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

CWI, Postbus 4079, 1009 AB Amsterdam, The Netherlands

Communicated by A.R. Meyer
Received July 1988
Revised November 1991

Abstract


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 o, 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 prove the completeness — and first-order logic and show how the notion of a formal proof of

*This is a substantial revision of part of our paper Modular specifications in process algebra — with curious queues, Report CS-R8821, CWI, 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 the revision was done during a visit from the second author at the Technical University of Munich.

**Present address: Computer Science Department, Stanford University, Stanford, CA 94305, USA. Email: rvg@cs.stanford.edu.

0304-3975/93/$06.00 © 1993 — Elsevier Science Publishers B.V. All rights reserved
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.

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 and Boudol [3] and the algebra of communicating processes (ACP) of Bergstra and 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 and Hoare [30] that we may have to accept that each application will require a 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 that 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 stating 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 [8], for example, it is shown that the combination of failure semantics and the priority operator is trace inconsistent. Another example can be found [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 $r$'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 then cannot 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 [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 and Wirsing [42] and Sannella and 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 to be 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 $\phi$ holds in the process algebra $A$, notation $A \models \phi$, or “formula $\phi$ holds in the class of process algebras $\mathcal{E}$”, notation $\mathcal{E} \models \phi$. Now we can formulate the goal that is pursued in the present section: to propose a method for proving assertions $A \models \phi$, or $\mathcal{E} \models \phi$.

1.2. Proving formulas from theories

Classical logic gave us the notion of a formal proof of a formula $\phi$ 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 $\phi$ holds in all algebras satisfying $T$. Here an algebra $A$ satisfies $T$, notation $A \models T$, if all formulas of $T$ hold in this algebra. Thus, if we want to prove $A \models \phi$, it suffices to prove $T \vdash \phi$ and $A \models T$ for a suitable theory $T$. Likewise, if we want to prove $\mathcal{E} \models \phi$, with $\mathcal{E}$ a class of algebras, if suffices to prove $T \vdash \phi$ and $\mathcal{E} \models T$.

At first sight, the method of proving $A \models \phi$ by means of a formal proof of $\phi$ out of $T$ seems very inefficient. Instead of verifying $A \models \phi$, one has to verify $A \models \psi$ 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 $\phi$ is more complicated than the formulas of $T$ and that a direct verification of $A \models \phi$ is much more work than the formal proof and all verifications $A \models \psi$ together. Secondly, it may occur that a single theory
T with $\mathcal{A} \models T$ is used to prove many formulas $\phi$, so that many verifications $\mathcal{A} \models \phi$ are balanced against many formal proofs of $\phi$ out of $T$ and a single set of verifications $\mathcal{A} \models \psi$. Especially when constructing formal proofs is considered easier than 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 $\phi$ 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 $\phi$ 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 $\phi$ 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 $\phi$ belongs to a certain class of formulas (the positive ones), it can. So, if $\phi$ is positive, we can use the following theorem: "If $\mathcal{A} \models T$, $T \models \phi$, $\phi$ 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 $U \models \phi$ for many formulas $\phi$ 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 "$H(T) \models \phi$" from "$T \models \phi$" and "$\phi$ is positive", and $H(T) \models \phi$ implies that $\phi$ 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 $M \models \phi$ such that in each step either the formula $\phi$ 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
proven by means of a module $M$ with $\mathcal{A} \models M$ and a formal proof of $\phi$ out of $M$. We now turn to the formal definitions.

1.4. Signatures

Let $\text{Names}$ be a given set of names and let $\mathbb{N}$ be the set of natural numbers.

A function declaration is an expression $f(n)$ with $f \in \text{Names}$ and $n \in \mathbb{N}$. A function declaration $f(0)$ of arity 0 is sometimes called a constant declaration.

A predicate declaration is an expression $p(n)$ with $p \in \text{Names}$ and $n \in \mathbb{N}$.

A signature $\sigma$ is a set of function and predicate declarations.

1.5. $\sigma$-Algebras

Let $\sigma$ be a signature. A $\sigma$-algebra $\mathcal{A}$ is a pair of a set $D_{\mathcal{A}}$ and a function $\sigma$ that maps

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

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

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

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

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

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

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

for all $p(n) \in \sigma$ 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 $h : \mathcal{A} \rightarrow \mathcal{B}$.

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

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

$f^{\mathcal{B}} = f^{\mathcal{A}}$ for all $f(n) \in \rho \cap \sigma$.

$p^{\mathcal{B}} = p^{\mathcal{A}}$ for all $p(n) \in \rho \cap \sigma$.

A congruence on a $\sigma$-algebra $\mathcal{A}$ is an equivalence relation $\equiv$ on $D_{\mathcal{A}}$ satisfying, for $x_i, y_i \in D_{\mathcal{A}}$,

- $\forall f(n) \in \sigma$: if $x_i \equiv y_i$ ($i = 1, \ldots, n$) then $f^{\mathcal{A}}(x_1, \ldots, x_n) \equiv f^{\mathcal{A}}(y_1, \ldots, y_n)$,
- $\forall p(n) \in \sigma$: if $x_i \equiv y_i$ ($i = 1, \ldots, n$) then $p^{\mathcal{A}}(x_1, \ldots, x_n) \equiv p^{\mathcal{A}}(y_1, \ldots, y_n)$.

(Sometimes (weak) congruences appear in the literature, that are not required to satisfy the congruence properties for predicates.) For $x \in D_{\mathcal{A}}$, the congruence class $[x]$ of $x$ is the set of all $y \in D_{\mathcal{A}}$ with $y \equiv x$. 

Modular specification of process algebras

299
For $A$ a $\sigma$-algebra and $\equiv$ a congruence on $A$, the $\sigma$-algebra $A/\equiv$, called $A$ modulo $\equiv$, is defined by

$$D_{A/\equiv} = \{ (x) : x \in D_A \},$$

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

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

Due to the congruence properties, this definition is independent of the choice of the representing $x_i \in (X_i)$.  

1.6. Logics

A logic $\mathcal{L}$ is a complex of prescriptions, defining for any signature $\sigma$
- a set $F^\vdash_{\mathcal{L}}$ of formulas over $\sigma$ such that $F^\vdash_{\mathcal{L}} \cap F^\vdash_{\mathcal{L}} = F^\vdash_{\sigma \cap \rho}$,
- a binary relation $|=_{\mathcal{L}}$ on $\sigma$-algebras $\times F^\vdash_{\mathcal{L}}$ such that for all $\rho$-algebras $A$ and $\phi \in F^\vdash_{\sigma \cap \rho}$, $A |=_{\mathcal{L}} \phi$ if and only if $A |=_{\mathcal{L}} \phi$, and
- a set $I^\vdash_{\mathcal{L}}$ of inference rules $\phi$ with $P \subseteq F^\vdash_{\mathcal{L}}$ and $\phi \in F^\vdash_{\mathcal{L}}$.

If $A |=_{\mathcal{L}} \phi$, we say that the $\sigma$-algebra $A$ satisfies the formula $\phi$, or that $\phi$ holds in $A$.

A theory $T$ is a set of formulas over $\sigma$. If $T$ is a theory over $\sigma$ and $A |=_{\mathcal{L}} \phi$ for all $\phi \in T$, we say that $A$ satisfies $T$, notation $A |=_{\mathcal{L}} T$. We also say that $A$ is a model of $T$. An inference rule $\phi$ with $P = \emptyset$ is called an axiom and will be denoted simply by $\phi$.

A logic $\mathcal{L}$ is sound if $\phi \in I^\vdash_{\mathcal{L}}$ implies $A |=_{\mathcal{L}} \phi$ for any $\sigma$-algebra $A$.

A formula $\phi \in F^\vdash_{\mathcal{L}}$ is preserved under subalgebras if $A \models_{\mathcal{L}} \phi$ implies $B \models_{\mathcal{L}} \phi$, for any subalgebra $B$ of $A$.

A formula $\phi \in F^\vdash_{\mathcal{L}}$ is preserved under homomorphisms if $A \models_{\mathcal{L}} \phi$ implies $B \models_{\mathcal{L}} \phi$, for any homomorphic image $B$ of $A$.

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 A.

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 $\sigma$ be a signature. The set $T(\sigma)$ of $\sigma$-terms is defined inductively by

$$x \in T(\sigma) \text{ for any variable } x,$$

if $F(f, n)$ is in $\sigma$ and $t_1 \in T(\sigma)$ for $i = 1, \ldots, n$ then $f(t_1, \ldots, t_n) \in T(\sigma)$.  

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 $\zeta : \mathcal{V} \to T(\sigma)$. By $t[\zeta]$ we denote the result of simultaneous substitution for $x \in \mathcal{V}$ of $\zeta(x)$ for all occurrences of $x$ in $t$. By $u/x$ 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 $\cdot \mapsto 1$ that takes every variable $x$ into an element of $\mathcal{D}_\mathcal{A}$. The $\zeta$-evaluation $[[t]]^\zeta_\mathcal{A}$ of a σ-term $t$ in $\mathcal{A}$ is defined by

- $[[x]]^\zeta = \zeta(x)$,
- $[[f(t_1, \ldots, t_n)]]^\zeta = f^\mathcal{A}([[t_1]]^\zeta, \ldots, [[t_n]]^\zeta)$.

### 1.6.2. Equational logic

The set $F^\sigma_{rel}$ of equations or equational formulas over $\sigma$ is defined by

- if $l_1 \in T(\sigma)$ for $i = 1, 2$ then $(t_1 = t_2) \in F^\sigma_{rel}$.

Let $\phi$ be an equation $(t_1 = t_2) \in F^\sigma_{rel}$. We say that $\phi$ is $\zeta$-true in a σ-algebra $\mathcal{A}$, notation $\mathcal{A}, \zeta \models_{eq}^\sigma \phi$, if $[[t_1]]^\zeta = [[t_2]]^\zeta$. $\phi$ is true in $\mathcal{A}$, notation $\mathcal{A}, \zeta \models_{eq}^\sigma \phi$, if $\mathcal{A}, \zeta \models_{eq}^\sigma \phi$ for all valuations $\zeta$.

An inference system $I^\sigma_{eq}$ for equational logic is displayed in Table 1, where $t, u$ and $v$ are terms over $\sigma$ and $x$ is a variable.

### 1.6.3. Conditional equational logic

The set $F^\sigma_{con}$ of atomic formulas over $\sigma$ is defined by

- if $\mathcal{R}(p, n)$ is in $\sigma$ and $t_1, \ldots, t_n \in T(\sigma)$ then $p(t_1, \ldots, t_n) \in F^\sigma_{con}$.

The set $F^\sigma_{con}$ of conditional equational formulas over $\sigma$ is defined by

- if $C \subseteq F^\sigma_{con}$ and $\phi \in F^\sigma_{con}$ then $(C \Rightarrow \phi) \in F^\sigma_{con}$.

Let $\zeta$ be a substitution. For $\mathcal{A} \models F^\sigma_{con}$, $C \subseteq F^\sigma_{con}$ and $\phi \in F^\sigma_{con}$, $\zeta[\cdot], C[\cdot]$ and $\phi[\cdot]$ are defined as the result of applying $\zeta$ to all terms in $\mathcal{A}$, $C$ and $\phi$, respectively.

The $\zeta$-truth of formulas $\phi \in F^\sigma_{con}$ in a σ-algebra $\mathcal{A}$ is defined by

- $\mathcal{A}, \zeta \models_{con}^\sigma \phi$ if $\phi \in F^\sigma_{con}$ and $\mathcal{A}, \zeta[\cdot] \models_{eq}^\sigma \phi$,
- $\mathcal{A}, \zeta \models_{con}^\sigma p(t_1, \ldots, t_n)$ if $([[t_1]]^\zeta, \ldots, [[t_n]]^\zeta) \in p^\mathcal{A}$,
- $\mathcal{A}, \zeta \models_{con}^\sigma C \Rightarrow \beta$ if $\mathcal{A}, \zeta[\cdot] \models_{eq}^\sigma \beta$ for some $\beta \in C$ or $\mathcal{A}, \zeta \models_{eq}^\sigma x$.

$\phi$ is true in $\mathcal{A}$, notation $\mathcal{A} \models_{con}^\sigma \phi$, if $\mathcal{A}, \zeta \models_{con}^\sigma \phi$ for all valuations $\zeta$.

<table>
<thead>
<tr>
<th>$t = t$</th>
<th>$u = v$</th>
<th>$t = u$</th>
<th>$u = v$</th>
<th>$u = v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v = u$</td>
<td>$t[u/x] = t[v/x]$</td>
<td>$v = t$</td>
<td>$a[t/x] = a[u/x]$</td>
<td>$v = u$</td>
</tr>
</tbody>
</table>
An inference system \( F_\text{eq}^\text{fin} \) for conditional equational logic is displayed in Table 2, where \( z \) and \( z \) are atomic formulas, \( C \) is a set of atomic formulas, \( \phi \) is a conditional equational formula, \( \varsigma \) is a substitution, \( t, u \) and \( v \) are terms over \( \sigma \) and \( x \) is a variable. A conditional equational formula \( 0 \Rightarrow x \) is denoted by \( x \).

The logic described in Table 2 is *infinite conditional equational logic*. Finitary conditional equational logic is obtained by the extra requirement that in conditional equational formulas \( C \Rightarrow x \) the set of conditions \( C \) should be finite. In that case the inference rule

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

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

1.6.4. Expressiveness

One can translate an equation \( zc F_\text{eq}^\text{fin} \) by a (finitary) conditional equational formula \( 0 \Rightarrow x \) and a finitary conditional equational formula \( \{ z_1, \ldots, z_n \} \Rightarrow x \) into a first-order formula \( \{ z_1, z_2, \ldots, z_n \} \Rightarrow x \) (see Appendix A). Using this translation we have \( F_\text{eq}^\text{fin} \subseteq F_\text{fin}^\text{eq} \subset F_\text{fin}^\text{all} \) and, furthermore, \( \mathcal{A} \models \text{eq}_a \phi \iff \mathcal{A} \models \text{eq}_a \phi \) for \( \phi \in F_\text{eq}^\text{fin} \) and \( \mathcal{A} \models \text{eq}_a \phi \iff \mathcal{A} \models \text{fin}_a \phi \) for \( \phi \in F_\text{eq}^\text{fin} \). This means that first-order logic with equality is more expressive than 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 \( \sigma \)-proof of a formula \( \phi \vdash F_\text{eq}^\text{fin} \) from a theory \( T \subseteq F_\text{eq}^\text{fin} \) using the logic \( \mathcal{L} \) is a well-founded, upwardly branching tree of which the nodes are labeled by \( \sigma \)-formulas, such that

- the root is labeled by \( \phi \), and
- if \( \psi \) 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 $P = \emptyset$, 
- or $\psi \in I^T_{\sigma}$.

If a $\sigma$-proof of $\phi$ from $T$ using $\mathcal{L}$ exists, we say that $\phi$ is $\sigma$-provable from $T$ by means of $\mathcal{L}$, notation $T \vdash ^{\mathcal{L}}_\sigma \phi$.

**Truth.** Let $\mathcal{C}$ be a class of $\sigma$-algebras and $\phi \in F^\mathcal{L}_{\sigma}$. Then $\phi$ is said to be true in $\mathcal{C}$, notation $\mathcal{C} \models ^{\mathcal{L}}_\sigma \phi$, if $\phi$ holds in all $\sigma$-algebras $\mathcal{A} \in \mathcal{C}$. By $\text{Alg}(\sigma, T)$ we denote the class of all $\sigma$-algebras satisfying $T$.

**Soundness theorem.** If $\mathcal{L}$ is sound then $T \vdash ^{\mathcal{L}}_\sigma \phi$ implies $\text{Alg}(\sigma, T) \models ^{\mathcal{L}}_\sigma \phi$.

**Proof.** Straightforward by induction. □

If no confusion is likely to result, the subscripts and superscripts of $|=\ $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.

**Theorem 1.1.** $T \vdash ^{\text{eq}}_\sigma \phi \iff \text{Alg}(\sigma, T) \models ^{\text{eq}}_\sigma \phi$.

An instance of an equation $(u=v) \in F^\text{eq\_1}_{\sigma}$ is an equation $t[u/x][\zeta] = t[v/x][\zeta]$. An equational $\sigma$-proof of an equation $(u=v) \in F^\text{eq\_1}_{\sigma}$ is a string $t_0 = t_1 = \cdots = t_n$ ($n \in \mathbb{N}$) of $\sigma$-terms $t_i$, with $t_0 = u$ and $t_n = v$, such that, for $i = 1, \ldots, n$, $(t_{i-1} = t_i)$ or $(t_i = t_{i-1})$ is an instance of an equation from $T$. Write $u \overset{T}{\sigma} v$ if such a proof exists.

**Theorem 1.2.** $u \overset{T}{\sigma} v \iff T \vdash ^{\text{eq\_1}}_\sigma (u=v)$.

**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 [40, 43].

A substitution instance of a conditional equational formula \( \phi \) is a conditional equational formula of the form \( \phi[A] \). A conditional equational \( \sigma \)-proof of a formula \( (C \Rightarrow x) \in F^\text{eq} \) from a theory \( T \subseteq F^\text{eq} \) is a well-founded, upwardly branching tree of which the nodes are labeled by atomic \( \sigma \)-formulas, such that

- the root is labeled by \( x \), and
- if \( \beta \) 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 \( D = \emptyset \), or
  - \( D \Rightarrow \beta \) is a substitution instance of a conditional equational formula \( \phi \) from \( T \) or the bottom part of Table 2.

An example of a (fragment of) a conditional equational proof is given in Fig. 2 in Section 3.1.1. Write \( C \overset{T}{\Rightarrow} x \) if such a proof exists.

**Theorem 1.3.** The following three statements are equivalent:

1. \( C \overset{T}{\Rightarrow} x \),
2. \( T \vdash \text{eq} C \Rightarrow x \),
3. \( \text{Alg}(\sigma, T) \vdash \text{eq} C \Rightarrow x \).

**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 difficult part of the proof, which is the implication (3) \( \Rightarrow \) (1).

Suppose \( \text{Alg}(\sigma, T) \vdash \text{eq} C \Rightarrow x \). Let \( \mathcal{G} \) be the \( \sigma \)-algebra with

\[
D_\mathcal{G} = T(\sigma),
\]

\[
f^\mathcal{G}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n) \quad \text{for} \quad f(f, n) \in \sigma, \quad \text{and}
\]

\[
(t_1, \ldots, t_n) \in p^\mathcal{G} \quad \text{iff} \quad \left( C \overset{T}{\sigma} p(t_1, \ldots, t_n) \right) \quad \text{for} \quad R(p, n) \in \sigma.
\]

Define the relation \( \equiv \) on \( D_\mathcal{G} \) by \( t \equiv u \) if \( (C \overset{T}{\sigma} (t = u)) \). By definition of \( C \overset{T}{\sigma} \), \( \equiv \) is a congruence; so, \( \mathcal{G}/\equiv \) is again a \( \sigma \)-algebra.

Any function \( \xi : \gamma' \rightarrow T(\sigma) \) is both a valuation in \( \mathcal{G} \) and a \( \sigma \)-substitution. Let \( \xi'/\equiv \) be the valuation in \( \mathcal{G}/\equiv \) defined by \( \xi'/\equiv(x) = (\xi(x))_\equiv \). Any valuation \( \xi' \) in \( \mathcal{G}/\equiv \) is of the form \( \xi'/\equiv \) for certain \( \xi \): take \( \xi(x) \in \xi'(x) \) for \( x \in \gamma' \). By induction, one easily establishes that, for any \( t \in T(\sigma) \), \( \llbracket t \rrbracket _{\mathcal{G}/\equiv} = (\llbracket t \rrbracket _{\mathcal{G}})'_\equiv = (\llbracket t \rrbracket _{\mathcal{G}})'_\equiv \).
Claim. For any $\beta \in F^2$, we have $(\mathcal{C} / =, \xi / = \models \beta) \iff (C \frac{T}{\mathcal{C}} \beta[\xi])$.

Proof of the Claim. Let $\beta$ 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

$$
\begin{align*}
\mathcal{C} / =, \xi / = \models p(t_1, \ldots, t_n) & \iff (\llbracket t_1 \rrbracket \xi / =, \ldots, \llbracket t_n \rrbracket \xi / =) \models p / = \\
& \iff (t_1[\xi], \ldots, t_n[\xi]) \models p / \\
& \iff \left( C \frac{T}{\mathcal{C}} p(t_1[\xi], \ldots, t_n[\xi]) \right) \\
& \iff \left( C \frac{T}{\mathcal{C}} p(t_1, \ldots, t_n)[\xi] \right).
\end{align*}
$$

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

$$
\begin{align*}
\mathcal{C} / =, \xi / = \models (t_1 = t_2) & \iff \llbracket t_1 \rrbracket \xi / = \llbracket t_2 \rrbracket \xi / \\
& \iff (t_1[\xi], t_2[\xi]) \models t_1 \equiv t_2[\xi] \\
& \iff \left( C \frac{T}{\mathcal{C}} p(t_1[\xi] = t_2[\xi]) \right) \\
& \iff \left( C \frac{T}{\mathcal{C}} p(t_1 = t_2)[\xi] \right). \qedhere
\end{align*}
$$

Proof of Theorem 1.3 (conclusion). Using this claim, we prove that $\mathcal{C} / = \models \beta$ for all $\beta \in \mathcal{C}$. The corresponding proof trees (one for each $\beta \in \mathcal{C}$) can be combined into one big proof tree, proving $C \frac{T}{\mathcal{C}} \gamma[\xi] \models \beta$ by applying the substitution instance $(D \Rightarrow \gamma)[\xi]$ of the conditional equational formula $(D \Rightarrow \gamma) \in \mathbb{T}$. Hence $\mathcal{C} / =, \xi / = \models \gamma$, which had to be proved.

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

The set \( \mathcal{M} \) of module expressions is defined inductively by

- if \( \sigma \) is a signature and \( T \) a theory over \( \sigma \) then \( (\sigma, T) \in \mathcal{M} \),
- if \( M \in \mathcal{M} \) and \( N \in \mathcal{M} \) then \( M + N \in \mathcal{M} \),
- if \( \sigma \) is a signature and \( M \in \mathcal{M} \) then \( \sigma \triangleleft M \in \mathcal{M} \),
- if \( M \in \mathcal{M} \) then \( H(M) \in \mathcal{M} \),
- if \( M \in \mathcal{M} \) then \( S(M) \in \mathcal{M} \).

Here \( + \) is the composition operator, allowing one to organize specifications in a modular way, and \( \triangleleft \) 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 [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, [36].

The visible signature \( \Sigma(M) \) of a module expression \( M \) is defined inductively by

\[
\begin{align*}
\Sigma(\sigma, T) &= \sigma, \\
\Sigma(M + N) &= \Sigma(M) \cup \Sigma(N), \\
\Sigma(\sigma \triangleleft M) &= \sigma \cap \Sigma(M), \\
\Sigma(H(M)) &= \Sigma(M), \\
\Sigma(S(M)) &= \Sigma(M).
\end{align*}
\]

Truth. The class \( \text{Alg}(M) \) of models of a module expression \( M \) is defined inductively by

- \( \mathcal{A} \) is a model of \( (\sigma, T) \) if it is a \( \sigma \)-algebra satisfying \( T \),
- \( \mathcal{A} \) is a model of \( M + N \) if it is a \( \Sigma(M + N) \)-algebra such that \( \Sigma(M) \triangleleft \mathcal{A} \) is a model of \( M \) and \( \Sigma(N) \triangleleft \mathcal{A} \) is a model of \( N \),
- \( \mathcal{A} \) is a model of \( \sigma \triangleleft M \) if it is the restriction of a model of \( M \) to the signature \( \sigma \),
- \( \mathcal{A} \) 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 \( \text{Alg}(M) \) is a generalization of \( \text{Alg}(\sigma, T) \) as defined earlier. All the elements of \( \text{Alg}(M) \) are \( \Sigma(M) \)-algebras. A \( \Sigma(M) \)-algebra \( \mathcal{A} \in \text{Alg}(M) \) is said to satisfy \( M \). A formula \( \phi \in F_{\Sigma(M)}^{\mathcal{A}} \) is satisfied by a module expression \( M \), notation \( M \models \phi \), if \( \text{Alg}(M) \models \phi \); thus, if \( \phi \) holds in all \( \Sigma(M) \)-algebras satisfying \( M \). A theory \( T \subseteq F_{\Sigma(M)}^{\mathcal{A}} \) is satisfied by a module expression \( M \), notation \( M \models T \), if \( M \models \phi \) for all \( \phi \in T \).

**Derivability.** A proof of a formula \( \phi \in F_{\Sigma(M)}^{\mathcal{A}} \) from a module expression \( M \) using the logic \( \mathcal{L} \) is a well-founded, upwardly branching tree of which the nodes are labeled by assertions \( N \models \psi \), such that

- the root is labeled by \( M \models \phi \), and
- if \( N \models \psi \) is the label of a node \( q \) and \( P \) is the set of labels of the nodes directly above \( q \), then \( \frac{P}{N \models \psi} \) is one of the inference rules of Table 3.

In Table 3, **positive** and **universal** are syntactic criteria, to be defined for each logic \( \mathcal{L} \) 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 \( N \models \psi \) for \( \frac{\psi}{N \models \psi} \), and omit braces in the conditions of inference rules. If a proof of \( \phi \) from \( M \) using \( \mathcal{L} \) exists, we say that \( \phi \) is **provable** from \( M \) by means of \( \mathcal{L} \), notation \( M \models \phi \).

**Lemma 1.4.** If \( M \models \phi \) then \( \phi \in F_{\Sigma(M)}^{\mathcal{A}} \).

**Proof.** By induction. The only nontrivial cases are the rules for \( + \) and \( \square \). These follow from \( F_\phi^\mathcal{A} \subseteq F_\phi^{\mathcal{A}'} \) and \( F_\phi^{\mathcal{A}'} \cap F_\phi^{\mathcal{A}''} \subseteq F_\phi^{\mathcal{A'}\mathcal{A}''} \), respectively. \( \square \)

**Theorem 1.5.** If \( \mathcal{L} \) is sound then \( M \models \phi \) implies \( M \models \phi \).

---

**Table 3**

<table>
<thead>
<tr>
<th>( (\sigma, T) \models \phi )</th>
<th>( M \models \phi )</th>
<th>( N \models \phi )</th>
<th>( M \models \phi )</th>
<th>( M \models \phi )</th>
<th>( H(M) \models \phi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( N \models \phi )</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
</tr>
<tr>
<td>( \sigma ; M \models \phi )</td>
<td>( \sigma ; M \models \phi )</td>
<td>( \sigma ; M \models \phi )</td>
<td>( \sigma ; M \models \phi )</td>
<td>( \sigma ; M \models \phi )</td>
<td>( \sigma ; M \models \phi )</td>
</tr>
<tr>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
</tr>
<tr>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
</tr>
<tr>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
</tr>
<tr>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
</tr>
<tr>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
</tr>
<tr>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
</tr>
<tr>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
<td>( M \models \phi ) ((j \in J))</td>
</tr>
</tbody>
</table>
Proof. By induction to the size of proof trees. Suppose \( M \vdash \phi \) by means of a proof tree whose last step is an application of an inference rule \( \frac{\quad}{\phi} \) 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 \( N \models \phi \). It has to be established that \( M \models \phi \), i.e. that, for all models \( \mathcal{A} \) of \( M \), we have \( \mathcal{A} \models \phi \). Now six cases can be distinguished.

1. \( P = \emptyset, M = (\sigma, T) \) and \( \phi \in T \).

Let \( \mathcal{A} \) be a model of \( M = (\sigma, T) \). Then \( \mathcal{A} \models \phi \) for all \( \phi \in T \); so, in particular, \( \mathcal{A} \models \phi \).

2. \( P = \{ M \dashv \vdash \phi \} \) and \( \phi \in I \).

Let \( \mathcal{A} \) be a model of \( M \). By induction \( \mathcal{A} \models \phi \) for \( \phi \in T \). Thus, \( \mathcal{A} \models \phi \), since \( \phi \) is sound.

3. \( M = N_1 + N_2 \) and \( P = \{ N_i \dashv \vdash \phi \} (i = 1 \text{ or } 2) \).

By Lemma 1.4, \( \phi \in F_{\rho \cap \sigma}^{\phi} \). Let \( \mathcal{A} \) be a model of \( M \). Taking \( \sigma = \Sigma(N_i) \) and \( \rho = \Sigma(M) \), by the requirement of a logic that for all \( \rho \)-algebras \( \mathcal{A} \) and \( \phi \in F_{\rho \cap \sigma}^{\phi} \), \( \mathcal{A} \models \phi \), we find \( \mathcal{A} \models \phi \).

4. \( M = \sigma \sqcap N, P = \{ N \dashv \vdash \phi \} \) and \( \phi \in F_{\rho \cap \sigma}^{\phi} \).

By Lemma 1.4, \( \phi \in F_{\rho \cap \sigma}^{\phi} \). Let \( \mathcal{A} \) be a model of \( M = \sigma \sqcap N \). Then there is a model \( \mathcal{B} \) of \( N \) with \( \sigma = \sigma \sqcap \sigma \mathcal{B} \). By induction, \( \mathcal{A} \models \phi \).

5. \( M = H(N), P = \{ N \dashv \vdash \phi \} \) and \( \phi \) is positive.

Let \( \mathcal{A} \) be a model of \( M = H(N) \). By induction, \( \mathcal{A} \models \phi \).

6. \( M = S(N), P = \{ N \dashv \vdash \phi \} \) and \( \phi \) is universal.

Let \( \mathcal{A} \) be a model of \( M = S(N) \). By induction, \( \mathcal{A} \models \phi \).

\[ \text{Modules. This paper deals with module expressions, which are syntactic objects, rather than modules, in the sense of semantic objects which are denoted by module expressions. Nevertheless, it may be helpful to have an idea what these modules could be. Here several options are available (see for instances [11]). One possibility is that a module expression denotes the class of its models, or the class of its countable models. Another possibility is that it denotes the set of all formulas which are derivable from it. However, the simplest and least abstract interpretation, which may be considered to be the one employed in this paper, is the one in which a module is an annotated and structured collection of axiom systems: a module expression simply denotes itself.} \]

1.11. Completeness

All logics mentioned in Section 1.6 and Appendix A are sound and complete:

\[ \text{Alg}(\sigma, T) \models \phi \iff T \models \phi. \]
As a corollary, using the translations of Section 1.6.4, we have

\[ T \models \equiv_{\sigma} \phi \iff T \models \equiv_{\sigma} \phi \quad \text{for } \phi \in F_{\sigma}^{\equiv} \]  

and

\[ T \models \equiv_{\sigma} \phi \iff T \models \text{freeq} \phi \quad \text{for } \phi \in F_{\sigma}^{\text{freeq}}. \]  

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 Theorems 3.4 and 3.9, respectively) and for first-order logic with equality this is still an open problem (as far as we know). Here a logic \( \mathcal{L} \) is complete if

\[ M \equiv \phi \Rightarrow M \equiv \phi \]  

It is easily proved that

\[ M \equiv \phi \Rightarrow M \equiv \phi \quad \text{for } \phi \in F_{\Sigma(M)}^{\equiv}, \]  

but the reverse directions do not hold (as is also shown by the counterexamples of Theorems 3.4 and 3.9). 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.

**Lemma 1.6.** If \( M \equiv \phi \) has been established before, then \( M \models \phi \) may occur as a leaf in a proof tree. Likewise, if \( T \equiv \phi \) has been established before, then \( \phi \) 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. □

**Lemma 1.7.** \((\sigma_0, T_0) + \cdots + (\sigma_n, T_n) \equiv \phi \Rightarrow (\bigcup_{i=0}^{n} \sigma_i, \bigcup_{i=0}^{n} T_i) \equiv \phi.\)

**Proof.** (\(\Rightarrow\)): Replace all modules that appear on the left of turn-styles in a proof of \( \phi \) 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 +.

(\(\Leftarrow\)): Also straightforward. □
Lemma 1.8. \( (\sigma, T) \models_{\mathcal{D}} \phi \iff T \models_{\mathcal{D}} \phi. \)

Proof. Trivial. \( \square \)

These three lemmas say that (parts of) proofs from modules that do not contain the operators \( \oplus, H \) 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 1.6 again and observation (1.2), 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 \((\sigma, T)\) will sometimes be introduced by mentioning only \(T\). In such cases \(\sigma\) is understood to be the signature of all functions and predicates appearing in \(T\). Outside this Section 1 and Appendix A inference rules \(\mathcal{F}\) do not occur, but all conditional equational formulas \(C \Rightarrow x\) are written as \(\xi\), as is usual. However, the suggested similarity between inference rules and conditional equational formulas is misleading: \(\mathcal{F}\) holds in an algebra \(\mathcal{A}\) if \((\mathcal{A}, \xi \models \psi \text{ for all } \psi \in P \text{ and all valuations } \xi)\) implies \((\mathcal{A}, \xi \models \phi \text{ for all valuations } \xi)\), while \(\mathcal{F}\) holds in \(\mathcal{A}\) if for all valuations \(\xi\): \((\mathcal{A}, \xi \models \beta \text{ for all } \beta \in C \text{ implies } \mathcal{A}, \xi \models x)\).

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 [10]. We will use, as much as possible, the notations and terminology from this reference.

2.1. ACP'

In this paper a central role will be played by the module ACP', the algebra of communicating processes with abstraction. A first parameter of ACP' 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 \(\delta\), denoting deadlock, inaction, a process that does nothing at all. In particular, \(\delta\) does not terminate successfully. We write \(A_\delta = A \cup \{\delta\}\).
Next we have a parallel composition operator $\parallel x \parallel y$ denotes the process corresponding to the parallel execution of $x$ and $y$. Execution of $x \parallel y$ 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 $ACP'$, which is a partial communication function $\gamma : A \times A \rightarrow A$ 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. By $\gamma(a,b) \downarrow$ 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: $ACP'(A,\gamma)$. However, in order to keep the notation simple, we will always omit the parameters if this can be done without causing confusion. In order to axiomatize the $\parallel$-operator, we use two auxiliary operators $\parallel_\perp$ (left-merge) and $\parallel_\|$ (communication merge). $x \parallel_\perp y$ is $x \parallel y$, but with the restriction that the first step comes from $x$, and $x \parallel_\| y$ and $x \parallel y$ but with a synchronization action as the first step.

Next we have for each $H \subseteq A$ an encapsulation operator $\delta_H$. The operator $\delta_H$ 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 $\tau \notin A$. We write $A_\tau$ for $A \cup \{\delta, \tau\}$. The fact that $\tau$ cannot synchronize makes, in some sense, this action unobservable. Therefore, it is often called the silent action. For each $I \subseteq A$ the language contains an abstraction or hiding operator $\tau_I$. This operator hides actions in $I$ by renaming them into $\tau$, thus expressing that certain actions in a system behavior cannot be observed.

In Table 4 we summarize the signature of module $ACP'$. Table 5 contains the theory of the module $ACP'$. All axioms in Table 5 are in fact axiom schemes in $a, b, H$.

<table>
<thead>
<tr>
<th>binary operators</th>
<th>+</th>
<th>alternative composition (sum)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>sequential composition (product)</td>
</tr>
<tr>
<td></td>
<td>$\parallel$</td>
<td>parallel composition (merge)</td>
</tr>
<tr>
<td></td>
<td>$\parallel_\perp$</td>
<td>left-merge</td>
</tr>
<tr>
<td></td>
<td>$\parallel_|$</td>
<td>communication merge</td>
</tr>
<tr>
<td>unary operators</td>
<td>$\delta_H$</td>
<td>encapsulation, for any $H \subseteq A$</td>
</tr>
<tr>
<td></td>
<td>$\tau_I$</td>
<td>abstraction, for any $I \subseteq A$</td>
</tr>
<tr>
<td>constants</td>
<td>$a$</td>
<td>for any atomic action $a \in A$</td>
</tr>
<tr>
<td></td>
<td>$\delta$</td>
<td>inaction, deadlock</td>
</tr>
<tr>
<td></td>
<td>$\tau$</td>
<td>silent action</td>
</tr>
</tbody>
</table>
and $I$. Here $a$ and $b$ range over $A_3^i$ (unless further restrictions are made in the table) and $H, I \subseteq A$. In a product $x \cdot y$ we will often omit the $·$. We take + to be less binding than other operations and $\cdot$ to be more binding than other operations. In case we are dealing with an associative operator, we also leave out parentheses. In this paper we present $\text{ACP}^r$ as a monolithic module. In [10] however, it is shown that $\text{ACP}^r$ can be viewed as the sum of a large number of submodules which are interesting in their own right. The module consisting of axioms $A_1$–5 only is called $\text{BPA}$ (from basic process algebra). If we add axioms $A_6$–7, we obtain $\text{BPA}_0$, and $\text{BPA}_0$ plus axioms $B_1$–2 gives $\text{BPA}_1$. The module $\text{ACP}$ consists of the axioms $A_1$–7, $\text{CF}_1$–2, $\text{CM}_1$–9 and $\text{D}_1$–4.

Let $D = \{d_1, \ldots, d_n\}$ be a finite set and let $t_{d_1}, \ldots, t_{d_n}$ be process expressions. We use the notation $\text{LED}_{D} t_d$ for the expression $t_{d_1} + \cdots + t_{d_n}$. Id$E D t_d = t$ by definition.

Proposition 2.1. $\text{ACP}^r \vdash a(t \cdot x \parallel y) = a(x \parallel y)$.

Proof. $a(t \cdot x \parallel y) \stackrel{\text{CM}_3}{=} (a(t \cdot x)) \parallel y \triangleright (a(x) \cdot t) \parallel y \stackrel{\text{B}_1}{=} a(x) \parallel y \stackrel{\text{CM}_3}{=} a(x \parallel y)$.

2.1.1. 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 $\text{ACP}^r$-axioms $A_1$, $A_2$ and $A_3$, respectively, it follows that $\subseteq$ is antisymmetrical, transitive and reflexive, and, hence, a partial order.

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

$$x \subseteq y \quad \Rightarrow \quad f(x) \subseteq f(y)$$
For reasons to be explained in Section 3, we prefer to state properties of ACP\(^*\)-processes in an equational rather than a conditional form. Therefore, we write the above monotonicity property as
\[ f(x) \leq f(x + y). \]

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\(^*\):
\[ x + z \leq x + y + z, \]
\[ x \cdot z \leq (x + y) \cdot z, \]
\[ x \sqcap z \leq (x + y) \sqcap z, \]
\[ \delta_H(x) \leq \delta_H(x + y), \]
\[ \tau_I(x) \leq \tau_I(x + y). \]

Due to branching time (see [23]), we cannot prove \[ z \cdot x \leq z \cdot (x + y), \]
\[ x \sqcap z \leq (x + y) \sqcap z \]
or \[ z \sqcup x \leq z \sqcup (x + y). \]
However, we do have the following weak form of monotonicity for the merge operator:
\[ \tau(x \parallel z) \leq \tau(x + y) \parallel z. \]

The last formula follows since
\[ \tau(x \parallel z) = \tau_x \sqcap z \leq (\tau x + y) \sqcap z \leq (\tau x + y) \parallel z. \]

2.2. Standard Concurrency

Often one adds to ACP\(^*\) the module SC of standard concurrency (Table 6). These axioms are not included in ACP\(^*\) because they can be derived for all closed ACP\(^*\)-terms.

**Proposition 2.2.** ACP\(^*\) + SC \(\vdash\)
(i) \[ x \parallel y = y \parallel x, \]
(ii) \[ x \parallel (y \parallel z) = (x \parallel y) \parallel z. \]

**Proof.** See [10]. \(\Box\)

2.3. Renamings

For every function \( f: A_1 \to A_1 \) with the property that \( f(\delta) = \delta \) and \( f(\tau) = \tau \), the module RN introduces a unary operator \( \rho_f \). Axioms for \( \rho_f \) are presented in Table 7 (where \( a \) ranges over \( A_1 \) and id is the identity function). Module RN is parametrized by \( A \).

For \( t \in A_1 \) and \( H \subseteq A \), we define mappings \( r_{\tau,H} : A_1 \to A_1 \) by
\[ r_{\tau,H}(a) = \begin{cases} t & \text{if } a \in H, \\ a & \text{otherwise.} \end{cases} \]
In the rest of this paper we will implicitly identify the operators \( \partial_r \) and \( \rho_{r,u} \), and also the operators \( \tau_i \) and \( \rho_{r,j} \): encapsulation is just renaming of actions into \( \delta \), and abstraction is renaming of actions into the silent step \( \tau \).

### 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 \( \gg \) and \( \gg \). In the process \( x \gg y \) the output of process \( x \) serves as input of process \( y \). Operator \( \gg \) is identical to operator \( \gg \) 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. Definition 2.3). In order to deal with such behaviors, it is useful to view \( \gg \) as the composition of two operators: the \( \gg \) operator and an abstraction operator that hides the internal communications. We will define the chaining operators in terms of the operators of \( \mathsf{ACP}^+ + \mathsf{RN} \). In this way we obtain a simple, finite axiomatization of the operators. The operator \( \gg \) occurs (in different notations) already in [28, 34].

Let for \( d \in D \), \( \ulcorner d \urcorner \) be the action of reading \( d \), and \( \lrcorner d \) be the action of sending \( d \). Furthermore, let \( \mathsf{ch}(D) \) be the following set of actions:

\[
\mathsf{ch}(D) = \{ \lrcorner d, [d,s(d), r(d), c(d) \mid d \in D] \}.
\]

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 \( \mathsf{ch}(D) \subseteq A \). The module should be used in a context with modules \( \mathsf{ACP}^+(A, \gamma) \) and \( \mathsf{RN}(A) \), where

\[
\text{range}(\gamma) \cap \{ \lrcorner d, [d, s(d), r(d) \mid d \in D] \} = \emptyset
\]

and communication on \( \mathsf{ch}(D) \) is defined by

\[
\gamma(s(d), r(d)) = c(d)
\]
(all other communications on or with \( \text{ch}(D) \) 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 \( a \in A_S \). Now the “concrete” chaining of processes \( x \) and \( y \), notation \( x \triangleright y \), is defined by means of the axiom \( (H = \{ s(d), r(d) \mid d \in D \}) \):

\[
x \triangleright y = \partial H(\rho(\uparrow s(x) \mid \rho(\downarrow r(y)))) \quad (\text{CH1}).
\]

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

\[
x \triangleright y = \tau I(x \triangleright y) \quad (\text{CH2}).
\]

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

The following laws can be easily proven from module \( \text{ACP}^+ + \text{SC} + \text{RN} + \text{CH}^+ \). Here \( d_0 \) and \( e_0 \) are elements of \( D \) and \( d \) and \( e \) are variables ranging over \( D \). L5 follows using Proposition 2.1:

\[
\uparrow d_0 \cdot x \triangleright \left( \sum_{e \in D} e \cdot y^e \right) = \tau(x \triangleright y^{d_0}), \quad (L1)
\]

\[
\uparrow d_0 \cdot x \triangleright \uparrow e_0 \cdot y = \uparrow e_0 \cdot (\uparrow d_0 \cdot x \triangleright y), \quad (L2)
\]

\[
\left( \sum_{d \in D} d \cdot x^d \right) \triangleright \left( \sum_{e \in D} e \cdot y^e \right) = \sum_{d \in D} \downarrow d \cdot \left( x^d \triangleright \left( \sum_{e \in D} e \cdot y^e \right) \right), \quad (L3)
\]

\[
\left( \sum_{d \in D} d \cdot x^d \right) \triangleright \uparrow e_0 \cdot y = \sum_{d \in D} \downarrow d \cdot (x^d \triangleright \uparrow e_0 \cdot y) + \uparrow e_0 \cdot \left( \left( \sum_{d \in D} d \cdot x^d \right) \triangleright y \right), \quad (L4)
\]

\[
a(\tau x \triangleright y) = a(x \triangleright y) = a(x \triangleright y). \quad (L5)
\]

The laws are equally valid when the operator \( \triangleright \) is replaced by \( \triangleright \), except for law L1, where, in addition, the \( \tau \) has to be replaced by \( c(d_0) \).

**Example.** Let \( D = \{0, 1\} \). Process \( \text{AND} \) reads two bits and then outputs 1 if both are 1, and 0 otherwise:

\[
\text{AND} = \downarrow 0 \cdot (\downarrow 0 \cdot \uparrow 0 + \downarrow 1 \cdot \uparrow 0) + \downarrow 1 \cdot (\downarrow 0 \cdot \uparrow 0 + \downarrow 1 \cdot \uparrow 1).
\]

Process \( \text{OR} \) reads two bits, outputs 0 if both are 0, and 1 otherwise:

\[
\text{OR} = \downarrow 0 \cdot (\downarrow 0 \cdot \uparrow 0 + \downarrow 1 \cdot \uparrow 1) + \downarrow 1 \cdot (\downarrow 0 \cdot \uparrow 1 + \downarrow 1 \cdot \uparrow 1).
\]
Process NEG reads a bit $b$ and outputs $1 - b$:

$$\text{NEG} = \downarrow 0 \cdot \uparrow 1 + \downarrow 1 \cdot \uparrow 0.$$  

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

$$(\text{NEG} \cdot \text{NEG} \triangleright \text{AND}) \triangleright \text{NEG} = \text{OR}.$$  

Note, however, that we do not have

$$(\text{NEG} \cdot \text{NEG} \triangleright \text{AND}) \triangleright \text{NEG} = \text{OR}$$  

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

### 2.5. Projection

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

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 $\pi : \mathbb{N} \times P \to P$ (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 $\alpha$ from processes to sets of actions is introduced, with axioms such as

$$\alpha(\delta) = \emptyset,$$

$$\alpha(ax) = \{a\} \cup \alpha(x),$$

$$\alpha(x + y) = \alpha(x) \cup \alpha(y).$$

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

<table>
<thead>
<tr>
<th>Table 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi_n(\tau) = \tau$</td>
</tr>
<tr>
<td>$\pi_n(ax) = \delta$</td>
</tr>
<tr>
<td>$\pi_{n+1}(ax) = a \cdot \pi_n(x)$</td>
</tr>
<tr>
<td>$\pi_n(\tau x) = \tau \cdot \pi_n(x)$</td>
</tr>
<tr>
<td>$\pi_n(x + y) = \pi_n(x) + \pi_n(y)$</td>
</tr>
</tbody>
</table>
algebraic solution. We view the alphabet of a process as a process again: \( \alpha \) 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 were represent a set of actions by a process. A set \( B \) of actions is represented by the process expression \( B = \sum_{a \in B} a \). So, the empty set is represented by \( \delta \), 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 2.1.1), fits with the notation for the subset predicate on sets.

The axioms in Table 9 can be used to compute the alphabet of finite processes. In the table \( \alpha \) ranges over the set \( A \) of actions which occurs as a parameter of module \( AB \). In order to compute the alphabet of infinite processes, we introduce an additional module \( AA \) (Table 10) which is also parametrized by \( A \).

It is not hard to see that the axioms of \( AA \) are derivable for all closed BPA-terms.

**Example (Baeten et al. [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 \( R \):

\[
\begin{align*}
\alpha(R) &= \alpha(Qb) = \alpha(\tau_{\{a\}}(P) \cdot b) = \alpha(\tau_{\{a\}}(P) \cdot \tau_{\{a\}}(b)) \\
&\overset{\text{TI4}}{=} \alpha(\tau_{\{a\}}(Pb)) \overset{\text{AB2}}{=} \tau_{\{a\}} \circ \delta_{\{a\}} \circ \alpha(Pb) \overset{\text{RN5}}{=} \delta_{\{a\}} \circ \alpha(Pb).
\end{align*}
\]

Since

\[
\alpha(Pb) = \alpha(aPb) = a + \alpha(Pb),
\]

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

\[
\alpha(\tau_{\{a\}}(Pb^n)) = \alpha(a^n \cdot \delta) \subseteq a
\]

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

\[
\alpha(R) = \delta_{\{a\}} \circ \alpha(Pb) = \delta_{\{a\}}(a) = \delta.
\]
Information about alphabets must be available if we want to apply the rules of Table 11. 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 (Table 11) is parametrized by $A$ and $\gamma$. Observe that axioms AA1 and RR1 together imply axiom RN4 of Table 7.

Axiom RR2, which describes the interaction between renaming and parallel composition, looks complicated, but that is only because it is so general. The axioms RR are derivable for closed 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 $\langle x \mid E \rangle$, 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 $\langle X \mid E \rangle$. Therefore, the convention will be adopted that once a recursive specification has been declared, $\langle x \mid E \rangle$ can be abbreviated by $X$. If this is done, $X$ is called a formal variable. Formal variables are denoted by capital letters. So, after the declaration $X = aX$, a statement $X = aaX$ should be interpreted as an abbreviation of $\langle X \mid X = aX \rangle = aa \langle X \mid X = aX \rangle$.

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

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

If the above convention is used, these formulas seem to be just the equations of $E$. The module REC is 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 nondeterminism 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 $a \in A_\ell$. 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.

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

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

**Definition 2.3.** Let $t$ be an open BPA$^*$-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$^*$-terms. For $x, y \in V_E$ we define:

\[ x \xrightarrow{w} y \iff \text{$y$ occurs unguarded in $x$.} \]

We call $E$ *guarded* if relation $\xrightarrow{\sim}$ is well-founded (i.e. there is no infinite sequence $x \xrightarrow{w} y \xrightarrow{w} z \xrightarrow{w} \cdots$).

**Theorem 2.4** (Recursive specification principle (RSP)). Let $E$ be a guarded recursive specification over the signature of BPA$^*$ and let $x \in V_E$. Then: BPA$^*$ + REC + PR + B + AIP$^-$ |- 

(RSP) \[ x = \langle x \mid E \rangle . \]

<table>
<thead>
<tr>
<th>Table 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0(x)$</td>
</tr>
<tr>
<td>$B_1(x)$</td>
</tr>
<tr>
<td>$B_0(x)$</td>
</tr>
<tr>
<td>$B_0(tx)$</td>
</tr>
<tr>
<td>$B_n(x)$</td>
</tr>
<tr>
<td>$B_{n+1}(ax)$</td>
</tr>
</tbody>
</table>
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 \( \{ p_y \mid y \in V_E \} \) and \( \{ q_y \mid y \in V_E \} \) which are both solutions of the equations of \( E \), that is, if we substitute for each occurrence of a variable \( y \) in \( E \) the corresponding processes \( p_y \) and \( q_y \), respectively, then we obtain valid statements. In such a situation RSP allows us to conclude that \( p_x = \langle x \mid E \rangle \) and also that \( q_x = \langle x \mid E \rangle \). Here it is important to note that \( \langle x \mid E \rangle \) 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.

**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

\[
\langle X \mid E \rangle = (a+b) \cdot \langle X \mid E \rangle = a \cdot \langle X \mid E \rangle + b \cdot \langle X \mid E \rangle
\]

\[
= a \cdot (a+b) \cdot \langle X \mid E \rangle + b \cdot \langle X \mid E \rangle,
\]

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

2.10. ACP\(_2\)

The combination of all modules presented thus far will be called ACP\(_2\). Formally, the module is defined by

\[
ACP_2 = ACP^+ + SC + RN + CH^+ + PR + AB
+ AA + RR + REC + B + AIP^-. 
\]

Branching bisimulation semantics, as described, for instance, in [10], gives a model for the module ACP\(_2\).

2.11. RSP\(^+\)

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

**Definition 2.5.** A process expression \( t \in T(\Sigma(ACP^+_2)) \) is called *guardedly specifiable* if there exists a guarded recursive specification \( E \) with \( x \in V_E \) such that

\[
ACP_2 \vdash t = \langle x \mid E \rangle.
\]
Theorem 2.6 (Generalized recursive specification principle (RSP+)). Let $\langle x \mid E \rangle$ be a guardedly specifiable process expression. Then $\text{ACP}_x \vdash (\text{RSP}^+) \frac{E}{x = \langle x \mid E \rangle}$.

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 $\text{ACP}_x \vdash \langle x \mid E \rangle = \langle y \mid F \rangle$.

Let $p$ be a conditional equational proof of $\langle x \mid E \rangle = \langle y \mid F \rangle$ from the theory $\text{ACP}_x$. Let $p'$ be the node-labeled tree obtained by replacing, for each $z \in V_F$, all occurrences of $\langle z \mid E \rangle$ in $p$ by $z$. We claim that $p'$ is a conditional equational proof of $E = \langle x \mid E \rangle$. 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 $\text{REC}$. 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 rules from Table 2 for symmetry and associativity of "=",

Remark. In the definition of the notion "guardedly specifiable", it is essential that the identity $t = \langle x \mid E \rangle$ is provable. If we would only require $t = \langle x \mid E \rangle$, then the corresponding version of RSP+ would not be provable from $\text{ACP}_x$, since this rule would then not be valid in the action relation model of [22]. Strictly speaking, in [22] a recursion construct $\langle x \mid E \rangle$ 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 $\langle x \mid \{x = x\}\rangle = \delta$ does not hold under this interpretation. However, if one interprets the construct $\langle x \mid E \rangle$ as a constant in the model of [22], then the most natural choice is to relate $\langle x \mid E \rangle$ to the bisimulation equivalence class of the term $\langle x \mid E \rangle$. Under this interpretation, $\langle x \mid \{x = x\}\rangle = \delta$. Hence, $\langle x \mid \{x = x\}\rangle = \langle y \mid \{y = \delta\}\rangle = \delta$. Since the specification $\{ y = \delta \}$ is guarded, this would mean that the expression $\langle x \mid \{x = x\}\rangle$ is guardedly specifiable. But then RSP+ gives that for arbitrary $z$: $z = \langle x \mid \{x = x\}\rangle = \delta$. This is clearly false.

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

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 \( \tau \)-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 Fig. 1. (In fact, the standard branching bisimulation semantics of [26] cannot be mapped homomorphically to failure semantics; see [14].)

For concrete process algebra (without \( \tau \)-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 represent the same process only in trace semantics, this guarantees only 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

\[
\text{branching bisimulation semantics with explicit divergence [26]}
\]

\[
\text{weak bisimulation semantics with explicit divergence [14]}
\]

\[
\text{ready trace semantics [8]}
\]

\[
\text{readiness semantics [37]} \quad \text{failure trace semantics [38]}
\]

\[
\text{failure semantics [14, 17, 21, 29]}
\]

\[
\text{trace semantics [28]}
\]

Fig. 1. The linear-time–branching-time spectrum.
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 identified only 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 \(ACP^f\), 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 \(ACP^f\). 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 module \(WBS\) (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 Fig. 1 in a setting with the language of \(BPA^f\). In fact, in combination with the laws of \(BPA^f\), the laws of Table 13 completely axiomatize the semantical notion of weak bisimulation for this language. Thus, any identity derived from \(ACP^f + WBS\) 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 \(ACP^f + WBS\) proves some very unintuitive identities. More specifically, we will introduce the notion of trace consistency and show that \(ACP^f + WBS\) is not trace consistent. However, first we have to introduce some auxiliary notation.

\textit{Notation for sequences.} Let \(K\) be any set. \(K^*\) 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 \(\lambda\) and sequence \(\rho \ast \sigma\) is the concatenation of sequences \(\rho\) and \(\sigma\). The sequence consisting only of \(k \in K\) is denoted by \(k\) as well.

| Table 13 |
|-----------------|---|
| \(xt = x\)      | T1 |
| \(tx + x = tx\) | T2 |
| \(a(tx + y) = a(tx + y) + ax\) | T3 |
Definition 3.1. Let $t$ be a closed expression in the signature of $\text{BPA}_t$. The \textit{trace set} $\text{tr}(t)$ of $t$ is defined inductively by
\begin{align*}
\text{tr}(\delta) &= \{\lambda\}, \\
\text{tr}(\tau) &= \{\lambda, \sqrt{\cdot}\}, \\
\text{tr}(a) &= \{\lambda, a, a\ast \sqrt{\cdot}\}, \\
\text{tr}(s+t) &= \text{tr}(s) \cup \text{tr}(t), \\
\text{tr}(s\cdot t) &= (\text{tr}(s) \cap \mathcal{A}^*) \cup \{\sigma \ast \rho \mid \sigma \ast \sqrt{\cdot} \in \text{tr}(s) \text{ and } \rho \in \text{tr}(t)\}. 
\end{align*}
Thus, trace sets are nonempty and prefix-closed. The special symbol $\sqrt{\cdot}$, which may occur at the end of a trace, denotes successful termination.

Definition 3.2. Let $M$ be a process module with $\Sigma(M) \equiv \Sigma(\text{BPA}_t)$. $M$ is called \textit{trace-consistent} if, for all closed $\text{BPA}_t$-expressions $s$ and $t$,
\[ M \models s = t \implies \text{tr}(s) = \text{tr}(t). \]
A model $\mathcal{M}$ of $M$ is \textit{trace-consistent} if, for all closed $\text{BPA}_t$-expressions $s$ and $t$,
\[ \mathcal{M} \models s = t \implies \text{tr}(s) = \text{tr}(t). \]
The module $\text{ACP}_t$ is trace-consistent because branching bisimulation semantics, as described in [10], gives a consistent model for this module. The module $\text{ACP}_t + \text{WBS}$, however, is in general not trace-consistent.

Proposition 3.3. Let $\text{ACP}_t$ and $\text{WBS}$ 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 $\text{ACP}_t + \text{WBS}$ is not trace-consistent.

Proof. We derive:
\[ c = \delta a + c = (\tau | b) a + c = (\tau a | b) + a | b \]
\[ = (\tau a + a) | b \not\equiv \tau a | b = (\tau | b) a = \delta. \]
This means that the module cannot be trace-consistent since
\[ \text{tr}(c) = \{\lambda, c, c\ast \sqrt{\cdot}\} \neq \{\lambda\} = \text{tr}(\delta). \]

This sudden inconsistency must be the result of a serious misunderstanding. And indeed, what is wrong is the use of $\text{ACP}_t$ in weak bisimulation semantics. It happens that weak bisimulation congruence (see [7]), which is the notion of equivalence used to construct algebras in the setting of weak bisimulation semantics, is a congruence for the $\ast$-operator but not for the $\mid$-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 $\parallel$ and $|$. The $\parallel$ 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 $\parallel$ and $|$ in ACP$^+$ 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$^+$: 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

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

denote the module $M$ in which the operators and predicates from signature $\sigma$ are hidden. Consider the module

$$sACP^+ = H(\{ f(\parallel, 2) \} \Delta ACP^+)$$

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$^+$ except for $|$, and since ACP$^+$ is sound w.r.t. branching bisimulation semantics, we conclude that $sACP^+$ is sound w.r.t. weak bisimulation semantics. Thus, $sACP^+$ 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 only positive formulas from module $sACP^+$ provable. The following example shows what goes wrong if we also allow nonpositive formulas. Analogous to the derivation which we used to show that ACP$^++$WBS is not trace-consistent, we prove that

$$\{ f(\parallel, 2) \} \Delta ACP^++WBS \not\vdash c = \delta.$$
Applying the results of Section 1.12 properly, this is one of the rare occasions where we have to construct a nontrivial conditional equational proof (Fig. 2). The leaves $c = (\tau a + a)\mid b$ and $\tau a\mid b = \delta$ in this proof tree have been established before using equational logic. Thus,

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

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

$$\{F(1, 2)\} A \text{ACP}^\tau \vdash \frac{\tau a + a = \tau a}{c = \delta}.$$ 

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

$$\{F(1, 2)\} A \text{ACP}^\tau + \text{WBS} \vdash c = \delta.$$ 

So, although the formulas provable from module $\{F(1, 2)\} A \text{ACP}^\tau$ contain no communication merge, some of them (which are nonpositive) cannot be combined consistently with the laws of weak bisimulation semantics.

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

**Theorem 3.4.** *Equational logic for modules is not complete.*

**Proof.** We prove that

$$\{F(1, 2)\} A \text{ACP}^\tau + \text{WBS} \models^{\text{rel}} c = \delta$$

even though

$$\{F(1, 2)\} A \text{ACP}^\tau + \text{WBS} \not\models^{\text{eqi}} c = \delta.$$ 

In the previous paragraph we have shown that

$$\{F(1, 2)\} A \text{ACP}^\tau + \text{WBS} \not\models^{\text{eqi}} c = \delta.$$
Due to the soundness of conditional equational logic and Theorem 1.5, this implies that

\[ \{ F(\mid, 2) \} \vartriangleleft ACP^\tau + \text{WBS} \models^{\text{eq}} c = \delta \]

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

\[ \{ F(\mid, 2) \} \vartriangleleft ACP^\tau + \text{WBS} \models^{\text{eq}} c = \delta \]

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

\[ H(\{ F(\mid, 2) \} \vartriangleleft ACP^\tau) + \text{WBS} \]

is trace-consistent and, therefore, does not prove \( c = \delta \). This result carries over to the setting of equational logic (cf. (1.4) in Section 1.11). 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,

\[ \{ F(\mid, 2) \} \vartriangleleft ACP^\tau + \text{WBS} \not \models^{\text{eq}} c = \delta. \]

\[ \square \]

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” a homomorphic mapping. One can think of the operation of action refinement which “lives” in certain noninterleaved 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 \( \tau \)) but not in failure trace semantics. Below, we show that the left-merge operator cannot be added to trace-consistent algebras in which the following axiom T4 is valid:

\[ \tau(x + y) = \tau x + y \quad (T4). \]

In a setting with the operators from BPA\(^\tau\), axiom T4 holds in all the semantics of Fig. 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\(^\tau\) can be added to the ready trace model for BPA\(^\tau\).
Proposition 3.5. Let $\text{sACP}'$ be parametrized by a set $A$ of actions which contains at least three elements $a$, $b$ and $c$. Then the module $\text{sACP}' + T4$ is not trace-consistent.

Proof. $\text{ACP}' + T4 \models \tau(ac + ca) + bc = \tau(\tau(ac + ca) + bc + c(\tau a + b))$, as

\[
\tau(\tau a + b) \parallel c = (\tau a + b) \parallel c = \tau(a \parallel c) + bc = \tau(ac + ca) + bc
\]

and

\[
\tau(\tau a + b) \parallel c = \tau((\tau a + b) \parallel c) = \tau(\tau(ac + ca) + bc + c(\tau a + b)).
\]

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

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 $\ast$. 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

\[
\text{SACP}' = H(\{ F(\bot, 2), F(\bot, 2) \Delta (\text{ACP}' + \text{SC})\}).
\]

This module is sound w.r.t. all semantics of Fig. 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 to be a natural idea to hide in a module all auxiliary operators using the $\Box$-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 $ACP'$, 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:

\[
(r(d) \triangleright (s(d)+s(e))) \triangleright r(e) = c(d) \cdot \delta,
\]

\[
r(d) \triangleright ((s(d)+s(e)) \triangleright r(e)) = c(e) \cdot \delta.
\]

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

\[
\frac{x \leq A', \; y \leq A', \; z \subseteq A'}{(x \triangleright y) \triangleright z = x \triangleright (y \triangleright z)} \quad \text{(CC)}.
\]

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 $ACP'$. The obvious proof strategy would be to eliminate the $\triangleright$'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 $\triangleright$'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 $\text{ch}(D)$, the alphabet $A$ contains actions

\[
\bar{H} = \{\bar{s}(d), \bar{r}(d) \mid d \in D\} \quad \text{and} \quad H = \{s(d), r(d) \mid d \in D\}.
\]
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 $\tilde{H} = H \cup \bar{H} \cup \bar{H}$. We assume that actions from $\tilde{H}$ do not synchronize with the other actions in the alphabet, and that $\text{range}(\gamma) \cap \tilde{H} = \emptyset$. On $\tilde{H}$ communication is given by $(d \in D)$:

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

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

**Theorem 3.6.** SACP⁺ + RN + CH⁺ + AB + AA + RR $\vdash$

$$
\alpha(x) \subseteq A^-, \alpha(y) \subseteq A^-, \alpha(z) \subseteq A^-
\therefore (x \not\succ y) \not\succ z = x \not\succ (y \not\succ z).
$$

**Theorem 3.7.** SACP⁺ + RN + CH⁺ + AB + AA + RR $\vdash$

$$
\alpha(x) \subseteq A^-, \alpha(y) \subseteq A^-, \alpha(z) \subseteq A^-
\therefore (x \not\succ y) \not\succ z = x \not\succ (y \not\succ z).
$$

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

$$
\text{CH}^- = \{ \text{IF}(a, 0) \mid a \in H \} \cup \{ \text{IF}(\rho_f, 1) \mid f : A^+_s \rightarrow A^+_s \}
$$

$$
\Delta(\text{SACP}^\text{+} + \text{RN} + \text{CH}^\text{+} + \text{AB} + \text{AA} + \text{RR}).
$$

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

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

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

The module CH⁺ can neither be used to prove any formula containing atoms in $\tilde{H}$ nor 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 $r(d) \not\succ (s(d) + s(e)) \not\succ r(e)$ still works in

¹ The fresh atom principle (FAP) states 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.
any model that is the restriction of a trace-consistent model of SACP' + RN + CH' + AB + AA + RR. Consequently, we have to modify 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

\[ CH = S(CH^{-}) + \langle \alpha(x) \subseteq A^{-} \rangle. \]

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 3.6 and 3.7 one can easily establish that module CH proves the general associativity of the chaining operators:

\[ CH \vdash x \triangleright (y \triangleright z) = (x \triangleright y) \triangleright x \]

and

\[ CH \vdash x \triangleright (y \triangleright z) = (x \triangleright y) \triangleright x. \]

Since in conditional equational logic all formulas are universal, the module CH− + \( \langle \alpha(x) \subseteq A^{-} \rangle \) 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.

**Proposition 3.8.** Let the module CH− be parametrized by an alphabet \( B = A^{-} \) such that there is at least one action \( \uparrow \text{d} \) in B. Then

\[ CH^{-} + \langle \alpha(x) \subseteq A^{-} \rangle \vdash_{\text{foleq}} \uparrow \text{d} = \delta. \]

**Proof.** The constituent components of CH− are parametrized with a set \( A \) of actions containing an action \( r(\text{d}) \). We claim that

\[ \text{SACP}^{+} + \text{RN} + \text{AB} \vdash_{\text{foleq}} (\alpha(r(\text{d})) \subseteq A^{-} \rightarrow \uparrow \text{d} = \delta). \]

The proof is constructed by applying on both sides of the equation \( \alpha(r(\text{d})) \subseteq A^{-} \rightarrow \uparrow \text{d} = \delta \), a renaming operator that takes \( r(\text{d}) \) into \( \uparrow \text{d} \) and all actions from \( A^{-} \) into \( \delta \). The formula \( (\alpha(r(\text{d})) \subseteq A^{-} \rightarrow \uparrow \text{d} = \delta) \) plays the role of \( \lnot (\alpha(r(\text{d})) \subseteq A^{-}) \), which cannot be proven from SACP' + RN + AB. From this one derives

\[ \text{SACP}^{+} + \text{RN} + \text{AB} \vdash_{\text{foleq}} \exists x (\alpha(x) \subseteq A^{-} \rightarrow \uparrow \text{d} = \delta) \]

(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

\[ CH^{-} \vdash_{\text{foleq}} \exists x (\alpha(x) \subseteq A^{-} \rightarrow \uparrow \text{d} = \delta) \]

and, thus,

\[ CH^{-} + \langle \alpha(x) \subseteq A^{-} \rangle \vdash_{\text{foleq}} \uparrow \text{d} = \delta. \]
Theorem 3.9. Infinitary conditional logic for modules is not complete.

Proof. Similar to the proof of Theorem 3.4, using the Proposition 3.8. □

3.3. SACP

Module SACP is an "improved" version of module ACP in which auxiliary operators are hidden in an appropriate way. It is defined by

\[ \text{SACP} = \text{SACP'} + \text{RN} + \text{CH} + \text{PR} + \text{AB} + \text{AA} + \text{RR} + \text{REC} + \text{B} + \text{AIP}^- . \]

Module SACP is parametrized by an alphabet \( A \) which does not include the auxiliary actions \( s(d), r(d), \ldots \). The rules RSP and RSP* can still be used in a setting with module SACP: we have SACP \( \vdash \) RSP and SACP \( \vdash \) RSP*. Also Proposition 2.1 and the results of Sections 2.1.1 and 2.4 carry over to the new setting. For this it is crucial that the properties of Section 2.1.1 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 [31, 44]),
- The implementation of languages with many-to-one synchronous communication, such as POOL (see [2, 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 [18, 29, 39].

4.1. An infinite specification of a queue

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. Further, $d$ ranges over $D$ and $\sigma$ ranges over $D^*$:

\[
\text{QUEUE} = Q_0 = \sum_{d \in D} \downarrow d \cdot Q_d,
\]

\[
Q_{\sigma \cdot d} = \sum_{e \in D} \downarrow e \cdot Q_{e \cdot \sigma \cdot d} + \uparrow d \cdot Q_e.
\]

Note that this infinite specification uses only the signature of BPA_4 (see Section 2.1). We have the following fact.

**Theorem 4.1.** Using read/send communication, the process QUEUE cannot be specified in ACP by finitely many recursion equations.

**Proof.** See [4, 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 [13, 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 BUF denote a buffer with capacity one:

\[
\text{BUF} = \sum_{d \in D} \downarrow d \cdot \text{BUF}_d,
\]

\[
\text{BUF}_d = \uparrow d \cdot \text{BUF}.
\]

In process verifications one often needs propositions like $\text{QUEUE} \triangleright \text{BUF} = \text{QUEUE}$. 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]:

\[
Q = \sum_{d \in D} \downarrow d \cdot (Q \triangleright \text{BUF}_d).
\]

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

**Theorem 4.2.** $\text{SACP}_A \models Q = \text{QUEUE}$. 
**Proof.** Define for \( n \in \mathbb{N} \) process \( Q^n \) as the chaining of \( Q \) with \( n \) empty buffers with capacity one:

- \( Q^0 = Q \).
- \( Q^{n+1} = Q^n \Rightarrow \text{BUF}1. \)  

By induction, we prove that, for all \( n \),

\[
Q^n = \sum_{d \in \mathcal{D}} \downarrow d \cdot (Q^n \Rightarrow \text{BUF}1^d).
\]

The case \( n = 0 \) follows trivially using (1). So, suppose that the statement has been shown for \( n \leq k \). Using the laws of Section 2.4, we derive

\[
Q^{k+1} \overset{4.2}{=} Q^k \Rightarrow \text{BUF}1 =
\]

\[
\overset{\text{ind}}{=} \left( \sum_{d \in \mathcal{D}} \downarrow d \cdot (Q^k \Rightarrow \text{BUF}1^d) \right) \Rightarrow \text{BUF}1
\]

\[
\overset{L3 + CH}{=} \sum_{d \in \mathcal{D}} \downarrow d \cdot (Q^k \Rightarrow (\text{BUF}1^d \Rightarrow \text{BUF}1))
\]

\[
\overset{L1}{=} \sum_{d \in \mathcal{D}} \downarrow d \cdot (Q^k \Rightarrow \tau (\text{BUF}1 \Rightarrow \text{BUF}1^d))
\]

\[
\overset{L3}{=} \sum_{d \in \mathcal{D}} \downarrow d \cdot (Q^k \Rightarrow (\text{BUF}1 \Rightarrow \text{BUF}1^d))
\]

\[
\overset{CH + 4.2}{=} \sum_{d \in \mathcal{D}} \downarrow d \cdot (Q^{k+1} \Rightarrow \text{BUF}1^d).
\]

This completes the proof of the induction step.

Define for \( \sigma \in \mathcal{D}^+ \) processes \( B^\sigma \) by

\[
B^d = \text{BUF}1^d,
\]

\[
B^{\sigma \cdot d} = B^\sigma \Rightarrow \text{BUF}1^d.
\]

By simple inductive arguments one can show that

\[
B^{d \cdot \sigma} = \text{BUF}1^d \Rightarrow B^\sigma,
\]

\[
B^{\sigma \cdot d} = \uparrow d \cdot (\text{BUF}1 \Rightarrow B^\sigma).
\]

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

\[
Q^n \overset{4.3}{=} \sum_{d \in \mathcal{D}} \downarrow d \cdot (Q^n \Rightarrow \text{BUF}1^d) \overset{4.4}{=} \sum_{d \in \mathcal{D}} \downarrow d \cdot (Q^n \Rightarrow B^d).
\]
Consider the following guarded recursive specification with variables $Q^\sigma$ for $\sigma \in D^*$ and $n \in \mathbb{N}$:

\[ Q^0 = \sum_{d \in D} \downarrow d \cdot Q_n^\sigma, \]

\[ Q^\sigma + d = \sum_{e \in D} \downarrow e \cdot Q^\sigma_{e, \sigma + d} + \uparrow d \cdot Q^\sigma_{n+1}. \]

Since QUEUE satisfies the defining equations of $Q^0$, the RSP gives that QUEUE = $Q^0$. Because $Q^0$ also satisfies the equations for $Q^2$, another application of RSP gives $Q = Q^0$. Consequently, QUEUE = $Q^0$. \hfill \Box

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.

**Corollary 4.3.** $\mathsf{SACP}_J^\sigma \vdash Q \triangleright BUFI = Q$.

**Proof.** From equations (4.1)–(4.3) it follows that

\[ Q \triangleright BUFI = \sum_{d \in D} \downarrow d \cdot ((Q \triangleright BUFI) \triangleright BUFI^d). \]

Since the specification for QUEUE is guarded, we know from Theorem 4.2 that $Q$ is guardedly specifiable. Therefore, since $Q \triangleright BUFI$ satisfies the defining equations of $Q$, we can use RSP* to conclude that $Q = Q \triangleright BUFI$. \hfill \Box

**Proposition 4.4.** $\mathsf{SACP}_J^\sigma \vdash Q = Q$. 

Proof.

\[ Q \triangleright Q = \sum_{d \in D} \downarrow d \cdot ((Q \triangleright \text{BUF}1^d) \triangleright Q) \]
\[ = \sum_{d \in D} \downarrow d \cdot (Q \triangleright (\text{BUF}1^d \triangleright Q)) \]
\[ = \sum_{d \in D} \downarrow d \cdot (Q \triangleright \tau \cdot (\text{BUF}1 \triangleright (Q \triangleright \text{BUF}1^d))) \]
\[ = \sum_{d \in D} \downarrow d \cdot (Q \triangleright (\text{BUF}1 \triangleright (Q \triangleright \text{BUF}1^d))) \]
\[ = \sum_{d \in D} \downarrow d \cdot ((Q \triangleright \text{BUF}1) \triangleright (Q \triangleright \text{BUF}1^d)) \]
\[ = \sum_{d \in D} \downarrow d \cdot ((Q \triangleright \text{BUF}1) \triangleright \text{BUF}1^d)) \]
\[ \text{(by Corollary 4.3)} \]
\[ = \sum_{d \in D} \downarrow d \cdot ((Q \triangleright Q) \triangleright \text{BUF}1^d). \]

Now apply RSP*.

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:

\[ \bar{Q} = \sum_{d \in D} \downarrow d \cdot (\bar{Q} \triangleright \uparrow d \cdot \bar{Q}). \]

A standard proof gives that \( \bar{Q} = \text{QUEUE} \). 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]: \( \uparrow \times D^* \). A problem with Pratt's specification, however, is that a neat axiomatization of the orthocurrence operator \( \times \) is not available. Our \( \bar{Q} \)-specification has the disadvantage that it does not allow for simple proofs of identities like \( \bar{Q} \triangleright \bar{Q} = \bar{Q} \).

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 to specify various types of faulty queues. One just has to replace the process \( \text{BUF}1 \) 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:

\[
FBUFl = \sum_{d \in D} !d \cdot FBUFl^d,
\]

\[
FBUFl^d = (\uparrow d + \tau) \cdot FBUFl.
\]

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

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

\[
FQ = \sum_{d \in D} !d \cdot (FQ \triangleright FBUFl^d).
\]

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

Lemma 4.5.

\[
\text{SACP}^\downarrow \vdash \tau \cdot (FBUFl^d \triangleright FBUFl) = \tau \cdot (FBUFl) \triangleright FBUFl^d.
\]
Proof. 
\[
\tau \cdot (\text{BUFD} \triangleright \text{BUFI}) = \tau \cdot (\tau \cdot (\text{BUFD} \triangleright \text{BUFI} + \text{BUFI} \triangleright \text{BUFD})) \\
= \tau \cdot (\tau \cdot (\tau \cdot (\text{BUFD} \triangleright \text{BUFI} + \text{BUFI} \triangleright \text{BUFD}))) \\
= \tau \cdot (\tau \cdot (\tau \cdot (\text{BUFD} \triangleright \text{BUFI} + \text{BUFI} \triangleright \text{BUFD}))) \\
\]

Compare the simple definition of FQ with the following BPA₁-specification of the same process.

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

\[
\text{FQUEUE} = FQ = \sum_{d \in D} \downarrow d \cdot FQ_d, \\
FQ_{\sigma \cdot d} = \sum_{e \in D} \uparrow e \cdot FQ_{\sigma \cdot e \cdot d} + \uparrow \cdot FQ_{\sigma} \cdot \sum_{\rho \in R(\sigma \cdot d)} \tau \cdot FQ_{\rho}.
\]

Theorem 4.6. \(\text{SACP}^*_F \models FQ = \text{FQUEUE}.\)

Proof. Analogous to the proof of Theorem 4.2. Use Lemma 4.5. \(\square\)

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

Proposition 4.7. \(\text{SACP}^*_F \models\)

(i) \(FQ \triangleright \text{BUFI} = FQ\),
(ii) \(\tau \cdot (\text{BUFD} \triangleright FQ) = \tau \cdot (\text{BUFD} \triangleright (FQ \triangleright \text{BUFD}))\),
(iii) \(Q \triangleright FQ = FQ \triangleright FQ = FQ\).

4.4. An identity that does not hold

In this section we will discuss the identity

\(FQ = Q \triangleright \text{BUFI}\).

"Intuitively", the processes FQ and Q \(\triangleright \text{BUFI}\) 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'. In fact, we have proved the following, even stronger, result.

**Theorem 4.8.** If parameter $D$ of operator $\triangleright$ contains more than one element, then $\text{SACP} \triangleq \text{WBS} \not\models \text{FQ} \approx \text{FBUFI}$. 

**Proof.** We show that the identity is not valid in the model of process graphs modulo weak bisimulation congruence of Baeten et al. [7] (see also [10]). Suppose that there exists a bisimulation between processes $\text{FQ}$ and $\text{FBUFI}$. Consider the situation in which process $\text{FQ}$ has read successively two different data, starting from the initial state. Because of the bisimulation, it must be possible for the process $\text{FBUFI}$ to read the same data in such a way that the resulting state is bisimilar to the state process $\text{FQ}$ has reached. Suppose that next process $\text{FQ}$ executes a $\tau$-step and forgets the second datum. We claim that process $\text{FBUFI}$ is not capable to perform a corresponding sequence of zero or more $\tau$-step. This is because there are only two possibilities:

1. $\text{FBUFI}$ forgets the second datum. But this means that also the first datum is forgotten. In the resulting state $\text{FBUFI}$ cannot output any datum (before reading one), whereas process $\text{FQ}$ can do this.

2. $\text{FBUFI}$ does not forget the second datum. In the resulting state $\text{FBUFI}$ can output this datum. Process $\text{FQ}$ cannot do that.

The argument is illustrated in Fig. 4. □

The next theorem states that, if we add law T4, the two faulty queues can be proven equivalent.
Theorem 4.9. SACP\_8^+ + WBS + T4 ⊢ FQ = Q ⊳ FBUF1.

Proof. Define the process QF by

\[ QF = QF = \sum_{d \in D} d \cdot QF_d, \]

\[ QF_{\sigma \ast d} = \sum_{e \in D} e \cdot QF_{\sigma \ast d} + (\mid d + \tau \mid) \cdot QF_e. \]

Analogous to the proof of Theorem 4.2, using in addition the identity \( Q \triangleright BUFl = Q \), one can prove from SACP\_8^+ that \( Q \triangleright FBUF1 = Q \).

The main trick in the proof is that we introduce yet another “view of queues”: process QF\( e \) 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 \( \sigma \) 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

\[ FS_\sigma = FBUF1, \]

\[ FS_{\sigma \ast d} = (\mid d + \tau \mid) \cdot FS_\sigma. \]

It is routine to prove from SACP\_8^+ that \( Q \triangleright FS_\sigma = QF_\sigma \). Until here one does not need the additional axioms. The crucial part in the proof is the following claim.

Claim. \( \tau \cdot FS_{\sigma \ast \sigma} \subseteq FS_{\sigma \ast d \ast \sigma} \).

Proof (by induction on the length of \( \sigma \)). If \( \sigma = \lambda \) then the claim holds trivially. Assume that the claim is proved for \( |\sigma| \leq n \). Suppose \( \sigma = \bar{\sigma} \ast e \) with \( |\bar{\sigma}| = n \). We derive

\[ \tau \cdot FS_{\bar{\sigma} \ast \bar{\sigma} \ast e} = \tau (\uparrow e \cdot FS_{\bar{\sigma} \ast \bar{\sigma} \ast e} + \tau \cdot FS_{\bar{\sigma} \ast \bar{\sigma}}) = \]

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

\[ \supseteq \uparrow e \cdot FS_{\bar{\sigma} \ast \bar{\sigma}} + \tau \cdot FS_{\bar{\sigma} \ast \bar{\sigma}} = \]

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

\[ \subseteq \uparrow e \cdot FS_{\bar{\sigma} \ast d \ast \bar{\sigma}} + \tau \cdot FS_{\bar{\sigma} \ast d \ast \bar{\sigma}} = FS_{\bar{\sigma} \ast d \ast \bar{\sigma} \ast e}. \]

This finishes the proof of the claim. As a corollary, we can prove that \( \tau \cdot QF_{\rho \ast \sigma} \subseteq QF_{\rho \ast d \ast \sigma} \).

\[ \tau \cdot QF_{\rho \ast \sigma} \subseteq (Q \triangleright FS_{\rho \ast \sigma}) \subseteq \]
(use the observation of Section 2.1.1 that \( \tau(x \parallel z) \subseteq (\tau x + y) \parallel z \)).

\[ \leq Q \equiv FS_{\rho \times d \times \sigma} = QF_{\rho \times d \times \sigma}. \]

We have shown that process \( QF_{\sigma} \) is indistinguishable from a process that can lose each datum at every moment. Using the notation of Section 4.3, we can write the following equation for processes \( QF_{\sigma \times d} \):

\[ QF_{\sigma \times d} = \sum_{e \in D} \downarrow e \cdot QF_{\epsilon \times \sigma \times d} + \uparrow d \cdot QF_{\sigma} + \sum_{\rho \in R(\sigma \times d)} \tau \cdot QF_{\rho}. \]

Application of RSP gives that the process \( FQUEUE \) of Section 4.3 equals process \( QF \). But according to Theorem 4.6 also \( FQUEUE = FQ \). Thus \( FQ = FQUEUE = QF = Q \equiv FBUFI. \]

5. 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 and Veltink [33].

In order to prove the associativity of the chaining operators, we needed auxiliary actions \( \tilde{s}(d), \tilde{r}(d) \), 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 do not like to rewrite their specification all the time when they work
on the verification. Therefore, we would like to have a proof principle stating 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 often be 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 lose data at every place is different from a FIFO-queue
which can only lose data at the end. Adding law T4, which holds in ready trace
semantics (and hence in failure semantics), made it possible to prove the two queues
equal.

Appendix A: 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 $F_{\sigma}^{\text{fo}}$ of 
*first-order formulas with equality* over $\sigma$ is defined by

$$F_{\sigma}^{\text{at}} = F_{\sigma}^{\text{fo}}$$

if $\phi \in F_{\sigma}^{\text{fo}}$ then $\neg \phi \in F_{\sigma}^{\text{fo}}$,

if $\phi$ and $\psi \in F_{\sigma}^{\text{fo}}$ then $(\phi \rightarrow \psi) \in F_{\sigma}^{\text{fo}}$,

if $\phi$ and $\psi \in F_{\sigma}^{\text{fo}}$ then $(\phi \wedge \psi) \in F_{\sigma}^{\text{fo}}$,

if $\phi$ and $\psi \in F_{\sigma}^{\text{fo}}$ then $(\phi \vee \psi) \in F_{\sigma}^{\text{fo}}$,

if $\phi$ and $\psi \in F_{\sigma}^{\text{fo}}$ then $(\phi \leftrightarrow \psi) \in F_{\sigma}^{\text{fo}}$,

if $x$ is a variable and $\phi \in F_{\sigma}^{\text{fo}}$ then $\forall x(\phi) \in F_{\sigma}^{\text{fo}}$,

if $\phi \in F_{\sigma}^{\text{fo}}$ then $\exists x(\phi) \in F_{\sigma}^{\text{fo}}$.

The $\xi$-truth of a formula $\phi \in F_{\sigma}^{\text{fo}}$ in a $\sigma$-algebra $A$ is defined by

$A, \xi \models_{\sigma}^{\text{fo}} \phi$ if $\phi \in F_{\sigma}^{\text{at}}$ and $A, \xi \models_{\sigma}^{\text{en}} \phi$,

$A, \xi \models_{\sigma}^{\text{fo}} \neg \phi$ if $A, \xi \models_{\sigma}^{\text{fo}} \phi$.
Modular specification of process algebras

$\mathcal{A}, \xi \models_{\text{foleq}} \phi \rightarrow \psi$ if $\mathcal{A}, \xi \not\models_{\text{foleq}} \phi$ or $\mathcal{A}, \xi \models_{\text{foleq}} \psi$.

$\mathcal{A}, \xi \models_{\text{foleq}} \phi \land \psi$ if $\mathcal{A}, \xi \models_{\text{foleq}} \phi$ and $\mathcal{A}, \xi \models_{\text{foleq}} \psi$.

$\mathcal{A}, \xi \models_{\text{foleq}} \phi \lor \psi$ if $\mathcal{A}, \xi \models_{\text{foleq}} \phi$ or $\mathcal{A}, \xi \models_{\text{foleq}} \psi$.

$\mathcal{A}, \xi \models_{\text{foleq}} \phi \rightarrow \psi$ if $\mathcal{A}, \xi \models_{\text{foleq}} \phi$ if and only if $\mathcal{A}, \xi \models_{\text{foleq}} \psi$.

$\mathcal{A}, \xi \models_{\text{foleq}} \forall x (\phi)$ if $\mathcal{A}, \xi' \models_{\text{foleq}} \phi$ for all valuations $\xi'$ with $\xi'(y) = \xi(y)$ for all variables $y \neq x$.

$\mathcal{A}, \xi \models_{\text{foleq}} \exists x (\phi)$ if $\mathcal{A}, \xi' \models_{\text{foleq}} \phi$ for some valuation $\xi'$ with $\xi'(y) = \xi(y)$ for all variables $y \neq x$.

$\phi$ is true in $\mathcal{A},$ notation $\mathcal{A} \models_{\text{foleq}} \phi,$ if $\mathcal{A}, \xi \models_{\text{foleq}} \phi$ for all valuations $\xi$.

An inference system $I'_{\eta}$ for first-order logic with equality is displayed in Table 14, where $\phi, \psi$ and $\rho$ are first-order formulas in $F'_{\eta}$, $x$ is an atomic formula in $F'_{\eta}$, $t, u$ and $v$ are terms over $\sigma$ and $x$ is a variable. An occurrence of a variable $x$ in a formula $\phi$ is bound if it occurs in a subformula $\forall x(\psi)$ or $\exists x(\psi)$ of $\phi$. Otherwise, it is free. A variable is free in a formula $\phi$ if all its occurrences in $\phi$ 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 $\phi$ if all free occurrences of variables in $t$ remain free in $\phi[t/x]$.

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 $\land, \lor$ and $\rightarrow$, 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 $\land, \lor$ and $\rightarrow$, and the universal ones are the formulas without quantifiers. Model theory (see, for instance, [36]) teaches us that a formula $\phi$ is preserved under homomorphisms (subalgebras) iff there is a positive (universal) formula $\psi$ with $\mathcal{A} \models_{\text{foleq}} \psi \iff \phi$.

Table 14

<table>
<thead>
<tr>
<th>$\phi, \phi \rightarrow \psi$ Modus ponens</th>
<th>$\phi$ Generalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall x(\phi)$</td>
<td>Deduction axioms</td>
</tr>
<tr>
<td>$\phi \rightarrow (\psi \rightarrow \phi)$</td>
<td>Axiom of the excluded middle</td>
</tr>
<tr>
<td>$[\forall x(\phi) \rightarrow {\phi \rightarrow (\psi \rightarrow \phi)}]$</td>
<td>Axiom of specialization</td>
</tr>
<tr>
<td>$\forall x(\phi) \rightarrow \forall x(\phi)$, if $t$ is free for $x$ in $\phi$</td>
<td>Axiom of contradiction</td>
</tr>
<tr>
<td>$(\phi \land \psi) \rightarrow \phi \land (\phi \lor \psi) \lor (\phi \lor \psi)$</td>
<td></td>
</tr>
<tr>
<td>$\phi \land (\phi \lor \psi) \land (\phi \lor \psi) \lor (\phi \land \psi) \rightarrow (\psi \rightarrow \psi)$</td>
<td></td>
</tr>
<tr>
<td>$\forall x(\phi) \rightarrow \forall x(\phi)$</td>
<td></td>
</tr>
<tr>
<td>$t = t$ &amp; $u = t$ $\rightarrow$ $v = t$</td>
<td></td>
</tr>
<tr>
<td>$(u = v) \rightarrow (v = u)$</td>
<td></td>
</tr>
<tr>
<td>$(u = v) \rightarrow (x[u/x] = x[v/x])$</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B. The associativity of the chaining operators

In this appendix we present the proofs of Theorems 3.6 and 3.7 about the associativity of the chaining operators. Define for \( v, w \in \{ \top, \bot, s, r, \tilde{s}, \tilde{r}, \epsilon, \tilde{\epsilon} \} \) 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 B.1.

\[
\begin{align*}
\text{SACP}^+ + \text{RN} + \text{CH}^+ + \text{AB} + \text{AA} + \text{RR} & \vdash \\
\partial_{\tilde{H}}(x) = x, \quad \partial_{\tilde{H}}(y) = y, \quad \partial_{\tilde{H}}(z) = z \\
\partial_{\tilde{H}}(\rho_{\bar{t}_x}(x) \parallel \rho_{\bar{t}_y}(y)) &= x \parallel y \Rightarrow \partial_{\tilde{H}}(\rho_{\bar{t}_x}(x) \parallel \rho_{\bar{t}_y}(y)).
\end{align*}
\]

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

\[
\begin{align*}
\partial_{\tilde{H}}(\rho_{\bar{t}_x}(x) \parallel \rho_{\bar{t}_y}(y)) &= (\text{Note 1 below, RR1}) \\
&= \partial_{\tilde{H}} \circ \rho_{\bar{t}_x} \circ \rho_{\bar{t}_y}(\rho_{\bar{t}_x}(x) \parallel \rho_{\bar{t}_y}(y)) = (\text{RN5, } y = \partial_{\tilde{H}}(y)) \\
&= \partial_{\tilde{H}} \circ \rho_{\bar{t}_x} \circ \rho_{\bar{t}_y}(\rho_{\bar{t}_x}(x) \parallel \rho_{\bar{t}_y}(y)) = (\text{Note 2, RR2}) \\
&= \partial_{\tilde{H}} \circ \rho_{\bar{t}_x} \circ \rho_{\bar{t}_y}(\rho_{\bar{t}_x}(x) \parallel \rho_{\bar{t}_y}(y)) = (\text{SC4, RN5, } x = \partial_{\tilde{H}}(x)) \\
&= \partial_{\tilde{H}} \circ \rho_{\bar{t}_x} \circ \rho_{\bar{t}_y}(\rho_{\bar{t}_x}(y) \parallel \rho_{\bar{t}_y}(x)) = (\text{as in Note 2, RR2}) \\
&= \partial_{\tilde{H}} \circ \rho_{\bar{t}_x} \circ \rho_{\bar{t}_y}(\rho_{\bar{t}_x}(y) \parallel \rho_{\bar{t}_y}(x)) = (\text{RN5}) \\
&= \partial_{\tilde{H}} \circ \partial_{\tilde{H}}(\rho_{\bar{t}_x}(y) \parallel \rho_{\bar{t}_y}(x)) = (\text{Note 3, RR1, SC4}) \\
&= \partial_{\tilde{H}}(\rho_{\bar{t}_x}(x) \parallel \rho_{\bar{t}_y}(y)) = (\text{CH1}) \\
&= x \parallel y.
\end{align*}
\]

Note 1. Let \( B = A - H \). We claim that \( \alpha(\rho_{\bar{t}_x}(x) \parallel \rho_{\bar{t}_y}(y)) \subseteq B \) (recall that \( B = \text{set} \sum_{b \subseteq B} b \)). This is proved as follows.

\[
\alpha(\rho_{\bar{t}_x}(x) \parallel \rho_{\bar{t}_y}(y)) \overset{AA2}{=} \alpha \circ \rho_{\bar{t}_x}(x) + \alpha \circ \rho_{\bar{t}_y}(y) + \alpha \circ \rho_{\bar{t}_x}(x) \parallel \alpha \circ \rho_{\bar{t}_y}(y) \subseteq
\]

(used that \( x \subseteq y \Rightarrow x \parallel z \subseteq y \parallel z \). Use further that \( x = \partial_{\tilde{H}}(x) \overset{RN5}{=} \partial_{\tilde{H}} \circ \partial_{\tilde{H}}(x) = \partial_{\tilde{H}}(x) \))

\[
\overset{AA1}{=} \alpha \circ \rho_{\bar{t}_x} \circ \partial_{\tilde{H}}(x) + \alpha \circ \rho_{\bar{t}_y} \circ \partial_{\tilde{H}}(y) + A | A \subseteq
\]


(use that range(y) ∩ H = ∅)
\[ R^5 \subseteq \alpha \circ \partial_H \circ \rho_{I_{f}}(x) + \alpha \circ \partial_H \circ \rho_{I_{f}}(y) + B \subseteq \]
\[ AA^3 + R^5 \subseteq \partial_H \circ \alpha \circ \rho_{I_{f}}(x) + \partial_H \circ \alpha \circ \rho_{I_{f}}(y) + B \subseteq \]
(Use that x ∋ y implies \( \rho_{I_{f}}(x) \subseteq \rho_{I_{f}}(y) \))
\[ AA^1 \subseteq \partial_H(A) + \partial_H(A) + B = B. \]
This finishes the proof of the claim.

Note 2. Application of axiom AA1 gives \( x \circ \rho_{I_{f}}(x) \subseteq A \) and \( x \circ \rho_{I_{f}}(y) \subseteq A \). In order to apply axiom RR2, we first have to check that, for all \( c \in A, \text{rr}(c) = \text{rr}(c) \). This is obviously the case. Because range(y) ∩ H = ∅, we have, for all \( b, c \in A, \text{rr} \circ \gamma(b,c) = \gamma(b,c) \). Now the last thing to be checked is that, for \( b, c \in A, \gamma(b,c) = \gamma(b, \text{rr}(c)) \).

Note 3. Let \( C = A - H \). We claim that \( \alpha(\rho_{I_{f}}(y) \parallel \rho_{I_{f}}(x)) \subseteq C \). 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 8.2.
\[ \text{SACP}' + \text{RN} + \text{CH}' + \text{AB} + \text{AA} + \text{RR} \models \]
\[ \partial_H(x) = x, \partial_H(y) = y, \partial_H(z) = z \]
\[ x \parallel (y \parallel z) \models (x \parallel y) \parallel z. \]

Proof. This is essentially Theorem 1.12.2 of [45]. We give a sketch of the proof.
\[ x \parallel (y \parallel z) \text{Lemma B.1} \models \partial_H(\rho_{I_{f}}(x) \parallel \rho_{I_{f}}(y) \parallel \rho_{I_{f}}(z)) \]
\[ R^5 \models \partial_H(\rho_{I_{f}}(x) \parallel \rho_{I_{f}}(y) \parallel \rho_{I_{f}}(z)) \]
\[ R^1 \models \partial_H \circ \partial_H(\rho_{I_{f}}(x) \parallel \rho_{I_{f}}(y) \parallel \rho_{I_{f}}(z)) \]
\[ R^2 \models \partial_H \circ \partial_H(\rho_{I_{f}}(x) \parallel \rho_{I_{f}}(y) \parallel \rho_{I_{f}}(z)) \]
\[ R^2 \models \partial_H \circ \partial_H(\rho_{I_{f}}(x) \parallel \rho_{I_{f}}(y) \parallel \rho_{I_{f}}(z)) \]
\[ R^1 \models \partial_H \circ \partial_H(\rho_{I_{f}}(x) \parallel \rho_{I_{f}}(y) \parallel \rho_{I_{f}}(z)) \]
\[ R^5 \models \partial_H \circ \partial_H(\rho_{I_{f}}(x) \parallel \rho_{I_{f}}(y) \parallel \rho_{I_{f}}(z)) \]
Theorem B.3.

\[ \text{SACP}^* + \text{RN} + \text{CH}^* + \text{AB} + \text{AA} + \text{RR} \vdash \]
\[ \quad \rho_H(x) = x, \quad \rho_H(y) = y, \quad \rho_H(z) = z. \]
\[ x \triangleright (y \triangleright z) = (x \triangleright y) \triangleright z. \]

Proof. Let \( I = \{ c(d) \mid d \in D \} \). We derive:
\[ x \triangleright (y \triangleright z) \overset{\text{CH2}}{=} \tau_I(x \triangleright (\tau_I(y \triangleright z))) \]
\[ \overset{\text{CH1}}{=} \tau_I \circ \rho_H(\rho_I(x) \parallel \rho_I(y \triangleright z)) \]
\[ \overset{\text{CH1}}{=} \tau_I \circ \rho_H(\rho_I(x) \parallel \rho_I(y \triangleright z)) \]
\[ \overset{\text{RNS5}}{=} \tau_I \circ \rho_H(\rho_I(x) \parallel \rho_I(y \triangleright z)) \]
\[ \overset{\text{RNS5}}{=} \tau_I \circ \rho_H(\rho_I(x) \parallel \rho_I(y \triangleright z)) \]
\[ \overset{\text{CH1}}{=} \tau_I(x \triangleright (y \triangleright z)) \]
\[ \overset{\text{Theorem B.2}}{=} \tau_I((x \triangleright y) \triangleright z) = \cdots = (x \triangleright y) \triangleright z. \quad \square \]

Theorems 3.6 and 3.7 follow from the above Theorems B.2 and B.3, respectively in combination with axiom RR1.

Acknowledgment

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.
References


