete process algebra (without τ-moves) these semantical notions have been defined in [23] relative to some very simple process language. If two process expressions p and q represent the same process in branching 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 branching bisimulation semantics to trace semantics, less and less distinctions are made. Now a user should state exactly in which properties of processes (s)he is interested. Suppose (s)he is only interested in traces and deadlock behavior, then we can tell that for this purpose 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 branching bisimulation semantics with explicit divergence), then they certainly have the same interesting properties, and probably some uninteresting ones as well. Hence a proof in branching bisimulation semantics with explicit divergence is just as good as one in failure semantics (or even better).

This is the reason that we prefer to carry out our proofs using a module, for instance $\mathrm{ACP}_\sharp^\tau$, which is sound with respect to branching bisimulation semantics with explicit divergence. However, if two processes are different in this semantics, we will never succeed in proving them equal from $\mathrm{ACP}_\sharp^\tau$. In such a case we may 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.

3.1.1. Weak bisimulation semantics and the communication merge. It is in the light of the above considerations that one should judge the appearance of the following module WBS:

WBS		
	$x\tau = x$	T1
	$\tau x + x = \tau x$	T2
	$a(\tau x + y) = a(\tau x + y) + ax$	T3

TABLE 13

The laws of this module do not hold in branching bisimulation semantics (we refer to [10, 26] for a detailed comparison of branching and weak bisimulation), but they do hold in all other semantics of Figure 1 in a setting with the language of BPA_δ^τ. In fact, in combination with the laws of BPA_δ^τ, the laws of Table 13 completely axiomatize the semantical notion of weak bisimulation for this language. Thus any identity derived from $\mathrm{ACP}_\sharp^\tau$ + WBS holds in weak bisimulation semantics and hence also in the coarser ones like failure and trace semantics, or so it seems ....

Even without explaining the semantical notions, it is possible to point out that there is a problem here. We will show that the module $\mathrm{ACP}_\sharp^\tau$ + WBS proves some very unintuitive identities. More specifically, we will introduce the notion of *trace consistency* and show that $\mathrm{ACP}_\sharp^\tau$ + WBS is not trace consistent. However, first we have to introduce some auxiliary

notation.

3.1.1.1. Notation for sequences. Let K be any set. $K^\star$ stands for the set of finite sequences of elements of K, and K^+ stands for the set of nonempty finite sequences over K. The empty sequence is denoted by λ and sequence $\rho\star\sigma$ is the concatenation of sequences ρ and σ. The sequence only consisting of $k\in K$ is denoted by k as well.

3.1.1.2. DEFINITION. Let t be a closed expression in the signature of BPA_δ^τ. The *trace set* $tr(t)$ of t is defined inductively by:

$$\begin{aligned} tr(\delta) &= \{\lambda\},\\ tr(\tau) &= \{\lambda,\surd\},\\ tr(a) &= \{\lambda,a,a\star\surd\},\\ tr(s+t) &= tr(s)\cup tr(t),\\ tr(s\cdot t) &= (tr(s)\cap A^*)\cup\{\sigma\star\rho\mid\sigma\star\surd\in tr(s) \text{ and } \rho\in tr(t)\}. \end{aligned}$$

Thus trace sets are nonempty and prefix closed. The special symbol $\surd$ which may occur at the end of a trace denotes successful termination.

3.1.1.3. DEFINITION. Let M be a process module with $\Sigma(M)\supseteq\Sigma(\mathrm{BPA}_\delta^\tau)$. M is called *trace consistent* if for all closed BPA_δ^τ-expressions s and t:

$$M\vdash s=t \quad \text{implies} \quad tr(s)=tr(t).$$

A model $\mathcal{A}$ of M is *trace consistent* if for all closed BPA_δ^τ-expressions s and t:

$$\mathcal{A}\models s=t \quad \text{implies} \quad tr(s)=tr(t).$$

The module $\mathrm{ACP}_\sharp^\tau$ is trace consistent because branching bisimulation semantics, as described in [10], gives a consistent model for this module. The module $\mathrm{ACP}_\sharp^\tau$ + WBS however is in general not trace consistent:

3.1.1.4. PROPOSITION. *Let* $\mathrm{ACP}_\sharp^\tau$ *and* WBS *be parametrized by a set A of actions which contains at least three different elements a, b and c with* $\gamma(a,b)=c$. *Then the module* $\mathrm{ACP}_\sharp^\tau$ + WBS *is not trace consistent.*

PROOF. We derive:

$$c=\delta a+c=(\tau\mid b)a+c=(\tau a\mid b)+a\mid b=$$

$$= (\tau a + a)|b \overset{T2}{=} \tau a|b = (\tau|b)a = \delta.$$

This means that the module cannot be trace consistent since

$$tr(c) = \{\lambda, c, c\star\surd\} \neq \{\lambda\} = tr(\delta).$$ □

This sudden inconsistency must be the result of a serious misunderstanding. And indeed, what's wrong is the use of ACP_δ^τ in weak bisimulation semantics. It happens that weak bisimulation congruence (see [7]), which is the notion of equivalence that is used to construct algebras in the setting of weak bisimulation semantics, is a congruence for the $+$-operator but not for the $|$-operator. This is the source of all the troubles. We should point out here that in [7] a communication merge operator is defined in a setting with weak bisimulation congruence. However, even though it plays a similar role, this operator is different from the communication merge on the domain of branching bisimulation semantics (as defined in [10]), in the sense that both operators cannot be related by a homomorphic mapping.

Module logic provides us with the tools to solve the above problem in a simple and rigorous way. First we make the observation that in practical applications one will never use the operators $\lfloor\!\lfloor$ and $|$ directly. The $\lfloor\!\lfloor$ and $|$ are only auxiliary operators, needed to give a finite complete axiomatization of the merge operator, but of no use for the specification of communicating processes. However, operators like $\lfloor\!\lfloor$ and $|$ in ACP^τ are needed to do calculations and without them even the most elementary equations cannot be derived.

Our solution to this problem is based on the following idea. Suppose one would like to prove an equation $p = q$ in which no $|$-operator occurs and which holds in weak bisimulation semantics but not in branching bisimulation semantics. Then we first prove an intermediate result from ACP^τ: one or more equations valid in branching bisimulation semantics (with explicit divergence) and in which no $|$ appears. Since, except for the $|$-operator, the branching bisimulation model can be mapped homomorphically on the weak bisimulation model, this intermediate result is preserved after carrying out the homomorphic mapping, and can be combined consistently with the module WBS. Thus the proof of $p = q$ can be completed. In our language of modules we describe this as follows. Let

$$\sigma\Delta M \equiv (\Sigma(M) - \sigma)\Box M$$

denote the module M in which the operators and predicates from

signature σ are hidden. Consider the module

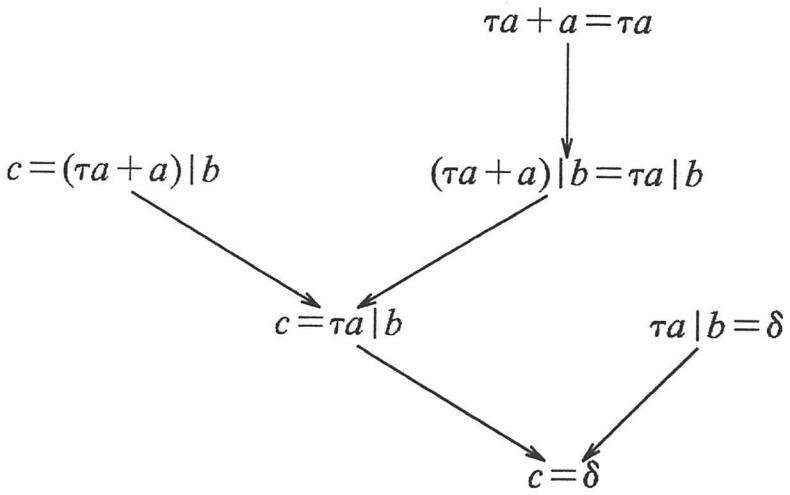
$$\mathrm{sACP}^\tau = H(\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^\tau).$$

This module does not contain the operator $|$ in its visible signature. Since weak bisimulation semantics can be obtained as a homomorphic image of branching bisimulation semantics for all operators of ACP^τ except for $|$, and since ACP^τ is sound w.r.t. branching bisimulation semantics, we conclude that sACP^τ is sound w.r.t. weak bisimulation semantics. Thus sACP^τ is a suitable module for proving statements in weak bisimulation semantics, and can be combined consistently with the module WBS.

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. Analogous to the derivation which we used to show that $\mathrm{ACP}^\tau + \mathrm{WBS}$ is not trace consistent, we prove that

$$\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^\tau + \mathrm{WBS} \vdash c=\delta.$$

Applying Section 1.12 properly, this is one of the rare occasions where we have to construct a nontrivial conditional equational proof:



The leafs $c=(\tau a+a)|b$ and $\tau a|b=\delta$ in this proof tree have been established before using equational logic. Thus

$$\mathrm{ACP}^\tau \vdash \frac{\tau a+a=\tau a}{c=\delta}.$$

One application of the rule for the export operator in Table 3 gives

$$\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^{\tau} \vdash \frac{\tau a + a = \tau a}{c = \delta}.$$

Consequently one can prove a trace inconsistency if one adds law T2:

$$\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^{\tau} + \mathrm{WBS} \vdash c = \delta.$$

So although the formulas provable from module $\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^{\tau}$ contain no communication merge, some of them (which are non-positive) cannot be combined consistently with the laws of weak bisimulation semantics.

The above observations allow us to prove the following incompleteness result:

3.1.1.5. THEOREM: *Equational logic for modules is not complete.*
PROOF. We prove that

$$\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^{\tau} + \mathrm{WBS} \models^{eql} c = \delta$$

even though

$$\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^{\tau} + \mathrm{WBS} \not\models^{eql} c = \delta.$$

In the previous paragraph we have shown that

$$\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^{\tau} + \mathrm{WBS} \vdash^{ceql} c = \delta.$$

Due to the soundness of conditional equational logic and Theorem 1.10.6, this implies

$$\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^{\tau} + \mathrm{WBS} \models^{ceql} c = \delta$$

which, since $c = \delta$ is an equational formula, is equivalent to

$$\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^{\tau} + \mathrm{WBS} \models^{eql} c = \delta$$

(as follows from an observation in Section 1.6.4). However, even though the equation $c = \delta$ holds in all models, it can not be proved with equational logic. In the previous paragraph, we have argued that in a setting with conditional equational logic, the module

$$H(\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^{\tau}) + \mathrm{WBS}$$

is trace consistent and therefore does not prove $c = \delta$. This result carries over to the setting of equational logic (cf. 1.11.4). The key observation we can make now is that, since in equational logic all formulas are positive, a module M proves an equation iff the module obtained from M by

removing all H's proves this equation. Thus

$$\{\mathbb{F}(|,2)\}\Delta\ \mathrm{ACP}^{\tau} + \mathrm{WBS} \nvdash^{eql} c=\delta. \qquad \square$$

3.1.2. The axiom T4 and the left-merge operator. We think that the example of the previous section, which shows how the H-construct can be used to solve a problem with the communication merge, is highly generic. It is a general phenomenon that if one tries to establish a homomorphism from one algebra to another, it might be that for certain operators on the source domain no corresponding versions exist on the target domain. Module logic provides a way to handle this type of situations on the logical level. It is not hard to come up with several other examples of process algebra operators that do not 'survive' an homomorphic mapping. One can think of the operation of action refinement which 'lives' in certain non-interleaved models of concurrency but for which no corresponding operator exists in the interleaving world (see [24]). In the introduction we already mentioned the example of the priority operator of [8], which can be defined in bisimulation semantics and ready trace semantics (at least in a setting without τ) but not in failure trace semantics. Below we will show that the left-merge operator cannot be added to trace consistent algebras in which the following axiom T4 is valid:

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

In a setting with the operators from $\mathrm{BPA}^{\tau}_{\delta}$, axiom T4 holds in all the semantics of Figure 1 except for branching and weak bisimulation semantics. So in particular it holds in ready trace semantics. The following example shows that not all operators from sACP^{τ} can be added to the ready trace model for $\mathrm{BPA}^{\tau}_{\delta}$.

3.1.2.1. PROPOSITION: *Let* sACP^{τ} *be parametrized by a set A of actions which contains at least three elements a, b and c. Then the module* $\mathrm{sACP}^{\tau}+\mathrm{T4}$ *is not trace consistent.*

PROOF. $\mathrm{ACP}^{\tau} + \mathrm{T4} \vdash \tau(ac+ca)+bc = \tau(\tau(ac+ca)+bc+c(\tau a+b))$, as

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

and

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

But this implies that the module is not trace consistent since $c \star b \notin tr(\tau(ac + ca) + bc)$ but $c \star b \in tr(\tau(\tau(ac + ca) + bc + c(\tau a + b)))$. □

The problem in this case is the left-merge operator which cannot be defined on the semantic domain of (for instance) ready trace semantics. Again we should note that operators which resemble our left-merge operator and which play a similar role in axiomatizing the merge operator *may* be defined on the domains of ready trace and other semantics. ACETO [1], for instance, has defined a kind of left-merge operator in a setting of testing (i.e. failure) semantics without $+$. However, given our belief that a framework for the specification and verification of concurrent and reactive systems should provide a user with a spectrum of semantic domains, and not just with a single domain, we think that in general it is preferable to have essentially only a single left-merge and a single communication merge operator, and not a whole spectrum of different versions of these operators.

The same approach that was used in Section 3.1.1 to solve the problem with the communication merge can of course be used also to solve the present problem with the left-merge operator. Consider the module

$$\mathrm{SACP}^\tau = H(\{\mathbb{F}(\lfloor\!\lfloor,2), \mathbb{F}(|,2)\}\Delta(\mathrm{ACP}^\tau + \mathrm{SC})).$$

This module is sound w.r.t all semantics of Figure 1 and can be combined consistently with the axiom T4.

3.2. The S-construct. In order to axiomatize certain operators which one would like to use for the specification of concurrent/reactive systems, one will often introduce other, auxiliary operators. These auxiliary operators are not intended for use in specifications. Therefore it seems a natural idea to hide in a module all auxiliary operators using the □-construct. In this way one accentuates the auxiliary status of these operators and one makes it impossible to use them in specifications. Just like the left-merge and the communication merge are used in order to axiomatize the parallel composition operator, also new atomic actions are often used for axiomatizing a new operator. This is especially the case when new operators are *defined* in terms of the basic combinators using a single axiom. As an example we mention the actions $s(d)$ and $r(d)$ which are used in the definition of the chaining operators. Defined operators can be very useful, but still one can just view them as notations. Since the set of actions occurs as a parameter in modules like ACP^τ, one may think that by varying this parameter the underlying

theory will not really change and that, after hiding of auxiliary actions no traces of them will be left in the resulting module. In this section we will argue that in general this is not the case and that in order to erase all traces of auxiliary operators one sometimes has to use in addition the S-construct. If one adds new atomic actions to a process module, then, in order to preserve trace consistency on the semantic level, one has to extend the domains of the algebras. If, in a subsequent step, one removes these auxiliary actions from the signature, one still has to face the fact that on the semantical level one has 'larger' algebras. In order to illustrate that this can be problematic, we consider the case of the chaining operators. One of the properties of these operators which we use most is that they are 'associative'. However, due to the auxiliary actions, the chaining operators are in general not associative in trace consistent models. Here is a counterexample:

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

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

We *do* have associativity under some very weak assumptions. In the model of branching bisimulation semantics, the following conditional law is valid (here $A' = A - \{s(d), r(d) \mid d \in D\}$):

$$\frac{\alpha(x) \subseteq A',\ \alpha(y) \subseteq A',\ \alpha(z) \subseteq A'}{(x \ggg y) \ggg z = x \ggg (y \ggg z)}\ \text{CC}$$

However, rather than using the above law, we would prefer a solution in which the auxiliary actions are hidden and, for the chaining operators, we have associativity in general. In this section we will see how this can be accomplished by means of the S-construct of our module logic.

Although the rule CC holds in the model of bisimulation semantics, we have not been able to prove it algebraically from the module $\mathrm{ACP}_\sharp^\tau$. The obvious proof strategy would be to eliminate the $\ggg$'s via an application of axiom CH1, then to move the various encapsulation and renaming operators to their proper place via axioms RN and RR, and then to reintroduce the $\ggg$'s again via CH1. Unfortunately this does not work because since both chaining operators synchronize processes via the same auxiliary actions, the side conditions of axiom RR2 do not hold, so we cannot apply this crucial axiom. 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

$$\overline{H} = \{\overline{s}(d),\overline{r}(d) \mid d \in D\} \quad \text{and} \quad \underline{H} = \{\underline{s}(d),\underline{r}(d) \mid d \in D\}.$$

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 $H = \{r(d),s(d) \mid d \in D\}$ and let $\hat{H} = H \cup \overline{H} \cup \underline{H}$. We assume that actions from H do not synchronize with the other actions in the alphabet, and that $range(\gamma) \cap \hat{H} = \varnothing$. On $\hat{H}$ communication is given by $(d \in D)$:

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

For the proof of the following two theorems we refer to Appendix II. Here $A^- = A - \hat{H}$.

3.2.1. THEOREM: $\text{SACP}^\tau + \text{RN} + \text{CH}^+ + \text{AB} + \text{AA} + \text{RR} \vdash$

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

3.2.2. THEOREM: $\text{SACP}^\tau + \text{RN} + \text{CH}^+ + \text{AB} + \text{AA} + \text{RR} \vdash$

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

We will now apply the module approach to remove all traces of the auxiliary atoms which were used in the definition of the chaining operators and to obtain general associativity. As a first step, consider the module

$$\text{CH}^- = (\{\mathbb{F}(a, 0) \mid a \in \hat{H}\} \cup \{\mathbb{F}(\rho_f, 1) \mid f : A_\delta^\tau \to A_\delta^\tau\})$$
$$\Delta(\text{SACP}^\tau + \text{RN} + \text{CH}^+ + \text{AB} + \text{AA} + \text{RR}).$$

This module is parametrized with an alphabet B satisfying $\uparrow d \in B \Leftrightarrow \downarrow d \in B$, and not containing the auxiliary actions $s(d)$, $r(d)$,... From B one derives the alphabet

$$A = B \cup \{s(d),r(d),\overline{s}(d),\overline{r}(d),\underline{s}(d),\underline{r}(d) \mid \uparrow d \in B\},$$

1. The *Fresh Atom Principle* (FAP) says that we can use new (or 'fresh') atomic actions in proofs. In [9], 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, then formally we have to assume that they were in the alphabet right from the beginning.

which is the parameter of its constituent components SACP$^\tau$, RN, CH$^+$, etc. Of course $B=A^-$.

The module CH$^+$ cannot be used to prove any formula containing atoms in $\hat{H}$ but also not to prove the general associativity of the chaining operators. The reason is that the auxiliary atoms, although removed from the language, are still present in the models of module CH$^-$. The counterexample $r(d)\ggg(s(d)+s(e))\ggg r(e)$ still works in any model that is the restriction of a trace consistent model of SACP$^\tau$ + RN + CH$^+$ + AB + AA + RR. Consequently, we have to adapt the class of models of CH$^-$ a bit. The right class of models can be denoted with the help of operator S: 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 at all. In these models the law $\alpha(x)\subseteq A^-$ holds. Thus module CH has consistent models. Using Theorems 3.2.1 and 3.2.2 one can easily establish that module CH proves the general associativity of the chaining operators:

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

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

Since in conditional equational logic all formulas are universal, the module CH$^-$ + $<\alpha(x)\subseteq A^->$ has the same derivational power as CH. In particular this module is trace consistent, even though its models are not. When using first order logic however, we see that the use of the S-construct in CH is essential.

3.2.3. PROPOSITION: Let the module CH$^-$ be parametrized by an alphabet $B=A^-$ such that there is at least one action $\uparrow d$ in B. Then

$$\mathrm{CH}^- + <\alpha(x)\subseteq A^-> \vdash^{foleq} \uparrow d=\delta.$$

PROOF. The constituent components of CH$^-$ are parametrized with a set A of actions containing an action $r(d)$. We claim:

$$\mathrm{SACP}^\tau + \mathrm{RN} + \mathrm{AB} \vdash^{foleq} (\alpha(r(d))\subseteq A^- \to \uparrow d=\delta).$$

The proof is constructed by applying at both sides of the equation $\alpha(r(d))+A^-=A^-$ a renaming operator that takes $r(d)$ into $\uparrow d$ and all actions from A^- into δ. The formula $(\alpha(r(d))\subseteq A^- \to \uparrow d=\delta)$ plays the role of $\neg(\alpha(r(d))\subseteq A^-)$, which cannot be proven from SACP$^\tau$ + RN + AB. From this one derives

$$\mathrm{SACP}^{\tau} + \mathrm{RN} + \mathrm{AB} \vdash^{foleq} \exists x(\alpha(x) \subseteq A^{-} \rightarrow \uparrow d = \delta)$$

(a formal derivation requires fluency in first order logic or the use of several lemmas). Since in this formula no hidden items occur we obtain

$$\mathrm{CH}^{-} \vdash^{foleq} \exists x(\alpha(x) \subseteq A^{-} \rightarrow \uparrow d = \delta)$$

and thus

$$\mathrm{CH}^{-} + <\alpha(x) \subseteq A^{-}> \vdash^{foleq} \uparrow d = \delta. \qquad \square$$

We conclude Section 3.2 with an incompleteness result:

3.2.4. THEOREM: *Infinitary conditional logic for modules is not complete.*
PROOF. Similar to the proof of Theorem 3.1.1.5, using the Proposition 3.2.3. $\square$

3.3. $\mathrm{SACP}^{\tau}_{\sharp}$. Module $\mathrm{SACP}^{\tau}_{\sharp}$ is an 'improved' version of module $\mathrm{ACP}^{\tau}_{\sharp}$ in which auxiliary operators are hidden in an appropriate way. It is defined by:

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

Module $\mathrm{SACP}^{\tau}_{\sharp}$ is parametrized by an alphabet A which does not include the auxiliary actions $s(d)$, $r(d)$,... The rules RSP and RSP^{+} can still be used in a setting with module $\mathrm{SACP}^{\tau}_{\sharp}$: we have $\mathrm{SACP}^{\tau}_{\sharp} \vdash \mathrm{RSP}$ and $\mathrm{SACP}^{\tau}_{\sharp} \vdash \mathrm{RSP}^{+}$. Also the results of Sections 2.1.2, 2.1.3 and 2.4.1 carry over to the new setting. For this it is crucial that the properties of Section 2.1.3 are stated in an equational and not in a conditional form.

4. CURIOUS QUEUES

The aim of this section is to present a somewhat more substantial example of the use of our module logic in process algebra verifications. We have chosen here to deal with some variants of FIFO queues with unbounded capacity. In the specification of concurrent systems these queues often play an important role. We give some examples:

- The semantical description of languages with asynchronous message passing such as CHILL (see [19]),
- The modeling of communication channels occurring in computer networks (see LARSEN & MILNER [31] and VAANDRAGER [44]),

- The implementation of languages with many-to-one synchronous communication, such as POOL (see AMERICA [2] and VAANDRAGER [45]),

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 [20]. Some other references are BROY [18], HOARE [29] and PRATT [39].

4.1. Also in the setting of ACP a lot of attention has been paid to the specification of queues. Below we present an infinite recursive specification of a queue. We assume a finite set D of data. In the equations we use the notations for sequences that were introduced in Section 3.1.1.1. Further, d ranges over D and σ ranges over D^*.

$$QUEUE = Q_\lambda = \sum_{d \in D} \downarrow d \cdot Q_d$$

$$Q_{\sigma * d} = \sum_{e \in D} \downarrow e \cdot Q_{e * \sigma * d} + \uparrow d \cdot Q_\sigma$$

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 [4] and BERGSTRA & TIURYN [15]. □

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] and BAETEN & BERGSTRA [5].

4.2. Definition of the queue by means of chaining. All process algebra verifications involving queues that we encountered in the literature were rather complex. For example, let $BUF1$ denote a buffer with capacity one:

$$BUF1 = \sum_{d \in D} \downarrow d \cdot BUF1^d$$

$$BUF1^d = \uparrow d \cdot BUF1$$

In process verifications one often needs propositions like $QUEUE \gg BUF1 = QUEUE$. However, proofs of such facts starting from the infinite specification happen to be rather complicated. We claim that the following specification of a queue by means of the (abstract) chaining operator allows for a simple proof of the above proposition and numerous other useful identities. A similar specification is also described by HOARE [29] (p. 158).

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

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

4.2.1. THEOREM: $SACP_\sharp^\tau \vdash Q = QUEUE$.
PROOF. Define for $n \in \mathbb{N}$ process Q^n as the chaining of Q with n empty buffers with capacity one:

$$Q^0 = Q, \tag{1}$$

$$Q^{n+1} = Q^n \gg BUF1. \tag{2}$$

By induction we prove that for all n:

$$Q^n = \sum_{d \in D} \downarrow d \cdot (Q^n \gg BUF1^d). \tag{3}$$

The case $n = 0$ follows trivially using (1). So suppose that the statement has been shown for $n \leqslant k$. Using the laws of Section 2.4.1 we derive:

$$Q^{k+1} \overset{2}{=} Q^k \gg BUF1 =$$

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

$$\overset{L3+CH}{=} \sum_{d \in D} \downarrow d \cdot (Q^k \gg (BUF1^d \gg BUF1)) =$$

$$\overset{L1}{=} \sum_{d \in D} \downarrow d \cdot (Q^k \gg \tau \cdot (BUF1 \gg BUF1^d)) =$$

$$\overset{L5}{=} \sum_{d\in D} \downarrow d\cdot(Q^k \gg (BUF1 \gg BUF1^d)) =$$

$$\overset{CH+2}{=} \sum_{d\in D} \downarrow d\cdot(Q^{k+1} \gg BUF1^d).$$

This completes the proof of the induction step.

Define for $\sigma\in D^+$ processes B^σ by:

$$B^d = BUF1^d, \tag{4}$$

$$B^{\sigma\star d} = B^\sigma \gg BUF1^d. \tag{5}$$

By simple inductive arguments one can show that

$$B^{d\star\sigma} = BUF1^d \gg B^\sigma, \tag{6}$$

$$B^{\sigma\star d} = \uparrow d\cdot(BUF1 \gg B^\sigma). \tag{7}$$

We can now derive the following recursive equations (from now on the laws L1-5 and the associativity of $\gg$ will be used without being mentioned explicitly):

$$Q^n \overset{3}{=} \sum_{d\in D} \downarrow d\cdot(Q^n \gg BUF1^d) \overset{4}{=} \sum_{d\in D} \downarrow d\cdot(Q^n \gg B^d).$$

$$Q^n \gg B^d \overset{3+4}{=} \sum_{e\in D} \downarrow e\cdot(Q^n \gg (BUF1^e \gg B^d)) + \uparrow d\cdot(Q^n \gg BUF1) =$$

$$\overset{2+6}{=} \sum_{e\in D} \downarrow e\cdot(Q^n \gg B^{e\star d}) + \uparrow d\cdot Q^{n+1}.$$

$$Q^n \gg B^{\sigma\star d} \overset{3+7}{=} \sum_{e\in D} \downarrow e\cdot(Q^n \gg (BUF1^e \gg B^{\sigma\star d})) + \uparrow d\cdot((Q^n \gg BUF1) \gg B^\sigma) =$$

$$\overset{2+6}{=} \sum_{e\in D} \downarrow e\cdot(Q^n \gg B^{e\star\sigma\star d}) + \uparrow d\cdot(Q^{n+1} \gg B^\sigma).$$

Consider the following guarded recursive specification with variables Q^n_σ for $\sigma\in D^*$ and $n\in\mathbb{N}$:

$$Q^n_\lambda = \sum_{d\in D} \downarrow d\cdot Q^n_d,$$

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

Since $QUEUE$ satisfies the defining equations of Q^0_λ, the RSP gives that $QUEUE = Q^0_\lambda$. Because Q^0 also satisfies the equations for Q^0_λ, another application of RSP gives $Q^0 = Q^0_\lambda$. Consequently $QUEUE = Q^0 = Q$. □

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. COROLLARY: $\mathrm{SACP}_\sharp^\tau \vdash Q \gg BUF1 = Q$.
PROOF. From equations (1), (2) and (3) in the proof of Theorem 4.2.1 it follows that

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

Since the specification for $QUEUE$ is guarded, we know from Theorem 4.2.1 that Q is guardedly specifiable. Therefore, since $Q \gg BUF1$ satisfies the defining equation of Q, we can use RSP^+ to conclude that $Q = Q \gg BUF1$. □

4.2.3. PROPOSITION: $\mathrm{SACP}_\sharp^\tau \vdash Q \gg Q = Q$.
PROOF. $Q \gg Q =$

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

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

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

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

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

$$\overset{4.2.2}{=} \sum_{d \in D} \downarrow d \cdot (Q \gg (Q \gg BUF1^d)) =$$

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

Now apply RSP^+. □

4.2.4. 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 concise. It is the shortest specification of a FIFO-queue known to the authors, except for a 5-character specification due to PRATT [39]: $\downarrow\uparrow \times D^*$. A problem with Pratt's specification however is that a neat axiomatization of the ortho-currence 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. A queue that can lose data. When dealing with communication protocols, one often encounters 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 also to specify various types of faulty queues. One just has to replace the process $BUF1$ in the definition of Q by a process that behaves like a buffer but can lose, damage or duplicate data.

Here 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:

$$\boxed{\begin{aligned} FBUF1 &= \sum_{d \in D} \downarrow d \cdot FBUF1^d \\ FBUF1^d &= (\uparrow d + \tau) \cdot FBUF1 \end{aligned}}$$

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 can 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 2. Each faulty buffer process is represented by a 'conductor'. These conductors also occur on the emblem of the Dutch REX project (Research and Education in Concurrent Systems). On the REX emblem one can see them engaged in their profession, which is parallel conducting of a symphony orchestra. In Figure 2, they are depicted while helping one of their colleagues who is moving, passing on to each other various types of boxes. These boxes of course correspond to the elements of the set D of data. As one can see the conductors are not really good in this type of work and now and then a box just slips out of their hands into the deep abyss that lurks right under the corridor in which all activity takes place. The aspect of process creation which is present in the specification of FQ is not captured in the figure. One should imagine that whenever a new box arrives at the beginning of the corridor, also a new conductor arrives and reluctantly starts to work by either passing on the box or letting it slip away.

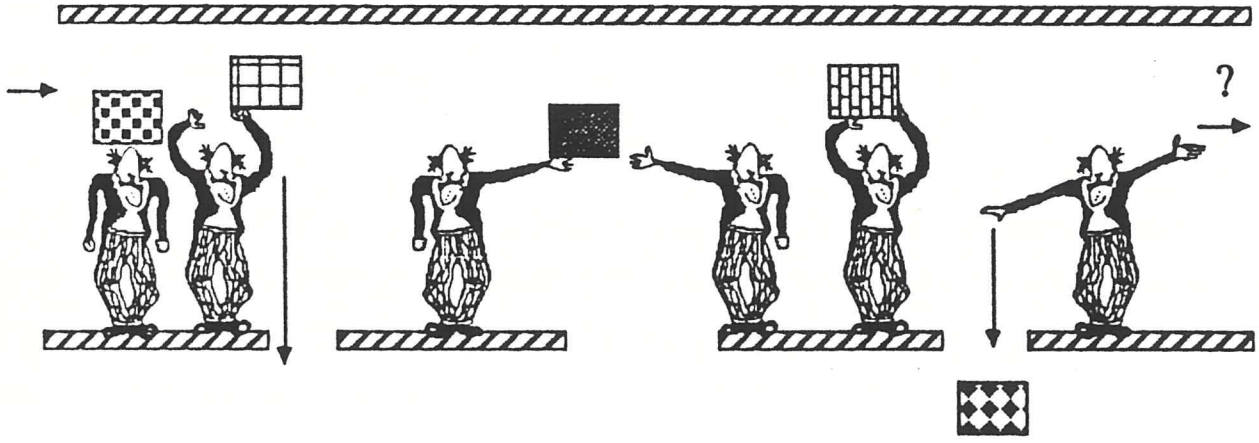


FIGURE 2. *The faulty queue*

4.3.1. LEMMA: $\mathrm{SACP}^{\tau}_{\sharp} \vdash$

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

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

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

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

$\overset{B2}{=} \tau\cdot(\tau\cdot(FBUF1 \gg FBUF1) + FBUF1 \gg FBUF1^{d}) =$

$= \tau\cdot(FBUF1 \gg FBUF1^{d}).$ □

Compare the simple definition of FQ with the following $\mathrm{BPA}^{\tau}_{\delta}$-specification of the same process.

4.3.2. For $\sigma \in D^{*}$, let $R(\sigma)$ be the finite set of residuals which can be obtained by deleting one arbitrary datum from σ. Now $FQUEUE$ is defined by the following recursive specification:

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

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

4.3.3. THEOREM: $\mathrm{SACP}^{\tau}_{\sharp} \vdash FQ = FQUEUE$.
PROOF. Analogous to the proof of Theorem 4.2.1. Use Lemma 4.3.1. □

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

4.3.4. PROPOSITION: $\mathrm{SACP}^{\tau}_{\sharp} \vdash$
i) $FQ \gg FBUF1 = FQ$,
ii) $\tau\cdot(FBUF1^{d} \gg FQ) = \tau\cdot(FBUF1 \gg (FQ \gg FBUF1^{d}))$,
iii) $Q \gg FQ = FQ \gg FQ = FQ$.

4.4. 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. We can think of no 'experiment' that distinguishes between the two processes. Still the identity cannot be proved from the module SACP^{τ}. In fact we have proved the following, even stronger result.

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

PROOF. We show that the identity is not valid in the model of process graphs modulo weak bisimulation congruence of BAETEN, BERGSTRA & KLOP [7] (see also BAETEN & WEIJLAND [10]). Suppose that there exists a bisimulation between processes FQ and $Q \gg FBUF1$. Consider the situation in which process FQ has read 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. Suppose that next 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 3. □

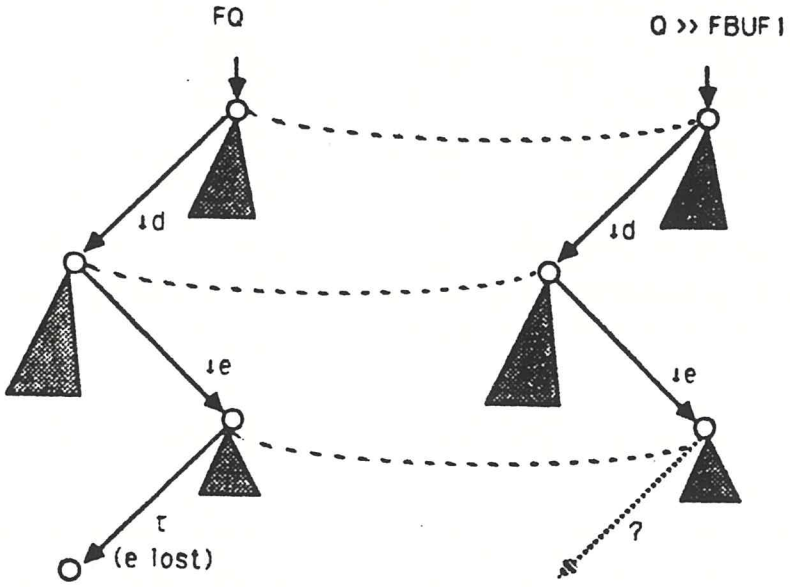


FIGURE 3.

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

4.4.2. THEOREM: $\mathrm{SACP}_{\sharp}^{\tau} + \mathrm{WBS} + \mathrm{T4} \vdash FQ = Q \gg FBUF1$.
PROOF. Define the process QF by:

$$QF = QF_{\lambda} = \sum_{d \in D} \downarrow d \cdot QF_d,$$

$$QF_{\sigma * d} = \sum_{e \in D} \downarrow e \cdot QF_{e * \sigma * d} + (\uparrow d + \tau) \cdot QF_{\sigma}.$$

Analogous to the proof of Theorem 4.2.1, using in addition the identity $Q \gg BUF1 = Q$, one can prove from $\mathrm{SACP}_{\sharp}^{\tau}$ that $Q \gg FBUF1 = QF$.

The main trick in the proof is that we introduce yet another 'view of queues': process QF_{σ} is split into two parts, a read-process and a (faulty) send process. The read-process takes care of reading new data and stores them in a queue. The send process outputs the data in σ or forgets them and after that it starts behaving like a faulty buffer with capacity one which receives input from the read process. The fact that the length of the sequence of data in the send process can only decrease (until the moment that it starts behaving like a faulty buffer) allows us to use induction.

Define the faulty send process for $\sigma \in D^*$ by:

$$FS_{\lambda} = FBUF1,$$

$$FS_{\sigma * d} = (\uparrow d + \tau) \cdot FS_{\sigma}.$$

It is routine to prove from $SACP_\sharp^\tau$ that $Q \gg FS_\sigma = QF_\sigma$. Until here one does not need the additional axioms. The crucial part in the proof is the following

CLAIM: $\tau \cdot FS_{\rho \star \sigma} \subseteq FS_{\rho \star d \star \sigma}$.

PROOF. By induction on the length of σ. If $\sigma = \lambda$ then the claim holds trivially. Assume the claim is proved for $|\sigma| \leqslant n$. Suppose $\sigma = \bar{\sigma} \star e$ with $|\bar{\sigma}| = n$. We derive:

$$\tau \cdot FS_{\sigma \star \bar{\sigma} \star e} = \tau \cdot (\uparrow e \cdot FS_{\sigma \star \bar{\sigma}} + \tau \cdot FS_{\sigma \star \bar{\sigma}}) =$$

(this is the only step where we use axiom T4)

$$= \uparrow e \cdot FS_{\sigma \star \bar{\sigma}} + \tau \cdot FS_{\sigma \star \bar{\sigma}} \subseteq$$

(because on the one hand $\uparrow e \cdot FS_{\sigma \star \bar{\sigma}} \subseteq \uparrow e \cdot FS_{\sigma \star d \star \bar{\sigma}}$ by induction hypothesis and axiom T3, and on the other hand $\tau \cdot FS_{\sigma \star \bar{\sigma}} \subseteq \tau \cdot FS_{\sigma \star d \star \bar{\sigma}}$ by induction hypothesis and axiom T2)

$$\subseteq \uparrow e \cdot FS_{\sigma \star d \star \bar{\sigma}} + \tau \cdot FS_{\sigma \star d \star \bar{\sigma}} = FS_{\sigma \star d \star \bar{\sigma} \star e}.$$

This finishes the proof of the claim. As a corollary we can prove that $\tau \cdot QF_{\rho \star \sigma} \subseteq QF_{\rho \star d \star \sigma}$:

$$\tau \cdot QF_{\rho \star \sigma} = \tau \cdot (Q \gg FS_{\rho \star \sigma}) \subseteq$$

(Use the observation of Section 2.1.3 that $\tau(x \| z) \subseteq (\tau x + y) \| z$)

$$\subseteq Q \gg FS_{\rho \star d \star \sigma} = QF_{\rho \star d \star \sigma}.$$

We have shown that process QF_σ is indistinguishable from a process that can lose each datum at every moment. Using the notation of Section 4.3.2 we can write the following equation for processes $QF_{\sigma \star d}$:

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

Application of RSP gives that the process $FQUEUE$ of Section 4.3.2 equals process QF. But according to Theorem 4.3.3 also $FQUEUE = FQ$. Thus $FQ = FQUEUE = QF = Q \gg FBUF1$. □

CONCLUSIONS AND OPEN PROBLEMS

In this paper we presented a language making it possible to give modular specifications of (classes of) process algebras. The language contains constructs $+$ and $\square$, which are standard in the theory of structured algebraic specifications, and moreover two new constructs H and S. Two applications have been presented of the new constructs: we showed how

the left- and communication merge operators 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 [11]: 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 [11] will probably not be achievable.

An important open problem for us is the question whether the proof system of Table 3 is complete for first order logic.

In this paper the modules are parametrized by a set of actions. These actions themselves do not have any structure. The most natural way to look towards actions like $s(d_0)$ however, is to view them as actions parametrized by data. We would like to include the notion of a parametrized action in our framework but it turns out that this is not trivial. Related work in this area has been done by MAUW [32] and MAUW & VELTINK [33].

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 unsatisfactory 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 4.4 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.

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. Furthermore we thank the referees for their useful suggestions.

APPENDIX I: FIRST ORDER LOGIC

In this appendix first order logic is defined, in order to allow comparison with the two logics of Section 1.

The set F_σ^{foleq} of *first order formulas with equality* over σ is defined by:
- $F_\sigma^{at} \subseteq F_\sigma^{foleq}$;
- if $\phi \in F_\sigma^{foleq}$ then $\neg\phi \in F_\sigma^{foleq}$;
- if ϕ and $\psi \in F_\sigma^{foleq}$ then $(\phi \to \psi) \in F_\sigma^{foleq}$;
- 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 is a variable and $\phi \in F_\sigma^{foleq}$ then $\forall x(\phi) \in F_\sigma^{foleq}$;
- if x is a variable and $\phi \in F_\sigma^{foleq}$ then $\exists x(\phi) \in F_\sigma^{foleq}$.

The *ξ-truth* of a formula $\phi \in F_\sigma^{foleq}$ in a σ-algebra $\mathcal{A}$ is defined by:
- $\mathcal{A},\xi \models_\sigma^{foleq} \phi$ if $\phi \in F_\sigma^{at}$ and $\mathcal{A},\xi \models_\sigma^{ceql} \phi$;
- $\mathcal{A},\xi \models_\sigma^{foleq} \neg\phi$ 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 $\mathcal{A},\xi \models_\sigma^{foleq} \psi$;
- $\mathcal{A},\xi \models_\sigma^{foleq} \forall x(\phi)$ if $\mathcal{A},\xi' \models_\sigma^{foleq} \phi$ for all valuations ξ' with $\xi'(y)=\xi(y)$ for all variables $y \neq x$;
- $\mathcal{A},\xi \models_\sigma^{foleq} \exists x(\phi)$ if $\mathcal{A},\xi' \models_\sigma^{foleq} \phi$ for some valuation ξ' with $\xi'(y)=\xi(y)$ for all variables $y \neq x$.

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 14. There ϕ, ψ and ρ are first order formulas in F_σ^{foleq}, α is an atomic formula in F_σ^{at}, 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*. A variable is *free* in a formula ϕ if all its occurrences in ϕ are free. $\phi[t/x]$ denotes the result of substituting u for all free occurrences of x in t. Now t *is free for* x *in* ϕ if all free occurrences of variables in t remain free in $\phi[t/x]$.

$\dfrac{\phi,\ \phi\to\psi}{\psi}$ *modus ponens*		$\dfrac{\phi}{\forall x(\phi)}$ *generalization*
$\phi\to(\psi\to\phi)$ $\{\phi\to(\psi\to\rho)\}\to\{(\phi\to\psi)\to(\phi\to\rho)\}$ $\{\forall x(\phi\to\psi)\}\to\{\phi\to\forall x(\psi)\}$, if x is not 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 specialization*
$(\phi\wedge\psi)\to\phi$	$\phi\to(\phi\vee\psi)$	$(\phi\leftrightarrow\psi)\to\{(\phi\to\psi)\wedge(\psi\to\phi)\}$
$(\phi\wedge\psi)\to\psi$	$\psi\to(\phi\vee\psi)$	$\{(\phi\to\psi)\wedge(\psi\to\phi)\}\to(\phi\leftrightarrow\psi)$
$\phi\to\{\psi\to(\phi\wedge\psi)\}$	$(\phi\vee\psi)\to(\neg\phi\to\psi)$	$\exists x(\phi)\leftrightarrow\neg\forall x(\neg\phi)$
$t=t$	$(u=v)\to(v=u)$	$\{(t=u)\wedge(u=v)\}\to(t=v)$
	$(u=v)\to(\alpha[u/x]\leftrightarrow\alpha[v/x])$	

TABLE 14

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 14 can be omitted.

In first order logic (with equality) the positive formulas are the ones without the connectives $\neg$, $\to$ and $\leftrightarrow$, and the universal ones are the formulas without quantifiers. Model theory (see for instance [36]) teaches us that a formula ϕ is preserved under homomorphisms (respectively subalgebras) iff there is a positive (respectively universal) formula ψ with $\vdash^{foleq} \psi\leftrightarrow\phi$.

APPENDIX II: THE ASSOCIATIVITY OF THE CHAINING OPERATORS

In this appendix we will present the proofs of Theorem 3.2.1 and Theorem 3.2.2 about the associativity of the chaining operators. Define for $v,w \in \{\uparrow,\downarrow,s,r,\overline{s},\overline{r},\underline{s},\underline{r}\}$ the renaming function vw by:

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

First we need an auxiliary lemma:

LEMMA. $\mathrm{SACP}^\tau + \mathrm{RN} + \mathrm{CH}^+ + \mathrm{AB} + \mathrm{AA} + \mathrm{RR} \vdash$

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

PROOF. We only prove the first equality. The second follows by symmetry.

$\partial_{\overline{H}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y)) =$ (Note 1 below, RR1)

$= \partial_{\overline{H}}\circ\rho_{s\overline{s}}\circ\rho_{r\overline{r}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y)) =$ (RN5, $y=\partial\hat{}_H(y)$)

$= \partial_{\overline{H}}\circ\rho_{s\overline{s}}\circ\rho_{r\overline{r}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{r\overline{r}}\circ\rho_{\downarrow r}(y)) =$ (Note 2, RR2)

$= \partial_{\overline{H}}\circ\rho_{s\overline{s}}\circ\rho_{r\overline{r}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow r}(y)) =$ (SC4, RN5, $x=\partial\hat{}_H(x)$)

$= \partial_{\overline{H}}\circ\rho_{r\overline{r}}\circ\rho_{s\overline{s}}(\rho_{\downarrow r}(y)\|\rho_{s\overline{s}}\circ\rho_{\uparrow s}(x)) =$ (as in Note 2, RR2)

$= \partial_{\overline{H}}\circ\rho_{r\overline{r}}\circ\rho_{s\overline{s}}(\rho_{\downarrow r}(y)\|\rho_{\uparrow s}(x)) =$ (RN5)

$= \partial_H\circ\partial_{\overline{H}}(\rho_{\downarrow r}(y)\|\rho_{\uparrow s}(x)) =$ (Note 3, RR1, SC4)

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

$= x \ggg y.$

Note 1. Let $B=A-H$. We claim $\alpha(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y)) \subseteq \boldsymbol{B}$ (recall that $\boldsymbol{B} =_{def} \sum_{b\in B} b$).

PROOF. $\alpha(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y)) =$

$$\overset{AA2}{=} \alpha\circ\rho_{\uparrow\overline{s}}(x)+\alpha\circ\rho_{\downarrow\overline{r}}(y)+\alpha\circ\rho_{\uparrow\overline{s}}(x)\,|\,\alpha\circ\rho_{\downarrow\overline{r}}(y) \subseteq$$

(Use that $x\subseteq y \Rightarrow x|z \subseteq y|z$. Use further $x=\partial\hat{}_H(x) \overset{RN5}{=} \partial_H\circ\partial\hat{}_H(x) = \partial_H(x)$.)

$$\overset{AA1}{\subseteq} \alpha\circ\rho_{\uparrow\overline{s}}\circ\partial_H(x)+\alpha\circ\rho_{\downarrow\overline{r}}\circ\partial_H(y)+\boldsymbol{A}\,|\,\boldsymbol{A} \subseteq$$

(Use that $range(\gamma) \cap H = \varnothing$.)

$$\overset{RN5}{\subseteq} \alpha \circ \partial_H \circ \rho_{\uparrow \overline{s}}(x) + \alpha \circ \partial_H \circ \rho_{\downarrow \overline{r}}(y) + \boldsymbol{B} \subseteq$$

$$\overset{AA3+RN5}{\subseteq} \partial_H \circ \alpha \circ \rho_{\uparrow \overline{s}}(x) + \partial_H \circ \alpha \circ \rho_{\downarrow \overline{r}}(y) + \boldsymbol{B} \subseteq$$

(Use that $x \subseteq y$ implies $\rho_f(x) \subseteq \rho_f(y)$.)

$$\overset{AA1}{\subseteq} \partial_H(\boldsymbol{A}) + \partial_H(\boldsymbol{A}) + \boldsymbol{B} = \boldsymbol{B}$$

This finishes the proof of the claim. □

Note 2. Application of axiom AA1 gives: $\alpha \circ \rho_{\uparrow \overline{s}}(x) \subseteq \boldsymbol{A}$ and $\alpha \circ \rho_{\downarrow r}(y) \subseteq \boldsymbol{A}$. In order to apply axiom RR2, we first have to check that for all $c \in A$: $\overline{rr}(c) = \overline{rr} \circ \overline{rr}(c)$. This is obviously the case. Because $range(\gamma) \cap H = \varnothing$, we have for all $b, c \in A$: $\overline{rr} \circ \gamma(b,c) = \gamma(b,c)$. Now the last thing to be checked is that for $b, c \in A$: $\gamma(b,c) = \gamma(b, \overline{rr}(c))$.

Note 3. Let $C = A - \overline{H}$. We claim: $\alpha(\rho_{\downarrow r}(y) \| \rho_{\uparrow s}(x)) \subseteq \boldsymbol{C}$. The proof is similar to the proof in Note 1.

This finishes the proof of the lemma. It should be noted however that although the proof looks equational, it is in fact a conditional equational proof. The appropriate adjustment of its format is left to (the imagination of) the reader. □

THEOREM 1. $SACP^{\tau} + RN + CH^{+} + AB + AA + RR \vdash$

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

PROOF. This is essentially Theorem 1.12.2 of [45]. We give a sketch of the proof.

$$x \ggg (y \ggg z) \overset{Lemma}{=} \partial_{\overline{H}}(\rho_{\uparrow \overline{s}}(x) \| \rho_{\downarrow \overline{r}} \circ \partial_{\underline{H}}(\rho_{\uparrow \underline{s}}(y) \| \rho_{\downarrow \underline{r}}(z))) =$$

$$\overset{RN5}{=} \partial_{\overline{H}}(\rho_{\uparrow \overline{s}}(x) \| \partial_{\underline{H}} \circ \rho_{\downarrow \overline{r}}(\rho_{\uparrow \underline{s}}(y) \| \rho_{\downarrow \underline{r}}(z))) =$$

$$\overset{RR1}{=} \partial_{\overline{H}} \circ \partial_{\underline{H}}(\rho_{\uparrow \overline{s}}(x) \| \partial_{\underline{H}} \circ \rho_{\downarrow \overline{r}}(\rho_{\uparrow \underline{s}}(y) \| \rho_{\downarrow \underline{r}}(z))) =$$

$$\overset{RR2}{=} \partial_{\overline{H}} \circ \partial_{\underline{H}}(\rho_{\uparrow \overline{s}}(x) \| \rho_{\downarrow \overline{r}}(\rho_{\uparrow \underline{s}}(y) \| \rho_{\downarrow \underline{r}}(z))) =$$

$$\overset{RR2}{=} \partial_{\overline{H}} \circ \partial_{\underline{H}}(\rho_{\uparrow \overline{s}}(x) \| \rho_{\downarrow \overline{r}}(\rho_{\downarrow \overline{r}} \circ \rho_{\uparrow \underline{s}}(y) \| \rho_{\downarrow \underline{r}}(z))) =$$

$$\overset{RR1}{=} \partial_{\overline{H}}\circ\partial_{\underline{H}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}\circ\rho_{\uparrow\underline{s}}(y)\|\rho_{\downarrow\underline{r}}(z)) =$$

$$\overset{RN5}{=} \partial_{\underline{H}}\circ\partial_{\overline{H}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\uparrow\underline{s}}\circ\rho_{\downarrow\overline{r}}(y)\|\rho_{\downarrow\underline{r}}(z)) =$$

$$\overset{RR1}{=} \partial_{\underline{H}}\circ\partial_{\overline{H}}(\rho_{\uparrow\underline{s}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\uparrow\underline{s}}\circ\rho_{\downarrow\overline{r}}(y))\|\rho_{\downarrow\underline{r}}(z)) =$$

$$\overset{RR2}{=} \partial_{\underline{H}}\circ\partial_{\overline{H}}(\rho_{\uparrow\underline{s}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y))\|\rho_{\downarrow\underline{r}}(z)) =$$

$$\overset{RR2}{=} \partial_{\underline{H}}\circ\partial_{\overline{H}}(\partial_{\overline{H}}\circ\rho_{\uparrow\underline{s}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y))\|\rho_{\downarrow\underline{r}}(z)) =$$

$$\overset{RR1}{=} \partial_{\underline{H}}(\partial_{\overline{H}}\circ\rho_{\uparrow\underline{s}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y))\|\rho_{\downarrow\underline{r}}(z)) =$$

$$\overset{RN5}{=} \partial_{\underline{H}}(\rho_{\uparrow\underline{s}}\circ\partial_{\overline{H}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y))\|\rho_{\downarrow\underline{r}}(z)) = (x\ggg y)\ggg z$$

□

THEOREM 2. $\mathrm{SACP}^{\tau} + \mathrm{RN} + \mathrm{CH}^{+} + \mathrm{AB} + \mathrm{AA} + \mathrm{RR} \vdash$

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

PROOF. Let $I=\{c(d)\,|\,d\in D\}$. We derive:

$$x\gg(y\gg z) \overset{CH2}{=} \tau_I(x\ggg(\tau_I(y\ggg z))) =$$

$$\overset{CH1}{=} \tau_I\circ\partial_H(\rho_{\uparrow s}(x)\|\rho_{\downarrow r}\circ\tau_I(y\ggg z)) =$$

$$\overset{RN5}{=} \partial_H\circ\tau_I(\rho_{\uparrow s}(x)\|\tau_I\circ\rho_{\downarrow r}(y\ggg z)) =$$

$$\overset{RR2}{=} \partial_H\circ\tau_I(\rho_{\uparrow s}(x)\|\rho_{\downarrow r}(y\ggg z)) =$$

$$\overset{RN5}{=} \tau_I\circ\partial_H(\rho_{\uparrow s}(x)\|\rho_{\downarrow r}(y\ggg z)) =$$

$$\overset{CH1}{=} \tau_I(x\ggg(y\ggg z)) =$$

$$\overset{Theorem1}{=} \tau_I((x\ggg y)\ggg z) = \cdots = (x\gg y)\gg z$$ □

Theorems 3.2.1 and 3.2.2 follow from the above Theorems 1 resp. 2 in combination with axiom RR1.

References

[1] L. Aceto (1990): *A theory of testing for ACP.* Computer Science Report 3/90, University of Sussex, Brighton.

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

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

[4] J.C.M. Baeten and J.A. Bergstra (1988): *Global renaming operators in concrete process algebra.* I&C 78(3), pp. 205-245.

[5] J.C.M. Baeten and J.A. Bergstra (1989): *Recursive process definitions with the state operator.* Report CS-R8920, Centrum voor Wiskunde en Informatica, Amsterdam, to appear in TCS.

[6] J.C.M. Baeten, J.A. Bergstra, and 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.

[7] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop (1987): *On the consistency of Koomen's fair abstraction rule.* Theoretical Computer Science 51(1/2), pp. 129-176.

[8] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop (1987): *Ready-trace semantics for concrete process algebra with the priority operator.* The Computer Journal 30(6), pp. 498-506.

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

[10] J.C.M. Baeten and W.P. Weijland (1990): *Process algebra*, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press.

[11] J.A. Bergstra, J. Heering, and P. Klint (1990): *Module algebra.* JACM 37(2), pp. 335-372.

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

[13] J.A. Bergstra and 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, and 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.

[15] J.A. BERGSTRA and J. TIURYN (1987): *Process algebra semantics for queues.* Fund. Inf. X, pp. 213-224, also appeared as: MC Report IW 241, Amsterdam 1983.

[16] G. BIRKHOFF (1935): *On the structure of abstract algebras.* In: Proceedings of the Cambridge Philosophical Society 31(4), pp. 433-454.

[17] S.D. BROOKES and 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 (1988): *Views on queues.* Sci. Comp. Prog. 11, pp. 65-86.

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

[20] T. DENVIR, W. HARWOOD, M. JACKSON, and M. RAY (1985): *The analysis of concurrent systems, Proceedings of a Tutorial and Workshop, Cambridge University 1983*, LNCS 207, Springer-Verlag.

[21] R. DE NICOLA and M. HENNESSY (1984): *Testing equivalences for processes.* Theoretical Computer Science 34, pp. 83-133.

[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 (1990): *The linear time - branching time spectrum.* In: Proceedings CONCUR 90, Amsterdam (J.C.M. Baeten & J.W. Klop, eds.), LNCS 458, Springer-Verlag, pp. 278-297.

[24] R.J. VAN GLABBEEK (1990): *Comparative concurrency semantics and refinement of actions.* Ph.D. Thesis, Free University, Amsterdam.

[25] R.J. VAN GLABBEEK and F.W. VAANDRAGER (1989): *Modular specifications in process algebra - with curious queues (extended abstract).* In: Algebraic Methods: Theory, Tools and Applications, Workshop Passau 1987 (M. Wirsing & J.A. Bergstra, eds.), LNCS 394, Springer-Verlag, pp. 465-506.

[26] R.J. VAN GLABBEEK and W.P. WEIJLAND (1990): *Branching time and*

abstraction in bisimulation semantics. Report TUM-I9052, Technical University of Munich, and Chapter III of [24].

[27] M. HENNESSY (1988): *Algebraic theory of processes*, MIT Press, Cambridge, Massachusetts.

[28] 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.

[29] C.A.R. HOARE (1985): *Communicating sequential processes*, Prentice-Hall International.

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

[31] K.G. LARSEN and R. MILNER (1987): *A complete protocol verification using relativized bisimulation*. In: Proceedings ICALP 87, Karlsruhe (Th. Ottmann, ed.), LNCS 267, Springer-Verlag, pp. 126-135.

[32] S. MAUW (1987): *An algebraic specification of process algebra, including two examples*. Report FVI 87-06, Dept. of Computer Science, University of Amsterdam, extended abstract appeared in: Algebraic Methods: Theory, Tools and Applications (M. Wirsing & J.A. Bergstra, eds.), LNCS 394, Springer-Verlag, pp. 507-554.

[33] S. MAUW and G.J. VELTINK (1990): *A process specification formalism*. Fundamenta Informaticae XIII, pp. 85-139.

[34] R. MILNER (1980): *A calculus of communicating systems*, LNCS 92, Springer-Verlag.

[35] R. MILNER (1989): *Communication and concurrency*, Prentice-Hall International.

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

[37] E.-R. OLDEROG and C.A.R. HOARE (1986): *Specification-oriented semantics for communicating processes*. Acta Informatica 23, pp. 9-66.

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

[39] V.R. PRATT (1986): *Modeling concurrency with partial orders*. International Journal of Parallel Programming 15(1), pp. 33-71.

[40] A. ROBINSON (1960): *On the mechanization of the theory of equations*. Bull. Res. Council Israel Sect. F 9F, pp. 47-70.

[41] D.T. SANNELLA and A. TARLECKI (1988): *Toward formal development of programs from algebraic specifications: implementations revisited*. Acta Informatica 25, pp. 233-281.

[42] D.T. SANNELLA and M. WIRSING (1983): *A kernel language for algebraic specification and implementation (extended abstract).* In: Proceedings International Conference 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.

[43] A. SELMAN (1972): *Completeness of calculii for axiomatically defined classes of algebras.* Algebra Universalis 2, pp. 20-32.

[44] F.W. VAANDRAGER (1986): *Verification of two communication protocols by means of process algebra.* Report CS-R8608, Centrum voor Wiskunde en Informatica, Amsterdam.

[45] F.W. VAANDRAGER (1990): *Process algebra semantics of POOL.* In: Applications of process algebra (J.C.M. Baeten, ed.), Cambridge University Press, pp. 173-236.

[46] F.W. VAANDRAGER (1990): *Algebraic techniques for concurrency and their application.* Ph.D. Thesis, University of Amsterdam.