Algebraic Techniques for Concurrency and their Application

Frits W. Vaandrager

Algebraic Techniques for Concurrency and their Application

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof.dr. S.K. Thoden van Velzen
in het openbaar te verdedigen in de Aula der Universiteit
(Oude Lutherse kerk, ingang Singel 411, hoek Spui),
op vrijdag 2 februari 1990 te 15.00 uur

door
Frits Willem Vaandrager
geboren te Voorschoten

Centrum voor Wiskunde en Informatica
1989

Promotor: prof.dr. J.A. Bergstra

Faculteit: Wiskunde en Informatica

The work on this thesis has been carried out at the Centre for Mathematics and Computer Science in the context of ESPRIT project no. 432, An Integrated Formal Approach to Industrial Software Development (METEOR), and RACE project no. 1046, Specification and Programming Environment for Communication Software (SPECS).

Acknowledgements

My promotor Jan Bergstra certainly belongs to the class of professors that run around from one meeting to another, managing a large group and trying to make it even bigger by raising funds all the time. In this way he created for me an extremely stimulating environment to work in with lots of bright colleagues and opportunities to travel abroad. I am very grateful for this. In addition I want to thank him for numerous discussions which, maybe due to lack of time, were always directed to the key issues immediately.

Furthermore, I would like to express my appreciation to the Centre for Mathematics and Computer Science, where I was employed in Jaco de Bakker's Department of Software Technology from 1985 up to now.

I am much indebted to my colleague Rob van Glabbeek. In most of the work I did during the last five years, traces can be found of his ping-pong speed thinking, creativity and original points of view on no matter what subject.

I also enjoyed particularly the collaboration with Jos Baeten and Jan Friso Groote.

Furthermore, I thank the other participants of the Process Algebra Meetings: Wiet Bouma, Jeroen Bruijning, Nicolien Drost, Henk Goeman, Jan Willem Klop, Karst Koymans, Sjouke Mauw, Kees Middelburg, Hans Mulder, Alban Ponse, Gerard Renardel, Piet Rodenburg, Gert Veltink, Jos Vrancken, Fer-Jan de Vries, Peter Weijland, Freek Wiedijk and Han Zuidweg. Much of the work in this thesis was firstly presented at the PAM and always when I gave a presentation there, the noisy audience managed to make some very good remarks.

Finally, I thank Robin Milner, Ernst-Rüdiger Olderog, Jos Baeten, Paul Klint and Paul Vitányi for their willingness to referee this thesis.

Contents

Introduction	1
Structured operational semantics and bisimulation as a congruence	19
Modular specifications in process algebra	75
Two simple protocols	115
Some observations on redundancy in a context	141
Process algebra semantics of POOL	165
Determinism → (event structure isomorphism = step sequence equivalence)	221
Samenvatting	243

Introduction

This thesis adresses the formal semantics of programming and specification languages for concurrent discrete event systems. Moreover it is concerned with various applications of such semantics.

A formal semantics is a mapping that relates to each program or specification in a language a computational structure (or a class of structures) in some semantic, mathematical domain. The intention is that a semantical object related to a program or specification models the behaviour of a computer system that executes the program, resp. the behaviour of a system that is described in the specification.

Both programming and specification languages are designed with some formal or intuitive semantics in mind. The distinguishing feature of a programming language however is that either there exists a physical machine which can realise a behaviour structure associated to a language element (with a high probability), or, at least, we know how to build such a machine in principle. For a fixed semantics, a programming language must be *executable*. For a specification language on the other hand, this is not required. It is hard to establish the exact boundary between programming and specification languages. Clearly a lot of languages can be and have been implemented. There are also languages for which we do not know how to implement them. Finally, given the existence of undecidable problems and assuming the Church-Turing thesis, some languages can never be implemented. In general however, certain parts of a specification language will always be executable.

In this thesis I will model systems in terms of the events that they generate in time. I will restrict attention to systems that are discrete in the sense that at any moment the set of events that have occurred is finite. Often the systems that will be considered consist of a number of components, each of them generating events independently or concurrently. Examples of concurrent discrete

event systems are parallel computers, operating systems, telephone switching systems, distributed database systems and delay insensitive circuits.

The first section of this introduction contains a discussion of the aim and scope of semantic theories. I will list some of the questions which in my view should motivate research on semantics and point out a number of areas where semantic theories have contributed. Then I will sketch briefly, in Section 2, the main ingredients of the semantic theory for concurrency as it has been developed over the last ten years. In Section 3, I will present an overview of the contents of this thesis. Referring to the first section, I will indicate what are the questions that will be addressed in the thesis. Section 2 will allow me to position my work within the theory of concurrency semantics.

1. Aim and scope of semantic theories

1.1. Language design. Formal semantics can be useful in the design of a language. In the case of functional and logic programming one could even say that the semantics, as mathematical theories, existed before the programming languages, and that these languages were created in an attempt 'to implement their semantics'.

A typical situation which occurs during language design is that implementors as well as (future) users want to introduce in a language lots of features which (1) are easy to implement, (2) often allow for short and fast programs, (3) do not fit at all in the semantic model of the language. Often in such a case, a semanticist will oppose incorporating this type of features in the language because they will lead to unstructured programs whose correctness is very hard to prove. As argued by America [3], it is a good rule of thumb to say that there is a problem in the language design whenever the description of a language feature that is considered to be inessential requires a special adaptation of the overall semantic model used to describe the language. A typical example is the goto-statement in languages like Pascal. In many cases however the gain in efficiency and practical usefulness obtained by extending a language with features that are outside the semantic model is so immense that they are just added, whether the semanticists appreciate it or not. The success of languages like LISP and Prolog for instance is due to a large extent to the imperative features which they incorporate. Therefore, only certain parts of real-life languages will have a neat underlying semantic theory. The job of the semanticist here is to help the language designer to make these parts as large as possible (without restricting the general expressiveness of the language too much), and to explain to programmers why they should try not to use certain constructs.

Informal language definitions as one can find in manuals are sometimes imprecise, ambiguous or incomplete. This can be ruled out by a formal language definition. One may hope that when persons who write language manuals base their work on a formal language definition, this will lead to a clearer and more systematic exposition. It is wrong to bother a user with all the details of a language implementation. If the language is provided with an

operational semantics, then (ideally) a simple and intuitive presentation of the abstract machine model underlying this semantics, can be used to explain the language to a user.

1.2. Correctness of programming language implementations. A formal semantics will often present us with a rather abstract view of the behaviour of a program. It is exactly this abstractness which makes it possible to reason about programs and their correctness. Eventually, of course, the aim of writing programs is to run them on a computer, i.e. a concrete, physical machine. All reasoning at the level of the abstract, mathematical semantics would be completely useless if there would not be a strong relationship between the mathematical semantics of a program and what happens in physical reality when the program is executed on a computer. If one has proved a program correct with respect to the mathematical semantics, then one wants to be quite sure that the output of the computer will be correct when running the program. In the case of current high level programming languages a semantics often provides a very abstract view of what goes on during execution of a program, and there is a huge distance between this view and what actually goes on in the machine. I think that a good theory of semantics can and should play a crucial role in bridging the gap between the two views. Often it will be necessary to provide, instead of a single mathematical semantics, a whole sequence of semantics for a language, ranging from a 'fully' abstract semantics used for reasoning about programs, to a semantics that relates to a program a mathematical object, a description of a machine that in its behaviour closely resembles the physical machine that will have to execute the program. It is part of a theory of semantics to establish the behavioural relationships between consecutive semantics in the sequence. I would like to stress that this is a mathematical activity. Establishing the relation between the machine model underlying the last semantics of the sequence and the physical machine is a task for physicists and the people who build the machines. Ideally, there is a strong mutual influence between semanticists and machine builders. The semanticists should tell the machine builders which abstract machines have nice computational properties and are worth implementing. On the other hand the machine builders have to tell the semanticists what type of machine models can be realised physically, and whether or not the models of the semanticists adequately describe the behaviour of computer equipment. In a time when on the one hand programming languages are based on increasingly more abstract concepts, and on the other hand the architectures of the machine on which these programs have to run become increasingly more complicated, a large effort is needed in the field of semantics to establish the behavioural relationships between the abstract models and the physical machines. That these relationships are by no means trivial, and sometimes even absent, will be illustrated in the section of this thesis which deals with the semantics of the language POOL.

Ideally, an abstract semantics provides a standard which can be used to say whether or not a physical machine correctly implements a language. At

present however, a rigorous proof that the behaviour of a physical machine is correctly modelled by some abstract semantics is completely out of scope in most cases. Only in particular instances (I am thinking of the implementation of the language occam on the transputer) it seems a feasible exercise right now. But even though a complete proof of correctness of an implementation is not feasible in most cases, it *is* possible to prove correctness of certain crucial parts. Research in the area of 'comparative concurrency semantics' has provided us with a lot of insight in the various possible behavioural relations between abstract machines.

It would be nice if, starting from a formal semantic model of some language, one could generate efficient implementations automatically. Unfortunately this does not seem to be feasible at the moment. However, in some cases it is possible to generate prototype implementations based on a formal semantics. Such prototypes can be useful in the language design phase.

Assuming that a machine correctly implements a language relative to some abstract semantics, this semantics in turn can be used to increase efficiency. If a user has written a program $P1$ and wants to execute this, then one may execute instead any program $P2$ which is semantically equivalent with $P1$. In particular one may choose $P2$ in such a way that it is more efficient than $P1$ (faster, less use of storage capacity, etc.).

1.3. Notions of implementation and proof systems. In the previous section I discussed the notion of implementing a programming language on a physical machine and the idea of showing that such an implementation is correct with respect to a mathematical semantics. I argued that often it is wise to introduce a number of intermediate mathematical semantics. In a natural way this leads to the introduction of an implementation relation between the elements of the various semantic domains. Often we have that two semantic mappings, an abstract and a concrete one, both map programs into the same semantic domain. In such cases we have to define an implementation relation on the semantic domain itself. Typically, such an implementation relation will be a pre-order, i.e. a transitive and reflexive relation. This type of implementation relation turns out to be extremely important, not only for proving correctness of machine implementations of programming languages. Given a notion of implementation for a specification language, one may try to establish that a specification which is not in the class of executable specifications, can be implemented (in the mathematical sense) by some member of the language which is in this class.

Specification languages which are not fully executable can still be very important, basically because of the connection with the stepwise refinement method. This method advocates a system construction route that starts with some high level, declarative, nonexecutable (or merely inefficient) specification, goes via a number of intermediate development steps which are provably correct with respect to some implementation relation, and ends with an efficiently executable program. The stepwise refinement method naturally leads to mixing programming parts with (nonexecutable) specification parts.

An important part of semantical theories should be (and is) devoted towards defining sensible notions of implementation, and the development of proof systems and associated decision procedures for establishing these implementation relations. The idea that all programs should be developed rigorously, using a stepwise refinement method, or should be verified formally is absurd, mostly it is just not worth the effort, but some programs are used in such critical applications (space shuttles, banking systems, etc.) that a large effort for proving correctness is justified. Even though much can still be improved, semantic theories have contributed substantially in this area. Typical issues which play a role here are the soundness and completeness of proof systems: all provable statements should be true and moreover any true statement that can be formulated in the language of the proof system should be provable. In designing implementation notions and proof systems, the semanticist is in interaction with the programmers and system designers. The semanticists should provide these people with simple, sound and complete proof systems which are still expressive enough for capturing important intuitions and proving relevant properties. Now and then a semanticist should reveal a serious bug in a system that has not been verified rigorously, thus stressing the importance of a more systematic approach to design and verification of programs.

Summarising, I sketched in this section a picture of a semanticist as someone who plays a role intermediate between the designers of a language, the implementors and the users (system designers and programmers). The semanticists produces mathematical theory which helps these people in doing their job. In the next section I want to say more about the internal structure of semantic theories.

2. Ingredients of semantic theories for concurrency

Below I list some important ingredients of semantic theories for programming and specification languages for concurrent discrete event systems. Here I profited from a similar listing which occurs in [23].

2.1. System models, semantic domains of computation structures. Many different semantic domains for modelling concurrent systems have been proposed. Just to give an idea, a few of them will be listed, together with some basic references:

- (labelled) transition systems [21].
- Petri nets [30]. There are many variants: C/E systems, P/T nets, safe nets, high-level Petri nets, timed Petri nets, etc.
- Event structures [35]. Again there are many variants: prime e.s., stable e.s., e.s. with binary conflict, etc.
- Mazurkiewicz traces [25].
- I/O automata [22].
- De Bakker-Zucker processes [7].
- Aczel's process domain of non-well-founded sets [1].
- The failure set model of [11].

Sometimes these models are equivalent, they just give different representations of what is essentially the same system behaviour. Often however, one model captures more features of system behaviour than another. In Petri nets for instance, one can describe that two events are causally independent (or concurrent). Independence of events is not a primitive notion in labelled transition systems. Similarly, labelled transition systems preserve the branching structure of a process, whereas this information is not fully preserved if one describes a system by a failure set.

2.2. Behavioural equivalences and notions of implementation. The semantics of concurrent and reactive systems is inherently more complex than for non-reactive systems. For non-reactive systems, it is clear what the observable behaviour of a system is: an input/output pair, leading to a semantic description of a system as a (partial) function, or in the nondeterministic case, a relation. For concurrent/reactive systems, there is no single, canonical notion of observable behaviour, but rather a multiplicity of such notions, leading to a multiplicity of behavioural equivalences: given a set $\mathbf{A}$ of observable properties, one can define an equivalence $\sim$ on the semantic domain by:

$$p \sim q \quad \text{iff} \quad \text{for any } A \in \mathbf{A} : p \text{ satisfies } A \Leftrightarrow q \text{ satisfies } A.$$

Often the relationship between two semantic domains can be characterised in terms of some (behavioural) equivalence: the elements of one semantic domain represent equivalence classes of the elements of another semantic domain.

Behavioural equivalences on a semantic domain form an important category of implementation relations. In fact this is the only type of implementation relations that will be considered in this thesis. Often it is argued that an implementation relation should not be symmetric: besides providing the service required by the specification, an implementation may do much more. The main reason why I have been able to prove correctness of implementations using a symmetric notion of implementation is that I considered languages with a built-in abstraction mechanism: this mechanism allows to disregard those behavioural aspects of an implementation which do not occur in the service specification. At present I do not know whether it is feasible to use symmetric implementation relations for the verification of larger systems.

2.3. Programming and specification languages with interpretations in semantic domains. We are faced with an almost infinite amount of programming and specification languages for concurrent systems. One can try to classify these languages by looking at the programming or specification paradigm they adhere to (object oriented, functional and logic programming, data flow computation, etc.). For each particular paradigm there will be a certain amount of semantic theory dealing with the peculiarities of that paradigm. For instance, in logic programming one will study the issue of resolution, in object-oriented programming one tries to understand what an 'object' is, etc.

Besides semantic theory that is specific to a single paradigm, there is also

theory dealing with concurrent languages in general. One can try to say something general about how to map a language to a semantic domain, how to give a compositional semantics, etc. Milner had the idea that for a proper understanding of the basic issues concerning the behaviour of concurrent systems it could be helpful to look for a simple language, with 'as few operators or combinators as possible, each of which embodies some distinct and intuitive idea, and which together give completely general expressive power' [27]. Besides Milner's calculi CCS and SCCS [26, 27], several other calculi have been developed with this idea in mind, such as TCSP [19], MEIJE [4] and ACP [8, 9]. These calculi are all very similar and this supports the idea that indeed some fundamental notions have been discovered. Work by DE SIMONE [31, 32] moreover supports the claim that these languages have a 'completely general expressive power'. This expressiveness makes that the languages are not fully executable and therefore are to be viewed as specification languages rather than as programming languages.

Starting with an example in [26], several high-level languages have been translated to CCS-like calculi. Inevitably, some of the structural properties of high-level languages get lost in such a translation. Often however, the rich semantic theory which is available for the basic calculi makes such translations really worth the effort.

2.4. Proof systems for showing implementation relations and semantic equivalence. It turns out that, at least for the basic calculi, most semantic equivalences and implementation relations can be characterised by means of simple (often even equational) axiomatisations. Of course one can always try to establish behavioural equivalence of two expressions at the level of the semantic domain. For a number of reasons however, I think that often it is advantageous to carry out verifications on the syntactic rather then the semantic level. I will come back to this issue in Section 3.2.

2.5. Property languages with satisfaction relations. When reasoning about semantic objects, there is a need for languages that can be used to express that a semantic object has a certain property. Such languages will be called *property languages*. In general, a property language is just a set of logic formulas. Further we have that the computational structures which form the semantic domain of programs can serve as models (in the logical sense) for the formulas. A semantical object (and hence a program) may or may not satisfy some formula. Some well known property languages are: temporal logic, Hennessy-Milner logic and trace logic. Sometimes, a single language can serve as a property language as well as a specification language. Consider, as an example, trace logic. A labelled transition system or a Petri net may or may not satisfy a certain trace formula. But on the other hand, working in a semantic model of trace sets, a trace formula can be interpreted as a trace set (namely the set of traces for which the formula holds).

2.6. (Compositional) proof systems for property languages. Given a property language and an associated satisfaction relation, it is useful to have a proof system for proving that a semantic object denoted by some program or specification satisfies a certain formula. These proof systems should preferably be *compositional*: it should be possible to prove a property of a composite system from properties of its components.

2.7. Comparative concurrency semantics and classification of properties. The incredible amount of available semantic models and behavioural equivalences asks for a systematic approach. Within the field of semantics, the discipline of comparative concurrency semantics aims at the construction of a lattice of process semantics, ordered by a relation 'makes at least as many identifications as'. Moreover the various features which can be described in a certain process semantics are to be identified. This will facilitate the task of finding an appropriate semantics for a given application.

Any semantics provides an answer to the following basic question: 'When do two expressions have the same meaning?' Often however, the ways in which two semantic mappings are defined are so completely different, that at first sight it is not clear at all that they both give the same answer to the above question. A comparative concurrency semantics should therefore try to characterise what is essentially a single semantics, in as many ways as possible. Below some of the possibilities are listed:

1. A characterisation in terms of equivalence classes of concrete semantic objects which somehow reflect the 'operational' behaviour associated to a program or specification.
2. A more abstract explicit representation, i.e. an interpretation in a semantic domain whose elements are not equivalence classes of some concrete domain. When the interpretation is moreover compositional and fixed point theory is used to deal with recursion in the language, this type of semantics is often called 'denotational'.
3. An algebraic characterisation of semantic equality: two terms are equal if their identity can be proved by means of given algebraic laws.
4. A logic characterisation. Two expressions are semantically equal iff certain concrete semantic objects associated to them satisfy the same properties.
5. A characterisation in terms of a 'button pushing scenario'. To each expression an abstract machine is associated. Two expressions are considered semantically equal iff an experimenter, given a repertoire of experiments (like pushing buttons) and a set of possible observations (like reading a terminal screen), cannot observe any difference between the machines.
6. A characterisation in terms of a simple observation criterion and a language for which the semantics is 'fully abstract'. If $Obs(p)$ denotes the set of observations one can do on expression p in some language L, then an equivalence $\sim$ on L is 'fully abstract' with respect to Obs iff:

$$p \sim q \quad \Leftrightarrow \quad \text{for all } L\text{-contexts } C[\,] : Obs(C[p]) = Obs(C[q]).$$

7. A characterisation by means of abstraction homomorphisms (see for instance [12]).

3. Overview of work in this thesis

Besides the introduction, this thesis consists of the following six papers:

1. (with Jan Friso Groote). *Structured operational semantics and bisimulation as a congruence*, Report CS-R8845, Centrum voor Wiskunde en Informatica, Amsterdam, 1988. Submitted to Information and Computation. An extended abstract appeared in: Proceedings ICALP 89, Stresa (G. Ausiello, M. Dezani-Ciancaglini & S. Ronchi Della Rocca, eds.), LNCS 372, Springer-Verlag, pp. 423-438, 1989.
2. (with Rob van Glabbeek). *Modular specifications in process algebra.* This paper is obtained by leaving out the sections on 'curious queues' from the paper: *Modular specifications in process algebra - with curious queues*, Report CS-R8821, Centrum voor Wiskunde en Informatica, Amsterdam, 1988. Submitted to Theoretical Computer Science. An extended abstract appeared in: Algebraic Methods: Theory, Tools and Applications (M. Wirsing & J.A. Bergstra, eds.), LNCS 394, Springer-Verlag, pp. 465-506.
3. *Two simple protocols*, to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 23-44.
4. *Some observations on redundancy in a context*, Report CS-R8812, Centrum voor Wiskunde en Informatica, Amsterdam, 1988, to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 237-260.
5. *Process algebra semantics of POOL*, Report CS-R8629, Centrum voor Wiskunde en Informatica, Amsterdam, 1986, to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 173-236.
6. *Determinism → (event structure isomorphism = step sequence equivalence)*, Report CS-R8839, Centrum voor Wiskunde en Informatica, Amsterdam, 1988. Submitted to Theoretical Computer Science.

These papers can be read independently, except that papers 3, 4 and 5 use the language and axioms of ACP_τ as presented in paper 2.

Below I will comment on the papers separately. It is not my aim to give a complete overview of the results that have been obtained. Each paper has an introductory section where the results of that particular are summarised and also a comparison is made with related work.

3.1. Structured operational semantics. The first paper in this thesis was written last. It is concerned with a certain type of conditional rules, used for defining transition system semantics of (programming) languages. About ten years ago, the semantics of concurrency was generally considered to be a difficult issue and for many languages one did not know how to obtain a simple and intuitively convincing semantics. But then suddenly these conditional rules appeared. Since then, nobody can claim any more that it is difficult to give at least one (operational) semantics to any programming language used in practice: with these rules it has become more or less trivial. It seems that the idea of using conditional rules for giving semantics to concurrent programming

languages arose first in Edinburgh. The first publication I know of on semantics of parallelism in which the rules occur is a paper by HENNESSY & PLOTKIN [18] from 1979. Clearly, Plotkin was the one who most emphatically stressed the importance of the rules. Anyway, by now they are used widely and certainly form the most popular way to give an operational semantics to parallel programming languages. Therefore it was a bit surprising that there was almost no general, theoretical work on 'Plotkin style rules'. Notable exceptions were the Ph.D. Thesis of DE SIMONE [31] (see also [32]) and a recent paper by BLOOM, ISTRAIL & MEYER [10]. Together with Jan Friso Groote, I studied the question if it would be possible to know, just by looking at the form of the rules used for describing a certain language, various basic properties of the induced transition system semantics. In the first paper of this thesis this problem is addressed and it is shown that indeed the general form of the conditional rules already determines many key properties such as whether or not bisimulation is a congruence, the effect of adding new language constructs and rules on the semantics, and the nature of a fully abstract model determined by the operational rules.

Maybe one reason why semanticists have paid almost no attention to a general theory of Plotkin style rules is the fact that they are so simple to use. Because semanticists do not like to deal with trivialities, they tend to spend only a very small amount of their time on giving a Plotkin style operational semantics for a particular language; after that they rush forward to more difficult questions, like compositionality and full abstractness. I hope that the first paper of this thesis convinces people that it *is* worthwhile to spend a substantial amount of time on operational semantics. If one selects semantic rules carefully (and this is not a trivial task) then compositionality, full abstraction and all the rest may follow more or less automatically.

3.2. Modular specifications in process algebra. In Section 2, I pointed out that there are many different types of process semantics. In general, there is no clear reason to prefer one type of semantics over another: what is optimal depends on the particular application one has in mind. Besides the variety in process semantics, there is also a huge variety in languages. In order to deal with this combined complexity, I will employ in this thesis, as much as possible, an algebraic, axiomatic approach, that is I prefer to reason about programs and specifications on the level of syntax, using (infinitary, conditional) equations, instead of working on the level of semantics, i.e. in terms of the semantic objects associated to expressions. In my view, the advantages of this approach are the following:

1. The use of an algebraic, axiomatic approach is highly *organising* and *unifying*. As described in Section 2.7, there are many different ways to give semantics to languages. Often the only meaningful way to compare different semantics is to look which expressions are identified in each of them. It is exactly this crucial information that can be expressed by means of axioms. Often it occurs, and this is very illuminating to see, that the difference between two process semantics, which have been defined by

different people, at different places, in a completely different style, can be characterised in terms of one or two simple axioms.
2. Results from mathematical logic and the theory of abstract data types can be used. That this is not just a theoretical possibility, is illustrated in the second paper, where some nontrivial results from the field of universal algebra are used to solve certain semantic problems in concurrency.
3. System verifications can be done independent of a particular model. If one has proved correctness of say a communication protocol, using a certain set of axioms, then one knows that the protocol will be correct with respect to all models of these axioms. Thus system verifications become *reusable*.

Of course the algebraic approach also has disadvantages. There are many important issues in concurrency which cannot be dealt with algebraically. If one places too much stress on algebra, then one will tend to disregard the other issues, and this endangers the applicability of the theory. Let me give some examples.

1. Binding of variables is needed if one wants to describe value passing between processes. In this thesis I use binding of variables as a kind of notation, which is not formally present in the language, just because I want to stay in the realms of algebra. This is not the type of solution that one would like to see in a full-grown methodology.
2. I think that property languages are very important, also for establishing implementation relations in more advanced applications. Property languages do not really fit into an algebraic/axiomatic framework.
3. A next weak point of the algebraic methodology is that almost all relevant decision procedures can be described best on the semantic level. In this thesis I present some system verifications, but I do not present any algorithms which could be used to let a computer do these verifications also. It is just completely unclear how one could do such a thing in an algebraic way. It should be noted here that the algebraic approach allows one to do certain verifications which certainly could not be done by any existing (model based) tool. Most computer tools are developed for doing finite state verifications. As soon as the state space becomes infinite, or if one wants to verify some very generic statement like that the implementation of a programming language is correct, tools crash immediately.
4. At present there are no convincing axiomatisations for non-interleaved models. Maybe this explains why people advocating an algebraic, axiomatic approach to concurrency mainly work in the setting of interleaving, even when they agree that non-interleaved models are interesting.

The axiomatic theory, which is presented in the second paper, is essentially the Algebra of Communicating Processes (ACP) of BERGSTRA & KLOP [8, 9] augmented with a number of operators and axioms to make specification and verification of larger systems feasible. The main contribution of this paper is a structured presentation of operators and axioms using a notion of *module*. A module is a small collection of operators and axioms describing some feature of concurrency. Modules can be combined in various ways using module

operators.

In a rather strong sense the axioms as presented in the second paper correspond to what is called rooted-τ-bisimulation equivalence in [6] and weak bisimulation congruence in [28]. The idea is that, whenever possible, verifications are carried out using the laws of (rooted-τ-)bisimulation semantics. If this turns out not to be possible, then one can always add some laws. The motivation for doing things in this way is that: (1) mathematically, bisimulation is a very pleasant notion, it is a natural first behavioural abstraction from transition systems; (2) the axioms which capture bisimulation semantics are simple; (3) at least for finite state systems, deciding bisimulation is easy, this in contrast with all (interesting) equivalences which identify more; (4) roughly speaking, bisimulation semantics is the most refined semantics in which nontrivial system verifications are possible. Doing it with fewer laws is not feasible at the moment, often there is no need to use more laws.

Currently, some work is done on axiomatising even finer equivalences. First, there is the work by VAN GLABBEEK & WEIJLAND [17] on the *branching bisimulation*, which is a variant of the semantics that underlies the axioms in this thesis but gives a more subtle treatment of the silent step τ. Second, there is a paper by DARONDEAU & DEGANO [14] which contains a very good idea about how to axiomatise non-interleaved equivalences. I am convinced that, when the ideas of both papers have been worked out, many system verifications can be performed in these more discriminating semantics.

Clearly however, there are cases where the interleaving bisimulation semantics already does not work because it makes unnecessary distinctions between processes. One of these cases will be discussed in the paper about POOL.

3.3. Two simple protocols. In the third paper of this thesis, simple versions of the alternating bit protocol and the positive acknowledgement with retransmission protocol are specified and verified in the framework of ACP. These examples together with many other similar case studies (see for instance [5]) clearly show that it is possible and also useful to describe and analyse small systems in terms of process algebra. Certain features are dealt with in a slightly ad hoc way (for instance: fairness by means of the so-called Koomen's Fair Abstraction Rule, and time-outs using a priority operator) but generally speaking I think that the modelling is reasonably convincing. A more serious problem is how this type of verifications can be scaled up so that they become useful for 'real' applications.

To begin with, there is a problem with the language that I used. I tend to view ACP as a kind of assembly language for concurrency. In order to give precise and structured specifications of larger systems it becomes necessary to have higher level, more sophisticated languages. Candidates are LOTOS [20] or the ACP-based PSF [24], but personally I think that these proposals are far from ideal. In the first place they do not provide the flexibility and expressiveness that I would like to have, in the second place system verification becomes highly problematic as soon as you start to use the specification constructs that these languages have on top of say the ACP framework as presented in the

second paper of this thesis. For instance, in LOTOS as well as PSF it is allowed to input a value over an infinite data type. There is almost no theory about deciding behavioural equivalence of expressions that contain such a construct.

But also when using the ACP axioms, verification becomes difficult when the systems under consideration grow larger. In [34], I described a case study dealing with the verification of a one bit sliding window protocol. This protocol is not trivial (in fact I managed to discover a small error in the description of the protocol in [33]) but when compared with many existing protocols it is small in size and complexity. For me this case study was very instructive. It showed that the ACP axioms basically allow for one type of verification only: brute force state space exploration using the expansion theorem. The verifier can bring in some cleverness by first expanding and minimizing certain subexpressions (a technique called *local replacement* in [34]). Still this is not the way in which one would like to reason about protocols: it is rather boring, provides not much insight and takes a lot of time. Machines are good in brute force calculations and they don't mind doing boring work. However, the phenomenon of combinatorial state space explosion will make that, when the protocols become a bit larger, also computers will not succeed in exploring all the states.

If one looks for some time at a protocol like the one bit sliding window protocol as presented in [34], one just 'sees' that it is correct: one has constructed a chain of arguments which somehow makes one believe in the correctness of the protocol. What one would like to have is a formal verification technique that allows one to formalise these arguments rather directly so that one can check whether the reasoning is correct. A first and very modest step towards such a verification technique is described in the fourth paper in this thesis.

3.4. Redundancy in a context. When I was involved in the extensive calculations of [34], it occurred to me that at a number of places it would help a lot if I could just drop certain summands in a process algebra expression. Intuitively, it was obvious that these summands could be omitted because they corresponded to behaviours of components in the system that could never be realised due to the context in which these components were placed. The summands were so to say redundant in the given context. When I tried to formalise the intuitive reasoning for showing redundancy of summands, it turned out that in all cases that I considered this could be achieved using properties of the sets of traces of processes. I proved the soundness of a rule saying that one can omit a certain summand in a process expression if the trace sets of some subexpressions have certain properties. In order to make this rule practically useful in verifications, I needed a property language for expressing that the traces of a process have some property, together with a proof system. Here I used a many-sorted first-order predicate logic, which is called *trace logic*. It was employed before by many others (see for instance [13, 29, 36]). In the fourth paper the idea of using trace logic for proving behavioural equivalence of process expressions is worked out and its usefulness is illustrated by means

of a verification of a small workcell architecture.

3.5. Process algebra semantics of POOL. The main case study in this thesis is reported in the fifth paper. There I describe a translation of the Parallel Object-Oriented Language POOL to the ACP language. Moreover some results are obtained about the correctness of implementations of POOL. There are a couple of remarks on the paper that I would like to make here.

One of the main ideas behind the translation from POOL to ACP was that it would provide us with a large number of semantics for POOL, one for each interpretation of the ACP language. In the paper it is pointed out that the Koomen's Fair Abstraction Rule from ACP does not give one the right notion of fairness for POOL. Since no other fairness notion was available for process algebra at the time at which the paper was written, the issue of fairness was left as an open problem. Recently much work has been done on giving semantics to ACP-like languages using Petri nets (see for instance [15, 16, 29]). Now I claim that the notion of place fairness, which is well known for Petri nets, gives exactly the right notion of fairness for POOL if we use the ACP translation given in this thesis together with an interpretation of ACP in the domain of Petri nets in the style of [15, 29] (In [16] only finite processes are discussed.)

In the paper on POOL I prove that a semantical description of POOL (as defined in [2]) based on handshaking communication between objects, is incompatible with an implementation where message queues are used. Since any implementation of POOL will use message queues, and moreover the language designers really want users to think about communication between objects in terms of handshaking, this result meant that there was an error in the language design. The error was due to the most complex construct in the POOL language, namely the 'select statement'. This select statement was a terrible construct anyhow, in the paper no less than three pages are needed to describe its semantics. For these reasons the select statement has been removed altogether in a more recent offspring of the POOL-family of languages. Instead this language contains a 'conditional answer statement'. The questions now is whether the new version of the language can be correctly implemented using queues. My conjecture is 'Yes', but this still requires a proof. I think that at this moment the proof techniques within the process algebra formalism are sufficiently strong to tackle this nontrivial but important problem.

3.6. Deterministic event structures. In computer science there is the extremely useful distinction between functional behaviour and performance. The idea is that for a given (distributed) system one first studies whether it is functionally correct, and only when this has been shown (ideally), one moves to questions concerning its time/space complexity. The axioms that are used in the previous chapters of the thesis correspond to what is often called *interleaving semantics*. In interleaving semantics the actions of different components in a parallel system are interleaved. A typical equation valid in interleaving semantics is $a \| b = a \cdot b + b \cdot a$: if one considers the parallel composition of actions a and b,

either first a occurs and then b, or b occurs first followed by a. So any intuition that $a\|b$ is 'faster' than $a\cdot b+b\cdot a$ or that in $a\|b$ the a and the b are 'causally independent' whereas in $a\cdot b+b\cdot a$ there are causal links, cannot be captured in interleaving semantics in terms of primitive notions. Therefore, interleaving semantics may be appropriate for dealing with functional behaviour, but it is not really suited for analysing performance. I think that one important reason why non-interleaved semantics for languages with concurrency are interesting is that they may help to solve this problem.

A well-known system model that can be used for giving non-interleaved semantics is the model of event structures. In [35], WINSKEL gives an exposition of the theory of event structures where he also describes how CCS-like languages can be interpreted on the domain of event structures. Now it is interesting to look for behavioural equivalences on event structures that still preserve features like real-time behaviour, causality and branching time. A multitude of equivalences have been proposed over the last years and it is a topic of current research to classify these equivalences and find out which are the most interesting ones. The concluding paper of this thesis is a contribution to this area. I prove that for an important class of processes, namely the *deterministic* ones, almost all of the non-interleaved equivalences that have been proposed in the literature coincide. More specifically, I will show that step sequence equivalence and event structure isomorphism agree on the domain of deterministic event structures. Since step sequence equivalence, which makes a lot of identifications and is almost an interleaving equivalence, can be axiomatised easily, this result can be used to obtain an algebraic characterisation of event structure isomorphism for deterministic systems.

REFERENCES

[1] P. ACZEL (1988): *Non-well-founded sets*, CSLI Lecture Notes No.14, Stanford University.

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

[3] P. AMERICA (1989): *The practical importance of formal semantics.* In: J.W. de Bakker, 25 jaar semantiek, liber amicorum, pp. 31-40.

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

[5] J.C.M. BAETEN (ED.) (1990): *Applications of process algebra*, to appear.

[6] J.C.M. BAETEN, J.A. BERGSTRA & J.W. KLOP (1987): *On the consistency of Koomen's fair abstraction rule.* Theoretical Computer Science 51(1/2), pp. 129-176.

[7] J.W. DE BAKKER & J.I. ZUCKER (1982): *Processes and the denotational semantics of concurrency.* I&C 54(1/2), pp. 70-120.

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

[9] J.A. BERGSTRA & J.W. KLOP (1989): *Process theory based on bisimulation semantics.* In: Proceedings REX School/Workshop on Linear Time,

Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout (J.W. de Bakker, W.-P. de Roever & G. Rozenberg, eds.), LNCS 354, Springer-Verlag, pp. 50-122.

[10] B. BLOOM, S. ISTRAIL & A.R. MEYER (1988): *Bisimulation can't be traced: preliminary report.* In: Conference Record of the 15th ACM Symposium on Principles of Programming Languages (POPL), San Diego, California, pp. 229-239.

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

[12] I. CASTELLANI (1987): *Bisimulations and abstraction homomorphisms.* Journal of Computer and System Sciences 34, pp. 210-235.

[13] Z. CHAOCHEN & C.A.R. HOARE (1981): *Partial correctness of communicating processes.* In: Proceedings 2nd Intern. Conf. on Distributed Comput. Systems, Paris.

[14] P. DARONDEAU & P. DEGANO (1988): *Causal trees.* Publication Interne No. 442, IRISA, Rennes Cedex, France, extended abstract appeared in: Proceedings ICALP 89, Stresa (G. Ausiello, M. Dezani-Ciancaglini & S. Ronchi Della Rocca, eds.), LNCS 372, Springer-Verlag, pp. 234-248.

[15] P. DEGANO, R. DE NICOLA & U. MONTANARI (1988): *A distributed operational semantics for CCS based on condition/event systems.* Acta Informatica 26(1/2), pp. 59-91.

[16] R.J. VAN GLABBEEK & F.W. VAANDRAGER (1987): *Petri net models for algebraic theories of concurrency.* In: Proceedings PARLE conference, Eindhoven, Vol. II (Parallel Languages) (J.W. de Bakker, A.J. Nijman & P.C. Treleaven, eds.), LNCS 259, Springer-Verlag, pp. 224-242.

[17] R.J. VAN GLABBEEK & W.P. WEIJLAND (1989): *Branching time and abstraction in bisimulation semantics (extended abstract).* In: Information Processing 89 (G.X. Ritter, ed.), Elsevier Science Publishers B.V. (North Holland), pp. 613-618.

[18] M. HENNESSY & G.D. PLOTKIN (1979): *Full abstraction for a simple programming language.* In: Proceedings 8th Symposium on Mathematical Foundations of Computer Science (MFCS) (J. Bečvář, ed.), LNCS 74, Springer-Verlag, pp. 108-120.

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

[20] ISO (1987): *Information processing systems - open systems interconnection - LOTOS - a formal description technique based on the temporal ordering of observational behaviour.* ISO/TC 97/SC 21 N DIS8807.

[21] R.M. KELLER (1976): *Formal verification of parallel programs.* Communications of the ACM 19(7), pp. 371-384.

[22] N. LYNCH & M. TUTTLE (1987): *Hierarchical correctness proofs for distributed algorithms.* In: Proceedings 6th ACM Symposium on Principles of Distributed Computing (PODC), Vancouver, Canada, August '87, pp. 137-151.

[23] Z. MANNA & A. PNUELI (1989): *The anchored version of the temporal framework*. In: Proceedings REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout (J.W. de Bakker, W.-P. de Roever & G. Rozenberg, eds.), LNCS 354, Springer-Verlag, pp. 201-284.

[24] S. MAUW & G.J. VELTINK (1988): *A process specification formalism*. Report P8814, Programming Research Group, University of Amsterdam, to appear in: Fundamenta Informaticae.

[25] A. MAZURKIEWICZ (1987): *Trace theory*. In: Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course, Bad Honnef, September 1986 (W. Brauer, W. Reisig & G. Rozenberg, eds.), LNCS 255, Springer-Verlag, pp. 279-324.

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

[27] R. MILNER (1983): *Calculi for synchrony and asynchrony*. Theoretical Computer Science 25, pp. 267-310.

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

[29] E.-R. OLDEROG (1988): *Nets, terms and formulas: three views of concurrent processes and their relationship*. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Fed. Rep. Germany.

[30] W. REISIG (1985): *Petri nets - an introduction*, EATCS Monographs on Theoretical Computer Science, Volume 4, Springer-Verlag.

[31] R. DE SIMONE (1984): *Calculabilité et expressivité dans l'algebra de processus parallèles Meije*. Thèse de 3^e cycle, Univ. Paris 7.

[32] R. DE SIMONE (1985): *Higher-level synchronising devices in* MEIJE-SCCS. Theoretical Computer Science 37, pp. 245-267.

[33] A.S. TANENBAUM (1981): *Computer networks*, Prentice-Hall International.

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

[35] G. WINSKEL (1987): *Event structures*. In: Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course, Bad Honnef, September 1986 (W. Brauer, W. Reisig & G. Rozenberg, eds.), LNCS 255, Springer-Verlag, pp. 325-392.

[36] J. ZWIERS (1989): *Compositionality, concurrency and partial correctness*, LNCS 321, Springer-Verlag.

Structured Operational Semantics and Bisimulation as a Congruence

Jan Friso Groote and Frits Vaandrager
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

In this paper we are interested in general properties of classes of transition system specifications in Plotkin style. The discussion takes place in a setting of labelled transition systems. The states of the transition systems are terms generated by a single sorted signature and the transitions between states are defined by conditional rules over the syntax. It is argued that in this setting it is natural to require that strong bisimulation equivalence is a congruence on the states of the transition systems. A general format, called the *tyft/tyxt* format, is presented for the rules in a transition system specification, such that bisimulation is always a congruence when all the rules fit this format. With a series of examples it is demonstrated that the *tyft/tyxt* format cannot be generalized in any obvious way. Another series of examples illustrates the usefulness of our congruence theorem. Briefly we touch upon the issue of modularity of transition system specifications. It is argued that certain pathological *tyft/tyxt* rules (the ones which are not *pure*) can be disqualified because they behave badly with respect to modularisation. Next we address the issue of full abstraction. We characterize the completed trace congruence induced by the operators in pure *tyft/tyxt* format as *2-nested simulation equivalence*. The pure *tyft/tyxt* format includes the format given by De Simone (1984,1985) but is incomparable to the GSOS format of Bloom, Istrail & Meyer (1988). However, it turns out that 2-nested simulation equivalence strictly refines the completed trace congruence induced by the GSOS format.

Key Words and Phrases: Structured Operational Semantics (SOS), transition system specifications, compositionality, labelled transition systems, bisimulation, congruence, process algebra, *tyft/tyxt* format, modularity of transition system specifications, full abstraction, testing, nested simulations, Hennessy-Milner logic, De Simone format, GSOS format.

1. Introduction

Plotkin (1981,1983) advocates a simple method for giving operational semantics to programming languages. The method, which is often referred to as *SOS* (for *Structured Operational Semantics*), is based on the notion of transition systems. The states of the transition systems are elements of some formal language that, in general, will extend the language for which one wants to give an operational semantics. The main idea of the method is to define the transitions between states by, what we call a *Transition System Specification (TSS)*: a set of conditional rules over the syntax of the language.

In recent years a large number of (concurrent) languages have been provided with an operational semantics using Plotkin's approach. Therefore it might be worthwhile to develop a *general* theory of structured operational semantics: to establish a hierarchy of 'formats' of transition system specifications and to investigate the expressiveness and nice properties of each format. We think that it is possible to develop such a general theory: many important properties of transition system specifications can be derived by just looking at the syntactic form of the rules. A general theory of SOS will be useful for several reasons. Firstly, certain results will become reusable so that one does not have to prove them for each individual language separately. Secondly, a general theory of SOS may lead to a better understanding of the relations between languages that have been provided with a semantics using the approach. Thirdly, one may hope that a general theory helps people in giving good operational semantics: if one knows that certain types of rules have bad properties, then one will try not to use them. Surprisingly, there are not so many papers that contain general results on SOS. We are only aware of the work of De Simone (1984,1985) and Bloom, Istrail & Meyer (1988).

The aim of this paper is to contribute to the general theory of structured operational semantics. We start from the requirement that strong bisimulation equivalence should be a congruence for the operators in a transition system specification. We then show how this requirement leads naturally to a certain format of rules, which we call the *tyft/tyxt* format. Next we analyze the properties of the *tyft/tyxt* format and make comparisons with related work.

In order to facilitate analysis, we restrict our attention to a specific type of transition systems: transitions are labelled and as states we have ground terms generated by a single sorted signature. This is an important subcase: the operational semantics of languages like CCS (Milner, 1980), TCSP (Olderog & Hoare, 1986), ACP (Van Glabbeek, 1987) and Meije (Boudol, 1985) has been described in essentially this way. However, there are also many examples of transition system specifications where the set of states is not specified by a single sorted signature, for instance the semantics for CSP as presented by Plotkin (1983) and the semantics for POOL of America, De Bakker, Kok & Rutten (1986). We hope that the insights derived from our analysis of a basic case will somehow generalize to more general settings.

1.1. Bisimulation as a congruence. A fundamental equivalence on the states of a labelled transition system is the strong bisimulation equivalence of PARK (1981). Strong bisimulation equivalence seems to be the *finest* extensional behavioural equivalence one would want to impose, i.e. two states of a transition system which are strongly bisimilar cannot be distinguished by external observation. This means that from an observational point of view, the transition systems generated by the SOS approach are too concrete as semantical objects. The objects that really interest us will be *abstract* transition systems where the states are bisimulation equivalence classes of terms, or maybe something even more abstract. If bisimulation is not a congruence then the function that computes the transitions associated to a phrase from the transitions associated to its components, depends on properties of the transition system which are generally considered to be irrelevant, such as the specific names of states. Hence we think that a transition system specification which leads to transition systems for which bisimulation is not a congruence should not be called *structured*: possibly it is compositional on the level of (concrete) transition systems but it is not compositional on the more fundamental level of transition systems modulo bisimulation equivalence.

This brings us to the first main question of this paper which is to find a format, as general as possible, for the rules in a transition system specification, such that bisimulation is always a congruence when all the rules have this format. We proceed in a number of steps.

In Section 2 of the paper definitions are given of some basic notions like signature, term and substitution. Section 3 contains a formal definition of the notion of a transition system specification (TSS). In Section 4 it is described how a TSS determines a transition system. Moreover the fundamental notion of strong bisimulation is introduced. The real work starts in Section 5, where we present a general format, called the *tyft/tyxt* format, for the inductive rules in a TSS and prove that bisimulation is always a congruence when all rules have this format (and a small additional requirement is met). With a series of examples it is demonstrated that this format cannot be generalized in any obvious way.

Section 6 contains some applications of our congruence theorem. We think that our result will be useful in many situations because it allows one to see immediately that bisimulation is a congruence. Thus it generalizes and makes less *ad hoc* the congruence proofs in (MILNER, 1983), (BAETEN & VAN GLABBEEK, 1987) and elsewhere. If the rules in a TSS do not fit our format then there is a good chance that something will be wrong: either bisimulation is not a congruence right away or the congruence property will get lost if more operators and rules are added.

1.2. Modularity of transition system specifications. Often one wants to add new operators and rules to a TSS. Therefore, a very natural and important operation on TSS's is to take their componentwise union. Given two specifications P_0 and P_1, let $P_0 \oplus P_1$ denote this union. A desirable property to have is that the outgoing transition of states in the transition system associated to P_0 are the same as the outgoing transitions of these states in the extended system $P_0 \oplus P_1$. This means that $P_0 \oplus P_1$ is a 'conservative extension' of P_0: any property which has been proved for the states in the old transition system remains valid (for the old states) in the enriched system. In Section 7 we show that, except for certain rules which are not *'pure'*, *tyft/tyxt* rules behave fine under modularisation. Fortunately, non-pure rules are quite pathological and we have never seen an application in which they are used.

1.3. Trace congruences. A central idea in the theory of concurrency is that processes which cannot be distinguished by observation, should be identified: the process semantics should be *fully abstract* with respect to some notion of testing (DE NICOLA & HENNESSY, 1984). Natural observations that one can make on a process are its *(completed) traces*. A *trace* of a process is a finite sequence of actions that can be performed during a run of the process. A trace is *completed* if it leads to a state from where no further actions are possible. Two processes are *(completed) trace congruent* with respect to some format of rules if they yield the same (completed) traces in any context that can be built from operations defined in this format. The first main result of Section 8 is a characterization, valid for image finite transition systems, of the completed trace congruence induced by the pure *tyft/tyxt* format as *2-nested simulation equivalence*. On the domain of image finite transition systems, 2-nested simulation coincides with the equivalence induced by the Hennessy-Milner logic formulas (HENNESSY & MILNER, 1985) with no [] in the scope of a $\langle\rangle$. Consequently the two trees in Figure 1, which are not bisimilar, cannot be distinguished by operators defined with pure *tyft/tyxt* rules. Also in Section 8, we characterize the trace congruence induced by the pure *tyft/tyxt* format as *simulation equivalence*.

1.4. Comparison with related work. In Section 9 we give an extensive comparison of our format with the format proposed by DE SIMONE (1984,1985) and the GSOS format of BLOOM, ISTRAIL & MEYER (1988). Roughly speaking, the situation is as displayed in Figure 2. The GSOS format and the pure *tyft/tyxt* format both generalize the format of De Simone. The GSOS format and our format are incomparable since the GSOS format allows negations in the premises, whereas all our rules are positive. On the other hand we allow for rules that give operators a lookahead and this is not allowed by the GSOS format. A simple example in (BLOOM, ISTRAIL & MEYER, 1988) shows that the combination of negation and lookahead is inconsistent in general. The point where the two formats diverge is characterized by the rules which fit the GSOS format but which contain no negation. We call the corresponding format *positive GSOS*.

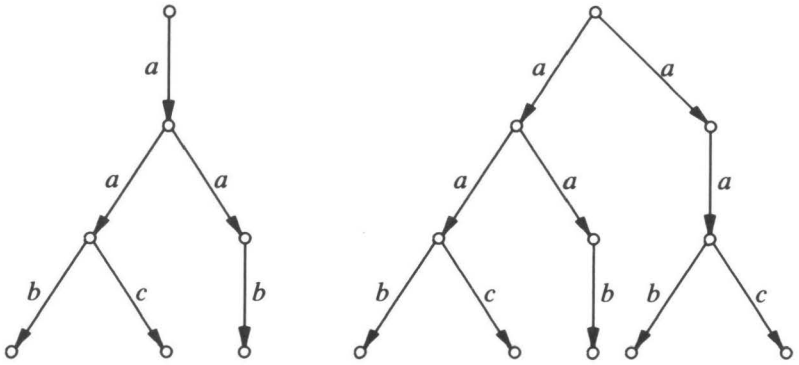

FIGURE 1. Pure *tyft/tyxt* congruent but not bisimilar

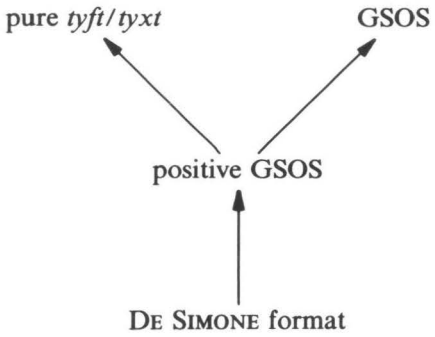

FIGURE 2

From results of DE SIMONE (1985) and BERGSTRA, KLOP & OLDEROG (1988) it follows that the completed trace congruence that corresponds to the format of De Simone coincides with *failure equivalence*. BLOOM, ISTRAIL & MEYER (1988) proved that the completed trace congruence induced by the GSOS format can be characterized by the class of Hennessy-Milner logic formulas in which only F may occur in the scope of a []. LARSEN & SKOU (1988) in turn showed that the equivalence induced by this class of logical formulas can be characterized as *2/3-bisimulation equivalence*. From these results we can conclude quite directly that the pure *tyft/tyxt* format can make more distinctions between processes than the GSOS format: 2-nested simulation refines 2/3-bisimulation. Now, interestingly, it turns out that the completed trace congruence induced by the *positive* GSOS format is also 2/3-bisimulation equivalence. So although it may be the case that in the general GSOS format can be used to define certain operations which cannot be defined using positive rules only, the use of negations in the definition of operators does not introduce any new distinctions between processes!

The notion of testing associated with the (positive) GSOS format allows one to observe *traces* of processes, to detect *refusals* and to make *copies* of processes at every moment. Our format allows one in addition to test whether some action is possible in the future: operators can have a *lookahead*. This can be seen as a weak form of *global testing* (ABRAMSKY, 1987).

A notable difference between the GSOS format and our format is that the GSOS format always leads to a computably finitely branching transition relation whereas our format does not necessarily do so. We argue that, even though finiteness and computability are very desirable properties, the statement of BLOOM, ISTRAIL & MEYER (1988) that any 'reasonably structured' specification should induce a computably finitely branching transition relation, is too strong and discards a large number of interesting applications.

ACKNOWLEDGEMENTS. We want to thank Bard Bloom for a very interesting and stimulating correspondence. Discussions with him had a pervasive influence on the contents of this paper. We also thank Rob van Glabbeek for many useful comments and inspiring discussions. Finally, we thank the referee of this paper.

2. PRELIMINARIES

In this paper we will work with a very simple notion of a signature. Only one sort is allowed; there are only function symbols and no predicate symbols; there is no overloading and no recursion construct. Throughout this paper we assume the presence of an infinite set V of *variables* with typical elements $x,y,z,...$

2.1. DEFINITION. A *(single sorted) signature* Σ is a pair (F,r) where:
- F is a set of *function symbols* disjoint with V,
- $r:F\to\mathbb{N}$ is a *rank function* which gives the arity of a function symbol; if $f\in F$ and $r(f)=0$ then f is called a *constant symbol.*

2.2. DEFINITION. Let $\Sigma=(F,r)$ be a signature. Let $W\subseteq V$ be a set of variables. The set of Σ-*terms* over W, notation $T(\Sigma,W)$, is the least set satisfying:
- $W\subseteq T(\Sigma,W)$,
- if $f\in F$ and $t_1,..,t_{r(f)}\in T(\Sigma,W)$, then $f(t_1,..,t_{r(f)})\in T(\Sigma,W)$.

$T(\Sigma,\varnothing)$ is abbreviated by $T(\Sigma)$ and $T(\Sigma,V)$ is abbreviated by $\mathbb{T}(\Sigma)$; elements from $T(\Sigma)$ are called *closed* or *ground terms*, elements from $\mathbb{T}(\Sigma)$ are called *open terms*. $Var(t)\subseteq V$ is the set of variables in a term $t\in\mathbb{T}(\Sigma)$.

2.3. DEFINITION. Let $\Sigma=(F,r)$ be a signature. A *substitution* σ is a mapping in $V\to\mathbb{T}(\Sigma)$. A substitution σ is extended to a mapping $\sigma:\mathbb{T}(\Sigma)\to\mathbb{T}(\Sigma)$ in a standard way by the following definition:
- $\sigma(f(t_1,..,t_{r(f)})) = f(\sigma(t_1),..,\sigma(t_{r(f)}))$ for $f\in F$ and $t_1,..,t_{r(f)}\in\mathbb{T}(\Sigma)$.

If σ and ρ are substitutions, then the substitution $\sigma\circ\rho$ is defined by:

$$\sigma\circ\rho(x) = \sigma(\rho(x)) \quad \text{for } x\in V.$$

2.4. NOTE. Observe that we have the following identities:

$$\sigma\circ\rho(t) = \sigma(\rho(t)) \qquad t\in\mathbb{T}(\Sigma)$$
$$\sigma(t) = t \qquad \text{for } t\in T(\Sigma)$$

3. TRANSITION SYSTEM SPECIFICATIONS

In this section a formal definition is given of the notion of a transition system specification. Also the notion of a proof of a transition from such a specification is defined.

3.1. DEFINITION. A *transition system specification (TSS)* is a triple (Σ,A,R) with Σ a signature, A a set of *labels* and R a set of *rules* of the form:

$$\frac{\{t_i \xrightarrow{a_i} t_i' \mid i\in I\}}{t \xrightarrow{a} t'}$$

where I is an index set, $t_i,t_i',t,t'\in\mathbb{T}(\Sigma)$ and $a_i,a\in A$ for $i\in I$. If r is a rule in the format above, then the elements of $\{t_i \xrightarrow{a_i} t_i' \mid i\in I\}$ are called the *premises* or *hypotheses* of r and $t \xrightarrow{a} t'$ is called the *conclusion* of r. A rule of the form $\frac{\emptyset}{t \xrightarrow{a} t'}$ is called an *axiom*, which, if no confusion can arise, is also written as $t \xrightarrow{a} t'$. An expression of the form $t \xrightarrow{a} t'$ with $a\in A$ and $t,t'\in\mathbb{T}(\Sigma)$ is called a *transition (labelled with a)*. The symbols $\phi,\psi,\chi,..$ will be used to range over transitions. The notions 'substitution', '*Var*' and 'closed' extend to transitions and rules as expected.

3.2. DEFINITION. Let $P=(\Sigma,A,R)$ be a TSS. A *proof* of a transition ψ from P is a well-founded, upwardly branching tree of which the nodes are labelled by transitions $t \xrightarrow{a} t'$ with $t,t'\in\mathbb{T}(\Sigma)$ and $a\in A$, such that:

- the root is labelled with ψ,
- if χ is the label of a node q and $\{\chi_i \mid i\in I\}$ is the set of labels of the nodes directly above q, then there is a rule $\frac{\{\phi_i \mid i\in I\}}{\phi}$ in R and a substitution $\sigma:V\to\mathbb{T}(\Sigma)$ such that $\chi=\sigma(\phi)$ and $\chi_i=\sigma(\phi_i)$ for $i\in I$.

If a proof of ψ from P exists, we say that ψ is *provable* from P, notation $P\vdash\psi$. A proof is *closed* if it only contains closed transitions.

3.3. LEMMA. *Let $P=(\Sigma,A,R)$ be a TSS, let $a\in A$ and let $t,t'\in T(\Sigma)$ such that $P\vdash t \xrightarrow{a} t'$. Then $t \xrightarrow{a} t'$ is provable by a closed proof.*

PROOF. As $P\vdash t \xrightarrow{a} t'$ there is a proof tree T for $t \xrightarrow{a} t'$. Define the substitution $\sigma:V\to T(\Sigma)$ by $\sigma(x)=t$ for all $x\in V$ (in fact, any closed term will do). Applying σ to all transitions in the proof T of $t \xrightarrow{a} t'$ yields a tree T' containing only closed transitions. Now one can easily check that T' is a proof of $t \xrightarrow{a} t'$. □

TSS's have been used mainly as a tool to give operational semantics to (concurrent) programming languages. As a running example we therefore present below a TSS for a simple process language.

3.4. EXAMPLE. Let $Act = \{a,b,c,..\}$ be a given set of *actions*. We consider the signature $\Sigma(\mathrm{BPA}_\delta^\epsilon)$ (Basic Process Algebra with δ and ϵ) as introduced in VRANCKEN (1986). $\Sigma(\mathrm{BPA}_\delta^\epsilon)$ contains constants a for each $a \in Act$, a constant δ that stands for *deadlock* or *inaction*, comparable to NIL in CCS and STOP in TCSP, and a constant ϵ that denotes the *empty process*, a process that terminates immediately and successfully. It is comparable to SKIP in TCSP and skip in CCS. Furthermore the signature contains binary operators $+$ (*alternative composition*) and $\cdot$ (*sequential composition*). As labels of transitions we take elements of $Act_\surd = Act \cup \{\surd\}$. Here $\surd$ (pronounce 'tick') is a special symbol used to denote the action of successful termination. At the end of a process this action indicates that execution has finished.
Define the TSS $P(\mathrm{BPA}_\delta^\epsilon)$ as $(\Sigma(\mathrm{BPA}_\delta^\epsilon), Act_\surd, R(\mathrm{BPA}_\delta^\epsilon))$ where $R(\mathrm{BPA}_\delta^\epsilon)$ is defined below in Table 1. In the table a ranges over $Act_\surd$, unless further restrictions are made. Infix notation is used for the binary function symbols.

1.	$a \xrightarrow{a} \epsilon$	$a \neq \surd$	2.	$\epsilon \xrightarrow{\surd} \delta$
3.	$\dfrac{x \xrightarrow{a} x'}{x+y \xrightarrow{a} x'}$		4.	$\dfrac{y \xrightarrow{a} y'}{x+y \xrightarrow{a} y'}$
5.	$\dfrac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$	$a \neq \surd$	6.	$\dfrac{x \xrightarrow{\surd} x' \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$

TABLE 1. The rules of $R(\mathrm{BPA}_\delta^\epsilon)$

One can easily check that the tree in Figure 3 constitutes a proof of the transition $(\epsilon \cdot (a+b)) \cdot c \xrightarrow{a} \epsilon \cdot c$ from $P(\mathrm{BPA}_\delta^\epsilon)$.

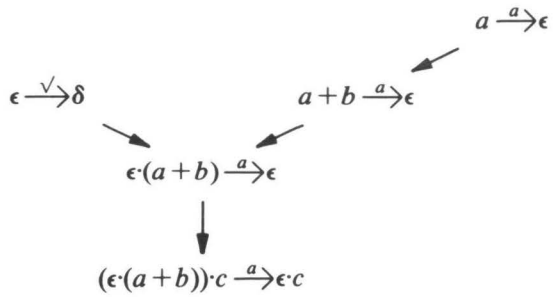

FIGURE 3

3.4.1. REMARK. Even though similar semantic interpretations have been given to (extensions of) $\Sigma(\mathrm{BPA}_\delta^\varepsilon)$ at a number of places, the rules of Table 1 seem to be new. VRANCKEN (1986) does not use inductive rules to give semantics to $\mathrm{BPA}_\delta^\varepsilon$. Instead, operations are defined directly on *process graphs*. In (BAETEN & VAN GLABBEEK, 1987) there are no transitions labelled with $\surd$. Instead, a unary termination predicate $\downarrow$ is used. The analogue of our rule 6 in their setting is:

$$\frac{x\downarrow,\ y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$$

Such a rule does not fit in the framework of this paper. We have chosen not to deal here with predicates like $\downarrow$ because the additional complexity would distract attention from the main issues in this paper. Moreover, a unary predicate $p(x)$ can always be coded in our setting by adding a new label a_p and rules such that:

$$p(x) \quad \Leftrightarrow \quad \exists y : x \xrightarrow{a_p} y.$$

We think that it will not be too difficult to extend the framework of this paper with predicates.

3.5. EXAMPLE. Our next example shows that the range of applications of TSS's is not restricted to the area of operational semantics: every Term Rewriting System (TRS) can be viewed as a TSS. Unfortunately, it seems that the intersection of the class of TSS's which correspond to TRS's and the class of TSS's for which bisimulation is a congruence is of no interest. A *Term Rewriting System (TRS)* is defined as a pair (Σ_0, R_0) with Σ_0 a signature and R_0 a set of *reduction* or *rewrite rules* of the form $r:(t,s)$ with r the *name* of the rewrite rule and $t,s \in \mathbb{T}(\Sigma_0)$. Here, t contains at least one function symbol and $Var(s) \subseteq Var(t)$.
A TRS (Σ_0, R_0) can be viewed as a TSS (Σ, A, R). Take $\Sigma = \Sigma_0$ as the signature and define the alphabet A as the set of all names r of rules $r:(t,s) \in R_0$. R contains for every r: $(t,s) \in R_0$ a rule:

$$t \xrightarrow{r} s$$

and for every function symbol f in Σ rules:

$$\frac{x \xrightarrow{r} y}{f(x_1,..,x,..,x_{r(f)}) \xrightarrow{r} f(x_1,..,y,..,x_{r(f)})}$$

to allow reductions in contexts. One can easily prove that there is a *one step rewrite* $t \longrightarrow_r s$ in the TRS (see (KLOP, 1987) for a definition) iff the corresponding TSS proves $t \xrightarrow{r} s$.

4. Transition systems and strong bisimulation equivalence

An operational semantics makes use of some sort of (abstract) machines and describes how these machines behave. Often one takes as machines simply nondeterministic automata in the sense of classical automata theory, also called labelled transition systems (Keller, 1976).

4.1. Definition. A *(nondeterministic) automaton* or *labelled transition system (LTS)* is a structure $(S,A,\rightarrow)$ where:
- S is a set of *states*,
- A is an *alphabet*,
- $\rightarrow \subseteq S \times A \times S$ is a *transition relation*.

Elements $(s,a,s') \in \rightarrow$ are called *transitions* and will be written as $s \xrightarrow{a} s'$. The intended interpretation is that from state s the machine can do an action a and thereby get into state s'.

4.1.1. Remark. Often transition systems are provided with an additional fourth component: the *initial state*. For our purpose it has some small technical advantages to work with transition systems that do not contain this ingredient. All considerations of this paper can trivially be extended to transition systems with initial state.

The notion of strong bisimulation equivalence as defined below is from Park (1981).

4.2. Definition. Let $\mathcal{A}=(S,A,\rightarrow)$ be a labelled transition system. A relation $R \subseteq S \times S$ is a *(strong) bisimulation* if for all s,t with $s\ R\ t$:
1. whenever $s \xrightarrow{a} s'$ for some a and s', then, for some t', also $t \xrightarrow{a} t'$ and $s' R\ t'$,
2. conversely, whenever $t \xrightarrow{a} t'$ for some a and t', then, for some s', also $s \xrightarrow{a} s'$ and $s' R\ t'$.

Two states $s,t \in S$ are *bisimilar* in $\mathcal{A}$, notation $\mathcal{A}: s \leftrightarrows t$, if there exists a bisimulation containing the pair (s,t). Note that bisimilarity is indeed an equivalence relation on states.

4.3. Definition (*TSS's, transition systems and bisimulation*). Let $P=(\Sigma,A,R)$ be a TSS. The transition system $TS(P)$ specified by P is given by:

$$TS(P) = (T(\Sigma),A,\rightarrow_P),$$

where relation $\rightarrow_P \subseteq T(\Sigma) \times A \times T(\Sigma)$ is defined by: $t \xrightarrow{a}_P t' \Leftrightarrow P \vdash t \xrightarrow{a} t'$.

We say that two terms $t,t' \in T(\Sigma)$ are *(P-)bisimilar*, notation $t \leftrightarrows_P t'$, if $TS(P): t \leftrightarrows t'$. We write $t \leftrightarrows t'$ if it is clear from the context what P is. Note that $\leftrightarrows_P$ is also an equivalence relation.

4.4. EXAMPLE. For the TSS $P(\mathrm{BPA}_\delta^\epsilon)$ of Example 3.4 we can derive the identities (a)-(e) below. In (f) it is shown that the left distributivity of $\cdot$ over $+$ does not hold in bisimulation semantics. Like in regular algebra we will often omit the $\cdot$ in a product $x{\cdot}y$ and we take $\cdot$ to be more binding than $+$.

(a) $\epsilon\epsilon \leftrightarrows \epsilon$
(b) $b \leftrightarrows b+b$
(c) $(\epsilon a+\epsilon b)(c(d\delta)+\delta) \leftrightarrows (a((c+\delta)d)+b(c(d+d)))\delta$
(d) $b\epsilon \leftrightarrows b$
(e) $\epsilon b \leftrightarrows b$
(f) $ab+ac \not\leftrightarrows a(b+c)$

The parts of the automaton belonging to (a),(b),(c) and (f) are drawn in Figure 4-6. A dotted line indicates that a pair of states is in the bisimulation relation. Furthermore, a state is always related to itself. In showing that two states are related, only the states that can be reached from these states are relevant and therefore only these states are drawn.

FIGURE 4. Examples 4.4(a) and 4.4(b)

In Figures 5/6 two separate automata are drawn instead of a combined one, to make the pictures clearer.

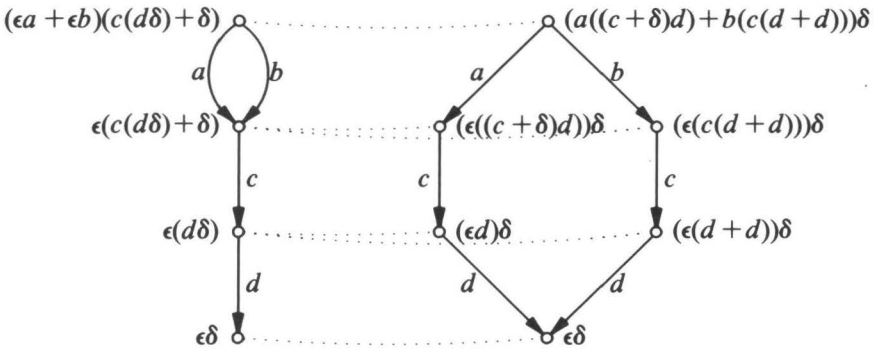

FIGURE 5. Example 4.4(c)

In Figure 6 the states $a(b+c)$ and $\epsilon(b+c)$ in the right transition system cannot be related to any of the states in the left transition system.

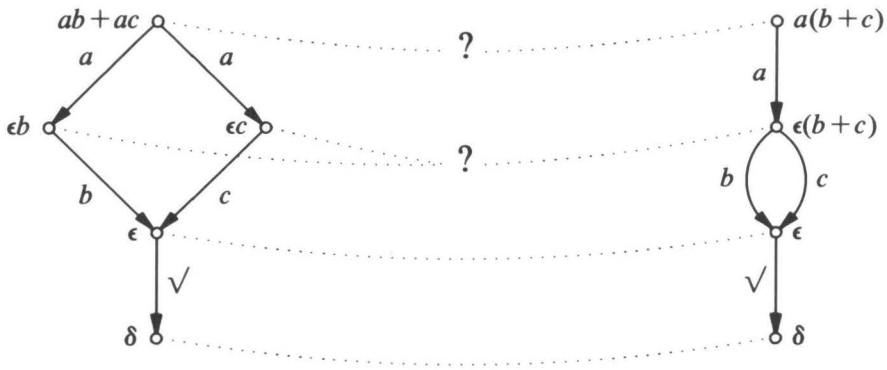

FIGURE 6. Example 4.4(f)

5. COMPOSITIONAL TRANSITION SYSTEM SPECIFICATIONS

TSS's do not always generate automata for which strong bisimulation is a congruence. A number of examples will follow in the sequel. But if the rules in TSS satisfy the format below (and an additional small technical requirement is met), strong bisimulation will turn out to be a congruence.

5.1. DEFINITION. Let $\Sigma=(F,r)$ be a signature and let $P=(\Sigma,A,R)$ be a TSS. A rule in R is in *tyft format* if it has the following form:

$$\frac{\{t_i \xrightarrow{a_i} y_i \mid i\in I\}}{f(x_1,..,x_n)\xrightarrow{a} t}$$

with I an index set, $f\in F$, $r(f)=n$, x_i $(1\leqslant i\leqslant n)$ and y_i $(i\in I)$ are all different variables from V, $a_i,a\in A$ and $t_i,t\in\mathbb{T}(\Sigma)$ for $i\in I$.

A rule in R is in *tyxt format* if it has the following form:

$$\frac{\{t_i \xrightarrow{a_i} y_i \mid i\in I\}}{x\xrightarrow{a} t}$$

with I an index set, x,y_i $(i\in I)$ all different variables from V, $a_i,a\in A$ and $t_i,t\in\mathbb{T}(\Sigma)$ for $i\in I$. P is in *tyft/tyxt format* if every rule in R is either in *tyft* format or in *tyxt* format. A transition system $\mathcal{A}$ is called *tyft/tyxt specifiable* if there exists a TSS P in *tyft/tyxt* format with $\mathcal{A}=TS(P)$.

5.2. NOTE. Observe that there does not have to be any relation at all between the premises and the conclusions in a rule satisfying our format. In fact our format explicitly requires the absence of certain relations between occurrences of variables in the premises and in the conclusion. Note that not only the TSS $P(\mathrm{BPA}_\delta^\varepsilon)$ of Example 3.4 is in *tyft/tyxt* format, but also any TSS obtained from $P(\mathrm{BPA}_\delta^\varepsilon)$ by dropping some rules. The transition system specifications related to term rewriting systems (see Example 3.5) are in general not in *tyft/tyxt* format.

5.3. EXAMPLE. Below we describe a TSS that models a simple typewriter that can be used to type strings and that has the option to delete the last character of the typed string using 'backspace'. The signature consists of the binary function symbol $\star$ denoting concatenation, and constant symbols λ (empty string) and $a,b,..,y,z$. As alphabet we take $A=\{a,b,..,y,z,\Delta\}$. Here, Δ stands for a backspace. Rules for the typewriter can be given as follows:

$$x \xrightarrow{a} x\star a \qquad \text{for } a\in\{a,b,..,y,z\}$$

$$a \xrightarrow{\Delta} \lambda \qquad \text{for } a\in\{a,b,..,y,z\}$$

$$x\star a \xrightarrow{\Delta} x \qquad \text{for } a\in\{a,b,..,y,z\}$$

This description of the typewriter is not in *tyft/tyxt* format, because the lhs of the last axiom contains two function symbols. A TSS for the typewriter in *tyft/tyxt* format is more involved. We need an auxiliary label *empty*, which denotes that an expression consists of the empty string. We also need more rules:

$$x \xrightarrow{a} x\star a \qquad \text{for } a\in\{a,b,..,y,z\}$$

$$a \xrightarrow{\Delta} \lambda \qquad \text{for } a\in\{a,b,..,y,z\}$$

$$\lambda \xrightarrow{empty} \lambda$$

$$\frac{x \xrightarrow{\Delta} x'}{y\star x \xrightarrow{\Delta} y\star x'}$$

$$\frac{x \xrightarrow{e} x' \quad y \xrightarrow{empty} y'}{x\star y \xrightarrow{e} x'} \qquad \text{for } e\in\{empty,\Delta\}$$

We come back to this example in Section 5.11.2.

5.4. Well-foundedness. A TSS with the rule:

$$\frac{f(x,y_2)\xrightarrow{a} y_1 \quad g(x',y_1)\xrightarrow{b} y_2}{x \xrightarrow{c} x'}$$

can be in *tyft/tyxt* format. However, we have a circular reference. In general y_1 will depend on $f(x,y_2)$ and thus on y_2 while y_2 depends on $g(x',y_1)$ and thus on y_1. We will exclude this type of dependencies, as they give rise to complicated TSS's. For this purpose the notion of a *dependency graph* is introduced.

5.4.1. DEFINITION. Let $P=(\Sigma,A,R)$ be a TSS. Let $S=\{t_i \xrightarrow{a_i} t_i' \mid i\in I\}$ be a set of transitions of P. The *dependency graph* of S is a directed (unlabelled) graph with:

- Nodes: $\bigcup_{i\in I} Var(t_i \xrightarrow{a_i} t_i')$,
- Edges: $\{<x,y> \mid x\in Var(t_i),\ y\in Var(t_i') \text{ for some } i\in I\}$.

A set of transitions is called *well-founded* if any backward chain of edges in the

dependency graph of these transitions is finite. A rule is called *well-founded* if the set of its premises is so. Finally, a TSS is called *well-founded* if all its rules are well-founded.

5.4.2. EXAMPLE. The dependency graph of the set of premises of the rule in Section 5.4 is given in Figure 7. The rule is not well-founded since the graph clearly contains a cycle.

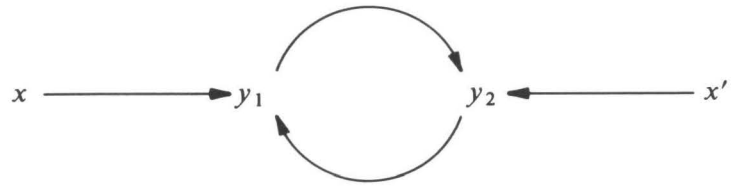

FIGURE 7

5.5. DEFINITION. Two TSS's P and P' are *transition equivalent* if $TS(P) = TS(P')$.

Hence, two TSS's are transition equivalent if they have the same signature, the same set of labels and if the sets of rules determine the same transition relation. The particular form of the rules is not important. In Example 3.4 for instance, we can replace rule 6 of Table 1 by the rule:

$$\frac{x \xrightarrow{\surd} \delta \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$$

The resulting TSS $P'(\mathrm{BPA}_\delta^\varepsilon)$ is transition equivalent to $P(\mathrm{BPA}_\delta^\varepsilon)$. This is because whenever $P(\mathrm{BPA}_\delta^\varepsilon)$ proves a transition of the form $t \xrightarrow{\surd} t'$, t' will be syntactically equal to δ. Observe that $P'(\mathrm{BPA}_\delta^\varepsilon)$ is not in *tyft/tyxt* format. We will come back to this in Section 5.13.

When dealing with closed terms, only the *tyft* format is necessary and the *tyxt* format is not needed. This is what the following lemma says.

5.6. LEMMA. *Let $P=(\Sigma,A,R)$ be a (well-founded) TSS in tyft/tyxt format. Then there is a transition equivalent (well-founded) TSS $P'=(\Sigma,A,R')$ in tyft format.*

PROOF. Let $\Sigma=(F,rank)$. Define R' by:

- every *tyft* rule of R is in R',
- for every *tyxt* rule $r \in R$ and for every function symbol $f \in F$, r_f is in R', where r_f is obtained by substituting $f(x_1,..,x_{rank(f)})$ for x in r with $\{x_1,..,x_{rank(f)}\} \subseteq V - Var(r)$.

If the old *tyxt* rules were well-founded, then the new rules will be well-founded too and in *tyft* format. Suppose that $t \xrightarrow{a} t'$ is a transition in $TS(P)$. Then, by definition of $TS(P)$ and Lemma 3.3, there is a closed proof from P of this transition. Now one can easily see that this is also a proof for $t \xrightarrow{a} t'$ from P'.

A similar argument gives that every transition of $TS(P')$ is also a transition of $TS(P)$. □

5.7. DEFINITION. Let $P=(\Sigma,A,R)$ be a TSS and let r be a rule in R. A variable in $Var(r)$ is called *free* if it does not occur in the left hand side of the conclusion or in the right hand side of a premise.

5.8. DEFINITION. Let $P=(\Sigma,A,R)$ be a TSS. A rule $r\in R$ is called *pure* if it is well-founded and contains no free variables. The TSS P is *pure* if all its rules are pure.

5.9. LEMMA. *Let $P=(\Sigma,A,R)$ be a well-founded TSS in tyft/tyxt format. Then there is a transition equivalent pure TSS $P'=(\Sigma,A,R')$ in tyft format.*
PROOF. By the previous lemma we can assume that P is in *tyft* format. Replace every rule with free variables by a set of new rules. The new rules are obtained by applying every possible substitution of closed terms for the free variables in the old rule. If the old rules were well-founded and in *tyft* format then the new rules will be pure and in *tyft* format. Now, every closed proof T for a transition $t_1 \xrightarrow{a} t_2$ from P is also a proof for $t_1 \xrightarrow{a} t_2$ from P' and vice versa. □

We now come to the first main theorem of this paper. It says that strong bisimulation is a congruence for all operators defined using a well-founded TSS in *tyft/tyxt* format.

5.10. THEOREM. *Let $\Sigma=(F,r)$ be a signature and let $P=(\Sigma,A,R)$ be a TSS. If P is well-founded and in tyft/tyxt format then strong bisimulation is a congruence for all function symbols, i.e. for all function symbols f in F and all closed terms $u_i,v_i\in T(\Sigma)$ $(1\leqslant i\leqslant r(f))$:*

$$\forall i\ u_i \leftrightarrows_P v_i \quad \Rightarrow \quad f(u_1,..,u_{r(f)}) \leftrightarrows_P f(v_1,..,v_{r(f)}).$$

Before we commence with the proof of this theorem, we present a number of examples which show that the condition in the theorem that the TSS is in *tyft/tyxt* format cannot be weakened in any obvious way. At present, we have no example to show that the condition that the TSS is well-founded cannot be missed: we just have not been able to prove the theorem without it. However, non-well-founded TSS's are quite pathological and we know of no application. In Section 7 it will be shown that non-well-founded rules are ill-behaved with respect to modularisation.

5.11. COUNTEREXAMPLES.

5.11.1. EXAMPLE. The first example shows that in general the variables in the source of
the conclusion must all be different. The crucial part of the example is a rule that one could call a *syntactical tester*. In case of the alternative composition, it

tests whether the left and right argument of the $+$ are syntactically identical. The TSS which we have in mind, is obtained by adding to $P(\mathrm{BPA}_\delta^\epsilon)$ the axiom: $x+x \xrightarrow{ok} \delta$. We then have $a \leftrightarrows a\epsilon$, but $a+a \not\leftrightarrows a+a\epsilon$ as a and $a\epsilon$ are not syntactically equal.

5.11.2. EXAMPLE. In general, not more than one function symbol may occur in the source of the conclusion. Take the TSS $P(\mathrm{BPA}_\delta^\epsilon)$ extended with the axiom $ab \xrightarrow{ok} \delta$. As in Example 4.4(b) $b \leftrightarrows b+b$, but in the new situation we do not have any more that $ab \leftrightarrows a(b+b)$ as $a(b+b)$ cannot do an initial ok-transition. Another example illustrating this point is obtained by adding the axiom $x+(y+z) \xrightarrow{ok} \delta$ to $P(\mathrm{BPA}_\delta^\epsilon)$. Again we have $b \leftrightarrows b+b$, but now it is not the case that $b+(b+b) \leftrightarrows b+b$.

As a last example of this kind we mention the typewriter of Section 5.3. The first specification is not in *tyft/tyxt* format, because it contains the axiom $x*a \xrightarrow{\Delta} x$ with $*$ and a function symbols. Now $\lambda*a \leftrightarrows a$ but $a*(\lambda*a) \not\leftrightarrows a*a$. Bisimulation *is* a congruence for the *tyft/tyxt* version of the typewriter. The reader may also check that the identities $\lambda*t \leftrightarrows t*\lambda \leftrightarrows t$ and $(s*t)*u \leftrightarrows s*(t*u)$ with s,t,u closed terms over the signature, hold for the second version of the typewriter but not for the first version.

5.11.3. EXAMPLE. Our next example shows that in the right hand side of a premise, function symbols are not allowed to occur. We can add *prefixing* operators $a{:}(\cdot)$ to $P(\mathrm{BPA}_\delta^\epsilon)$ for each $a \in Act$ and define the operational meaning of these operators with rules:

$$a{:}x \xrightarrow{a} x.$$

If we now add moreover the rule:

$$\frac{x \xrightarrow{a} \delta}{a{:}x \xrightarrow{ok} \delta}$$

we have problems because $a{:}a{:}\delta \not\leftrightarrows a{:}a{:}(\delta+\delta)$ even though $\delta \leftrightarrows \delta+\delta$.

5.11.4. EXAMPLE. The variables in the right hand sides of the arrows in the premises must in general be different. This is shown by adding the rule:

$$\frac{x \xrightarrow{a} y \quad x' \xrightarrow{a} y}{x \cdot x' \xrightarrow{ok} \delta} \quad a \neq \surd$$

to $P(\mathrm{BPA}_\delta^\epsilon)$. Now $a \leftrightarrows a\epsilon$, but $aa \not\leftrightarrows (a\epsilon)a$.

5.11.5. EXAMPLE. If variables in the left hand side of the conclusion and the right hand side of the premises coincide, problems can arise too. Add the rule:

$$\frac{x \xrightarrow{a} y}{x+y \xrightarrow{ok} \delta}$$

to $P(\mathrm{BPA}_\delta^\epsilon)$ and observe that $\epsilon\epsilon \leftrightarrows \epsilon$, but $a+\epsilon\epsilon \not\leftrightarrows a+\epsilon$.

5.12. We now will prove Theorem 5.10.

PROOF. Let $\Sigma=(F,r)$ be a signature and let $P=(\Sigma,A,R_0)$ be a well-founded TSS in *tyft/tyxt* format. We have to prove that $\leftrightarrows_P$ is a congruence. Let $R\subseteq T(\Sigma)\times T(\Sigma)$ be the least relation satisfying:

- $\leftrightarrows_P \subseteq R$,
- for all function symbols f in F and terms u_i,v_i $(1\leqslant i\leqslant r(f))$ in $T(\Sigma)$:

$$\forall i\ u_i R v_i \quad \Rightarrow \quad f(u_1,..,u_{r(f)})\,R\,f(v_1,..,v_{r(f)}).$$

It is enough to show $R\subseteq\leftrightarrows_P$ because then $R=\leftrightarrows_P$ and it follows from the definition of R that $\leftrightarrows_P$ is a congruence for all f in F. In order to prove $R\subseteq\leftrightarrows_P$ it is enough to show that R is a bisimulation. For reasons of symmetry it is even enough to show only one half of the transfer property: if $u\,R\,v$ and $u\xrightarrow{a}_P u'$ then there is a v' such that $v\xrightarrow{a}_P v'$ and $u'R\,v'$. If $u\,R\,v$ then by definition of R either $u\leftrightarrows_p v$ or, for some function symbol f in F: $u\equiv f(u_1,..,u_{r(f)})$ and $v\equiv f(v_1,..,v_{r(f)})$ with $u_i\,R\,v_i$ for all i. As $\leftrightarrows_p$ trivially satisfies the transfer property, only the second option needs to be checked. Summarizing, we have to prove the following statement:

Whenever $P\vdash f(u_1,..,u_{r(f)})\xrightarrow{a}u'$ *and* $u_i\,R\,v_i$ *for* $1\leqslant i\leqslant r(f)$ *then there is a* v' *such that* $P\vdash f(v_1,..,v_{r(f)})\xrightarrow{a}v'$ *and* $u'R\,v'$.

Lemma 3.3 says that there is a proof T of $f(u_1,..,u_{r(f)})\xrightarrow{a}u'$ that only contains closed transitions. We will prove the statement with ordinal induction on the structure of T. Lemma 5.9 allows us to assume throughout the proof that the rules in R_0 are pure and in *tyft* format.

Let r be the last rule used in proof T, in combination with a substitution σ. Assume that r is equal to:

$$\frac{\{t_i\xrightarrow{a_i}y_i\mid i\in I\}}{f'(x_1,..,x_{r(f')})\xrightarrow{a}t}$$

It follows that: 1) $f'\equiv f$
2) $\sigma(x_i)=u_i$ for $1\leqslant i\leqslant r(f)$
3) $\sigma(t)=u'$

Our aim is to use the rule r again in the proof of $f(v_1,..,v_{r(f)})\xrightarrow{a}v'$ for some v' by finding a proper substitution σ'. Consider the dependency graph G of the premises of r. Because r is *tyft*, each node in G has at most finitely many incoming edges. Because G is well-founded we can define for each node x of G, $depth(x)\in\mathbb{N}$ as the length of the maximal backward chain of edges (use König's lemma). Define

$$\begin{aligned} X &= \{x_i\mid 1\leqslant i\leqslant r(f)\}\\ Y &= \{y_i\mid i\in I\}\\ Y_n &= \{y\in Y\mid depth(y)=n\} \quad \text{for } n\geqslant 0 \end{aligned}$$

Observe that for any variable $x \in X$: $depth(x)=0$, and that the sets Y_n form a partition of Y. We will define a substitution σ' that satisfies the following properties:

$$\sigma'(x_i)=v_i \qquad \text{for } 1 \leqslant i \leqslant r(f) \tag{1}$$

$$\sigma(y)\, R\, \sigma'(y) \qquad \text{for } y \in X \cup Y \tag{2}$$

$$P \vdash \sigma'(t_i \xrightarrow{a_i} y_i) \qquad \text{for } i \in I \tag{3}$$

Substitution σ' will be constructed in a stepwise fashion. To begin with we define:

$$\sigma'(x_i) = v_i \qquad \text{for } 1 \leqslant i \leqslant r(f)$$

$$\sigma'(y) = \sigma(y) \qquad \text{for } y \in V - (X \cup \bigcup_{n>0} Y_n)$$

We still have to define σ' on $\bigcup_{n>0} Y_n$. As soon as σ' has been defined for all variables in $X \cup Y_0 \cup \cdots \cup Y_m$ $(m \geqslant 0)$, we can state the following properties $\alpha(m)$ and $\beta(m)$:

$$\alpha(m): \quad \sigma(y)\, R\, \sigma'(y) \quad \text{for } y \in X \cup Y_0 \cup \cdots \cup Y_m$$

$$\beta(m): \quad P \vdash \sigma'(t_i \xrightarrow{a_i} y_i) \quad \text{for } y_i \in Y_0 \cup \cdots \cup Y_m$$

One can easily check that $\alpha(0)$ and $\beta(0)$. Let $n>0$. Suppose that σ' has been defined already for all variables in $X \cup Y_0 \cup \cdots \cup Y_{n-1}$ in such a way that properties $\alpha(n-1)$ and $\beta(n-1)$ hold. We show how to define σ' on all variables of Y_n such that $\alpha(n)$ and $\beta(n)$ hold. This is sufficient for completing the definition of a σ' that satifies properties 1-3: property 1 is met by definition, property 2 and 3 follow because σ' satisfies properties $\alpha(n)$ resp. $\beta(n)$ for all $n \in \mathbf{N}$.

Pick an element $y^* \in Y_n$. There is a unique $i \in I$ with $y^* = y_i$. Because $y_i \in Y_n$ and rule r is pure, $Var(t_i) \subseteq X \cup Y_0 \cup \cdots \cup Y_{n-1}$. Now use that σ' satisfies $\alpha(n-1)$ to obtain that for all variables $y \in Var(t_i)$: $\sigma(y)\, R\, \sigma'(y)$. Next we use the following

FACT. *Let $t \in \mathbb{T}(\Sigma)$ and let $\rho, \rho' : V \to T(\Sigma)$ be substitutions such that for all x in $Var(t)$: $\rho(x)\, R\, \rho'(x)$. Then $\rho(t)\, R\, \rho'(t)$.*

PROOF. Straightforward induction on the structure of term t using the definition of R. □

We obtain that $\sigma(t_i)\, R\, \sigma'(t_i)$. Since also $P \vdash \sigma(t_i) \xrightarrow{a_i} \sigma(y_i)$, we can distinguish, by definition of R, between two cases:

1) $\sigma(t_i) \leftrightarrows_P \sigma'(t_i)$. In this case we can find a $w \in T(\Sigma)$ such that $P \vdash \sigma'(t_i) \xrightarrow{a_i} w$ and $\sigma(y_i)\, R\, w$. We then define $\sigma'(y^*) = \sigma'(y_i) = w$.
2) There is a function symbol g in F and there are terms w_j, w_j' for $1 \leqslant j \leqslant r(g)$ such that:

$$\sigma(t_i) = g(w_1, .., w_{r(g)}),$$

$\sigma'(t_i)=g(w_1',..,w_{r(g)}')$ and

$w_j R w_j'$ for $1 \leqslant j \leqslant r(g)$.

But now we can apply the induction hypothesis which gives that we can find a w such that $P \vdash g(w_1',..,w_{r(g)}') \xrightarrow{a} w$ and $\sigma(y_i) R w$. We define $\sigma'(y^*)=\sigma'(y_i)=w$.

In the same way we can define σ' for the other elements of Y_n. It is not hard to see that after this $\alpha(n)$ and $\beta(n)$ hold.

Let for $i \in I$, T_i be a proof of $\sigma'(t_i \xrightarrow{a} y_i)$. Construct a proof T' with root $\sigma'(f(x_1,..,x_{r(f)}) \xrightarrow{a} t)$ and as direct subtrees the proofs T_i $(i \in I)$. Define $v'=\sigma'(t)$. Clearly T' is a proof for $f(v_1,..,v_{r(f)}) \xrightarrow{a} v'$. Since for all $x \in Var(t)$: $\sigma(x) R \sigma'(x)$ (use that r is pure), it follows by an application of the previously stated fact that $\sigma(t) R \sigma'(t)$ or, equivalently, $u' R v'$. □

5.13. The implication in Theorem 5.10 cannot be reversed. So given a TSS for which bisimulation is a congruence, this TSS need not be well-founded and in *tyft/tyxt* format. This is obvious because for any TSS, a transition equivalent TSS can be obtained by adding all derivable transitions as rules. And if bisimulation is a congruence for the one it is a congruence for the other. If one starts from a well-founded TSS in *tyft/tyxt* format, the result will in general not be *tyft/tyxt*. For instance, in the case of $P(\mathrm{BPA}_\delta^\epsilon)$ one adds the rule $a \cdot (x+y) \xrightarrow{a} \epsilon \cdot (x+y)$.

Even after removing derivable rules, a TSS for which bisimulation is a congruence need not be well-founded and in *tyft/tyxt* format. The TSS $P'(\mathrm{BPA}_\delta^\epsilon)$ described in Section 5.5 contains no derivable rules and is not in *tyft/tyxt* format. But, as observed in that section, it is transition equivalent to the TSS $P(\mathrm{BPA}_\delta^\epsilon)$ which *is* in *tyft/tyxt* format. Hence, bisimulation equivalence is a congruence.

It is worth noting that if one adds new operators and rules to $P'(\mathrm{BPA}_\delta^\epsilon)$, the congruence property can get lost, even if the rules for the new operators are *tyft*. In order to see this, consider the TSS obtained by adding to $P'(\mathrm{BPA}_\delta^\epsilon)$ *encapsulation* or *restriction* operators ∂_H for $H \subseteq Act$ and the *tyft* rules:

$$\frac{x \xrightarrow{a} x'}{\partial_H(x) \xrightarrow{a} \partial_H(x')} \quad a \notin H$$

We then obtain $a \leftrightarrows \partial_{\{b\}}(a)$, but $a \cdot b \not\leftrightarrows \partial_{\{b\}}(a) \cdot b$.

The examples above do not rule out the following weakened variant of the reverse implication of Theorem 5.10: if P is a TSS for which bisimulation is a congruence, then $TS(P)$ can be specified by a well-founded TSS in *tyft/tyxt* format. Below we present a TSS that eliminates this variant of the reverse implication. Consider the TSS P that has constant symbols a,b and δ, a binary function symbol f, labels a,b,c and rules:

$$a \xrightarrow{a} \delta$$

$$b \xrightarrow{a} \delta$$

$$b \xrightarrow{b} \delta$$

$$f(a) \xrightarrow{c} \delta$$

The last rule is not *tyft/tyxt*, but it is not hard to see that $\leftrightarrows_P$ is a congruence. We claim that there exists no TSS in *tyft/tyxt* format that is transition equivalent to P. In order to prove this claim, it is, by Lemma 5.6 and the proof of Lemma 5.9, sufficient to show that no TSS P' in *tyft* format and without free variables in the rules can be transition equivalent to P. Suppose there would be such a P'. Since $P' \vdash f(a) \xrightarrow{c} \delta$, there is a closed proof T of $f(a) \xrightarrow{c} \delta$ such that only the root of T is labelled with $f(a) \xrightarrow{c} \delta$. The other nodes in T are labelled with either $a \xrightarrow{a} \delta$, $b \xrightarrow{a} \delta$ or $b \xrightarrow{b} \delta$. Let r be the last rule used in T, in combination with a substitution σ. Rule r must be of the form:

$$\frac{\{t_i \xrightarrow{a_i} t_i' \mid i \in I\}}{f(x) \xrightarrow{c} t}$$

It is not hard to see that for $i \in I$, t_i must be equal to x, a or b. Clearly $\sigma(x) = a$. Let σ' be the same as σ except that $\sigma'(x) = b$. Let $i \in I$. Then $\sigma'(t_i \xrightarrow{a_i} t_i')$ is either $a \xrightarrow{a} \delta$, $b \xrightarrow{a} \delta$ or $b \xrightarrow{b} \delta$. Moreover $\sigma'(t) = \delta$. Thus we can construct a proof from P' of transition $f(b) \xrightarrow{c} \delta$ by taking r as a last rule with substitution σ' and appending proofs of $a \xrightarrow{a} \delta$, $b \xrightarrow{a} \delta$ and $b \xrightarrow{b} \delta$ on top of that at the appropriate places. Contradiction.

Also in this case we have that adding *tyft* rules may destroy the congruence property (take the axiom $a \xrightarrow{b} \delta$).

5.14. REMARK. The examples of Section 5.13 show that there is another reason for using TSS's in *tyft/tyxt* format, namely their extensibility, without endangering congruence properties. It seems that, whenever a TSS contains a non *tyft/tyxt* rule, we can extend this TSS (except for some trivial cases, for instance if the non *tyft/tyxt* rules are derivable) with a number of *tyft* rules in such a way that for the resulting TSS bisimulation is not a congruence.

6. SOME APPLICATIONS

In this section we give some examples of TSS's and applications of the congruence theorem.

6.1. The silent move. In process algebra it is current practice to have a constant 'τ' representing an internal machine step that cannot be observed. In order to describe the 'invisible' nature of τ, the notions of *observation congruence* (MILNER, 1980) and *rooted-τ-bisimulation* (BERGSTRA & KLOP, 1988) have been introduced. As observed by VAN GLABBEEK (1987) it is not necessary to introduce a new notion of bisimulation: one can just work with the standard notion of strong bisimulation if one is willing to add some Plotkin style rules that capture the notion of a hidden, internal machine step.
Below we assume that τ is an element of the set Act of actions that figures as a parameter of the TSS $P(BPA_\delta^\S)$. The TSS $P(BPA_{\epsilon\delta}^\tau)$ is obtained by adding to

$P(BPA^{\epsilon}_{\delta})$ the rules of Table 2 ($a \in Act_{\surd}$).

7.	$a \xrightarrow{a} \tau$	$a \neq \surd$
8.	$\dfrac{x \xrightarrow{a} y \quad y \xrightarrow{\tau} z}{x \xrightarrow{a} z}$	
9.	$\dfrac{x \xrightarrow{\tau} y \quad y \xrightarrow{a} z}{x \xrightarrow{a} z}$	

TABLE 2. Rules for the silent move τ

One possible interpretation that one can give to a transition $t \xrightarrow{a} t'$ ($a \neq \tau$) is that the system that is modelled can evolve from state t to state t' during a certain positive time interval in which an occurrence of action a can be observed. Then $t \xrightarrow{\tau} t'$ means that no action can be observed during such an interval. Rule 8 and 9 can be viewed as logical consequences of this interpretation. It is consistent with the interpretation of transitions and the rules of Table 1 and Table 2 to assume that execution of a *process* a takes a positive amount of time; the observation of the *action* a however takes place at the beginning. Rule 7 says that when the action a is observed, the process a that executes this action may still perform some internal activity before it terminates successfully.

The TSS $P(\mathrm{BPA}^{\tau}_{\epsilon\delta})$ is in pure *tyft/tyxt* format. Thus strong bisimulation is a congruence. One can prove that the theory $\mathrm{BPA}^{\tau}_{\epsilon\delta}$, as presented in Table 3 (a ranges over Act), is a sound and complete axiomatisation of the model induced by the TSS $P(\mathrm{BPA}^{\tau}_{\epsilon\delta})$ modulo strong (!) bisimulation.

$x+y = y+x$	A1	$a\tau = a$	T1
$x+(y+z) = (x+y)+z$	A2	$\tau x + x = \tau x$	T2
$x+x = x$	A3	$a(\tau x+y) = a(\tau x+y)+ax$	T3
$(x+y)z = xz+yz$	A4		
$(xy)z = x(yz)$	A5		
$x+\delta = x$	A6		
$\delta x = \delta$	A7		
$\epsilon x = x$	A8		
$x\epsilon = x$	A9		

TABLE 3. The axiom system $\mathrm{BPA}^{\tau}_{\epsilon\delta}$

This means that, if $\leftrightarrows_{r\tau\delta}$ denotes rooted-τ-bisimulation (i.e. observation congruence), we have the following situation:

$$P(\mathrm{BPA}^{\tau}_{\epsilon\delta}) \vdash s \leftrightarrows t \;\Leftrightarrow\; P(BPA^{\epsilon}_{\delta}) \vdash s \leftrightarrows_{r\tau\delta} t \;\Leftrightarrow\; \mathrm{BPA}^{\tau}_{\epsilon\delta} \vdash s = t.$$

In Figure 8-10 we give three examples corresponding to the τ-laws of MILNER (1980). In Figure 8 two separate transition systems are drawn. In Figures 8

and 10 a may not equal τ. In Figure 9 the relevant states of $\tau+\epsilon$ and τ are drawn, as the equation $\tau+\epsilon=\tau$ is equivalent to the axiom T2. It is left to the reader to check that the transition systems are strongly bisimilar.

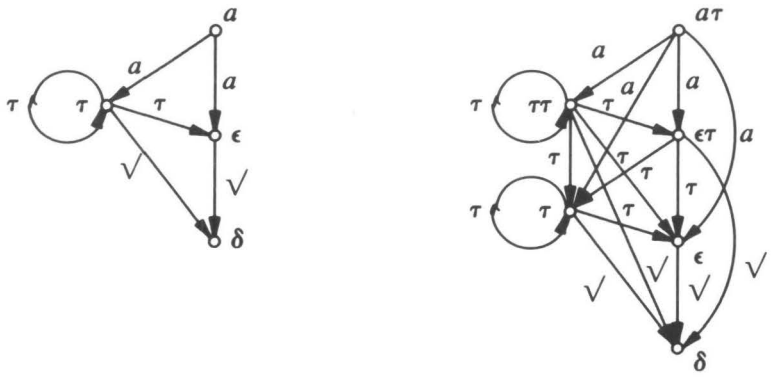

FIGURE 8 ($a = a\tau$)

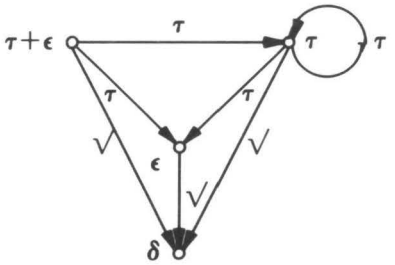

FIGURE 9 ($\tau+\epsilon = \tau$)

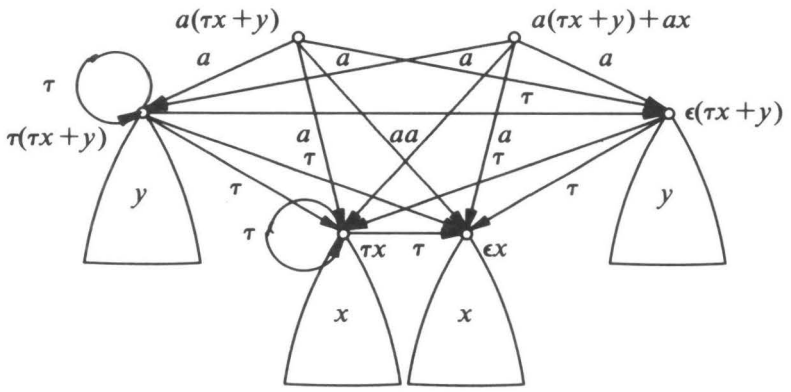

FIGURE 10 ($a(\tau x+y) = a(\tau x+y)+ax$)

6.2. Recursion. There are many ways to deal with recursion in process algebra. One approach is to introduce a set Ξ of *process names*. Elements of Ξ are added to the signature of the TSS as constant symbols. The recursive definitions of the process names are given by a set $E=\{X \Leftarrow t_X \mid X \in \Xi\}$ of *declarations*. Here the t_X are ground terms over the signature of the TSS (hence, they may contain process names in Ξ). If $X \Leftarrow t_X$ is a declaration, then this means that the behaviour of process X is given by its *body* t_X. Formally this is expressed by adding to the TSS rules:

$$\frac{t_X \xrightarrow{a} y}{X \xrightarrow{a} y}$$

for every declaration $X \Leftarrow t_X$. Now observe that these rules are pure *tyft*. Hence it follows that if one adds recursion to a well-founded TSS in *tyft/tyxt* format in the way described above, bisimulation remains a congruence.

A slightly different way of dealing with recursion is followed by OLDEROG & HOARE (1986) and HENNESSY (1988). Here axioms $X \xrightarrow{\tau} t_X$ appear saying that by some internal activity, a process name can expand to its body. Also this type of rules satisfy our format.

6.3. The state operator. In many cases where operational semantics of a language is defined using Plotkin style rules, values play a role (see for instance (AMERICA ET AL., 1986) and (PLOTKIN, 1983)). Here, states of the transition system are generally configurations, i.e. pairs $<t,\sigma>$ of a process expression t and a valuation σ. In this section we argue that it is often possible to give inductive rules for these languages within the *tyft/tyxt* format using the *extended state operator* Λ_σ of BAETEN & BERGSTRA (1988).

We will add the state operator to the setting of $\mathrm{BPA}^{\tau}_{\varepsilon\delta}$ of Section 6.1. Let S be a set of *states*. For each $\sigma \in S$ we add a function symbol Λ_σ to the signature. An expression $\Lambda_\sigma(t)$, represents a process that transforms the state σ during successive transitions of t as specified by a function $\mathit{effect}: S \times Act \times Act \to S$ while influencing the actual labels of the transitions of t as specified by a function $\mathit{action}: Act \times S \to 2^{Act}$. $\mathit{action}(a,\sigma)$ defines the set of actions that can be performed by $\Lambda_\sigma(t)$ if t performs an a. $\mathit{effect}(\sigma,a,b)$ defines the resulting state if $\Lambda_\sigma(t)$ actually transforms under $b \in \mathit{action}(a,\sigma)$. Note that the extra argument b is necessary as the action function defines a set of possible actions that can be performed by $\Lambda_\sigma(t)$. The environment may determine which action from this set actually will occur. The functions *effect* and *action* are *inert* for τ, i.e. $\mathit{action}(\tau,\sigma)=\{\tau\}$ and $\mathit{effect}(\sigma,\tau,a)=\sigma$ for every $a \in Act$. The rules for the state operator are ($\sigma \in S$; $a,b \in Act$):

$$\frac{x \xrightarrow{a} x'}{\Lambda_\sigma(x) \xrightarrow{b} \Lambda_{\mathit{effect}(\sigma,a,b)}(x')} \quad b \in \mathit{action}(a,\sigma)$$

$$\frac{x \xrightarrow{\surd} x'}{\Lambda_\sigma(x) \xrightarrow{\surd} \Lambda_\sigma(x')}$$

Clearly the above rules are pure *tyft*, so bisimulation will be a congruence. As

a typical application we consider a small subset of CSP. Actions in Act are of the form τ, $g!e$, $g?v$ or $[v:=e]$ where v ranges over a set $\mathcal{V}$ of program variables and e ranges over natural number expressions built from $\mathcal{V}$, constants for the natural numbers and the usual operations such as $+$, $-$, $\times$. $g!e$ means 'write the value of expression e to channel g', $g?v$ means 'read a value from channel g and assign this value to v' and $[v:=e]$ means: 'assign the value of expression e to v'. We assume the presence of an interpretation function $[\![\cdot]\!]$ that, given a valuation σ of the variables, yields for each expression a natural number. As state space S we take all valuations in $\mathcal{V}\to\mathbb{N}$. Let $\sigma[n/v]$ be the valuation σ except for the fact that variable v is mapped on n. Now we can define the functions *action* and *effect* as follows:

$$\begin{array}{ll} action(\sigma, g!e) = \{g![\![e]\!]^\sigma\} & effect(\sigma, g!e, g!n) = \sigma \\ action(\sigma, g?v) = \{g?n \mid n\in\mathbb{N}\} & effect(\sigma, g?v, g?n) = \sigma[n/v] \\ action(\sigma, [v:=e]) = \{\tau\} & effect(\sigma, [v:=e], \tau) = \sigma[[\![e]\!]^\sigma/v] \end{array}$$

Function *effect* is inert in the cases that are not specified. As an example consider a process that is capable of reading a value from channel g_1 and sending the square of that value to channel g_2:

$$\Lambda_\sigma(g_1?v\cdot[w:=v\times v]\cdot g_2!w)$$

A particular sequence of transitions of this process is:

$$\Lambda_\sigma(g_1?v\cdot[w:=v\times v]\cdot g_2!w) \xrightarrow{g_1?3} \Lambda_{\sigma[3/v]}(\epsilon\cdot[w:=v\times v]\cdot g_2!w) \xrightarrow{\tau}$$
$$\Lambda_{\sigma[3/v,9/w]}(\epsilon\cdot g_2!w) \xrightarrow{g_2!9} \Lambda_{\sigma[3/v,9/w]}(\epsilon) \xrightarrow{\surd} \Lambda_{\sigma[3/v,9/w]}(\delta)$$

It is not difficult to extend the combination of $BPA^\tau_{\epsilon\delta}$ and the state operator with a parallel combinator. Then, communication can be defined such that we have value passing between several processes. We will not give a detailed elaboration of this because that would go beyond the scope of this article. However, we would like to stress that in some sense the extended state operator is more powerful than the approach with a global state using configurations. The extended state operator can in a very natural way be used to model that certain data are local to some processes.

7. Modular properties of transition system specifications

Often one wants to add new operators and rules to a given TSS. Therefore, a very natural operation on TSS's is to take their componentwise union. Given two TSS's P_0 and P_1 we use the notation $P_0\oplus P_1$ to denote the resulting system. A nice property to have in such a situation is that the outgoing transitions in $TS(P_0)$ of terms in the signature of P_0 are the same as the outgoing transitions of these terms in $TS(P_0\oplus P_1)$. This means that $P_0\oplus P_1$ is a *conservative extension* of P_0: any property which has been proved for the states in the old transition system remains valid (for the old states) in the enriched system.

In this section we study the question what restrictions we have to impose on

P_0 and P_1 in order to obtain conservativity. First we give the basic definitions.

7.1. DEFINITION. Let $\Sigma_i=(F_i,r_i)$ $(i=0,1)$ be two signatures such that $f\in F_0\cap F_1 \Rightarrow r_0(f)=r_1(f)$. The *sum* of Σ_0 and Σ_1, notation $\Sigma_0\oplus\Sigma_1$, is the signature:

$$\Sigma_0\oplus\Sigma_1 = (F_0\cup F_1,\lambda f.\textit{if } f\in F_0 \textit{ then } r_0(f) \textit{ else } r_1(f)).$$

7.2. DEFINITION. Let $P_i=(\Sigma_i,A_i,R_i)$ $(i=0,1)$ be two TSS's with $\Sigma_0\oplus\Sigma_1$ defined. The *sum* of P_0 and P_1, notation $P_0\oplus P_1$, is the TSS:

$$P_0\oplus P_1 = (\Sigma_0\oplus\Sigma_1,A_0\cup A_1,R_0\cup R_1).$$

7.3. DEFINITION. Let $P_i=(\Sigma_i,A_i,R_i)$ $(i=0,1)$ be two TSS's with $P=P_0\oplus P_1$ defined. Let $P=(\Sigma,A,R)$. We say that P is a *conservative extension* of P_0 and that P_1 *can be added conservatively* to P_0 if for all $s\in T(\Sigma_0)$, $a\in A$ and $t\in T(\Sigma)$:

$$P\vdash s\xrightarrow{a}t \;\Leftrightarrow\; P_0\vdash s\xrightarrow{a}t.$$

Note that the implication $P\vdash s\xrightarrow{a}t \Leftarrow P_0\vdash s\xrightarrow{a}t$ holds trivially.

7.4. REMARK. Let $P_i=(\Sigma_i,A_i,R_i)$ $(i=0,1)$ be two TSS's with $P=P_0\oplus P_1$ a conservative extension of P_0. Then P is also a conservative extension of P_0 up to bisimulation, i.e. for $s,t\in T(\Sigma_0)$:

$$s\leftrightarroweq_P t \;\Leftrightarrow\; s\leftrightarroweq_{P_0} t.$$

7.5. COUNTEREXAMPLES. We want to study the question in which cases a TSS P_1 can be added conservatively to a TSS P_0. However, we will restrict ourselves to the case where both P_0 and P_1 are in *tyft/tyxt* format. Below, 5 examples are presented that illustrate different situations where we do not have conservativity.

7.5.1. EXAMPLE. If P_1 has a rule with a function symbol that already occurs in Σ_0 in the lhs of the conclusion, then problems arise quite soon. If $P_0=P(\mathrm{BPA}^{\varepsilon}_{\delta})$ and P_1 contains a single rule:

$$x+y\xrightarrow{ko}\delta$$

then $\delta\leftrightarroweq_{P_0}\delta+\delta$ but not $\delta\leftrightarroweq_{P_0\oplus P_1}\delta+\delta$.

7.5.2. EXAMPLE. Conservativity can get lost if free variables occur in a premise of a rule in P_0. In order to see this consider the TSS P_0 with constant symbols a, b, a label a and rules:

$$a \xrightarrow{a} a$$

$$\frac{x \xrightarrow{a} y}{b \xrightarrow{a} y}$$

It is not hard to see that $a \leftrightarrows b$. However, if we add constant symbols c,d and a rule $c \xrightarrow{a} d$ it follows that $a \not\leftrightarrows b$.

7.5.3. EXAMPLE. Conservativity can get lost also if free variables occur in the conclusion of a rule in P_0. Let the signature of P_0 consists of two constant symbols a and b. The set of labels contains only a and there are two axioms:

$$a \xrightarrow{a} a$$

$$b \xrightarrow{a} x$$

It is not hard to see that $a \leftrightarrows_{P_0} b$. However, if we add a TSS P_1 which contains a constant symbol c and no rules, then $a \not\leftrightarrows_{P_0 \oplus P_1} b$.

7.5.4. EXAMPLE. Conservativity up to bisimulation can be violated if we add *tyxt* rules to a given TSS. Let P_0 consist of $P(\mathrm{BPA}_\delta^\varepsilon)$. In P_0 we have $a \leftrightarrows a+\delta$. This is no longer true if we add a TSS P_1 which contains a single axiom $x \xrightarrow{\surd} x$.
Another example of this kind is given by the rules 8 and 9 in Table 2 of Section 6.1. Consider $P(\mathrm{BPA}_\delta^\varepsilon)$ to which rule 7 has been added. None of the τ-laws holds in this system. However, if rules 8 and 9 are added the τ-laws do hold. Hence, rules 8 and 9 do not preserve conservativity up to bisimulation.

7.5.5. EXAMPLE. Our last example shows that non-well-foundedness of P_0 can disturb conservativity. Suppose P_0 consists of $P(\mathrm{BPA}_\delta^\varepsilon)$ and a circular (non-well-founded) rule:

$$\frac{x_1+y_1 \xrightarrow{ok} y_2 \quad x_2+y_2 \xrightarrow{ok} y_1}{x_1+x_2 \xrightarrow{ko} y_1+y_2}$$

One can easily see that $\delta \leftrightarrows_{P_0} \delta+\delta$. However, adding a TSS P_1 with a single axiom $ok \xrightarrow{ok} ok$ makes that $\delta \not\leftrightarrows_{P_0 \oplus P_1} \delta+\delta$.

The next theorem shows that in some sense the examples above give a complete overview of the situations in which we do not have conservativity.

7.6. THEOREM. *Let $P_0=(\Sigma_0,A_0,R_0)$ be a TSS in pure tyft/tyxt format and let $P_1=(\Sigma_1,A_1,R_1)$ be a TSS in tyft format such that there is no rule in R_1 that contains a function symbol from Σ_0 in the left hand side of its conclusion. Let $P=P_0\oplus P_1$ be defined. Then P_1 can be added conservatively to P_0.*

PROOF. We use the same type of strategy as in the proof of Theorem 5.10. Let $P=(\Sigma,A,R)$. Let $s\in T(\Sigma_0)$, $a\in A$ and $s'\in T(\Sigma)$ with $P\vdash s\xrightarrow{a}s'$. Let T be a proof of $s\xrightarrow{a}s'$ from P. With ordinal induction on the structure of T we prove that T is also a proof of $s\xrightarrow{a}s'$ from P_0. Let r be the last rule which is used in T. Because $s\in T(\Sigma_0)$ and all rules of P_1 are *tyft* and contain no function symbols from Σ_0 in the left hand side of their conclusions, r must be in R_0. Suppose r is pure *tyft* (the case that r is pure *tyxt* is completely analogous and omitted). Suppose in particular that r is equal to:

$$\frac{\{t_i\xrightarrow{a_i}y_i\mid i\in I\}}{f(x_1,..,x_{r(f)})\xrightarrow{a}t}$$

Let σ be the substitution that relates rule r to the last step in proof T. We then have:

$$\sigma(f(x_1,..,x_{r(f)}))=s,$$
$$\sigma(t)=s'.$$

Consider the dependency graph G of the premises of r. Like in the proof of Theorem 5.10 we define for each node x of G, $depth(x)\in\mathbb{N}$ as the length of the maximal backward chain of edges. Further we define:

$$X = \{x_i\mid 1\leqslant i\leqslant r(f)\}$$
$$Y = \{y_i\mid i\in I\}$$
$$Y_n = \{y\in Y\mid depth(y)=n\}\quad \text{for } n\geqslant 0$$

With induction on n we prove that $\sigma(x)$ is in $T(\Sigma_0)$ for all $x\in X\cup Y$. Because $s\in T(\Sigma_0)$ and $\sigma(f(x_1,..,x_{r(f)})=s$, $\sigma(x)\in T(\Sigma_0)$ for all $x\in X$. Let $n\in\mathbb{N}$ and suppose that $\sigma(x)\in T(\Sigma_0)$ for all $x\in X\cup Y_0\cup\cdots\cup Y_{n-1}$. Let $y^*\in Y_n$. There is a unique $i\in I$ with $y^*=y_i$. Because $y_i\in Y_n$ and rule r is pure, $Var(t_i)\subseteq X\cup Y_0\cup\cdots\cup Y_{n-1}$. But now we can apply the induction hypothesis: since $s_i=\sigma(t_i)\in T(\Sigma_0)$, $s_i'=\sigma(y_i)\in T(\Sigma_0)$ too. Since y^* is chosen arbitrarily, $\sigma(y)\in T(\Sigma_0)$ for all $y\in Y_n$. This finishes the induction on n so that we have shown that $\sigma(y)\in T(\Sigma_0)$ for all $x\in X\cup Y$. Since $Var(t)\subseteq X\cup Y$, we may conclude $s'=\sigma(t)\in T(\Sigma_0)$. □

7.7. In our view the counterexamples which show that the original system has to be pure and no rule from the added system may contain a function symbol of the original system in the lhs of its conclusion are quite strong. It will be difficult to strengthen Theorem 7.6 by weakening these constraints. Because modularity is an important and desirable property and because TSS's which are not pure are ill-behaved with respect to modularity, one might decide, for this reason, to call such TSS's unstructured.

The main reason we had for including Theorem 7.6 in this paper is that we need it in the next section. It is clear that a lot more can be said about modular properties of TSS's than we have done here.

8. Trace congruences

In this section we study the trace congruences induced by the pure *tyft/tyxt* format. Intuitively, two processes s and t are (completed) trace congruent if for any context $C[\,]$ which can be defined using the pure *tyft/tyxt* format, the (completed) traces of $C[s]$ and $C[t]$ are the same. It seems reasonable to require that, whenever new function symbols and rules are added to a TSS in order to build a context which can distinguish between terms, these new ingredients may not change the original transition system: the extension should be conservative. If it would be allowed to introduce new transitions in the original transition system, then we could add rules like:

$$\frac{x \xrightarrow{I'm(s)} x',\ y \xrightarrow{I'm(t)} y'}{x+y \xrightarrow{I'm(s+t)} x'+y'}$$

and make that syntactically different terms always have outgoing transitions with different labels. As a result completed trace congruence would just be syntactic equality between terms.

The results of the previous section show that for a TSS in *tyft/tyxt* format it is in general rather difficult to determine a class of TSS's which can be added to it conservatively. Consequently it is also difficult to characterize the completed trace congruence induced by this format. However, for TSS's in pure *tyft/tyxt* format such a class exists: by Theorem 7.6 every TSS in *tyft* format can be added conservatively to a TSS in pure *tyft/tyxt* format. For this reason we decided to work on a characterization of the completed trace congruence induced by the pure *tyft/tyxt* format and leave the general *tyft/tyxt* format for what it is. We think that this is not a serious restriction because:

- We have never seen an application of a TSS with non-well-founded rules or rules with free variables.
- Well-foundedness is used anyhow in the proof of Theorem 5.10. The proof of Lemma 5.9 shows that for every well-founded TSS in *tyft/tyxt* format there exists an equivalent TSS in pure *tyft/tyxt* format.
- TSS's in *tyft/tyxt* format that are not pure, are ill-behaved with respect to modularisation and therefore not much effort should be spent in proving theorems about them.

8.1. Definition. Let $\mathcal{A}=(S,A,\rightarrow)$ be a LTS. A state $s\in S$ is a *termination node*, notation $s\not\rightarrow$, if there are no $t\in S$ and $a\in A$ with $s\xrightarrow{a}t$. A sequence $a_1\star\cdots\star a_n\in A^*$ is a *completed trace* of s if there are states $s_0,..,s_n\in S$ such that $s_0=s$ and $s_0\xrightarrow{a_1}s_1\xrightarrow{a_2}..\xrightarrow{a_n}s_n\not\rightarrow$. $CT(s)$ is the set of all completed traces of s. Two states $s,t\in S$ are *completed trace equivalent* if $CT(s)=CT(t)$. This is denoted as $s\equiv_{CT}t$.

8.2. DEFINITION. Let $\mathcal{F}$ be some format of TSS rules. Let $P=(\Sigma,A,R)$ be a TSS in $\mathcal{F}$ format. Two terms $s,t\in T(\Sigma)$ are *completed trace congruent with respect to* $\mathcal{F}$ rules, notation $s\equiv_{\mathcal{F}} t$, if for every TSS $P'=(\Sigma',A',R')$ in $\mathcal{F}$ format which can be added conservatively to P and for every $\Sigma\oplus\Sigma'$-context $C[\,]$: $C[s]\equiv_{CT} C[t]$. s and t are *completed trace congruent within* P, notation $s\equiv_P t$, if for every Σ-context $C[\,]$: $C[s]\equiv_{CT} C[t]$.

8.3. NOTE. In the sequel we will define a number of equivalence relations on the states of transition systems. If $P=(\Sigma,A,R)$ is a TSS and s,t are terms in $T(\Sigma)$ then, whenever we say that s and t are equivalent according to a certain equivalence relation, what we mean is that the states s and t of the transition system $TS(P)$ are equivalent according to this relation.

8.4. Overview of results of Section 8. ABRAMSKY (1987) and BLOOM, ISTRAIL & MEYER (1988) give Plotkin style rules to define operators with which one can distinguish between any pair of non-bisimilar processes. We cannot obtain this result with pure *tyft/tyxt* rules, but we will show that the notion of completed trace congruence with respect to pure *tyft/tyxt* rules exactly coincides with *2-nested simulation equivalence* for all *image finite* processes. What we in fact will prove is best illustrated by Figure 11.

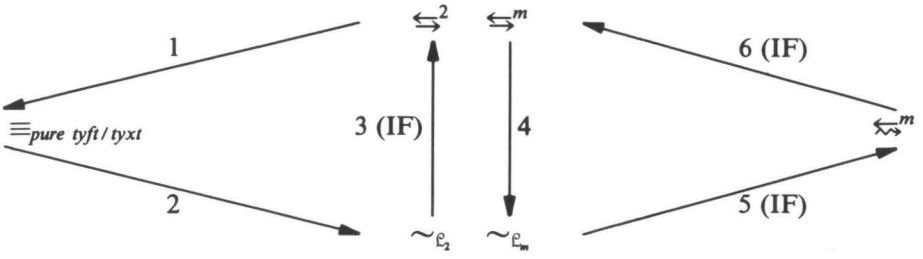

FIGURE 11

The arrows indicate set inclusion. 'IF' stands for Image Finite and indicates that we need image finiteness of processes for the proofs of inclusions 3, 5 and 6. For $m\in\mathbb{N}$, $\leftrightarrows^m$ is m-nested simulation equivalence. $\sim_{\mathcal{L}_m}$ is the equivalence induced by the set $\mathcal{L}_m$ of Hennessy-Milner formulas in which no negation symbol occurs nested m times or more. In the right corner of Figure 11 we have an auxiliary equivalence notion $\leftrightsquigarrow^m$. In Sections 8.5-8.7 these notions are made precise and the inclusions are proved. It immediately follows that both triangles collapse for image finite transitions systems. In particular we will prove the following Theorem 8.4.2.

8.4.1. DEFINITION. An LTS $\mathcal{L}=(S,A,\rightarrow)$ is *image finite* if for all $s\in S$ and $a\in A$ the set $\{t \mid s \xrightarrow{a} t\}$ is finite.

8.4.2. THEOREM. *Let $P=(\Sigma,A,R)$ be a TSS in pure tyft/tyxt format such that $TS(P)$ is image finite. Let $s,t\in T(\Sigma)$. Then:*

$$s \equiv_{\text{pure tyft/tyxt}} t \quad \Leftrightarrow \quad s \leftrightarrows^2 t \quad \Leftrightarrow \quad s \sim_{\wp_2} t.$$

PROOF. Direct from Theorem 8.5.8, Corollary 8.6.7 and Corollary 8.7.6 of this section. □

We are quite sure that, if one uses infinitary Hennessy-Milner logic as in (MILNER, 1989), the restriction of image finiteness in Theorem 8.4.2 can be dropped. Because we wanted to keep the presentation as simple as possible, we preferred to leave this generalization as an exercise to the reader.

In Section 8.8 we show that, using the results that were needed to characterize the completed trace congruence for the pure *tyft/tyxt* format, it is easy to prove that the trace congruence with respect to this format coincides with simulation equivalence for image finite processes.

Bloom, Istrail & Meyer have studied the completed trace congruence induced by *tree rules*. Tree rules differ from pure *tyft/tyxt* rules in that they may only have variables in the premises and there may not be a single variable in the left hand side of a conclusion. Hence, one could also call this type of rules 'pure *xyft* rules'. They proved the following theorem (BLOOM, 1988):

8.4.3. THEOREM (BLOOM, ISTRAIL & MEYER). *Let $P=(\Sigma,A,R)$ be a TSS in tree rule format such that $TS(P)$ is image finite. Let $s,t\in T(\Sigma)$. Then:*

$$s \equiv_{\text{tree rules}} t \quad \Leftrightarrow \quad s \sim_{\wp_2} t.$$

This result, which is close to our characterization theorem, has not been published. A sketch of the proof is included at the end of this section. We were aware of the result of Bloom, Istrail & Meyer before we proved the characterization theorem for the pure *tyft/tyxt* format. However, all proofs in this section are entirely our own.

8.5. Nested simulation equivalences.

8.5.1. DEFINITION. Let $\mathcal{L}=(S,A,\rightarrow)$ be a LTS. A relation $R \subseteq S\times S$ is called a *simulation* if it satisfies:

whenever $s\,R\,t$ and $s \xrightarrow{a} s'$ then, for some $t'\in S$, also $t \xrightarrow{a} t'$ and $s'R\,t'$.

s can be simulated by t, notation $s \subseteq_{\rightarrow} t$, if there is a simulation containing the pair (s,t). s and t are *simulation equivalent*, notation $s \leftrightarrows t$, if $s \subseteq_{\rightarrow} t$ and $t \subseteq_{\rightarrow} s$.

Note the difference between simulation equivalence and bisimulation

equivalence: in the case of a bisimulation equivalence, there should be single relation which is a simulation relation in two directions; in the case of simulation equivalence it is required that there are two simulation relations, one for each direction.

8.5.2. DEFINITION. Let $\mathcal{A}=(S,A,\rightarrow)$ be a LTS and let α be an ordinal number. We define the relation $\subseteq^{\alpha} \subseteq S \times S$ inductively as follows:

$s \subseteq^{\alpha} t$ iff for each $\beta<\alpha$ there is a simulation relation $R \subseteq (\subseteq^{\beta})^{-1}$ with $s\,R\,t$.

Two states s and t are *α-nested simulation equivalent*, notation $s \leftrightarrows^{\alpha} t$, if $s \subseteq^{\alpha} t$ and $t \subseteq^{\alpha} s$.

8.5.3. LEMMA. *Let $\mathcal{A}=(S,A,\rightarrow)$ be a LTS. Let α,β be ordinal numbers with $\beta<\alpha$. Let $s,t \in S$. Then:*

0. $\subseteq^{0} = S \times S$
1. $\subseteq^{1} = \subseteq$ *and* $\leftrightarrows^{1} = \leftrightarrows$
2. $\subseteq^{\alpha} \subseteq \subseteq^{\beta}$
3. $\subseteq^{\alpha} \subseteq (\subseteq^{\beta})^{-1}$
4. $\leftrightarrows^{\alpha} \subseteq \subseteq^{\alpha} \subseteq \leftrightarrows^{\beta}$
5. $\leftrightarrow \subseteq \leftrightarrows^{\beta}$
6. $s \subseteq^{\alpha+1} t$ *iff there is a simulation relation* $R \subseteq (\subseteq^{\alpha})^{-1}$ *with* $s\,R\,t$.
7. *if α is a limit ordinal, then* $s \subseteq^{\alpha} t$ *iff for all* $\beta<\alpha$: $s \subseteq^{\beta} t$.

PROOF. Straightforward using the definitions. □

Besides the above lemma, there are a lot of other interesting facts about nested simulations that one may try to prove. In particular it is interesting to see what are the exact relationships between nested simulation equivalences and bisimulation equivalence. Below some results are presented which clarify these relationships. Since these results are a bit outside the scope of this paper, all proofs have been omitted.

8.5.4. COUNTEREXAMPLE. Below we present a counterexample which shows that the inclusion of Lemma 8.5.3.5 is strict. In order to present the example it is useful (although not necessary) to introduce the *summation* operator $\sum$. This operator, which for instance occurs in (MILNER, 1989), does not fit the framework of this paper because it may have an arbitrary, possibly infinite number of arguments. If t_i ($i \in I$) are terms, then $\sum_{i \in I} t_i$ is a term too. Its behaviour is described by rules (for all $a \in A$, $j \in I$):

$$\frac{t_j \xrightarrow{a} y}{\sum_{i \in I} t_i \xrightarrow{a} y}$$

One has to assume an upperbound on the cardinality of the index set I in order to make the collection of terms setlike. In our framework the operator $\sum$ can be coded by viewing $\sum_{i \in I} t_i$ as a constant. Besides the $\sum$ operator, we will use δ and $+$ as in $P(\mathrm{BPA}^{\varepsilon}_{\delta})$ and prefixing operators $a:(.)$ as in Section 5.11.3.

We define the following terms:

$$s_0 = c:\delta$$
$$t_0 = s_0 + b:\delta$$
$$s_{\alpha+1} = a:t_\alpha$$
$$t_{\alpha+1} = s_{\alpha+1} + a:s_\alpha$$

If α is a limit ordinal, then:

$$S_\alpha = \sum_{\beta<\alpha} d:s_\beta$$
$$s_\alpha = \sum_{\beta<\alpha} d:(S_\alpha + d:t_\beta)$$
$$t_\alpha = s_\alpha + d:S_\alpha$$

Below in Figure 12 a part of the transition system is displayed:

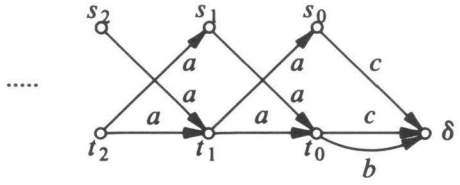

FIGURE 12

One can prove that for every ordinal α: $s_\alpha \leftrightarrows^\alpha t_\alpha$ and $s_\alpha \not\leftrightarrows t_\alpha$. However, within a fixed transition system $\leftrightarrows^\alpha$ and $\leftrightarrows$ will coincide when α is large enough:

8.5.5. THEOREM. *Let $\mathcal{A}=(S,A,\rightarrow)$ be a LTS and let α be the smallest regular cardinal larger than the cardinality of all sets $\{s' \mid s \xrightarrow{a} s'\}$ $(a \in A, s \in S)$. Then*

$$s \leftrightarrows^\alpha t \quad \Leftrightarrow \quad s \leftrightarrows t.$$

This theorem implies in particular that for image finite transition systems the intersection for all $m \in \mathbb{N}$ of m-nested simulation equivalence coincides with bisimulation equivalence.
Another implication is that if, relative to some transition system, $\leftrightarrows^\alpha$ is different from $\leftrightarrows$, $\leftrightarrows^\beta$ and $\leftrightarrows^\gamma$ are different for all $\beta<\gamma\leqslant\alpha$.

8.5.6. Nested simulations and completed trace equivalence. Simulation equivalence does not refine completed trace equivalence. Take for example the simulation equivalent processes a and $a\delta+a$. The completed trace sets are $\{a\star\surd\}$ and $\{a,a\star\surd\}$, respectively. However, it is not hard to see that for $m\geqslant 2$, m-nested simulation equivalence does refine completed trace equivalence.

8.5.7. LEMMA. *Let $\Sigma=(F,r)$ be a signature and let $P=(\Sigma,A,R)$ be a TSS. If P is well-founded and in tyft/tyxt format, then for all ordinals α, $\leftrightarrows^\alpha$ is a congruence for all function symbols in F.*

PROOF. Completely analogous to the proof of Theorem 5.10. Let P be well-founded and in tyft/tyxt format. It is sufficient to show that for all ordinals α, all $f\in F$ and all closed terms $u_i,v_i\in T(\Sigma)$ $(1\leqslant i\leqslant r(f))$:

$$\forall i\ u_i \leftrightarrows^\alpha v_i \quad\Rightarrow\quad f(u_1,..,u_{r(f)}) \leftrightarrows^\alpha f(v_1,..,v_{r(f)}).$$

We prove this statement with induction on α. Let α be an ordinal and suppose the statement is proved for $\beta<\alpha$. Let $R\subseteq T(\Sigma)\times T(\Sigma)$ be the least relation satisfying:

- $\leftrightarrows^\alpha \subseteq R$,
- for all function symbols f in F and terms u_i,v_i $(1\leqslant i\leqslant r(f))$ in $T(\Sigma)$:

$$\forall i\ u_i\, R\, v_i \quad\Rightarrow\quad f(u_1,..,u_{r(f)})\, R\, f(v_1,..,v_{r(f)}).$$

It is enough to show $R\subseteq\leftrightarrows^\alpha$. Let $\beta<\alpha$. Since, by Lemma 8.5.3.3, $\leftrightarrows^\alpha\subseteq(\leftrightarrows^\beta)^{-1}$, and because, by induction hypothesis, $\leftrightarrows^\beta$ is a congruence we have that $R\subseteq(\leftrightarrows^\beta)^{-1}$. In order to show $R\subseteq\leftrightarrows^\alpha$, it remains to be shown that R is a simulation relation, i.e. if $u\,R\,v$ and $u\xrightarrow{a}_P u'$ then there is a v' such that $v\xrightarrow{a}_P v'$ and $u'R\,v'$. The proof of this fact can in essence be copied from the proof of Theorem 5.10. □

The next theorem states the validity of inclusion 1.

8.5.8. THEOREM *(inclusion 1). Let $P=(\Sigma,A,R)$ be a TSS that is in pure tyft/tyxt format. Then:*

$$\leftrightarrows^2 \subseteq \equiv_{\text{pure tyft/tyxt}}.$$

PROOF. Let $s,t\in T(\Sigma)$ with $s\leftrightarrows^2 t$. Let $P'=(\Sigma',A',R')$ be a TSS in pure *tyft/tyxt* format that can be added conservatively to P and let $C[\,]$ be a $\Sigma\oplus\Sigma'$-context. Since $P\oplus P'$ is a conservative extension of P, $s\leftrightarrows^2 t$ within $TS(P\oplus P')$. Now we use that $\leftrightarrows^2$ is a congruence for operators in pure *tyft/tyxt* format (Lemma 8.5.7) and get $C[s]\leftrightarrows^2 C[t]$. Since $\leftrightarrows^2$ refines completed trace congruence: $C[s]\equiv_{CT} C[t]$. Because P' and $C[\,]$ were chosen arbitrarily this gives us: $s\equiv_{\text{pure tyft/tyxt}} t$. □

8.6. Testing Hennessy-Milner formulas. Next we give the definitions of Hennessy-Milner logic (HML) and prove the second inclusion in Figure 11. Most definitions are standard and can also be found in (HENNESSY & MILNER, 1985). The notion of HML-formulas of alternation depth m seems to be new, although the set of HML-formulas of alternation depth 1 (the formulas without negation) is exactly the set $\mathfrak{N}$ of (HENNESSY & MILNER, 1985).

8.6.1. DEFINITION. The set $\mathcal{L}$ of *Hennessy-Milner logic (HML) formulas* (over a given alphabet $A = \{a,b,\ldots\}$) is given by the following grammar:

$$\phi ::= T \mid \phi \wedge \phi \mid \neg\phi \mid \langle a \rangle \phi.$$

Let $\mathcal{A}=(S,A,\rightarrow)$ be a LTS. The satisfaction relation $\models \subseteq S \times \mathcal{L}$ is the least relation such that:

- $s \models T$ for all $s \in S$,
- $s \models \phi \wedge \psi$ iff $s \models \phi$ and $s \models \psi$,
- $s \models \neg\phi$ iff not $s \models \phi$,
- $s \models \langle a \rangle \phi$ iff for some $t \in S$: $s \xrightarrow{a} t$ and $t \models \phi$.

We adopt the following notations:

- F stands for $\neg T$,
- $\phi \vee \psi$ stands for $\neg(\neg\phi \wedge \neg\psi)$,
- $[a]\phi$ stands for $\neg\langle a \rangle \neg\phi$.

It is not difficult to see that any HML formula is logically equivalent to a formula in the language $\mathcal{L}'$ which is generated by the following grammar:

$$\phi ::= T \mid F \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi \mid [a]\phi.$$

8.6.2. DEFINITION. Let $\mathcal{A}=(S,A,\rightarrow)$ be a LTS and let $\mathcal{K}$ be a set of HML formulas. With $\sim_{\mathcal{K}}$ we denote the equivalence relation on S induced by $\mathcal{K}$:

$$s \sim_{\mathcal{K}} t \quad \Leftrightarrow \quad (\forall \phi \in \mathcal{K}: s \models \phi \Leftrightarrow t \models \phi).$$

We will call this relation $\mathcal{K}$ *formula equivalence.*

We recall a fundamental result of HENNESSY & MILNER (1985):

8.6.3. THEOREM (HENNESSY & MILNER). *Let $\mathcal{A}=(S,A,\rightarrow)$ be an image finite LTS. Then for all $s,t \in S$:*

$$s \leftrightarrows t \quad \Leftrightarrow \quad s \sim_{\mathcal{L}} t.$$

8.6.4. DEFINITION. For $m \in \mathbb{N}$ define the set $\mathcal{L}_m$ of HML-formulas by:

- $\mathcal{L}_0$ is empty,
- $\mathcal{L}_{m+1}$ is given by the following grammar:

$$\phi ::= \neg\psi \,(\text{for } \psi \in \mathcal{L}_m) \mid T \mid \phi \wedge \phi \mid \langle a \rangle \phi.$$

We leave it as an exercise to the reader to check that the equivalence induced by $\mathcal{L}_m$ formulas is the same as the equivalences induced by the sets $\mathcal{K}^{\langle\rangle}_m$ and $\mathcal{K}^{[]}_m$ which are given by:

- $\mathcal{K}^{\langle\rangle}_0 = \mathcal{K}^{[]}_0 = \varnothing$.
- $\mathcal{K}^{\langle\rangle}_{m+1}$ is defined by:

$$\phi ::= \psi \,(\text{for } \psi \in \mathcal{K}^{[]}_m) \mid T \mid F \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi.$$

- $\mathcal{K}^{[]}_{m+1}$ is defined by:

$$\phi ::= \psi (\text{for } \psi \in \mathcal{K}_m^{\langle\rangle}) \mid T \mid F \mid \phi \wedge \phi \mid \phi \vee \phi \mid [a]\phi.$$

8.6.5. EXAMPLE. Consider the terms s_i, t_i as defined in Section 8.5.4. Define for $0<m<\omega$ the formula $\varphi_m \in \mathcal{L}_m$ by: $\varphi_1 = \langle b\rangle T \wedge \langle c\rangle T$ and $\varphi_{m+1} = \langle a\rangle \neg \varphi_m$. It is easily checked that for $i \geqslant 0$: $s_i \not\models \varphi_{i+1}$ and $t_i \models \varphi_{i+1}$.

8.6.6. THEOREM *(Testing $\mathcal{L}_2$ formulas). Let $P_0 = (\Sigma_0, A_0, R_0)$ be a TSS in pure tyft/tyxt format. Then there is a TSS $P_1 = (\Sigma_1, A_1, R_1)$ in pure tyft format, which can be added conservatively to P_0, such that completed trace congruence within $P_0 \oplus P_1$ is included in $\mathcal{L}_2$ formula equivalence.*

PROOF. P_1 is constructed in the following way. The set A_1 consists of A_0 together with 5 new labels:

$$A_1 = A_0 \cup \{ok, left, right, syn, skip\}.$$

Signature Σ_1 contains a constant δ, unary function names $a:$ for each $a \in A_1$, and binary function symbols $+$ and *Sat*. Observe that the signature is finite if the alphabet A_0 is finite. For δ and $+$ we have just the same rules as in BPA_δ^ξ and $a:$ denotes prefixing like in Example 5.11.3. The most interesting operator is the operator *Sat*. Its first argument is intended to be a coding of some $\mathcal{L}_2$ formula. The *Sat* operator tests whether its second argument satisfies the $\mathcal{L}_2$ formula which is represented by its first argument. The rules of the *Sat* operator are given in Table 4. In the table a ranges over A_1. Because P_1 is in *tyft* format, $\Sigma_0 \cap \Sigma_1 = \varnothing$ and P_0 is pure *tyft/tyxt*, it follows with Theorem 7.6 that P_1 is a conservative extension of P_0.

$\dfrac{x \xrightarrow{skip} x'}{Sat(x,y) \xrightarrow{ok} Sat(x',y)}$	1
$\dfrac{x \xrightarrow{left} x_l,\ Sat(x_l,y) \xrightarrow{ok} y_l \quad x \xrightarrow{right} x_r,\ Sat(x_r,y) \xrightarrow{ok} y_r}{Sat(x,y) \xrightarrow{ok} y_l + y_r}$	2
$\dfrac{x \xrightarrow{syn} x',\ x' \xrightarrow{a} x'' \quad y \xrightarrow{a} y',\ Sat(x'',y') \xrightarrow{ok} y''}{Sat(x,y) \xrightarrow{ok} y''}$	3

TABLE 4. A test system for $\mathcal{L}_2$ formulas

$\mathcal{L}_2$ formulas are encoded using the following rules:

$$C_T = skip:\delta,$$
$$C_{\phi\wedge\psi} = left:C_\phi + right:C_\psi,$$
$$C_{\neg\phi} = skip:C_\phi,$$
$$C_{\langle a\rangle\phi} = syn:a:C_\phi.$$

We claim that for $\phi \in \mathcal{L}_2$, $Sat(C_\phi,t)$ has a completed trace *ok* iff $t \models \phi$. With this claim, which we will prove below, we can finish the proof of Theorem 8.6.6: whenever for some $s,t \in T(\Sigma_0 \oplus \Sigma_1)$ with $s \not\sim_{\mathcal{L}_2} t$, then there is an $\mathcal{L}_2$ formula ϕ_0 such that $s \models \phi_0$ and $t \not\models \phi_0$ (or vice versa). Using the claim this means $Sat(C_{\phi_0},s) \not\equiv_{CT} Sat(C_{\phi_0},t)$.

Before we present a formal proof of the claim, we give some intuition about how $Sat(C_\phi,t)$ tests the formula ϕ on t. If $\phi = T$, testing is straightforward: $C_T = skip:\delta$ and *skip* indicates to *Sat* that it can do an *ok* step (rule 1). Hence, $Sat(skip:\delta,t) \xrightarrow{ok} Sat(\delta,t)$ and it is not hard to check that $Sat(\delta,t)$ cannot do a next step.

Testing of $\wedge$ and $\langle a \rangle$ is almost as straightforward as testing the formula T and resembles the definition of $\models$. The intuitive meaning of the constant symbols *left:*, *right:* and *syn:* is respectively: transform to the left/right part of a formula and synchronize the next action of the coded formula and the tested process. Testing $\neg$ contains a little trick. First, the positive part of a formula is tested, which possibly yields a first *ok* and then the negative parts are tested. This can give rise to another *ok*. For instance the test $Sat(C_{\neg\phi},t)$ performs an initial *ok* step as its positive part is empty and then tests for the $\mathcal{L}_1$ formula ϕ whether $t \models \phi$. If there is no negative part that holds, the test does not yield another *ok* action and there is a completed trace *ok*. If a negative part is true, the test will yield another *ok* step and the *ok* trace is extended to the trace $ok \star ok$, which is not ok because now $ok \notin MT(Sat(C_\phi,t))$. Next we will give a formal proof of the claim.

LEMMA. *Let* $t \in T(\Sigma_0 \oplus \Sigma_1)$ *and let* $\phi \in \mathcal{L}_1$. *Then:*

i) $t \models \phi \Rightarrow CT(Sat(C_\phi,t)) = \{ok\}$,

ii) $t \not\models \phi \Rightarrow CT(Sat(C_\phi,t)) = \varnothing$.

PROOF. Induction on the structure of ϕ.

a) ϕ is T. Then $t \models \phi$. The only move of $Sat(C_\phi,t)$ is $Sat(C_\phi,t) \xrightarrow{ok} Sat(\delta,t)$ and $Sat(\delta,t)$ has no outgoing transitions. Both implications hold.

b) ϕ is $\phi_1 \wedge \phi_2$. If $t \models \phi$ then $t \models \phi_1$ and $t \models \phi_2$. By induction $CT(Sat(C_{\phi_1},t)) = \{ok\}$ and $CT(Sat(C_{\phi_2},t)) = \{ok\}$. Since all outgoing transitions of $Sat(C_\phi,t)$ are proved using rule 2 in Table 4, one can easily see that $CT(Sat(C_\phi,t)) = \{ok\}$. If on the other hand $t \not\models \phi$ then either $t \not\models \phi_1$ or $t \not\models \phi_2$. Hence by induction either $CT(Sat(C_{\phi_1},t)) = \varnothing$ or $CT(Sat(C_{\phi_2},t)) = \varnothing$. Thus $Sat(C_\phi,t)$ can have no outgoing transitions and $CT(Sat(C_\phi,t)) = \varnothing$.

c) ϕ is $\langle a \rangle \phi'$. If $t \models \phi$ then there is a t' such that $t \xrightarrow{a} t'$ and $t' \models \phi'$. By induction $CT(Sat(C_{\phi'},t')) = \{ok\}$. Outgoing transitions of $Sat(C_\phi,t)$ can only be proved using rule 3 and inspection of this rule allows us to conclude that $CT(Sat(C_\phi,t)) = \{ok\}$. If $t \not\models \phi$ then for all t' with $t \xrightarrow{a} t'$, $t' \not\models \phi'$. Hence by induction $CT(Sat(C_{\phi'},t')) = \varnothing$. But this implies $CT(Sat(C_\phi,t)) = \varnothing$ since rule 3 cannot be applied. □

CLAIM. *Let* $t \in T(\Sigma_0 \oplus \Sigma_1)$ *and let* $\phi \in \mathcal{L}_2$. *Then:*

$$t \models \phi \iff ok \in CT(Sat(C_\phi, t)).$$

PROOF. $\Rightarrow$) Induction on the structure of ϕ.

a) ϕ is $\neg\psi$, $\psi \in \mathcal{L}_1$. We have $t \not\models \psi$. By the lemma above, $CT(Sat(C_\psi, t)) = \varnothing$. By rule 1: $Sat(C_\phi, t) \xrightarrow{ok} Sat(C_\psi, t)$. Hence ok is in $CT(Sat(C_\phi, t))$.

b) ϕ is T. Rule 1 gives $Sat(C_T, t) \xrightarrow{ok} Sat(\delta, t) \not\longrightarrow$. Hence $ok \in CT(Sat(C_\phi, t))$.

c) ϕ is $\phi_1 \wedge \phi_2$. Since $t \models \phi$ we also have $t \models \phi_1$ and $t \models \phi_2$. By induction $ok \in CT(Sat(C_{\phi_1}, t))$ and $ok \in CT(Sat(C_{\phi_2}, t))$. Since all outgoing transitions of $Sat(C_\phi, t)$ are proved using rule 2, one can easily see that $ok \in CT(Sat(C_\phi, t))$.

d) $\phi = \langle a \rangle \phi'$. Since $t \models \langle a \rangle \phi'$, there is a t' such that $t \xrightarrow{a} t'$ and $t' \models \phi'$. Induction gives that $ok \in CT(Sat(C_{\phi'}, t'))$. Hence there is a termination node t'' such that $Sat(C_{\phi'}, t') \xrightarrow{ok} t''$. Now an application of rule 3 gives that $ok \in CT(Sat(C_\phi, t))$.

$\Leftarrow$) Induction on the structure of ϕ.

a) ϕ is $\neg\psi$, $\psi \in \mathcal{L}_1$. If $Sat(C_\phi, t)$ does a move, then the last rule applied in the proof must have been rule 1 and the transition must be $Sat(C_\phi, t) \xrightarrow{ok} Sat(C_\psi, t)$. Because $ok \in CT(Sat(C_\phi, t))$, $Sat(C_\psi, t)$ can have no outgoing transitions. Since $\psi \in \mathcal{L}_1$, the lemma allows us to conclude that $t \not\models \psi$. Hence $t \models \phi$.

b) ϕ is T. Since $t \models T$ the implication holds.

c) ϕ is $\phi_1 \wedge \phi_2$. If $Sat(C_\phi, t)$ does a move then the last rule applied in the proof of this transition must have been rule 2. Since $ok \in CT(Sat(C_\phi, t))$, it must be that $ok \in CT(Sat(C_{\phi_1}, t))$ and $ok \in CT(Sat(C_{\phi_2}, t))$. But this means that we can apply the induction hypothesis to obtain $t \models \phi_1$ and $t \models \phi_2$. Hence $t \models \phi$.

d) ϕ is $\langle a \rangle \phi'$. If $Sat(C_\phi, t)$ does a move then the last rule applied in the proof must have been rule 3. So, because $ok \in CT(Sat(C_\phi, t))$, there are t', t'' with $t \xrightarrow{a} t'$, $Sat(C_{\phi'}, t') \xrightarrow{ok} t''$ and t'' a termination node. This implies that $ok \in CT(Sat(C_{\phi'}, t'))$. By induction $t' \models \phi'$. Hence $t \models \phi$. □

This completes the proof of Theorem 8.6.6. □

8.6.7. COROLLARY *(inclusion 2). Let P be a TSS in pure tyft/tyxt format. Then:*

$$\equiv_{pure\ tyft/tyxt} \subseteq \sim_{\mathcal{L}_2}.$$

8.7. In this section it will be shown that the inclusions 4, 5 and 6 hold. As an immediate corollary it follows that inclusion 3 holds.

8.7.1. THEOREM *(inclusion 4). Let* $\mathcal{A}=(S,A,\rightarrow)$ *be a LTS. Then for all* $s,t\in S$ *and* $m\in\mathbf{N}$*:*

$$s \leftrightarrows^m t \;\Rightarrow\; s \sim_{\mathcal{L}_m} t.$$

PROOF. Suppose that $s \subseteq^m t$ and $s \models \phi$ for some $\phi\in\mathcal{L}_m$. We prove $t \models \phi$ with induction on m. The case $m=0$ is trivial. So suppose $m>0$. We prove $t\models\phi$ with induction on the structure of ϕ.

a) ϕ is $\neg\psi$, $\psi\in\mathcal{L}_{m-1}$. By definition of $s \subseteq^m t$ we have $t \subseteq^{m-1} s$. Application of the induction hypothesis gives $t \not\models \psi$ and hence $t\models\phi$.
b) ϕ is T. In this case $t\models\phi$ trivially holds.
c) ϕ is $\phi_1\wedge\phi_2$. From $s\models\phi$ it follows that $s\models\phi_1$ and $s\models\phi_2$. By induction $t\models\phi_1$ and $t\models\phi_2$. Hence, $t\models\phi$.
d) ϕ is $\langle a\rangle\phi'$. There exists an s' such that $s\xrightarrow{a}s'$ and $s'\models\phi'$. Since $s\subseteq^m t$, there exists an m-nested simulation R containing (s,t). Hence, for some $t'\in S$, $t\xrightarrow{a}t'$ and $s'R\,t'$. So $s'\subseteq^m t'$. By induction $t'\models\phi'$ and thus $t\models\phi$. □

We define $\leftrightarrows^m$ and $\leftrightarrows_n^m$ as auxiliary notions. Roughly speaking, $s\leftrightarrows_n^m t$ means that s and t are m-nested simulation equivalent to depth n. $\leftrightarrows^m$ is the intersection of $\leftrightarrows_n^m$ for all n.

8.7.2. DEFINITION. Let $\mathcal{A}=(S,A,\rightarrow)$ be a LTS. Define for $m,n\in\mathbf{N}$ relations $\subseteq_n^m \subseteq S\times S$ by:

- $s\subseteq_0^m t$ always,
- $s\subseteq_n^0 t$ always,
- $s\subseteq_{n+1}^{m+1} t$ iff $t\subseteq_{n+1}^m s$ and whenever $s\xrightarrow{a}s'$ then there is a t' such that $t\xrightarrow{a}t'$ and $s'\subseteq_n^{m+1} t'$.

We write:

- $s\leftrightarrows_n^m t$ if $s\subseteq_n^m t$ and $t\subseteq_n^m s$,
- $s\leftrightarrows^m t$ if for all n: $s\leftrightarrows_n^m t$,
- $s\subseteq^m t$ if for all n: $s\subseteq_n^m t$.

8.7.3. LEMMA. *Let* $m,n\in\mathbf{N}$. *Then* $\subseteq_{n+1}^m \subseteq \subseteq_n^m$ *and* $\leftrightarrows_{n+1}^m \subseteq \leftrightarrows_n^m$.
PROOF. Straightforward simultaneous induction on m and n. □

8.7.4. THEOREM *(inclusion 5). Let* $\mathcal{A}=(S,A,\rightarrow)$ *be a LTS which is image finite. Then for all* $s,t\in S$ *and* $m\in\mathbf{N}$*:*

$$s\sim_{\mathcal{L}_m} t \;\Rightarrow\; s\leftrightarrows^m t.$$

PROOF. Suppose that $s \not\leftrightarrows^m t$. With induction on m we show that there is a $\phi\in\mathcal{L}_m$ such that $s\models\phi$ but $t\not\models\phi$. It cannot be that $m=0$. So take $m>0$. Since $s\not\leftrightarrows^m t$, there must be an n such that $s\not\leftrightarrows_n^m t$. With induction on n we show that there exists a ϕ such that $s\models\phi$ but not $t\models\phi$.
It cannot be that $n=0$. Take $n>0$. If $t\not\subseteq_n^{m-1}s$ then we can find, by induction hypothesis, a $\psi\in\mathcal{L}_{m-1}$ such that $t\models\psi$ and $s\not\models\psi$. Hence $s\models\neg\psi$ (the formula $\neg\psi$ is in $\mathcal{L}_m$) and $t\not\models\neg\psi$. If, on the other hand, $t\subseteq_n^{m-1}s$, then it must be that for

some $a \in A$ and $s' \in S$ with $s \xrightarrow{a} s'$ we have that for all t' with $t \xrightarrow{a} t'$: $s' \not\subseteq^m_{n-1} t'$. Now a first possibility is that there is no t' such that $t \xrightarrow{a} t'$. In this situation $s \models \langle a \rangle T$, $t \not\models \langle a \rangle T$ and we are done. The other possibility is that there is a nonzero, but due to the image finiteness, finite number of states $t_1,..,t_p$ that can be reached from t by an a-transition. Since $s' \not\subseteq^m_{n-1} t_i$ for $1 \leqslant i \leqslant p$, we have by induction that there are $\phi_i \in \mathcal{L}_m$ such that $s' \models \phi_i$ and $t_i \not\models \phi_i$. Consider the $\mathcal{L}_m$-formula $\phi = \phi_1 \wedge .. \wedge \phi_p$. Since $s' \models \phi$ and $t_i \not\models \phi$, $s \models \langle a \rangle \phi$ and $t \not\models \langle a \rangle \phi$. □

8.7.5. THEOREM *(inclusion 6). Let $\mathcal{A} = (S, A, \rightarrow)$ be a LTS which is image finite. Then for all $s,t \in S$ and $m \in \mathbf{N}$:*

$$s \leftrightsquigarrow^m t \quad \Rightarrow \quad s \leftrightarrows^m t.$$

PROOF. Suppose that $s \subsetneq^m t$. With induction on m we prove that $s \subseteq^m t$. The case $m = 0$ is trivial. So suppose $m > 0$. We prove that $\subsetneq^m$ is an m-nested simulation relation. Whenever $v \subsetneq^m w$ then for all n, $v \subsetneq^m_n w$. Hence by definition of $\subsetneq^m$, $w \subsetneq^{m-1}_n v$ for all n. Thus $w \subsetneq^{m-1} v$ and by induction $w \subseteq^{m-1} v$. So the relation $\subsetneq^m$ is contained in the relation $(\subseteq^{m-1})^{-1}$. It remains to be shown that $\subsetneq^m$ is a simulation relation. Suppose $v \subsetneq^m w$ and $v \xrightarrow{a} v'$. Since for all $n > 0$, $v \subsetneq^m_n w$ there is for each n a w_n such that $w \xrightarrow{a} w_n$ and $v' \subsetneq^m_{n-1} w_n$. Due to the image finiteness there must be a w^* that occurs infinitely often in the sequence $w_1, w_2, ...$ Because for all n $\subsetneq^m_{n-1} \supseteq \subsetneq^m_n$ by Lemma 8.7.3, we have that for all $n > 0$, $v \subsetneq^m_{n-1} w^*$ and therefore $v \subsetneq^m w^*$. This concludes the proof that $\subsetneq^m$ is an m-nested simulation. □

8.7.6. COROLLARY. *Let $\mathcal{A} = (S, A, \rightarrow)$ be a LTS which is image finite. Then for all $s,t \in S$ and $m \in \mathbf{N}$:*

$$s \leftrightarrows^m t \quad \Leftrightarrow \quad s \leftrightsquigarrow^m t \quad \Leftrightarrow \quad s \sim_{\mathcal{L}_m} t.$$

PROOF. Immediate from Theorems 8.7.1, 8.7.4 and 8.7.5. □

8.8. Trace congruence. Using the above results, we can easily characterize the 'trace congruence' induced by pure *tyft/tyxt* rules as simulation equivalence or $\mathcal{L}_1$ formula equivalence (for image finite LTS's). We just repeat the argumentation above for trace congruence instead of completed trace congruence. First the notion of trace congruence is defined.

8.8.1. DEFINITION. Let $\mathcal{A} = (S, A, \rightarrow)$ be a LTS. A sequence $a_1 \star .. \star a_n \in A^*$ is a *trace* of s if there are states $s_1, s_2, .., s_n \in S$ such that $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} .. \xrightarrow{a_n} s_n$. $T(s)$ is the set of all traces of s. Two states $s,t \in S$ are *trace equivalent* if $T(s) = T(t)$. This is denoted $s \equiv_T t$.

8.8.2. DEFINITION. Let $\mathcal{F}$ be some format of TSS rules. Let $P=(\Sigma,A,R)$ be a TSS in $\mathcal{F}$ format. Two terms $s,t\in T(\Sigma)$ are *trace congruent with respect to* $\mathcal{F}$ rules, notation $s\equiv^T_{\mathcal{F}} t$, if for every TSS $P'=(\Sigma',A',R')$ in $\mathcal{F}$ format which can be added conservatively to P and for every $\Sigma\oplus\Sigma'$-context $C[\,]$: $C[s]\equiv_T C[t]$.

8.8.3. THEOREM. *Let $P=(\Sigma,A,R)$ be a TSS in pure tyft/tyxt format such that $TS(P)$ is image finite. Let $s,t\in T(\Sigma)$. Then:*

$$s\equiv^T_{\text{pure tyft/tyxt}} t \quad\Leftrightarrow\quad s\leftrightarrows t \quad\Leftrightarrow\quad s\sim_{\mathcal{L}_1} t.$$

PROOF. In fact most of the work has already been done. The equivalence of $\leftrightarrows$ and $\sim_{\mathcal{L}_1}$ follows from Corollary 8.7.6. The implication $s\equiv^T_{\text{pure tyft/tyxt}} t \Rightarrow s\leftrightarrows t$ follows by Lemma 8.5.7 and the observation that simulation equivalence refines trace equivalence. The reverse implication can be proved using the same test system as in the proof of Theorem 8.6.6. □

8.9. Characterization theorem for tree rules. The characterization Theorem 8.4.3 for tree rules of Bloom, Istrail & Meyer follows from Theorem 8.5.8, Corollary 8.7.6 and the following Theorem 8.9.1. In fact this combination gives a result which is even stronger than the result of Bloom, Istrail & Meyer as we allow more general rules in the original system and our test system is finite if the alphabet of the old system is finite (they did not look at finite test systems for $\mathcal{L}_2$ formulas). The next theorem also strengthens Theorem 8.6.6 because now only tree rules are used. But, as the proof of this theorem is rather tricky, we liked to give the simpler variant first.

8.9.1. THEOREM. *Let $P_0=(\Sigma_0,A_0,R_0)$ be a TSS in pure tyft/tyxt format. Then there is a TSS $P_1=(\Sigma_1,A_1,R_1)$ in tree rule format, which can be added conservatively to P_0, such that completed trace congruence within $P_0\oplus P_1$ is included in $\mathcal{L}_2$ formula equivalence. Moreover, if alphabet A_0 is finite, then the components of P_1 are finite too.*

PROOF (sketch). The alphabet A_1 consists of A_0 together with 8 new labels:

$$A_1=A_0\cup\{ok,ko,left,right,size,neg,\langle\rangle,i\}.$$

Σ_1 contains δ, $+$ and prefix-operators $a:$ for every $a\in A_1$. In R_1 we find the usual rules for these operators. Furthermore Σ_1 contains binary operators $\|_H$ which model parallel composition with synchronisation of actions in a set $H\subseteq A_1$. For these operators R_1 contains rules ($a\in A_1$):

$$\frac{x\xrightarrow{a}x'}{x\|_H y\xrightarrow{a}x'\|_H y}\quad a\notin H \qquad \frac{y\xrightarrow{a}y'}{x\|_H y\xrightarrow{a}x\|_H y'}\quad a\notin H$$

$$\frac{x\xrightarrow{a}x',\ y\xrightarrow{a}y'}{x\|_H y\xrightarrow{a}x'\|_H y'}\qquad a\in H$$

Next Σ_1 contains a binary operator *Sat* which tests whether its second

argument satisfies the $\mathcal{L}_2$ formula which is encoded using the rules below. Further it contains the auxiliary operators *Context*, *skip-i* and *ok-to-ko*. The rules in R_1 for these operators are displayed in Table 5 (where $a \in A_1$). If A_0 is finite then clearly A_1, Σ_1 and R_1 are finite too.

$$\frac{x \xrightarrow{ok} x'}{Sat(x,y) \xrightarrow{ok} \delta} \qquad \frac{x \xrightarrow{i} x'}{Context(x,y) \xrightarrow{i} Context(x', skip\text{-}i(y))}$$

$$\frac{x \xrightarrow{left} x_l,\ x \xrightarrow{right} x_r}{Sat(x,y) \xrightarrow{i} Sat(x_l,y) \|_{\{ok\}} Sat(x_r,y)} \qquad \frac{x \xrightarrow{ok} x'}{Context(x,y) \xrightarrow{ok} ok\text{-}to\text{-}ko(y)}$$

$$\frac{x \xrightarrow{\langle\rangle} x' \xrightarrow{a} x'',\ y \xrightarrow{a} y'}{Sat(x,y) \xrightarrow{i} Sat(x'',y')} \qquad \frac{x \xrightarrow{i} x' \xrightarrow{a} x''}{skip\text{-}i(x) \xrightarrow{a} x''}$$

$$\frac{x \xrightarrow{size} x',\ x \xrightarrow{neg} x''}{Sat(x,y) \xrightarrow{i} Context(x', Sat(x'',y))} \qquad \frac{x \xrightarrow{ok} x'}{ok\text{-}to\text{-}ko(x) \xrightarrow{ko} \delta}$$

TABLE 5. A test system for $\mathcal{L}_2$ formulas with tree rules only

Let the mapping $s: \mathcal{L}_1 \to \mathbf{N}$ be given by:

$$s(T) = 0$$
$$s(\phi \wedge \psi) = 1 + s(\phi) + s(\psi)$$
$$s(\langle a \rangle \phi) = 1 + s(\phi)$$

and let the Σ_1 terms S_n $(n \geq 0)$ be given by:

$$S_0 = ok:\delta$$
$$S_{n+1} = i:S_n$$

$\mathcal{L}_2$ formulas are coded as follows:

$$C_T = ok:\delta$$
$$C_{\phi \wedge \psi} = left:C_\phi + right:C_\psi$$
$$C_{\neg\phi} = size:S_{s(\phi)} + neg:C_\phi$$
$$C_{\langle a \rangle \phi} = \langle\rangle:a:C_\phi$$

We will now briefly explain the way in which the above construction works. We have the following claim:

CLAIM. *Let $\phi \in \mathcal{L}_2$ and $t \in T(\Sigma_0 \oplus \Sigma_1)$. Then $t \models \phi$ iff $Sat(C_\phi, t)$ has a completed trace with an ok action but without a ko action.*

It is not hard to see that the above claim is correct in case $\phi \in \mathcal{L}_1$. This is a direct consequence of the next lemma which can be proved easily by means of induction on the structure of ϕ:

LEMMA 1. *Let* $\phi \in \mathcal{L}_1$ *with* $s(\phi)=n$ *and let* $t \in T(\Sigma_0 \oplus \Sigma_1)$. *Then:*
- $t \models \phi \Rightarrow \{i^n \star ok\} \subseteq CT(Sat(C_\phi,t)) \subseteq \{i^n \star ok\} \cup \{i^m \mid 1 \leqslant m \leqslant n\}$,
- $t \not\models \phi \Rightarrow CT(Sat(C_\phi,t)) \subseteq \{i^m \mid 1 \leqslant m \leqslant n\}$.

The problem is what to do with negations. The key idea of our solution is that if one applies the *skip-i* operator $s(\phi)$ times on $Sat(C_\phi,t)$, the trace set of the resulting process consists of *ok* if $t \models \phi$ and will be empty otherwise. So what we have to do is to place $s(\phi)$ times a *skip-i* operator around $Sat(C_\phi,t)$ in a structured way and next apply a renaming of *ok* into *ko*. This is of course done using the binary operator *Context*. The first argument of this operator gives instructions on how to build a context around the second argument. In case a formula $\neg\phi$ has to be tested, our construction works in such a way that (after some *i*-steps) always an *ok* step will be generated, whereas a subsequent *ko* action is generated only when the tested process satisfies ϕ. One can prove the following lemma:

LEMMA 2. *Let* $\phi \in \mathcal{L}_1$ *with* $s(\phi)=n$ *and let* $t \in T(\Sigma_0 \oplus \Sigma_1)$. *Then:*
- $t \models \phi \Rightarrow CT(Context(S_n,Sat(C_\phi,t))) = \{i^n \star ok \star ko\}$,
- $t \not\models \phi \Rightarrow CT(Context(S_n,Sat(C_\phi,t))) = \{i^n \star ok\}$.

Using Lemma 2, the Claim can be proved with straightforward induction on the structure of ϕ. Theorem 8.9.1 is an immediate consequence of the Claim. □

9. COMPARISON WITH OTHER FORMATS

In this section we will give an extensive comparison of our format with the formats proposed by DE SIMONE (1984,1985) and BLOOM, ISTRAIL & MEYER (1988). First both formats are described.

9.1. DEFINITION. Let $\Sigma=(F,r)$ be a signature and let A be a set of labels. A *De Simone rule* (over Σ and A) takes the form:

$$\frac{\{x_i \xrightarrow{a} y_i \mid i \in I\}}{f(x_1,..,x_l) \xrightarrow{a} t}$$

where:
- $f \in F$ and $r(f)=l$,
- $I \subseteq \{1,..,l\}$,
- $x_1,..,x_l$ and y_i $(i \in I)$ are distinct variables,

Let for $1 \leqslant i \leqslant l$ $x_i'=y_i$ if $i \in I$ and $x_i'=x_i$ otherwise.
- t is a term in $T(\Sigma,\{x_1',..,x_l'\})$ in which each x_i' occurs at most once.

Clearly the De Simone format as presented above is included in our *tyft/tyxt* format. One should note however that De Simone assumes in addition that the set of labels is an (infinite) commutative monoid. Moreover he includes (unguarded) recursion in the language together with the standard fixed point rules.

9.2. DEFINITION. Let $\Sigma=(F,r)$ be a signature and let A be a set of labels. A *GSOS rule* (over Σ and A) takes the form:

$$\frac{\{x_i \xrightarrow{a_{ij}} y_{ij} \mid 1\leqslant i\leqslant l,\ 1\leqslant j\leqslant m_i\} \cup \{x_i \xrightarrow{b_{ij}} \not\rightarrow \mid 1\leqslant i\leqslant l,\ 1\leqslant j\leqslant n_i\}}{f(x_1,..,x_l) \xrightarrow{a} t}$$

where the variables are all distinct, $f\in F$, $l=r(f)$, $m_i,n_i\geqslant 0$, $a_{ij},b_{ij}\in A$ and t is a term in $T(\Sigma,\{x_i,y_{ij} \mid 1\leqslant i\leqslant l,\ 1\leqslant j\leqslant m_i\})$.

A *GSOS rule system* is a triple (Σ,A,R) with Σ a signature, A a set of labels and R a set of GSOS rules over Σ and A.

We should mention here that the above definition is simplified in order to make comparison possible and only gives an approximation of the notion of a GSOS rule system as it is defined by BLOOM, ISTRAIL & MEYER (1988). There a GSOS rule system contains some additional ingredients for dealing with guarded recursion and there are a number of finiteness constraints. The feature which distinguishes GSOS rules from the other rules in this paper is the possibility of *negative* premises. This makes that it is not immediately clear how (and if) a GSOS rule system determines a transition relation.

9.2.1. DEFINITION. Let (Σ,A,R) be a GSOS rule system. A transition relation $\rightarrow \subseteq T(\Sigma)\times A\times T(\Sigma)$ *agrees with* the rules in R if:

- Whenever an instantiation by a substitution σ of the premises of a rule is true of the relation, then the instantiation of the conclusion by σ is true as well.
- Whenever $t \xrightarrow{a} t'$ is true, then there is a rule r and an instantiation σ such that $t \xrightarrow{a} t'$ is the instantiation of the conclusion of r by σ, and the instantiations of the premises of r by σ are true.

It is not hard to show that for any GSOS rule system, there is a unique transition relation which agrees with the rules. If a GSOS rule system only contains positive rules then it is a TSS according to our definition. Moreover in this case the unique transition relation which agrees with the rules according to the definition above is just the same relation as the one defined in Definition 3.2 using the notion of proof trees of transitions.

The following example from BLOOM, ISTRAIL & MEYER (1988) shows that in general the GSOS format cannot be combined consistently with the *tyft/tyxt* format. There are 4 operators in the signature: f, g, c and d. We have an action a and the following rules:

$$\frac{x \xrightarrow{a} y \quad y \xrightarrow{a} z}{f(x) \xrightarrow{a} d}$$

$$\frac{x \xrightarrow{a} \not\rightarrow}{g(x) \xrightarrow{a} d}$$

$$c \xrightarrow{a} g(f(c))$$

There is no transition relation which agrees with these rules. In particular, $f(c)$ can move iff it cannot move.[1]

9.3. EXAMPLES. Below we list some examples that illustrate the differences between the formats.

9.3.1. Global closure properties. Rules in *tyxt* format fit neither De Simone's format nor the GSOS format. One could say that *tyxt* rules, like for instance the τ rules of Table 2, express certain 'global closure properties', a form of operational behaviour which is in general independent of the particular function symbol at the head of a term.

9.3.2. Contexts. Often it is very useful to have function symbols in the left hand side of a premise. However, this is not allowed by the De Simone or GSOS format. In Section 6.2 we saw that these rules can be used to model recursion. Also in the system of Table 4 for testing $\mathcal{L}_2$ formulas, this type of rules play an important role. In (BAETEN & VAN GLABBEEK, 1987), operators ϵ_K are described that erase all actions from a set $K \subseteq Act$. We can add these operators to $P(\mathrm{BPA}^{\epsilon}_{\delta})$ together with the following rules from (BAETEN & VAN GLABBEEK, 1987):

$$\frac{x \xrightarrow{a} y}{\epsilon_K(x) \xrightarrow{a} \epsilon_K(y)} \quad a \notin K$$

$$\frac{x \xrightarrow{a} y \quad \epsilon_K(y) \xrightarrow{b} z}{\epsilon_K(x) \xrightarrow{b} z} \quad a \in K$$

The same type of trick can also be used to describe the 'atomic version operator'. This operator was introduced by DE BAKKER & KOK (1988) for giving semantics to concurrent Prolog. Here we will give our own variant of this operator, using our own notation. The interested reader who wants to know how this type of operators can be used to give semantics to concurrent Prolog is referred to (DE BAKKER & KOK, 1988). Take as starting point the signature of $\mathrm{BPA}^{\epsilon}_{\delta}$. But as labels of transitions we now don't take elements of $Act_{\surd}$, but elements of the set of finite sequences over $Act_{\surd}$. Write a for the sequence consisting of the single symbol $a \in Act_{\surd}$. With $\sigma\sigma'$ we denote the concatenation of the sequences σ and σ'. The set of rules of the TSS contains the rules of $R(\mathrm{BPA}^{\epsilon}_{\delta})$ (but now the labels should be interpreted as sequences!) and moreover the following rules:

$$\frac{x \xrightarrow{\surd} y}{[x] \xrightarrow{\surd} y}$$

1. In (GROOTE, 1989), it is investigated in which cases a specification in *'ntyft/ntyxt'* format is consistent. A general method, based on the stratification technique in logic programming, is presented to show consistency of sets of rules. It is shown that various results from this paper extend smoothly to a setting where rules may contain negative premises.

$$\frac{x \xrightarrow{a} y \quad [y] \xrightarrow{\sigma} z}{[x] \xrightarrow{a\sigma} z}$$

The rules express that only successful sequences, i.e. sequences ending on $\surd$, can happen in the scope of an atomic version operator. The rules are in *tyft* format. Hence, strong bisimulation is a congruence in this setting.

9.3.3. Lookahead. All operators defined with the De Simone or GSOS format have a lookahead of at most 1. Hence the following operator, which can be viewed as the inverse of the split operator of VAN GLABBEEK & VAANDRAGER (1987), cannot be defined:

$$\frac{x \xrightarrow{a^+} y \quad y \xrightarrow{a^-} z}{combine(x) \xrightarrow{a} combine(z)}$$

Other examples of operators with a lookahead are the ϵ_K and the atomic version operator as described above. As a last example we mention the *abstraction* or *hiding* operator from ACP_τ (here $I \subseteq Act$):

$$\frac{x \xrightarrow{a} x'}{\tau_I(x) \xrightarrow{\tau} \tau_I(x')} \quad a \in I$$

$$\frac{x \xrightarrow{a} x'}{\tau_I(x) \xrightarrow{a} \tau_I(x')} \quad a \notin I$$

If we add these rules to the system $P(BPA^\tau_{\epsilon\delta})$ as described in Section 6.1, then we can derive:

$$\tau_{\{i\}}(i \cdot a) \xrightarrow{a} \tau_{\{i\}}(\epsilon).$$

Observe that the *rules* that contain a function symbol τ_I all have a lookahead of 1 (i.e. the length of the maximal path in the dependency graphs of the rules is 1). As *operators* on transition systems the τ_I have an unbounded lookahead, due to the presence of *tyxt* rules with a lookahead of 2 in $P(BPA^\tau_{\epsilon\delta})$.

9.3.4. Copying. In contrast to De Simone's format, the GSOS format and also our format can describe operators which copy their arguments. The system call *fork* of UNIX (1986) is a typical example of an operation that one would like to describe using copying. One can think of a rule like:

$$fork(x) \xrightarrow{\tau} parent(x) \| child(x).$$

Below we present another example where copying occurs naturally. It describes an operational semantics of the natural numbers which is based on the idea of counting: the process associated to an integer expression performs as many actions as the value which is denoted by this expression under the standard interpretation. We consider the signature containing a constant symbol 0, a unary function symbol *succ* and binary function symbols $+$ and $\times$. There is only one transition label, namely 1. The operational semantics of the operators is described by the following rules:

$$succ(x) \xrightarrow{1} x$$

$$\frac{x \xrightarrow{1} x'}{x+y \xrightarrow{1} x'+y} \qquad \frac{y \xrightarrow{1} y'}{x+y \xrightarrow{1} x+y'}$$

$$\frac{x \xrightarrow{1} x' \quad y \xrightarrow{1} y'}{x \times y \xrightarrow{1} (x' \times y') + (x' + y')}$$

Observe that two expressions denote the same value under the standard interpretation iff they are bisimilar.

9.3.5. Branching. The ability to copy arguments is not the only difference between De Simone's format and the GSOS format. A rule like:

$$\frac{x \xrightarrow{a} x', \; x \xrightarrow{b} x''}{f(x) \xrightarrow{a} f(x')}$$

fits De Simone's format but not the GSOS format. In this rule we see a *branching* in the dependency graph at node x.

9.3.6. Catalysis. A similar example is obtained if we add to $P(\mathrm{BPA}_\delta^\varepsilon)$ the following rule which fits the GSOS format:

$$\frac{x \xrightarrow{ok} x', \; y \xrightarrow{a} y'}{Cat(x,y) \xrightarrow{a} Cat(x,y')}$$

Here we have a situation, not allowed by De Simone's format, where a potential *ok*-action of the first component makes it possible for the second component to proceed. But when it proceeds the first component remains unchanged. Hence, one can view the first component as a *catalyst* of the second component.

9.3.7. Priorities. In (Baeten, Bergstra & Klop, 1986) an operator is introduced to describe priorities in ACP, whereby some actions have priorities over others in a non-deterministic choice. The operator turns out to be quite interesting and has been used in a number of applications. In (Baeten, Bergstra & Klop, 1986) the operator is defined using equations, but if one uses Plotkin-style rules then it is inevitable to use negative hypotheses.

Consider the GSOS rule system $P(\mathrm{BPA}_\delta^\varepsilon)$ and assume that the set $Act_\surd$ of labels is finite. Assume furthermore that a partial order $>$ is given on $Act_\surd$ such that $\surd$ is not in the ordering. Now we can add a unary operator θ to the rule system, with for each $a \in A_\surd$ a rule:

$$\frac{x \xrightarrow{a} x', \; \forall b > a : x \xrightarrow{b} \not\rightarrow}{\theta(x) \xrightarrow{a} \theta(x')}$$

The rule expresses that in the scope of a θ-operator an a action can occur unless an action with a higher priority is possible. Cleaveland & Hennessy (1988) describe priorities using *tyxt* rules with negative hypotheses. Another example of an operator that is defined using rules with negative premises is the *broadcast* operator as described by Pnueli (1985).

9.4. Completed trace congruences. The differences between the formats presented thus far can be understood also if we look at the completed trace congruences which they induce. In Section 8 we saw that the trace congruence induced by (variants) of the pure *tyft/tyxt* format coincides with $\mathcal{L}_2$ formula equivalence.
The main theorem which De Simone proved about his format is that all operators defined using his type of inductive rules can also be defined by MEIJE-SCCS 'architectural' expressions. Similar results have not yet been proved for the GSOS or the *tyft/tyxt* format. Now it is a standard result that the completed trace congruence induced by languages like MEIJE-SCCS, ACP, CSP, etc., coincides with *failure equivalence* ($\equiv_F$) (see for instance (BERGSTRA, KLOP & OLDEROG, 1988)). Hence the completed trace congruence induced by De Simone's format is failure equivalence (it is not too difficult to give a direct proof of this fact).
BLOOM, ISTRAIL & MEYER (1988) characterized the completed trace congruence induced by their format in terms of the equivalence corresponding to the following set of formulas:

9.4.1. DEFINITION. The set $\mathcal{D}$ of *denial*[1] *(HML) formulas* (over a given alphabet $A=\{a,b,\ldots\}$) is given by the following grammar:

$$\phi ::= T \mid \phi \wedge \phi \mid [a]F \mid \langle a \rangle \phi.$$

9.4.2. THEOREM (BLOOM, ISTRAIL & MEYER). *Let $P=(\Sigma,A,R)$ be a GSOS rule system such that the associated transition system is image finite. Then:* $\equiv_{GSOS} = \sim_{\mathcal{D}}$.

Some additional insight is provided by the following characterization of denial equivalence which is due to LARSEN & SKOU (1988).

9.4.3. DEFINITION. Let $\mathcal{A}=(S,A,\rightarrow)$ be a LTS. A relation $R \subseteq S \times S$ is a *2/3-bisimulation*, also called a *ready simulation*, if it satisfies:
1. whenever $s\,R\,t$ and $s \xrightarrow{a} s'$ then, for some $t' \in S$, also $t \xrightarrow{a} t'$ and $s'\,R\,t'$,
2. whenever $s\,R\,t$ and $t \xrightarrow{a} t'$ then, for some $s' \in S$, also $s \xrightarrow{a} s'$.

Two states $s,t \in S$ are *2/3-bisimilar* (or *ready simulation equivalent*) in $\mathcal{A}$ if there exists a 2/3-bisimulation containing the pair (s,t) and a 2/3-bisimulation containing the pair (t,s).

1. The formulas as defined in (BLOOM, ISTRAIL & MEYER, 1988) were called *limited modal formulas* and may also contain F and $\vee$. However, it is easily proved that this addition does not increase the distinguishing power.

9.4.4. THEOREM (LARSEN & SKOU). *Let $\mathcal{A}=(S,A,\rightarrow)$ be an image finite LTS. Then two states are 2/3-bisimilar just in case they satisfy exactly the same denial formulas.*

It is a trivial exercise to show that:

$$\leftrightarrows^2 \subseteq \leftrightarrows_{2/3} \subseteq \equiv_F \subseteq \equiv_{CT}$$

The examples of Figures 13, 14 and 15 show that these inclusions are strict.

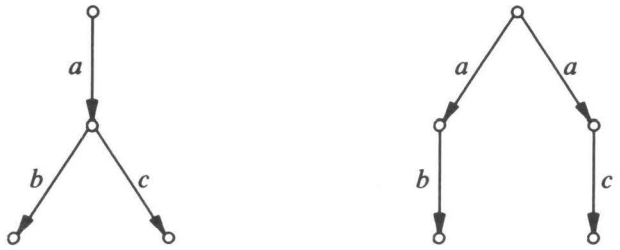

FIGURE 13. Completed trace equivalent but not De Simone congruent

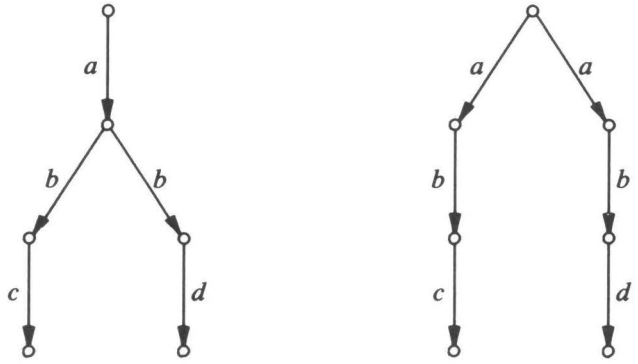

FIGURE 14. De Simone congruent but not GSOS congruent

9.4.5. Testing denial formulas. The question arises whether all features of the GSOS format are really needed in order to test denial formulas. In particular it is interesting to know whether the negative premises add anything to the discriminating power of the format. Surprisingly, as was first observed by ROB VAN GLABBEEK (1988), this is not the case: GSOS congruence coincides with positive GSOS congruence. Below we present a system in positive GSOS format for testing denial formulas. The system is simpler than the original system of Rob van Glabbeek. Moreover our system has the advantage of being finite in case the alphabet of the old system is finite.

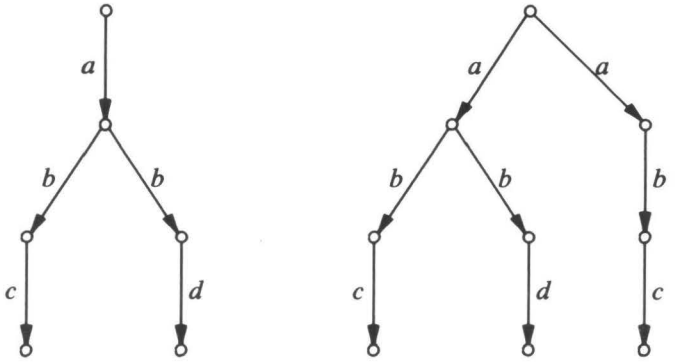

FIGURE 15. GSOS congruent but not pure *tyft/tyxt* congruent

9.4.6. THEOREM. *Suppose we have a TSS* $P_0=(\Sigma_0,A_0,R_0)$ *in GSOS format. Then there exists a TSS* $P_1=(\Sigma_1,A_1,R_1)$ *in GSOS format with all premises positive and non-branching, which can be added conservatively to* P_0*, such that completed trace congruence within* $P_0 \oplus P_1$ *is included in denial equivalence. Moreover, if alphabet* A_0 *is finite, then the components of* P_1 *are finite too.*

PROOF. The set A_1 consists of A_0 together with 6 new labels:

$$A_1 = A_0 \cup \{ok, ko, left, right, [], \langle\rangle\}.$$

Signature Σ_1 contains a constant δ, unary function names $a:$ for each $a \in A_1$, and binary function symbols $+$, $\|$, Sat, $Sat_{[]}$, $Sat_{\langle\rangle}$ and Sat_{right}. The rules for δ, $a:$ and $+$ are as usual. $\|$ is just arbitrary interleaving. The Sat operator tests whether its second argument satisfies the denial formula which is represented by its first argument. The rules for the $\|$-operator and the various Sat-operators are given in Table 6. In the table, a ranges over A_1. One can check that P_1 can be added conservatively to P_0.

$$\frac{x \xrightarrow{[]} x'}{Sat(x,y) \xrightarrow{[]} Sat_{[]}(x',y)} \qquad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{a} y'}{Sat_{[]}(x,y) \xrightarrow{ko} \delta}$$

$$\frac{x \xrightarrow{\langle\rangle} x'}{Sat(x,y) \xrightarrow{\langle\rangle} Sat_{\langle\rangle}(x',y)} \qquad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{a} y'}{Sat_{\langle\rangle}(x,y) \xrightarrow{ok} Sat(x',y')}$$

$$\frac{x \xrightarrow{left} x'}{Sat(x,y) \xrightarrow{left} Sat(x',y) \| Sat_{right}(x,y)} \qquad \frac{x \xrightarrow{right} x'}{Sat_{right}(x,y) \xrightarrow{right} Sat(x',y)}$$

$$\frac{x \xrightarrow{a} x'}{x \| y \xrightarrow{a} x' \| y} \qquad \frac{y \xrightarrow{a} y'}{x \| y \xrightarrow{a} x \| y'}$$

TABLE 6. A test system for denial formulas

Denial formulas are encoded using the following rules:

$$C_T = \delta$$
$$C_{\phi \wedge \psi} = left:C_\phi + right:C_\psi$$
$$C_{[a]F} = []:a:\delta$$
$$C_{\langle a \rangle \phi} = \langle\rangle:a:C_\phi$$

CLAIM. *Let $t \in T(\Sigma_0 \oplus \Sigma_1)$ and let ϕ be a denial formula. Then $t \models \phi$ iff $Sat(C_\phi, t)$ has a completed trace with as many ok's as ϕ has $\langle a \rangle$'s, and no ko.*
PROOF. Rather straightforward induction on the structure of ϕ. □

9.4.7. Comparison of testing abilities. The notion of testing which underlies CCS/CSP/ACP, and hence De Simone's format, is well-known (see for instance (DE NICOLA & HENNESSY, 1984) and (BERGSTRA, KLOP & OLDEROG, 1988)): these languages allow one to observe *traces*, *deadlock* and to *block* actions from a certain moment onwards. This makes it possible to detect *refusals* indirectly: one concludes that a certain action can be refused after an initial trace because deadlock occurs if all the other actions are blocked. The construction in the proof of Theorem 9.4.6 clearly shows which notion of testing underlies the (positive) GSOS format: the format allows one to observe *traces* of processes, to detect *refusals* and to make *copies* of processes at every moment. In the general GSOS format refusals can be observed directly: one can define a context which performs an *ok* step if its argument cannot do a certain action. In the positive GSOS format refusals can also be observed, but only indirectly. The key feature which distinguishes the positive GSOS format from the De Simone format is the capacity to make copies of processes at every moment. Observe that the only rule in Table 6 that does not fit De Simone's format is the rule dealing with the *left* action. In this rule the x and y are copied. In many situations copying is a natural operation which can be realised physically by for instance a core dump procedure.

The construction in the proof of Theorem 8.9.1 shows that the additional testing power needed to bring one from denial equivalence to $\mathcal{L}_2$ formula equivalence only consists of the ability to see whether some action is possible in the future: there should be operations with a *lookahead* (in fact the proof of Theorem 8.9.1 shows that a lookahead of 2 is already enough). Using operators with a lookahead one can investigate *all* branches of a process for positive information and one can see whether a certain *tree* is possible. In particular one can see whether there exists a branch in which a certain action is present. In the same way as one can observe in De Simone's format that a certain action is refused because deadlock occurs when the other actions are blocked, one can conclude in the *tyft/tyxt* format that a tree is refused. The ability to see in the future of a process can be considered as a weak form of *global testing*. Global testing is the same as what MILNER (1981) calls *controlling the weather conditions*. ABRAMSKY (1987) describes global testing as: "the ability to enumerate all (of finitely many) possible 'operating environments' at each stage of the test, so as to guarantee that all nondeterministic branches will be

pursued by various copies of the subject process". Because an operator with lookahead is not capable to see negative information (like the absence of some action) directly, and because it is also not able to force that all nondeterministic branches are pursued by some number of copies, lookahead does not give one the full testing power of global testing. Since global testing is needed in order to distinguish between processes which are not bisimilar, this explains why the fully abstract semantics induced by our format is still below bisimulation equivalence. Global testing in the above sense seems very unrealistic as a testing ability and in direct conflict with the observational viewpoint of concurrent systems. Recently however, LARSEN & SKOU (1988) have pointed out that if one assumes that every transition in a transition system has a certain minimum probability of being taken, an observer can - due to the probabilistic nature of transitions - with arbitrary high degree of confidence, assume that all transitions have been examined, simply by repeating an experiment many times (using the copying facility). This idea gives some plausibility to the notion of global testing. In fact LARSEN & SKOU (1988) deviced some testing algorithms which allow them, with a probability arbitrary close to 1, to distinguish between processes that are not bisimilar.

Unless one believes in fortune telling as a technique which has some practical relevance for computer science, lookahead as a testing notion is not very realistic. Still, this lookahead pops up naturally if one looks at the maximal format of rules for which bisimulation is a congruence and we showed that rules with a lookahead are often useful. Therefore we think that, just like bisimulation equivalence, $\pounds_2$ formula equivalence is an interesting equivalence that is worth studying, even though it does not correspond to a very natural notion of testing.

9.4.8. Finiteness and decidability. In their paper 'Bisimulation can't be traced', BLOOM, ISTRAIL & MEYER (1988) argue that bisimulation equivalence *cannot* be reduced to completed trace congruence with respect to any *reasonably structured* system of process constructing operations. They present the GSOS format, which they believe to be the most general format leading to reasonably structured systems, and then show that the congruence induced by this format is denial formula equivalence. Although the pure *tyft/tyxt* format cannot trace bisimulation equivalence, it can trace more of it than the GSOS format. This implies that not all pure *tyft/tyxt* rules are structured according to the definition of BLOOM, ISTRAIL & MEYER (1988). And indeed what's wrong in their opinion with our rules is that they might lead to transition systems with a transition relation which is infinitely branching or not computable. The various finiteness constraints which are present in the definition of the GSOS format in (BLOOM, ISTRAIL & MEYER, 1988), are motivated by the requirement that the transition relation should be computably finitely branching. We think that, although it is certainly important to have finiteness and decidability, it is much too strong to call any TSS leading to a transition relation which does not have these properties 'not reasonably structured' (this is what BLOOM, MEYER & ISTRAIL (1988) seem to do). Since our format gives us the expressiveness to

describe the invisible nature of τ (see Section 6.1) it is to be expected that, in general, we also have the infinite branching and undecidability of the models of CCS/ACP$_\tau$ based on observational congruence. If one disqualifies infinitary and undecidable TSS's right from the start, then one misses a large number of interesting applications. Of course the question what type of TSS's do lead to computably finitely branching transition systems is a very interesting one. It seems that if one generalises the positive GSOS format in the direction of the *tyft/tyxt* format, infinite branching arises quite soon. The following example for instance, which is due to Bard Bloom, illustrates that function symbols in the premises are 'dangerous'.

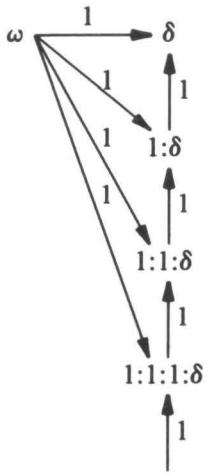

FIGURE 15

In the example we have prefixing and δ as usual and moreover a constant ω with rules:

$$\omega \xrightarrow{1} \delta \qquad \frac{\omega \xrightarrow{1} x}{\omega \xrightarrow{1} 1{:}x}$$

The part of the transition system which is displayed in Figure 15 shows that ω has an infinite number of outgoing transitions. Another example illustrating the same point is obtained by adding recursion to $P(BPA^{\epsilon}_{\delta})$ in the style of Section 6.2 with the 'unguarded' recursive definition $X \Leftarrow Xa + a$. It is easy to give examples of *tyxt* rules or tree rules which lead to infinite branching or undecidability. It is an open question to find a format in between positive GSOS and *tyft/tyxt* which always leads to computably finitely branching transition relations.

In our view one reason why rules with a lookahead are important is that they make it possible to have different levels of granularity of actions and to express that an action at one level can be composed of several smaller actions at a lower level. The system of Table 6 for testing denial equivalence is an

excellent example of a situation where the GSOS format forces one to do in two steps what one would like to do in only one.

References

[1] S. Abramsky (1987): *Observation equivalence as a testing equivalence.* Theoretical Computer Science 53, pp. 225-241.

[2] P. America, J.W. de Bakker, J.N. Kok & J.J.M.M. Rutten (1986): *Operational semantics of a parallel object-oriented language.* In: Conference Record of the 13th ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida, pp. 194-208.

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

[4] J.C.M. Baeten, J.A. Bergstra & J.W. Klop (1986): *Syntax and defining equations for an interrupt mechanism in process algebra.* Fund. Inf. IX(2), pp. 127-168.

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

[6] J.W. de Bakker & J.N. Kok (1988): *Uniform abstraction, atomicity and contractions in the comparative semantics of concurrent Prolog.* In: Proceedings Fifth Generation Computer Systems 1988 (FGCS 88), Tokyo, Japan, pp. 347-355.

[7] J.A. Bergstra & J.W. Klop (1988): *A complete inference system for regular processes with silent moves.* In: Proceedings Logic Colloquium 1986 (F.R. Drake & J.K. Truss, eds.), North Holland, Hull, pp. 21-81, also appeared as: Report CS-R8420, Centrum voor Wiskunde en Informatica, Amsterdam 1984.

[8] J.A. Bergstra, J.W. Klop & E.-R. Olderog (1988): *Readies and failures in the algebra of communicating processes.* SIAM Journal on Computing 17(6), pp. 1134-1177.

[9] B. Bloom (November 1988): *Personal communication.*

[10] B. Bloom, S. Istrail & A.R. Meyer (1988): *Bisimulation can't be traced: preliminary report.* In: Conference Record of the 15th ACM Symposium on Principles of Programming Languages (POPL), San Diego, California, pp. 229-239.

[11] G. Boudol (1985): *Notes on algebraic calculi of processes.* In: Logics and Models of Concurrent Systems (K. Apt, ed.), NATO ASI Series F13, Springer-Verlag, pp. 261-303.

[12] R. Cleaveland & M. Hennessy (1988): *Priorities in process algebra.* In: Proceedings 3th Annual Symposium on Logic in Computer Science (LICS), Edinburgh, pp. 193-202.

[13] R. De Nicola & M. Hennessy (1984): *Testing equivalences for processes.* Theoretical Computer Science 34, pp. 83-133.

[14] R.J. van Glabbeek (1987): *Bounded nondeterminism and the*

approximation induction principle in process algebra. In: Proceedings STACS 87 (F.J. Brandenburg, G. Vidal-Naquet & M. Wirsing, eds.), LNCS 247, Springer-Verlag, pp. 336-347.

[15] R.J. VAN GLABBEEK (November 1988): *Personal communication.*

[16] R.J. VAN GLABBEEK & F.W. VAANDRAGER (1987): *Petri net models for algebraic theories of concurrency.* In: Proceedings PARLE conference, Eindhoven, Vol. II (Parallel Languages) (J.W. de Bakker, A.J. Nijman & P.C. Treleaven, eds.), LNCS 259, Springer-Verlag, pp. 224-242.

[17] J.F. GROOTE (1989): *Transition system specifications with negative premises.* Report CS-R89.., Centrum voor Wiskunde en Informatica, Amsterdam, to appear. Extended abstract submitted to LICS 90.

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

[19] M. HENNESSY & R. MILNER (1985): *Algebraic laws for nondeterminism and concurrency.* JACM 32(1), pp. 137-161.

[20] R.M. KELLER (1976): *Formal verification of parallel programs.* Communications of the ACM 19(7), pp. 371-384.

[21] J.W. KLOP (1987): *Term rewriting systems: a tutorial.* Bulletin of the EATCS 32, pp. 143-182.

[22] K.G. LARSEN & A. SKOU (1988): *Bisimulation through probabilistic testing.* R 88-29, Institut for Elektroniske Systemer, Afdeling for Matematik og Datalogi, Aalborg Universitetscenter.

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

[24] R. MILNER (1981): *Modal characterisation of observable machine behaviour.* In: Proceedings CAAP 81 (G. Astesiano & C. Bohm, eds.), LNCS 112, Springer-Verlag, pp. 25-34.

[25] R. MILNER (1983): *Calculi for synchrony and asynchrony.* Theoretical Computer Science 25, pp. 267-310.

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

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

[28] D.M.R. PARK (1981): *Concurrency and automata on infinite sequences.* In: Proceedings 5th GI Conference (P. Deussen, ed.), LNCS 104, Springer-Verlag, pp. 167-183.

[29] G.D. PLOTKIN (1981): *A structural approach to operational semantics.* Technical Report DAIMI FN-19, Computer Science Department, Aarhus University.

[30] G.D. PLOTKIN (1983): *An operational semantics for CSP.* In: Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts - II, Garmisch, 1982 (D. Bjørner, ed.), North-Holland, Amsterdam, pp. 199-225.

[31] A. PNUELI (1985): *Linear and branching structures in the semantics and logics of reactive systems.* In: Proceedings ICALP 85, Nafplion (W. Brauer, ed.), LNCS 194, Springer-Verlag, pp. 15-32.

[32] R. DE SIMONE (1984): *Calculabilité et expressivité dans l'algebra de processus parallèles Meije*. Thèse de 3^e cycle, Univ. Paris 7.

[33] R. DE SIMONE (1985): *Higher-level synchronising devices in* MEIJE-SCCS. Theoretical Computer Science 37, pp. 245-267.

[34] UNIX (1986): *Programmer's Reference Manual, 4.3 BSD edition*, Computer Systems Research Group, University of California, Berkeley.

[35] J.L.M. VRANCKEN (1986): *The algebra of communicating processes with empty process*. Report FVI 86-01, Dept. of Computer Science, University of Amsterdam.

Modular Specifications in Process Algebra

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

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

Key Words & Phrases: process algebra, concurrency, modular algebraic specifications, export operator, union of modules, homomorphism operator, subalgebra operator, chaining operator.
Note. This paper is essentially the same as [14], except that Sections 4 (on queues) and 5 (on the CABP protocol) have been left out.

Introduction

During the last decade, a lot of research has been done on *process algebra*: the branch of theoretical computer science concerned with the modelling of concurrent systems as elements of an algebra. Besides the Calculus of Communicating Systems (CCS) of Milner [18, 19], several related formalisms have been developed, such as the theory of Communicating Sequential Processes (CSP) of Hoare [16], the Meije calculus of Austry & Boudol [1] and the Algebra of Communicating Processes (ACP) of Bergstra & Klop [8-10].

When work on process algebra started, many people hoped that it would be possible to come up, eventually, with the 'ultimate' process algebra, leading to a 'Church thesis' for concurrent computation. This process algebra, one imagined, should contain only a few fundamental operators and it should be suited to model all concurrent computational processes. Moreover there should be a calculus for this model making it possible to prove the identity of processes algebraically, thus proving correctness of implementations with respect to specifications. As far as we know, the ultimate process algebra has

not yet been found, but we will not exclude that it will be discovered in the near future.

Two things however, have become clear in the meantime: (1) it is doubtful whether algebraic system verification, as envisaged in [18], will be possible in this model, and (2) even if the ultimate process algebra exists, this certainly does not mean that all other process algebras are no longer interesting. We elaborate on this below.

A central idea in process algebra is that two processes which cannot be distinguished by observation should preferably be identified: the process semantics should be fully abstract with respect to some notion of testing (see [12, 18]). This means that the choice of a suitable process algebra may depend on the tools an environment has to distinguish between certain processes. In different applications the tools of the environment may be different, and therefore different applications may require different process algebras. A large number of process semantics are not fully abstract with respect to any (reasonable) notion of testing (bisimulation semantics and partial order semantics, for instance). Still these semantics can be very interesting because they have simple definitions or correspond to some strong operational intuition. Our hypothetical ultimate process algebra will make very few identifications, because it should be resistant against all forms of testing. Therefore not many algebraic laws will be valid in this model and algebraic system verification will presumably not be possible (specification and implementation correspond to different processes in the model).

Another factor which plays a role has to do with the operators of process algebras. For theoretical purposes it is in general desirable to work with a single, small set of fundamental operators. We doubt however that a unique optimal and minimal collection exists. What is optimal depends on the type of results one likes to prove. This becomes even more clear if we look towards practical applications. Some operators in process algebra can be used for a wide range of applications, but we agree with JIFENG & HOARE [17] that we may have to accept that each application will require derivation of specialised laws (and operators) to control its complexity.

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

This paper is about process algebras, their mutual relationships, and strategies to prove that a formula is valid in a process algebra. Still, we do not present any particular process algebra in this paper. We only define classes of models of process modules. One reason for doing this is that a detailed description of particular process algebras would make this paper too long. Another reason is that there is often no clear argument for selecting a particular process algebra. In such situations we are interested in assertions saying

that a formula is valid in all algebras satisfying a certain theory. A number of times we need results stating that some formulas *cannot* be proven from a certain module. A standard way to prove this is to give a model of the module where the formulas are not true. For this reason we will often refer to particular process algebras which have been described elsewhere in the literature.

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

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

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

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

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

It turns out that certain pairs of modules are incompatible in a very strong sense: with the combination of two modules strange and counter-intuitive identities can be derived. In BAETEN, BERGSTRA & KLOP [4], for example, it is shown that the combination of failure semantics and the priority operator is inconsistent in the sense that an identity can be derived which says that a particular process that can do a b-action after it has done an a-action, equals a process that cannot do this. Another example can be found in BERGSTRA, KLOP & OLDEROG [11], where it is pointed out that the combination of failure semantics and Koomen's Fair Abstraction Rule (KFAR) is inconsistent.

In the first section of this paper we present, 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. The axiom system ACP contains auxiliary operators $\lfloor\!\lfloor$ and $\mid$ (left-merge and communication-merge) which drastically simplify computations and have some desirable 'metamathematical' consequences (finite axiomatisability[1]; greater suitability for term rewriting analysis). These auxiliary operators can be defined in a large class of process algebras. However, it turns out that in a setting with the silent step τ the left-merge cannot be added consistently to all algebras (for instance not to the usual variants of failure semantics). Now one may think that this result means that someone who is doing failure semantics with τ's cannot profit from the nice properties of the left-merge. However, we will show in this paper that use of the module approach makes it possible to do failure semantics with τ's but still benefit from the left-merge in verifications. The idea is that verifications take place on two levels: the level of bisimulation semantics where the left-merge can be used, and a level of for instance failure semantics, where no left-merge is present. The failure model can be obtained from the bisimulation model by removing the auxiliary operators and taking a homomorphic image. Now we use the observation that certain formulas (the 'positive' ones without auxiliary operators) are preserved under this procedure. A consequence of this application is that even if bisimulation semantics is not considered to be an appropriate process semantics (since it is not fully abstract with respect to any reasonable notion of testing), it still can be useful as an expedient for proving formulas in failure semantics.
2. As already pointed out above, one would like to have, from a theoretical point of view, as few operators or combinators as possible. On the other hand, when dealing with applications, it is often very rewarding to introduce new operators. This paradox can be resolved if the new operators are definable in terms of the more elementary ones. In that case the new operators can be considered as notations which are useful, but do not complicate the underlying theory. A problem with defining operators in terms of other operators is that often auxiliary atomic actions are needed in the definition. These auxiliary actions can then not be used in any other place, because that would disturb the intended semantics of the operator. In the laws that can be derived for the defined operator, the auxiliary actions occur prominently. These 'side effects' are often quite

1. Recently, FARON MOLLER [20] from Edinburgh showed that in bisimulation semantics the merge operator cannot be finitely axiomatised without auxiliary operators.

unpleasant. One may think that side effects are unavoidable and that someone who really does not like them should define new operators directly in the algebras (even though this is in conflict with the desire to have as few operators as possible). However, we will show that the module approach can be used to solve also this problem: with the restriction operator we remove the auxiliary actions from the signature and then we apply the subalgebra operator in order to 'move' to algebras where the auxiliary actions are not present at all.

The concept of hiding auxiliary operators in a module in some formal way is quite familiar in the literature (see BERGSTRA, HEERING & KLINT [7] for example), but the use of module operators H and S, and their application in combining modules that would be incompatible otherwise, is, as far as we know, new. The H and S operations are in spirit related to the **abstract** operation of SANNELLA & WIRSING [25] and SANNELLA & TARLECKI [24], which also extends the model class of a module.

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

1. MODULE LOGIC

In this paper, as in many other papers about process algebra, we use formal calculi to prove statements about concurrent systems. In this section we answer the following questions:

- Which kind of calculi do we use?
- What do we understand by a proof?

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

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

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

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

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

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

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

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

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

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

$$(\mathbf{R}:p \subseteq S_1 \times \cdots \times S_n) \in \sigma \Rightarrow (\mathbf{S}:S_i) \in \sigma \ (i = 1,\ldots,n)$$

A function declaration $\mathbf{F}:f:\rightarrow S$ of arity 0 is sometimes called a *constant declaration* and written as $\mathbf{F}:f\in S$.

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

$(\mathbf{S}:S)\in\sigma$ to a set $S^{\mathcal{A}}$,

$(\mathbf{F}:f:S_1\times\cdots\times S_n\rightarrow S)\in\sigma$ to a function $f^{\mathcal{A}}_{S_1\times\cdots\times S_n\rightarrow S}:S_1^{\mathcal{A}}\times\cdots\times S_n^{\mathcal{A}}\rightarrow S^{\mathcal{A}}$,

$(\mathbf{R}:p\subseteq S_1\times\cdots\times S_n)\in\sigma$ to a predicate $p^{\mathcal{A}}_{S_1\times\cdots\times S_n}\subseteq S_1^{\mathcal{A}}\times\cdots\times S_n^{\mathcal{A}}$.

Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras. $\mathcal{B}$ is a *subalgebra* of $\mathcal{A}$ if $S^{\mathcal{B}}\subseteq S^{\mathcal{A}}$ for all $(\mathbf{S}:S)\in\sigma$, if moreover $f^{\mathcal{A}}_{S_1\times\cdots\times S_n\rightarrow S}$ restricted to $S_1^{\mathcal{B}}\times\cdots\times S_n^{\mathcal{B}}\rightarrow S^{\mathcal{B}}$ is just $f^{\mathcal{B}}_{S_1\times\cdots\times S_n\rightarrow S}$ for all $\mathbf{F}:f:S_1\times\cdots\times S_n\rightarrow S$ in σ, and if $p^{\mathcal{A}}_{S_1\times\cdots\times S_n}$ restricted to $S_1^{\mathcal{B}}\times\cdots\times S_n^{\mathcal{B}}$ is just $p^{\mathcal{B}}_{S_1\times\cdots\times S_n}$ for all $\mathbf{R}:p\subseteq S_1\times\cdots\times S_n$ in σ.

A *homomorphism* $h:\mathcal{A}\rightarrow\mathcal{B}$ consists of mappings $h_S:S^{\mathcal{A}}\rightarrow S^{\mathcal{B}}$ for all $\mathbf{S}:S$ in σ, such that

$$h_S(f^{\mathcal{A}}_{S_1\times\cdots\times S_n\rightarrow S}(x_1,\ldots,x_n)) = f^{\mathcal{B}}_{S_1\times\cdots\times S_n\rightarrow S}(h_{S_1}(x_1),\ldots,h_{S_n}(x_n))$$
$$\text{for all } (\mathbf{F}:f:S_1\times\cdots\times S_n\rightarrow S)\in\sigma \text{ and all } x_i\in S_i^{\mathcal{A}}(i=1,\ldots,n)$$

$$p^{\mathcal{A}}_{S_1\times\cdots\times S_n}(x_1,\ldots,x_n) \Leftrightarrow p^{\mathcal{B}}_{S_1\times\cdots\times S_n}(h_{S_1}(x_1),\ldots,h_{S_n}(x_n))$$
$$\text{for all } (\mathbf{R}:p\subseteq S_1\times\cdots\times S_n)\in\sigma \text{ and all } x_i\in S_i^{\mathcal{A}}(i=1,\ldots,n)$$

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

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

$$S^{\mathcal{B}} = S^{\mathcal{A}} \text{ for all } (\mathbf{S}:S)\in\rho\cap\sigma$$

$$f^{\mathcal{B}}_{S_1\times\cdots\times S_n\rightarrow S} = f^{\mathcal{A}}_{S_1\times\cdots\times S_n\rightarrow S} \text{ for all } (\mathbf{F}:f:S_1\times\cdots\times S_n\rightarrow S)\in\rho\cap\sigma$$

$$p^{\mathcal{B}}_{S_1\times\cdots\times S_n} = p^{\mathcal{A}}_{S_1\times\cdots\times S_n} \text{ for all } (\mathbf{R}:p\subseteq S_1\times\cdots\times S_n)\in\rho\cap\sigma$$

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

- a set $F^{\mathfrak{L}}_{\sigma}$ of *formulas* over σ such that $F^{\mathfrak{L}}_{\sigma}\cap F^{\mathfrak{L}}_{\rho}=F^{\mathfrak{L}}_{\sigma\cap\rho}$,
- a binary relation $\models^{\mathfrak{L}}_{\sigma}$ on σ-algebras$\times F^{\mathfrak{L}}_{\sigma}$ such that for all ρ-algebras $\mathcal{A}$ and $\phi\in F^{\mathfrak{L}}_{\sigma\cap\rho}$: $\sigma\square\mathcal{A}\models^{\mathfrak{L}}_{\sigma\cap\rho}\phi \Leftrightarrow \mathcal{A}\models^{\mathfrak{L}}_{\rho}\phi$
- and a set $I^{\mathfrak{L}}_{\sigma}$ of *inference rules* $\dfrac{H}{\phi}$ with $H\subseteq F^{\mathfrak{L}}_{\sigma}$ and $\phi\in F^{\mathfrak{L}}_{\sigma}$.

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

A logic $\mathfrak{L}$ is *sound* if $\frac{H}{\phi} \in I_\sigma^{\mathfrak{L}}$ implies $\mathcal{A} \models_\sigma^{\mathfrak{L}} H \Rightarrow \mathcal{A} \models_\sigma^{\mathfrak{L}} \phi$ for any σ-algebra $\mathcal{A}$.

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

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

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

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

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

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

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

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

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

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

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

Here $+$ is the composition operator, allowing to organise specifications in a modular way, and $\square$ is the export operator, restricting the visible signature of

a module, thereby hiding auxiliary items. These operators occur in some form or other frequently in the literature on software engineering. Our notation is taken from BERGSTRA, HEERING & KLINT [7] in which also additional references can be found. The homomorphism operator H and the subalgebra operator S are, as far as we know, new in the context of algebraic specifications. Of course they are well known in model theory, see for instance MONK [21].

The *visible signature* $\Sigma(M)$ of a module M is defined inductively by:
- $\Sigma(\sigma,T) = \sigma$,
- $\Sigma(M+N) = \Sigma(M) \cup \Sigma(N)$,
- $\Sigma(\sigma \Box M) = \sigma \cap \Sigma(M)$,
- $\Sigma(H(M)) = \Sigma(M)$,
- $\Sigma(S(M)) = \Sigma(M)$.

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

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

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

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

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

SOUNDNESS THEOREM. *If* $\mathfrak{L}$ *is sound then* $M \vdash^{\mathfrak{L}} \phi$ *implies* $M \models^{\mathfrak{L}} \phi$.

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

TABLE 1

PROOF. With induction. Again the only nontrivial cases are the rules for $+$ and $\Box$. These follow since for all ρ-algebras $\mathcal{A}$ and $\phi \in F^{\mathcal{E}}_{\sigma \cap \rho}$: $\sigma \Box \mathcal{A} \models \phi \Rightarrow \mathcal{A} \models \phi$ and $\sigma \Box \mathcal{A} \models \phi \Leftarrow \mathcal{A} \models \phi$ respectively. □

2. PROCESS ALGEBRA

This is not an introductory paper on process algebra. We only give a listing of some important process modules. For an introduction to the ACP formalism we refer the reader to [8-10].

2.1. ACP_τ. In this paper a central role will be played by the module ACP_τ, the Algebra of Communicating Processes with abstraction. A first parameter of ACP_τ is a finite set A of *actions*. For each action $a \in A$ there is a constant a in the language, representing the process, starting with an a-action and terminating (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 itsself, or by the environment.

We have a special constant δ, denoting *deadlock, inaction,* a process that cannot do anything at all. In particular δ does not terminate succesfully. We write $A_\delta = A \cup \{\delta\}$.

Next we have a *parallel composition* operator $\|$. $x \| y$ denotes the process corresponding to the parallel execution of x and y. Execution of $x \| y$ either starts with a step from x, or with a step from y, or with a *synchronisation* of an

action from x and an action from y. Synchronisation of actions is described by the second parameter of ACP_τ, which is is a binary communication function $\gamma: A_\delta \times A_\delta \to A_\delta$ that is commutative, associative and has δ as zero element:

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

If $\gamma(a,b)=c\neq\delta$ this means that actions a and b can synchronise. The synchronous performance of a and b is then regarded as a performance of the communication action c. Formally we should add the parameters to the name of a module: $ACP_\tau(A,\gamma)$. However, in order to keep notation simple, we will always omit the parameters if this can be done without causing confusion. In order to axiomatise the $\|$-operator we use two auxiliary operators $\lfloor\!\lfloor$ (*left-merge*) and $|$ (*communication merge*). $x \lfloor\!\lfloor y$ is $x\|y$, but with the restriction that the first step comes from x, and $x|y$ is $x\|y$ but with a synchronisation action as the first step.

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

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

In Table 2 we summarize the signature of module ACP_τ.

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

TABLE 2

Table 3 contains the theory of the module ACP_τ. In this paper we present ACP_τ as a monolithic module. In [10] however, it is shown that ACP_τ can be viewed as the sum of a large number of sub-modules which are interesting in their own right. The module consisting of axioms A1-5 only is called BPA

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

TABLE 3

(from Basic Process Algebra). If we add axioms A6-7 we obtain BPA_δ, and BPA_δ plus axioms T1-3 gives $BPA_{\tau\delta}$. The module ACP consists of the axioms A1-7, CF, CM1-9 and D1-4, i.e. the left column of Table 3. All axioms in Table 3 are in fact axiom schemes in a, b, H and I. Here a and b range over A_δ (unless further restrictions are made in the table) and $H,I\subseteq A$. In a product $x\cdot y$ we will often omit the $\cdot$. We take $\cdot$ to be more binding than other operations and $+$ to be less binding than other operations. In case we are dealing with an associative operator, we also leave out parentheses.

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

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

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

2.1.4. Monotony. Most of the operators of ACP_τ are monotonous with respect to the summand inclusion ordering. Using essentially the distributivity of the operators over $+$, one can show that if $x \subseteq y$, ACP_τ proves:
- $x+z \subseteq y+z$,
- $x \cdot z \subseteq y \cdot z$,
- $x \mathbin{\lfloor\!\lfloor} z \subseteq y \mathbin{\lfloor\!\lfloor} z$,
- $x \mid z \subseteq y \mid z$,
- $\partial_H(x) \subseteq \partial_H(y)$,
- $\tau_I(x) \subseteq \tau_I(y)$.

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

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

2.2. Standard Concurrency. Often one adds to ACP_τ the following module SC of Standard Concurrency ($a \in A_\delta$), which is parametrised by A. A proof that these axioms hold for all closed recursion-free terms can be found in [9].

SC

$(x \mathbin{\lfloor\!\lfloor} y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} (y \| z)$	SC1
$(x \mid ay) \mathbin{\lfloor\!\lfloor} z = x \mid (ay \mathbin{\lfloor\!\lfloor} z)$	SC2
$x \mid y = y \mid x$	SC3
$x \| y = y \| x$	SC4
$x \mid (y \mid z) = (x \mid y) \mid z$	SC5
$x \| (y \| z) = (x \| y) \| z$	SC6

TABLE 4

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

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

TABLE 5

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

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

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

2.4. Chaining operators. A basic situation we will encounter is one in which processes input and output values in a domain D. Often we want to 'chain' two processes in such a way that the output of the first one becomes the input of the second. In order to describe this, we define *chaining* operators $\ggg$ and $\gg$. In the process $x\ggg y$ the output of process x serves as input of process y. Operator $\gg$ is identical to operator $\ggg$, but hides in addition the communications that take place at the internal communication port. The reason for introducing two operators is a technical one: the operator $\gg$ (in which we are interested most) often leads to the possibility of an infinite sequence of internal actions corresponding to hidden synchronisations between the two arguments of the operator (a form of *unguarded recursion*, cf. Sections 2.8.1 and 2.12.1). In order to deal with such behaviours, it is useful to view $\gg$ as the composition of two operators: the $\ggg$ operator and an abstraction operator that hides the communications of $\ggg$. We will define the chaining operators in terms of the operators of ACP_τ+RN. In this way we obtain a simple, finite axiomatisation of the operators. The operator $\gg$ occurs (in a different notation) already in HOARE [15] and MILNER [18].

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

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

Here $r(d)$, $s(d)$ and $c(d)$ $(d\in D)$ are auxiliary actions which play a role in the definition of the chaining operators. The module for the chaining operators is parametrised by an action alphabet A satisfying $ch(D)\subseteq A$. The module should occur in a context with a module $\mathrm{ACP}_\tau(A,\gamma)$ where

$$range(\gamma)\cap\{\downarrow d,\uparrow d,s(d),r(d)\mid d\in D\} = \varnothing$$

and communication on $ch(D)$ is defined by

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

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

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

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

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

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

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

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

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

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

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

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

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

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

These processes can be composed using chaining operators. It is not too hard to prove:

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

Note however that we do not have

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

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

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

For each recursive specification E and $x \in V_E$, the module REC introduces a constant $<x \mid E>$, denoting the x-component of a solution of E.

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

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

$<x \mid E> = <t_x \mid E>$	REC

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

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

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

TABLE 6

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

$\mathbf{F}: \pi: \mathbb{N} \times P \to P$.

2.7. Boundedness. The predicate $B_n \subseteq P$ $(n \in \mathbb{N})$ states that the nondeterminism displayed by a process before its n^{th} atomic 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 7 $(a \in A_\delta)$. Module B is parametrised by A.

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

TABLE 7

Boundedness predicates were introduced in [13].

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

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

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

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

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

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

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

2.8.2. THEOREM *(Recursive Specification Principle* (RSP)).
$ACP_\tau + REC + PR + B + AIP^- \vdash$

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

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

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

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

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

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

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

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

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

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

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

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

We define

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

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

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

2.9.2. EXAMPLE. The transition diagram of Figure 1 represents a cluster in a recursive specification. The nodes represent variables in the recursive specification, labelled edges represent summands, and the triangles denote exits. The sets $\{1,2,3\}$, $\{4,5,6,7\}$, $\{8\}$ and $\{1,2,3,4,5,6,7,8\}$ are examples of conservative clusters. Cluster $\{1,2,3,4,5,6,7\}$ is not conservative since exit Z cannot be reached from cluster variables 4, 5, 6 and 7.

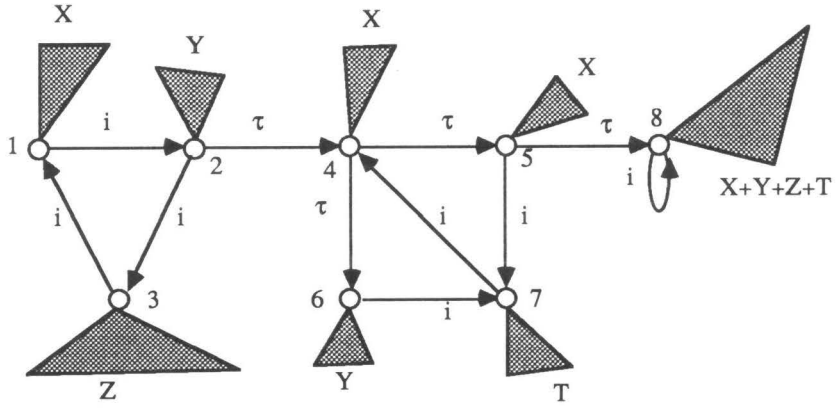

FIGURE 1

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

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

2.9.4. THEOREM. $\mathrm{ACP}_\tau + \mathrm{RN} + \mathrm{REC} + \mathrm{RSP} + \mathrm{KFAR} \vdash \mathrm{CFAR}$.
PROOF. See [26]. □

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

$$\alpha(\delta) = \varnothing$$

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

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

In this approach the question arises what axioms should be adopted for the set-operators $\cup$, $\cap$, etc. One option, which is implicitly adopted in previous papers on process algebra, is to take the equalities which are true in set theory. This collection is unstructured and too large for our purposes. Therefore we

propose a different, more algebraic solution. We view the alphabet of a process as a *process*; the alphabet operator α goes from sort P to sort P. Process $\alpha(x)$ is the alternative composition of the actions which can be performed by x. In this way we represent a set of actions by a process. A set B of actions is represented by the process expression $\boldsymbol{B}=_{def}\sum_{b\in B} b$. So the empty set is represented by δ, a singleton-set $\{a\}$ by the expression a, and a set $\{a,b\}$ by expression $a+b$. Set union corresponds to alternative composition. The process algebra axioms A1-3 and A6 correspond to similar axioms for the set union operator. The notation $\subseteq$ for summand inclusion between processes (Section 2.1.2), fits with the notation for the subset predicate on sets.

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

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

TABLE 8

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

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

TABLE 9

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

2.10.1. EXAMPLE. (from [2]). Let $p=\langle X\mid\{X=aX\}\rangle$, and define $q=\tau_{\{a\}}(p)$, $r=q\cdot b$ (with $b\neq a$). What is the alphabet of r? We derive:

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

Since

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

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

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

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

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

Information about alphabets must be available if we want to apply the following rules. These rules, which are a generalisation of the conditional axioms of [2], occur in a slightly different form also in [27]. Rules like these are an important tool in system verifications based on process algebra. Module RR is parametrised by A and γ.

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

TABLE 10

Observe that axioms AA1 and RR1 together imply axiom RN4 of Table 5. Axiom RR2, which describes the interaction between renaming and parallel composition, looks complicated, but that is only because it is so general. The axioms RR are derivable for closed recursion-free terms.

2.10.2. LEMMA: *(Conditional Axioms* (CA)*): Let CA be the theory consisting of the conditional axioms in Table 11. Then:* ACP_τ + RN + AB + RR $\vdash$ CA.

PROOF: We prove three of the rules. The others can be dealt with similarly.

CA3: Choose $a \in \alpha(x)$. Then $a \notin H$. This means that $r_{\delta,H}(a) = a$. Because a was chosen arbitrarily, we can aply rule RR1, which gives $\rho_{r_{\delta,H}}(x) = \partial_H(x) = x$.

CA5: Follows immediately from the observation

$$r_{\delta,H} = r_{\delta,H_1} \circ r_{\delta,H_2}$$

and application of axiom RN5 of Table 5.

CA1: Choose $c \in \alpha(y)$. We have:

$$r_{\delta,H}(c) = r_{\delta,H} \circ r_{\delta,H}(c)$$

Choose $b \in \alpha(x)$. If $c \notin H$ then $r_{\delta,H}(c) = c$ and the condition of rule RR2 is fulfilled. If $c \in H$ then either $\gamma(b,c)$ equals δ (so that we have

$\dfrac{\alpha(x)\mid(\alpha(y)\cap H)\subseteq H}{\partial_H(x\|y)=\partial_H(x\|\partial_H(y))}$	CA1	$\dfrac{\alpha(x)\mid(\alpha(y)\cap I)=\varnothing}{\tau_I(x\|y)=\tau_I(x\|\tau_I(y))}$	CA2
$\dfrac{\alpha(x)\cap H=\varnothing}{\partial_H(x)=x}$	CA3	$\dfrac{\alpha(x)\cap I=\varnothing}{\tau_I(x)=x}$	CA4
$\dfrac{H=H_1\cup H_2}{\partial_H(x)=\partial_{H_1}\circ\partial_{H_2}(x)}$	CA5	$\dfrac{I=I_1\cup I_2}{\tau_I(x)=\tau_{I_1}\circ\tau_{I_2}(x)}$	CA6
$\dfrac{H\cap I=\varnothing}{\tau_I\circ\partial_H(x)=\partial_H\circ\tau_I(x)}$	CA7		

TABLE 11

$r_{\delta,H}\circ\gamma(b,c)=\delta$), or $\gamma(b,c)\in H$, so that again $r_{\delta,H}\circ\gamma(b,c)=\delta$. But in case $c\in H$ we also have

$$r_{\delta,H}\circ\gamma(b,r_{\delta,H}(c))=r_{\delta,H}\circ\gamma(b,\delta)=\delta$$

This means that we can apply rule RR2. □

2.10.3. REMARK. In most of the situations where we want to apply axiom CA1, H does not contain results of communications: $(A\mid A)\cap H=\varnothing$. Further actions from $\alpha(x)$ will not communicate with actions from H. In these cases the following weakened version of axiom CA1 is already strong enough:

$$\frac{\alpha(x)\mid H=\varnothing}{\partial_H(x\|y)=\partial_H(x\|\partial_H(y))}\ \text{CA1}^{\star}$$

2.11. $\text{ACP}_\tau^\sharp$. The combination of all modules presented thus far, except for KFAR, will be called $\text{ACP}_\tau^\sharp$ (the system $\text{ACP}_\tau^\sharp$ as presented here slightly differs from a system with the same name occurring in [10]). The module is defined by:

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

Bisimulation semantics, as described in for instance [3], gives a model for the module $\text{ACP}_\tau^\sharp$ + KFAR. Work of BERGSTRA, KLOP & OLDEROG [11] showed that in a large number of interesting models KFAR is not valid. Therefore we have chosen not to include KFAR in the 'standard' module $\text{ACP}_\tau^\sharp$.

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

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

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

We have the following theorem:

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

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

2.12.3. Remarks. In the definition of the notion 'guardedly specifiable', it is essential that the identity $p = \langle Y \mid F \rangle$ is *provable*. If we would only require that $p = \langle Y \mid F \rangle$, then the corresponding version of RSP^+ would not be provable from $\mathrm{ACP}_\tau^\sharp$, since this rule would then not be valid in the action relation model of [13]. In this model we have the identity $\langle X \mid \{X = X\} \rangle = \delta$.[1] 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 expression $\langle X \mid \{X = X\} \rangle$ is guardedly specifiable. But then RSP^+ gives that for arbitrary x: $x = \langle X \mid \{X = X\} \rangle = \delta$. This is clearly false.

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

3. APPLICATIONS OF THE MODULE APPROACH IN PROCESS ALGEBRA

3.1. The auxiliary status of the left-merge.

1. Strictly speaking, this is not correct. In [13], 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 [13], then the most natural choice is to relate to $\langle X \mid E \rangle$ the bisimulation equivalence class of the term $\langle X \mid E \rangle$. Under this interpretation $\langle X \mid \{X = X\} \rangle = \delta$.

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

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

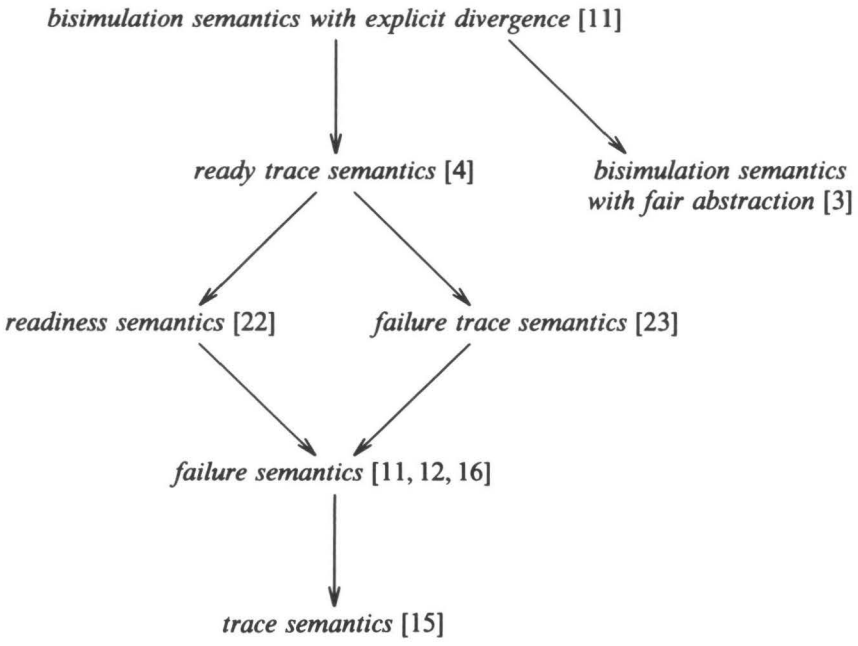

FIGURE 2. *The linear time - branching time spectrum*

If two process expressions p and q represent the same process in bisimulation

semantics with explicit divergence, they have many properties in common; if they only represent the same process in trace semantics, this only guarantees that they share some of these properties; and, descending from bisimulation semantics with explicit divergence to trace semantics, less and less distinctions are made. Now a user should state exactly in which properties of processes (s)he is interested. Suppose (s)he is only interested in traces and deadlock behaviour, then we can tell that for this purpose failure semantics suffices. This means that if processes p and q are proven equal in failure semantics, this guarantees that they have the same relevant properties. If they are only identified in trace semantics (somewhere in the lattice below failure semantics) such a conclusion cannot be drawn, but if they are identified in a semantics finer than failure semantics (such as bisimulation semantics with explicit divergence), then they certainly have the same interesting properties, and probably some uninteresting ones as well. Hence a proof in bisimulation semantics with explicit divergence is just as good as one in failure semantics (or even better).

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

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

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

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

3.1.2. An inconsistency.

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

$$M \vdash x = y,$$

the sets of complete traces agree:

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

A *complete trace* is a finite sequence of actions, ending with a symbol $\surd$ or δ indicating successful resp. unsuccessful termination. A formal definition of the set $\mathit{trace}(x)$ is given in [11]. Here we only give some examples, which should

make the notion sufficiently clear:

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

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

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

the sets of complete traces agree. The module $ACP_\tau^\sharp$ + KFAR is consistent because bisimulation semantics with fair abstraction, as described in [3], gives a consistent model for this module. However, KFAR is not valid in any of the other semantics of Figure 2.

3.1.2.2. PROPOSITION.
$ACP_\tau+T4 \vdash \tau(ac+ca)+bc = \tau(\tau(ac+ca)+bc+c(\tau a+b))$.
PROOF.

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

Proposition 3.1.2.2 shows that module $ACP_\tau+T4$ is not consistent. This sudden inconsistency must be the result of a serious misunderstanding. And indeed, what's wrong is the use of ACP_τ in the less discriminating models (say in failure semantics). It happens that, in a setting with τ, failure equivalence (or ready trace equivalence for that matter) is not a congruence for the left-merge $\between$, and this causes all the trouble.

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

Let $sacp_\tau$ be the signature obtained from $\Sigma(ACP_\tau)$ by stripping the left-merge and communication-merge:

$$sacp_\tau = \Sigma(ACP_\tau) - \{\mathbf{F} : \between : P\times P\to P, \mathbf{F} : | : P\times P\to P\}$$

Failure equivalence as in [11], etc. are congruences for the operators of $sacp_\tau$. However, the operators $\between$ and $|$ in ACP_τ are needed to axiomatise the $\|$-operator, and without them even the most elementary equations cannot be derived. Our solution to this problem is based on the following idea. Suppose we want to prove an equation $p=q$ in the signature $sacp_\tau$ that holds in ready trace semantics (and hence in failure semantics) but not in bisimulation semantics. Then we first prove an intermediate result from ACP_τ: one or more equations holding in bisimulation semantics (with explicit divergence) and in which no $\between$ and $|$ appear. This intermediate result is preserved after mapping the bisimulation model homomorphically on the ready trace or failure model, and can be combined consistently with the axiom T4. Thus the proof of $p=q$ can

be completed. In our language of modules we can describe this as follows. The module

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

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

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

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

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

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

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

3.2. Associativity of the chaining operator. ACP_τ is a universal specification formalism in the sense that in bisimulation semantics every finitely branching, effectively presented process can be specified in ACP_τ by a finite system of recursion equations (see [3]). Still it often turns out that adding new operators to the theory facilitates specification and verification of concurrent systems. In general, adding new operators and laws can have far reaching consequences for the underlying mathematical theory. Often however, new operators are *definable* in terms of others operators and the axioms are *derivable* from the other axioms. In that case the new operators can be considered as notations which are useful, but do not complicate the underlying theory in any way. Examples of definable operators are the projection operators and the process creation operator of [6].

Just like the left-merge and the communication-merge are needed in order to axiomatise the parallel composition operator, new atomic actions are often needed if we want to define a new operator in terms of more elementary operators. As an example we mention the actions $s(d)$ and $r(d)$ which we need in the definition of the chaining operators. These auxiliary atoms will never be used in process specifications. Unfortunately they have the unpleasant property that they occur in some important algebraic laws for the new operators. One of the properties of the chaining operators we use most is that they are 'associative'. However, due to the auxiliary actions, the chaining

operators are not associative in general. We do not have general associativity in the model of bisimulation semantics. Counterexample:

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

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

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

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

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

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

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

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

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

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

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

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

1. The *Fresh Atom Principle* (FAP) says that we can use new (or 'fresh') atomic actions in proofs. In [5], it is shown that FAP holds in bisimulation semantics. We have not included FAP in the theoretical framework of this paper. Therefore, if we need certain 'fresh' atoms in a proof, we have to assume that they were in the alphabet right from the beginning.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\overset{AA3,RN4}{\subseteq} \partial_H\circ\alpha\circ\rho_{\uparrow\bar{s}}(x)+\partial_H\circ\alpha\circ\rho_{\downarrow\bar{r}}(y)+\boldsymbol{B}\subseteq$

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

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

This finishes the proof of the claim.

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

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

This finishes the proof of the lemma. □

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

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

PROOF. This is essentially Theorem 1.12.2 of [27]. We give a sketch of the proof.

$$\begin{aligned}
x\ggg(y\ggg z) &= \partial_{\overline{H}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}\circ\partial_{\underline{H}}(\rho_{\uparrow\underline{s}}(y)\|\rho_{\downarrow\underline{r}}(z))) = \\
&\overset{RN5}{=} \partial_{\overline{H}}(\rho_{\uparrow\overline{s}}(x)\|\partial_{\underline{H}}\circ\rho_{\downarrow\overline{r}}(\rho_{\uparrow\underline{s}}(y)\|\rho_{\downarrow\underline{r}}(z))) = \\
&\overset{RR1}{=} \partial_{\overline{H}}\circ\partial_{\underline{H}}(\rho_{\uparrow\overline{s}}(x)\|\partial_{\underline{H}}\circ\rho_{\downarrow\overline{r}}(\rho_{\uparrow\underline{s}}(y)\|\rho_{\downarrow\underline{r}}(z))) = \\
&\overset{RR2}{=} \partial_{\overline{H}}\circ\partial_{\underline{H}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(\rho_{\uparrow\underline{s}}(y)\|\rho_{\downarrow\underline{r}}(z))) = \\
&\overset{RR2}{=} \partial_{\overline{H}}\circ\partial_{\underline{H}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(\rho_{\downarrow\overline{r}}\circ\rho_{\uparrow\underline{s}}(y)\|\rho_{\downarrow\underline{r}}(z))) = \\
&\overset{RR1}{=} \partial_{\overline{H}}\circ\partial_{\underline{H}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}\circ\rho_{\uparrow\underline{s}}(y)\|\rho_{\downarrow\underline{r}}(z)) = \\
&\overset{RN5}{=} \partial_{\underline{H}}\circ\partial_{\overline{H}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\uparrow\underline{s}}\circ\rho_{\downarrow\overline{r}}(y)\|\rho_{\downarrow\underline{r}}(z)) = \\
&\overset{RR1}{=} \partial_{\underline{H}}\circ\partial_{\overline{H}}(\rho_{\uparrow\underline{s}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\uparrow\underline{s}}\circ\rho_{\downarrow\overline{r}}(y))\|\rho_{\downarrow\underline{r}}(z)) = \\
&\overset{RR2}{=} \partial_{\underline{H}}\circ\partial_{\overline{H}}(\rho_{\uparrow\underline{s}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y))\|\rho_{\downarrow\underline{r}}(z)) = \\
&\overset{RR2}{=} \partial_{\underline{H}}\circ\partial_{\overline{H}}(\partial_{\overline{H}}\circ\rho_{\uparrow\underline{s}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y))\|\rho_{\downarrow\underline{r}}(z)) = \\
&\overset{RR1}{=} \partial_{\underline{H}}(\partial_{\overline{H}}\circ\rho_{\uparrow\underline{s}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y))\|\rho_{\downarrow\underline{r}}(z)) = \\
&\overset{RN5}{=} \partial_{\underline{H}}(\rho_{\uparrow\underline{s}}\circ\partial_{\overline{H}}(\rho_{\uparrow\overline{s}}(x)\|\rho_{\downarrow\overline{r}}(y))\|\rho_{\downarrow\underline{r}}(z)) = (x\ggg y)\ggg z
\end{aligned}$$

□

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

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

PROOF. Let $I=\{c(d)\mid d\in D\}$.

$$\begin{aligned}
x\gg(y\gg z) &\overset{CH2}{=} \tau_I(x\ggg(\tau_I(y\ggg z))) = \\
&\overset{CH1}{=} \tau_I\circ\partial_H(\rho_{\uparrow s}(x)\|\rho_{\downarrow r}\circ\tau_I(y\ggg z)) =
\end{aligned}$$

$$\overset{RN5}{=} \partial_H \circ \tau_I(\rho_{\uparrow s}(x) \| \tau_I \circ \rho_{\downarrow r}(y \ggg z)) =$$
$$\overset{RR2}{=} \partial_H \circ \tau_I(\rho_{\uparrow s}(x) \| \rho_{\downarrow r}(y \ggg z)) =$$
$$\overset{RN5}{=} \tau_I \circ \partial_H(\rho_{\uparrow s}(x) \| \rho_{\downarrow r}(y \ggg z)) =$$
$$\overset{CH1}{=} \tau_I(x \ggg (y \ggg z)) =$$
$$\overset{3.2.1.2}{=} \tau_I((x \ggg y) \ggg z) = \cdots = (x \gg y) \gg z \qquad \square$$

3.2.2. Removing auxiliary atoms. We will now apply the module approach to remove completely the auxiliary atoms which were used in the definition of the chaining operators. What we want to obtain is a module where 'inside' the auxiliary atoms are used to define the chaining operator but where 'outside' they are no longer visible and moreover chaining is associative in general. Below we will employ the notation:

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

Consider the module:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3. $SACP_\tau^\sharp$. Module $\mathrm{SACP}_\tau^\sharp$ is an 'improved' version of module $\mathrm{ACP}_\tau^\sharp$. It is defined by:

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

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

4. Conclusions and open problems

In this paper we presented a language making it possible to give modular specifications of process algebras. The language contains operations $+$ and $\Box$, which are standard in the theory of structured algebraic specifications, and moreover two new operators H and S. Two applications have been presented of the new operators: we showed how the left-merge operator can be hidden if this is needed and we described how the chaining operators can be defined in a clean way in terms of more elementary operators such that a setting is created where the chaining operators are associative. It is clear that there are many 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 [7]: 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 [7] will probably not be achievable.

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

In this paper the modules are parametrised by a set of actions. These actions themselves do not have any structure. The most natural way to look towards actions like $s1(d_0)$ however, is to see them as actions parametrised by data. We would like to include the notion of a parametrised action in our framework but it turns out that this is not trivial.

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

ACKNOWLEDGEMENTS

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

APPENDIX: LOGICS

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

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

For any $(\mathbb{S}:S) \in \sigma$ the set T_S^σ of *σ-terms* of sort S is defined inductively by:

- $x_S \in T_S^\sigma$ for any σ-variable x_S.
- If $\mathbb{F}:f:S_1 \times \cdots \times S_n \to S$ is in σ and $t_i \in T_{S_i}^\sigma$ for $i = 1,\ldots,n$ then $f_{S_1 \times \cdots \times S_n \to S}(t_1,\ldots,t_n) \in T_S^\sigma$.

The *ξ-evaluation* $[\![t]\!]^\xi \in S^{\mathcal{A}}$ of a σ-term $t \in T_S^\sigma$ in a σ-algebra $\mathcal{A}$ (with ξ a valuation) is defined by:

- $[\![x_S]\!]^\xi = \xi(x_S) \in S^{\mathcal{A}}$.
- $[\![f_{S_1 \times \cdots \times S_n \to S}(t_1,\ldots,t_n)]\!]^\xi = f^{\mathcal{A}}_{S_1 \times \cdots \times S_n \to S}([\![t_1]\!]^\xi,\ldots,[\![t_n]\!]^\xi)$.

2. Equational logic. The set F_σ^{eql} of *equations* or *equational formulas* over σ is defined by:

- If $t_i \in T_S^\sigma$ for $i=1,2$ and certain $\mathbb{S}:S$ in σ, then $(t_1=t_2)\in F_\sigma^{eql}$.

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

An inference system I_σ^{eql} for equational logic is displayed in Table 12 below. There t, u and v are terms over σ and x is a variable. Furthermore $t[u/x]$ is the result of substituting u for all occurrences of x in t. Of course u and x should be of the same sort. Finally an inference rule $\dfrac{H}{\phi}$ with $H=\varnothing$ is called an *axiom* and denoted simply by ϕ.

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

TABLE 12

3. Conditional equational logic. The set F_σ^{at} of *atomic formulas* over σ is defined by:

- If $t_i\in T_S^\sigma$ for $i=1,2$ and certain $\mathbb{S}:S$ in σ, then $(t_1=t_2)\in F_\sigma^{at}$.
- If $\mathbb{R}:p\subseteq S_1\times\cdots\times S_n$ is in σ and $t_i\in T_{S_i}^\sigma$ for $i=1,\ldots,n$ then $p_{S_1\times\cdots\times S_n}(t_1,\ldots,t_n)\in F_\sigma^{at}$.

The set F_σ^{ceql} of *conditional equational formulas* over σ is defined by:

- If $C\subseteq F_\sigma^{at}$ and $\alpha\in F_\sigma^{at}$ then $(C\Rightarrow\alpha)\in F_\sigma^{ceql}$.

The *ξ-truth* of formulas $\phi\in F_\sigma^{at}\cup F_\sigma^{ceql}$ in a σ-algebra $\mathcal{A}$ is defined by:

- $\mathcal{A},\xi\models_\sigma^{ceql} t_1=t_2$ if $[\![t_1]\!]^\xi=[\![t_2]\!]^\xi$.
- $\mathcal{A},\xi\models_\sigma^{ceql} p_{S_1\times\cdots\times S_n}(t_1,\ldots,t_n)$ if $p^{\mathcal{A}}_{S_1\times\cdots\times S_n}([\![t_1]\!]^\xi,\ldots,[\![t_n]\!]^\xi)$.
- $\mathcal{A},\xi\models_\sigma^{ceql} C\Rightarrow\alpha$ if $\mathcal{A},\xi\not\models_\sigma^{ceql}\beta$ for some $\beta\in C$ or $\mathcal{A},\xi\models_\sigma^{ceql}\alpha$.

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

An inference system I_σ^{ceql} for conditional equational logic is displayed in Table 13 below. There α and α_i are atomic formulas, C is a set of atomic formulas, ϕ is a conditional equational formula, t_i, t, u and v are terms over σ and x_i and x are variables. Furthermore $\alpha[u/x]$ is the result of substituting u for all occurrences of x in α. Of course u and x should be of the same sort. Likewise $\phi[t_i/x_i\ (i\in I)]$ is the result of simultaneous substitution for $i\in I$ of t_i for all occurrences of x_i in ϕ. An inference rule $\dfrac{\varnothing}{\phi}$ is again denoted by ϕ and a conditional equational formula $\varnothing\Rightarrow\alpha$ by α.

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

TABLE 13

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

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

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

4. First order logic. The set F_σ^{foleq} of *first order formulas with equality* over σ is defined by:

- If $t_i\in T_S^\sigma$ for $i=1,2$ and certain $\mathbb{S}:S$ in σ, then $(t_1=t_2)\in F_\sigma^{foleq}$.
- If $\mathbb{R}:p\subseteq S_1\times\cdots\times S_n$ is in σ and $t_i\in T_{S_i}^\sigma$ for $i=1,\ldots,n$ then $p_{S_1\times\cdots\times S_n}(t_1,\ldots,t_n)\in F_\sigma^{foleq}$.
- If $\phi\in F_\sigma^{foleq}$ then $\neg\phi\in F_\sigma^{foleq}$.
- If ϕ and $\psi\in F_\sigma^{foleq}$ then $(\phi\rightarrow\psi)\in F_\sigma^{foleq}$.
- If ϕ and $\psi\in F_\sigma^{foleq}$ then $(\phi\wedge\psi)\in F_\sigma^{foleq}$.
- If ϕ and $\psi\in F_\sigma^{foleq}$ then $(\phi\vee\psi)\in F_\sigma^{foleq}$.
- If ϕ and $\psi\in F_\sigma^{foleq}$ then $(\phi\leftrightarrow\psi)\in F_\sigma^{foleq}$.
- If x_S is a σ-variable and $\phi\in F_\sigma^{foleq}$ then $\forall x_S(\phi)\in F_\sigma^{foleq}$.
- If x_S is a σ-variable and $\phi\in F_\sigma^{foleq}$ then $\exists x_S(\phi)\in F_\sigma^{foleq}$.

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

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

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

An inference system I_σ^{foleq} for first order logic with equality is displayed in Table 14 below. There ϕ, ψ and ρ are elements of F_σ^{foleq}, α is an atomic formula (constructed by means of the first two clauses in the definition of F_σ^{foleq} only), t, u and v are terms over σ and x is a variable. An occurrence of a variable x in a formula ϕ is *bound* if it occurs in a subformula $\forall x(\psi)$ or $\exists x(\psi)$ of ϕ. Otherwise it is *free*. $\phi[t/x]$ denotes the result of substituting u for all free occurrences of x in t. Of course u and x should be of the same sort. Now t *is free for* x *in* ϕ if all free occurrences of variables in t remain free in $\phi[t/x]$. As before an inference rule $\dfrac{H}{\phi}$ with $H = \varnothing$ is called an *axiom* and denoted simply by ϕ.

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

TABLE 14

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

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

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

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

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 modules. Equational logic and conditional equational logic are not complete anymore and for first order logic with equality this is still an open problem (as far as we know). Here a logic $\mathfrak{L}$ is complete if $M \models^{\mathfrak{L}} \phi \Rightarrow M \vdash^{\mathfrak{L}} \phi$. It is easily shown that

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

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

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

8. Positive and universal formulas. In equational logic all formulas are both positive and universal. In conditional equational logic all formulas are universal and the positive formulas are the atomic ones. In first order logic with equality the positive formulas are the ones without the connectives $\neg$ and $\rightarrow$ and the universal ones are the formulas without quantifiers. Model theory (see for instance [21]) teaches us that a formula ϕ is preserved under homomorphisms (respectively subalgebras) iff there is a positive (respectively universal) formula ψ with $\vdash^{foleq} \psi \leftrightarrow \phi$.

References

[1] D. Austry & G. Boudol (1984): *Algèbre de processus et synchronisations.*

Theoretical Computer Science 30(1), pp. 91-131.

[2] J.C.M. BAETEN, J.A. BERGSTRA & J.W. KLOP (1987): *Conditional axioms and α/β calculus in process algebra.* In: Formal Description of Programming Concepts - III, Proceedings of the third IFIP WG 2.2 working conference, Ebberup 1986 (M. Wirsing, ed.), North-Holland, Amsterdam, pp. 53-75.

[3] J.C.M. BAETEN, J.A. BERGSTRA & J.W. KLOP (1987): *On the consistency of Koomen's fair abstraction rule.* Theoretical Computer Science 51(1/2), pp. 129-176.

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

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

[6] J.A. BERGSTRA (1985): *A process creation mechanism in process algebra.* Logic Group Preprint Series Nr. 2, CIF, State University of Utrecht, to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 81-88.

[7] J.A. BERGSTRA, J. HEERING & P. KLINT (1988): *Module algebra (revised version).* Report P8823, Programming Research Group, University of Amsterdam, to appear in: JACM. This report is a revised version of CWI Report CS-R8617, Amsterdam 1986.

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

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

[10] J.A. BERGSTRA & J.W. KLOP (1989): *Process theory based on bisimulation semantics.* In: Proceedings REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout (J.W. de Bakker, W.-P. de Roever & G. Rozenberg, eds.), LNCS 354, Springer-Verlag, pp. 50-122.

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

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

[13] R.J. VAN GLABBEEK (1987): *Bounded nondeterminism and the approximation induction principle in process algebra.* In: Proceedings STACS 87 (F.J. Brandenburg, G. Vidal-Naquet & M. Wirsing, eds.), LNCS 247, Springer-Verlag, pp. 336-347.

[14] R.J. VAN GLABBEEK & F.W. VAANDRAGER (1988): *Modular specifications in process algebra - with curious queues.* Report CS-R8821, Centrum voor

Wiskunde en Informatica, Amsterdam, extended abstract appeared in: Algebraic Methods: Theory, Tools and Applications (M. Wirsing & J.A. Bergstra, eds.), LNCS 394, Springer-Verlag, pp. 465-506.

[15] C.A.R. HOARE (1980): *Communicating sequential processes.* In: On the construction of programs - an advanced course (R.M. McKeag & A.M. Macnaghten, eds.), Cambridge University Press, pp. 229-254.

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

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

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

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

[20] F. MOLLER (1989): *Axioms for concurrency.* Ph.D. Thesis, Report CST-59-89, Department of Computer Science, University of Edinburgh.

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

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

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

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

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

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

[27] F.W. VAANDRAGER (1986): *Process algebra semantics of POOL.* Report CS-R8629, Centrum voor Wiskunde en Informatica, Amsterdam, to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 173-236.

Two Simple Protocols

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

After some introductory remarks about the specification and verification of distributed systems in the framework of process algebra, simple versions of the Alternating Bit Protocol and the Positive Acknowledgement with Retransmission protocol are discussed.

Key Words & Phrases: process algebra, concurrency, communication protocols, specification, verification, fairness, time outs.

Note. Without further reference, we will use in this paper the notation and axioms that are described in the second paper of this thesis (*Modular specifications in process algebra*).

1. General introduction

In the ACP formalism we can also define (specify) networks of processes which cooperate in an *asynchronous* way. We can do this by looking at the communication channels in the network as processes which communicate in a *synchronous* way with the processors to which they are connected. Almost always, this synchronous communication will take place according to the *handshaking paradigm*: exactly two processes participate in every communication. When we specify communications of this type we will employ a *read/send* communication function: Let $\mathbb{D}$ be a finite set of *data* which can be communicated between processes, and let $\mathbb{P}$ be a finite set of *locations* (or *ports*) where synchronous communication can take place. The alphabet of atomic actions now consists of *read actions* $rp(d)$, *send actions* $sp(d)$ and *communication* (or *synchronisation*) *actions* $cp(d)$ for $p \in \mathbb{P}$ and $d \in \mathbb{D}$. As the only synchronisations we have: $\gamma(rp(d),sp(d)) = cp(d)$.

A typical system that can be specified in this way in ACP is depicted in Figure 1. This graphical representation was first used by Jan Willem Klop. The corresponding process expression is then for instance:

$$\partial_H(P_1 \| P_2 \| P_3 \| P_4 \| P_5 \| C_1 \| C_2 \| C_3 \| C_4 \| C_5).$$

Let us consider for a moment the issue of the physical interpretation of expressions of this type and the question about the nature of the events in reality that are modelled by the read and send actions. In general we will describe with expressions P_1, P_2,... and C_1, C_2,... the behaviour of physical objects. P_1 and P_2 for example correspond with personal computers, P_3 and P_4 with disk drives and P_5 with a printer. C_1 up to C_5 describe cables of a network

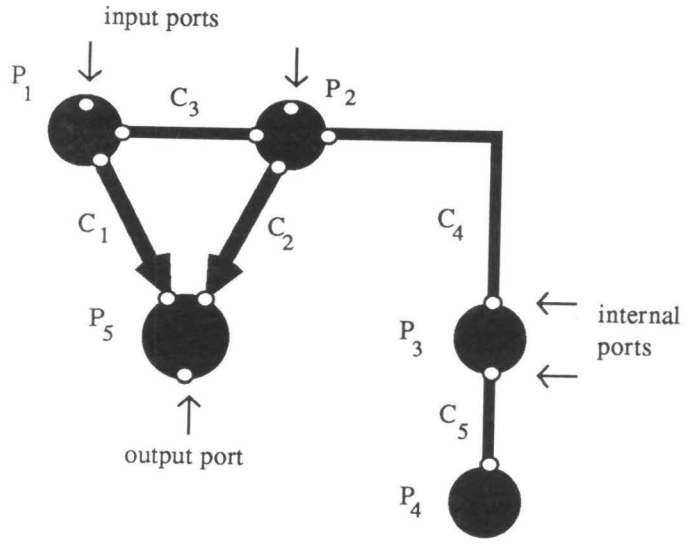

FIGURE 1

connecting all these machines together. All the components have a spatial extent. Now we associate with each port name $p \in \mathbf{P}$ a point in space on the border line between two (or more) components (see Figure 2).

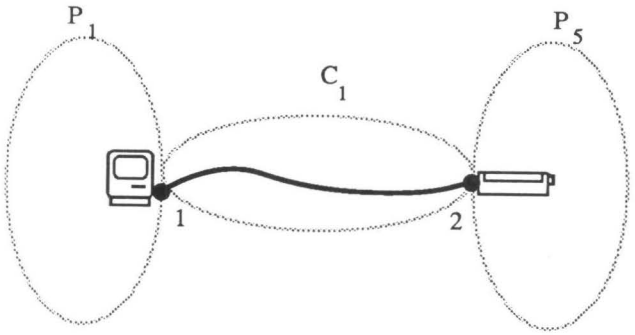

FIGURE 2

When process P_1 performs an action $s\,1(d_0)$ we relate this to the transmission of a datum d_0 by the personal computer. At the physical level this means that at the location (port 1) where cable C_1 is connected to the computer variations occur in the electric voltage during a certain amount of time. Because d_0 can have a considerable size (think of a file which is sent to the printer) the transmission can take some time. The instantaneous event associated with $r\,1(d_0)$ occurs at a moment the cable 'knows' that a datum d_0 has been transmitted at port 1. Such a moment occurs when P_1 has almost finished

transmission of this datum. The complementary event $s\,1(d_0)$ happens at the moment that the computer has made so much progress with the transmission of d_0 that the environment has enough information to 'know' that it is d_0 indeed. By defining the events in the right way we can ensure that $s\,1(d_0)$ and $r\,1(d_0)$ coincide. It is impossible that $s\,1(d_0)$ occurs and $r\,1(d_0)$ does not, or the other way around. Therefore we can consider the occurrence of $s\,1(d_0)$ and $r\,1(d_0)$ as a single event. This is precisely what we express in process algebra with the communication function and the encapsulation operator. Notice that the above interpretation of read and send actions is not in conflict with the intuition presented in [10] that the instantaneous event associated with an atomic process should be situated at the beginning of that process. Apparently a command **print**(d_0) that one can give to the computer corresponds to a process $\tau{\cdot}s\,1(d_0)$. At the moment process C_1 knows that d_0 has been transmitted and the event $c\,1(d_0)$ occurs, the execution of processes $s\,1(d_0)$ and $r\,1(d_0)$ will not yet be finished. One possible scenario is that execution of $s\,1(d_0)$ finishes before the end of the execution of $r\,1(d_0)$.

In process theory the only aspect of a system that is considered is its external behaviour. Two systems with identical external behaviour should be identified in principle. From the point of view of process algebra there is no difference between a labourer assembling bicycle pumps, and a robot performing the same job. Unless attention is paid in the formal specification to sophisticated details like fluctuations in productivity due to nocturnal excesses, the approaching weekend, depressions because of the monotony of the job, etc..

In order to realise a certain external behaviour (the *specification*), often a complex internal structure (the *implementation*) is needed. This brings us to the important issue of *abstraction*. We are interested in a technique which makes it possible to *abstract* from the internal structure of a system, so that we can derive statements about the external behaviour. Abstraction is an indispensable tool for managing the complexity of process verifications. This is because abstraction allows for a reduction of the complexity (the number of states) of subprocesses. This makes it possible to verify large processes in a *hierarchical* way. A typical verification consists of a proof that, after abstraction, an implementation *IMP* behaves like the much simpler process *SPEC* which serves as system specification:

$$ABS(IMP) = SPEC.$$

In process algebra we model abstraction by making the distinction between two types of actions, namely *external* or *observable* actions and the *internal* or *hidden* action τ, and by introducing explicit abstraction operators τ_I which transform observable actions into the hidden action (see Figure 3).

Fundamental within the ACP-formalism is the *algebraic* approach. A verification consists of a proof of a statement of the form:

$$\mathrm{ACP}_\tau + \cdots \vdash \tau_I(IMP) = SPEC.$$

The idea is that 'users' can stay in the realm of the formal system and execute algebraic manipulations, without the need for an excursion into the semantics.

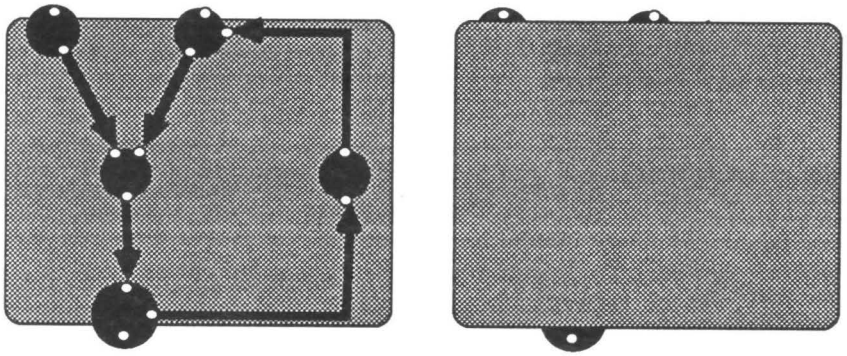

FIGURE 3

2. THE ALTERNATING BIT PROTOCOL

The most widely studied communication protocol in existence is undoubtedly the Alternating Bit Protocol (ABP, [4]). Whenever somewhere in this world someone introduces a new formalism for concurrent processes, you can count on it that the practical applicability of the formalism is illustrated by means of a specification and verification of a variant of the ABP. As a first test-case for a concurrency theory the protocol is very appropriate indeed: the protocol can be described in a few words, but its formal specification and verification constitutes a non-trivial problem. However, for real practical application of a concurrency theory much more is needed. In the analysis of realistic protocols one encounters various problems of scale which cannot be observed when dealing with the ABP.

We do not want to break with the traditions concerning the ABP, and will start here with a discussion of a simple variant of the ABP in the setting of process algebra. In the setting of process algebra, more complex protocols have been dealt with in [1, 12].

Other discussions of the Alternating Bit Protocol can be found in [4, 13, 15, 17, 20, 21]. In the context of ACP the protocol was verified for the first time in [6]. The discussion of the ABP here is based on a streamlined version of the proof, given by the author, which can be found in [8]. Variants of the ABP are discussed in the setting of process algebra in [11, 14].

2.1. Specification.

The Alternating Bit Protocol can be visualised as follows:

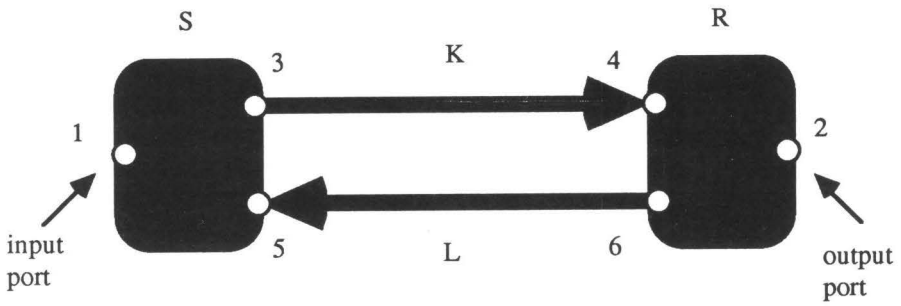

FIGURE 4

Let D be a finite set of data. Elements of D are to be transmitted by the protocol from port 1 to port 2. There are four components: a sender S, a receiver R, and two channels K and L.

2.1.1. Component S. S starts by Reading a Message (RM) at port 1. Then a *frame* consisting of the message from D and a control bit is transmitted via channel K (SF = Send Frame), until a correct acknowledgement has arrived via channel L (RA = Receive Acknowledgement). In equations we will always use the symbol d to denote elements from the set D, b denotes an element from $B = \{0,1\}$, and f finally is used for frames in $D \times B$. In Table 1 we 'declare' the recursive specification that gives the behaviour of component S. After a variable has been declared we will use it without mentioning the corresponding specification.

$$
\begin{aligned}
S &= RM^0 \\
RM^b &= \sum_{d \in D} r1(d) \cdot SF^{db} \\
SF^{db} &= s3(db) \cdot RA^{db} \\
RA^{db} &= (r5(1-b) + r5(ce)) \cdot SF^{db} + r5(b) \cdot RM^{1-b}
\end{aligned}
$$

TABLE 1. Recursive specification for component S

Graphically we can depict process S as in Figure 5. In a certain sense the figure is inaccurate: instead of a node SF^{e0} for each element e in D, there is only a single node SF^{d0}. Between each pair of nodes we draw only one edge, which however can be labelled with more than one action. Figure 5 can be considered as a 'projection' of the transition diagram belonging to S.

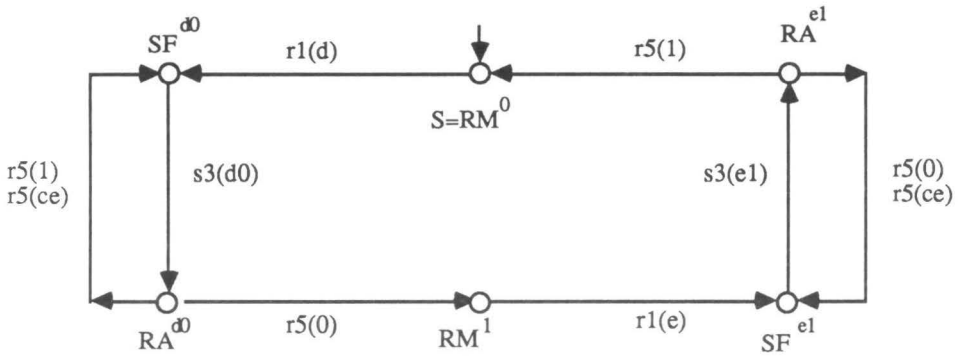

FIGURE 5

2.1.2. Component K. We assume that two things can happen if we send a frame into channel K: (1) the message is communicated correctly, (2) the message is damaged in transit. We assume that if something goes wrong with the message, the receiver hardware will detect this when it computes a *checksum* (ce = checksum error). Further the channels are supposed to be fair in the sense that they will not produce an infinite consecutive sequence of erroneous outputs. These are plausible assumptions we have to make in order to prove correctness of a protocol that is based on unreliable message passing. Data transmission channel K communicates frames in the set $D \times B$ from port 3 to 4. We give the defining equations (Table 2) and the corresponding diagram (Figure 6).

$$K = \sum_{f \in D \times B} r3(f) \cdot K^f$$

$$K^f = (\tau \cdot s4(ce) + \tau \cdot s4(f)) \cdot K$$

TABLE 2. Defining equations for channel K

The τ's in the second equation express that the choice whether or not a frame f is to be communicated correctly, is nondeterministic and cannot be influenced by one of the other components.

2.1.3. Component R. R starts by Receiving a Frame (RF) via channel K. If the control bit of the frame is correct, then the message contained in the frame is sent to port 2 (SM = Send Message). Component R Sends Acknowledgements (SA) via channel L. Figure 7 gives the transition diagram for R.

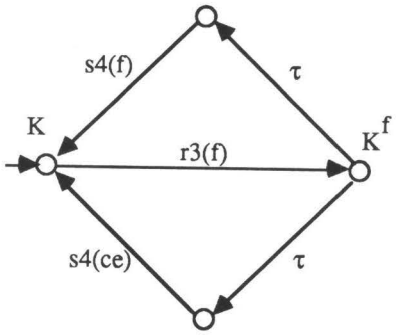

FIGURE 6

$$
\begin{aligned}
R &= RF^0 \\
RF^b &= \Big(\sum_{d\in D} r4(d(1-b))+r4(ce)\Big)\cdot SA^{1-b} + \sum_{d\in D} r4(db)\cdot SM^{db} \\
SA^b &= s6(b)\cdot RF^{1-b} \\
SM^{db} &= s2(d)\cdot SA^b
\end{aligned}
$$

TABLE 3. Recursive specification for component R

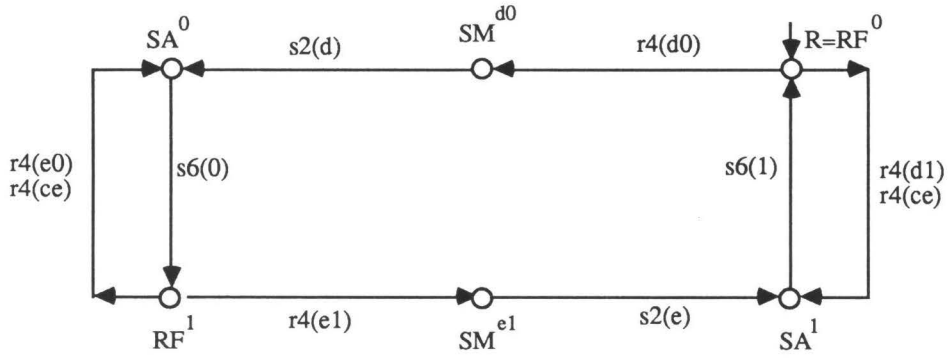

FIGURE 7

2.1.4. Component L. The task of acknowledgement transmission channel L is to communicate boolean values from R to S. The channel may yield error outputs but again we assume that this is detected, and that moreover the channel is fair. See Figure 8 for the diagram.

$$L = \sum_{b \in B} r6(b) \cdot L^b$$

$$L^b = (\tau \cdot s5(ce) + \tau \cdot s5(b)) \cdot L$$

TABLE 4. Defining equations for channel L

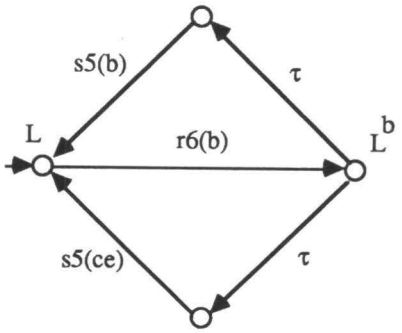

FIGURE 8

2.1.5. Sets. Define $\mathbf{D} = D \cup (D \times B) \cup B \cup \{ce\}$. $\mathbf{D}$ is the set of 'generalised' data (i.e. plain data, frames, bits, error) that occur as parameters of atomic actions. We use the notation $g \in \mathbf{D}$. The second parameter of atomic actions is the set $\mathbf{P} = \{1,2,...,6\}$ of ports. We use symbol p for elements of $\mathbf{P}$. Communication follows the read/send scheme. This leads to an alphabet

$$A = \{sp(g), rp(g), cp(g) | p \in \mathbf{P}, g \in \mathbf{D}\}$$

and communications $\gamma(sp(g), rp(g)) = cp(g)$ voor $p \in \mathbf{P}$, $g \in \mathbf{D}$. Define the following two subsets of A:

$$H = \{sp(g), rp(g) | p \in \{3,4,5,6\}, g \in \mathbf{D}\},$$

$$I = \{cp(g) | p \in \{3,4,5,6\}, g \in \mathbf{D}\}.$$

Now the ABP is described by

$$ABP = \tau_I \circ \partial_H(S \| K \| R \| L)$$

This is a correct description in the sense that the specifications of the

components S, K, R and L are guarded and consequently the specification of the ABP as a whole has a unique solution.

2.2. Verification.
Verification of the ABP amounts to a proof that:
(1) the protocol will eventually send at port 2 all and only data it has read at port 1,
(2) the protocol will output data at port 2 in the same order as it has read them at port 1.
This means that, in order to verify the protocol, it is sufficient to prove the following theorem.

2.2.1. THEOREM. $\mathrm{ACP}_\tau + \mathrm{SC} + \mathrm{REC} + \mathrm{RSP} + \mathrm{CA} + \mathrm{CFAR} \vdash$

$$ABP = \sum_{d\in D} r1(d) \cdot s2(d) \cdot ABP.$$

PROOF. Let $I' = \{cp(g) \mid p \in \{3,4,5\},\ g \in \mathbb{D}\}$. We will use $[x]$ as a notation for $\tau_{I'} \circ \partial_H(x)$. I' is defined in such a way that we can derive a guarded system of equations for $[x]$. Consider the following system of recursion equations in Table 5.

(0)	$X = X_1^0$
(1)	$X_1^b = \sum_{d\in D} r1(d) \cdot X_2^{db}$
(2)	$X_2^{db} = \tau \cdot X_3^{db} + \tau \cdot X_4^{db}$
(3)	$X_3^{db} = c6(1-b) \cdot X_2^{db}$
(4)	$X_4^{db} = s2(d) \cdot X_5^{db}$
(5)	$X_5^{db} = c6(b) \cdot X_6^{db}$
(6)	$X_6^{db} = \tau \cdot X_5^{db} + \tau \cdot X_1^{1-b}$

TABLE 5. Recursion equations for X

The transition diagram of X is displayed in Figure 9. We claim that with the above mentioned axioms one can prove that $X = [S \| K \| R \| L]$. We prove this by showing that $[S \| K \| R \| L]$ satisfies the same recursion equations (0)-(6) as X does. In the computations below, the bold-face part denotes the part of the expression currently being 'rewritten'.

$$[S \| K \| R \| L] = [RM^0 \| K \| RF^0 \| L] \qquad (0)$$

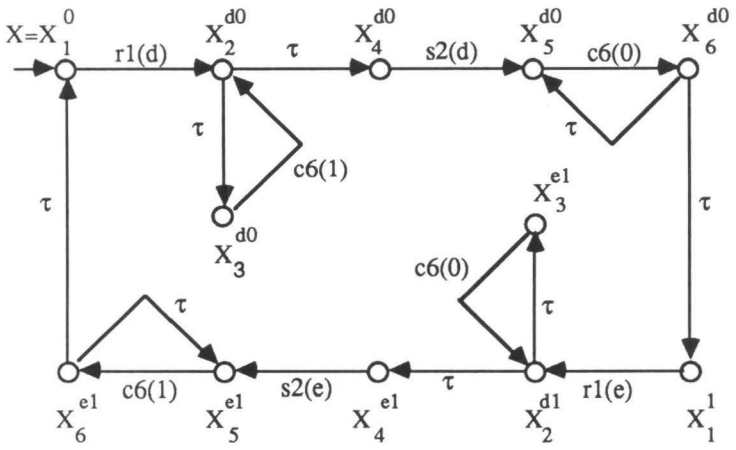

FIGURE 9

$$[\mathbf{RM^b}\|K\|RF^b\|L] = \sum_{d\in D} r1(d)\cdot[\mathbf{SF^{db}}\|\mathbf{K}\|RF^b\|L] = \tag{1}$$
$$= \sum_{d\in D} r1(d)\cdot\tau\cdot[RA^{db}\|K^{db}\|RF^b\|L] =$$
$$= \sum_{d\in D} r1(d)\cdot[RA^{db}\|K^{db}\|RF^b\|L]$$

$$[RA^{db}\|\mathbf{K^{db}}\|RF^b\|L] = \tau\cdot[RA^{db}\|\mathbf{s4(ce)\cdot K}\|\mathbf{RF^b}\|L] + \tag{2}$$
$$+ \tau\cdot[RA^{db}\|\mathbf{s4(db)\cdot K}\|\mathbf{RF^b}\|L] =$$
$$= \tau\cdot[RA^{db}\|K\|SA^{1-b}\|L] + \tau\cdot[RA^{db}\|K\|SM^{db}\|L]$$

$$[RA^{db}\|K\|\mathbf{SA^{1-b}}\|\mathbf{L}] = c6(1-b)\cdot[RA^{db}\|K\|RF^b\|\mathbf{L^{1-b}}] = \tag{3}$$
$$= c6(1-b)\cdot(\tau\cdot[\mathbf{RA^{db}}\|K\|RF^b\|\mathbf{s5(ce)\cdot L}] +$$
$$+ \tau\cdot[\mathbf{RA^{db}}\|K\|RF^b\|\mathbf{s5(1-b)\cdot L}]) =$$
$$= c6(1-b)\cdot\tau\cdot\tau\cdot[\mathbf{SF^{db}}\|\mathbf{K}\|RF^b\|L] =$$
$$= c6(1-b)\cdot\tau\cdot\tau\cdot\tau\cdot[RA^{db}\|K^{db}\|RF^b\|L] =$$
$$= c6(1-b)\cdot[RA^{db}\|K^{db}\|RF^b\|L]$$

$$[RA^{db}\|K\|\mathbf{SM^{db}}\|L] = s2(d)\cdot[RA^{db}\|K\|SA^b\|L] \tag{4}$$

$$[RA^{db}\|K\|\mathbf{SA^b}\|\mathbf{L}] = c6(b)\cdot[RA^{db}\|K\|RF^{1-b}\|L^b] \tag{5}$$

$$[RA^{db}\|K\|RF^{1-b}\|\mathbf{L^b}] = \tau\cdot[\mathbf{RA^{db}}\|K\|RF^{1-b}\|\mathbf{s5(ce)\cdot L}] + \tag{6}$$
$$+ \tau\cdot[\mathbf{RA^{db}}\|K\|RF^{1-b}\|\mathbf{s5(b)\cdot L}] =$$
$$= \tau\cdot[SF^{db}\|K\|RF^{1-b}\|L] + \tau\cdot[RM^{1-b}\|K\|RF^{1-b}\|L]$$

$$[\mathbf{SF^{db}}\|\mathbf{K}\|RF^{1-b}\|L] = \tau\cdot[RA^{db}\|\mathbf{K^{db}}\|RF^{1-b}\|L] = \tag{7}$$
$$= \tau\cdot(\tau\cdot[RA^{db}\|\mathbf{s4(ce)}\cdot\mathbf{K}\|\mathbf{RF^{1-b}}\|L] +$$
$$+ \tau\cdot[RA^{db}\|\mathbf{s4(db)}\cdot\mathbf{K}\|\mathbf{RF^{1-b}}\|L]) =$$
$$= \tau\cdot[RA^{db}\|K\|SA^{b}\|L]$$

Now substitute (7) in (6) and apply RSP. Using conditional axiom CA6 we have $ABP = \tau_I([S\|K\|R\|L]) = \tau_I(X) = \tau_I(X_1^0)$. Further, an application of CFAR gives $\tau_I(X_2^{db}) = \tau\cdot\tau_I(X_4^{db})$ and $\tau_I(X_5^{db}) = \tau\cdot\tau_I(X_1^{1-b})$. Hence,

$$\tau_I(X_1^b) = \sum_{d\in D} r1(d)\cdot\tau_I(X_2^{db}) = \sum_{d\in D} r1(d)\cdot\tau_I(X_4^{db}) =$$
$$= \sum_{d\in D} r1(d)\cdot s2(d)\cdot\tau_I(X_5^{db}) = \sum_{d\in D} r1(d)\cdot s2(d)\cdot\tau_I(X_1^{1-b})$$

and thus

$$\tau_I(X_1^0) = \sum_{d\in D} r1(d)\cdot s2(d)\cdot\sum_{e\in D} r1(e)\cdot s2(e)\cdot\tau_I(X_1^0) \quad \text{and}$$
$$\tau_I(X_1^1) = \sum_{d\in D} r1(d)\cdot s2(d)\cdot\sum_{e\in D} r1(e)\cdot s2(e)\cdot\tau_I(X_1^1).$$

Applying RSP again yields $\tau_I(X_1^0) = \tau_I(X_1^1)$ and therefore

$$\tau_I(X_1^0) = \sum_{d\in D} r1(d)\cdot s2(d)\cdot\tau_I(X_1^0).$$

This finishes the proof of the theorem. □

2.3. REMARK.
Channels K and L can contain only one datum at a time. Now one can say that this is no problem because S and R will never send a message into a channel when the previous one is still there. If S and R would do this then our process algebra modelling would be incorrect. Because they don't, there is no problem. This argument is correct for the ABP, but one should be careful in more complex situations: if one *implicitly* uses assumptions about the behaviour of a system in the specification of that system, then there is a risk that a verification shows that the system has certain 'wonderful' properties which in reality it has not. We give an example. Consider the situation where a process S first sends three threatening letters into channel K followed by a violent attempt to eliminate process R. Suppose K is a 1-datum-buffer. The system starts and S sends the first threatening letter into the channel. Now receiver R at the other side of the channel is very busy doing other things, and has no time to read messages from K. Only after a long, long time R looks if there is mail in K. Of course R is really shocked by the contents of the letter, and immediately tries to eliminate S. Only after this has succeeded, it reads from K again. Because S becomes dangerous only after the third message has been sent, process R will not get into trouble. The crucial point is now that

this would have been different if K were a FIFO-queue.

3. The PAR protocol (part 1)

In this section we will describe a protocol that is very similar to the ABP, although there is a fundamental difference. The protocol, that is described in [18], is called PAR, which stands for *Positive Acknowledgement with Retransmission*. In the protocol the sender waits for an acknowledgement before a new datum is transmitted. Instead of two different acknowledgements, like in the ABP, the PAR protocol only uses one type of acknowledgement (hence the word 'Positive'). This discussion of the PAR protocol is a revised version of Sections 3 and 4 of [19].

3.1. Specification.

The diagram that describes the architecture of the PAR protocol is almost identical to the diagram for the ABP, with as only difference that on one side of the sender a small *timer process* has been added.

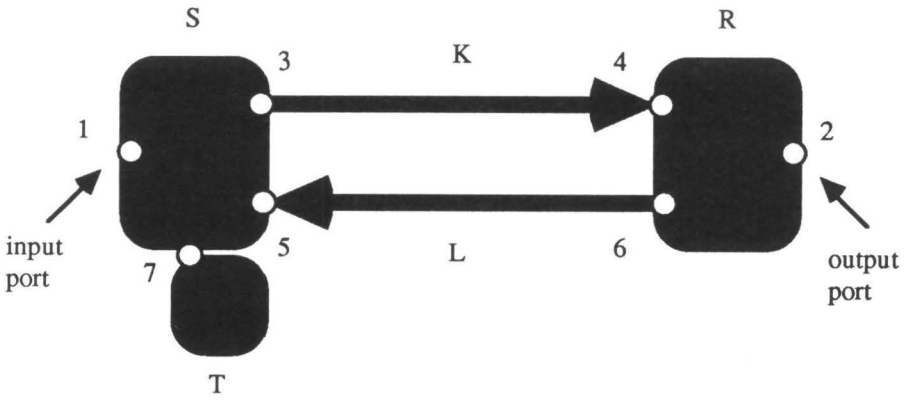

Figure 10

Thus, there are five components:
S: Sender
T: Timer
K: Data transmission channel
R: Receiver
L: Acknowledgement transmission channel

3.1.1. Sets. Let D be a finite set of data. Elements of D are to be transmitted by the PAR protocol from port 1 to port 2. Let $B=\{0,1\}$. Frames in $D\times B$ are transmitted by channel K. Define $\mathbb{D} = D\cup(D\times B)\cup\{ac,ce,st,to\}$ (ac = acknowledgement, ce = checksum error, st = start timer, to = time out). For the interaction with their environment, the components use ports from a set $\mathbb{P} = \{1,2,...,6,7\}$. $\mathbb{P}$ and $\mathbb{D}$ occur as parameters of atomic actions.

Alphabet A and communication function γ are defined using the read/send scheme. In addition, A contains two other actions i and j which do not communicate.

3.1.2. The channels. If a message is sent into channel K or L, three things can happen:
(1) the message is communicated correctly,
(2) the message is damaged in transit,
(3) the message gets lost in the channel.

Channels K and L are described by the following equations in Table 6.

K	$= \sum_{f \in D \times B} r3(f) \cdot K^f$
K^f	$= (i \cdot s4(f) + i \cdot s4(ce) + i) \cdot K$
L	$= r6(ac) \cdot L^{ac}$
L^{ac}	$= (j \cdot s5(ac) + j \cdot s5(ce) + j) \cdot L$

TABLE 6. Definition for channels K and L

The reason why we use actions i and j, instead of the τ as it was done in the specification of the ABP, will become clear further on.

3.1.3. The sender. In the specification of the sender process S (Table 7) we use formal variables RH^n, SF^{dn}, ST^{dn}, WS^{dn} $(d \in D, n \in B)$:

$RH=$ Read a message from the Host at port 1. The host process, which is not specified here, furnishes the sender with data.
$SF=$ Send a Frame in channel K at port 3.
$ST=$ Start the Timer.
$WS=$ Wait for Something to happen. Here there are three possibilities: (1) an acknowledgement frame arrives undamaged, (2) something damaged comes in, or (3) the timer goes off. If a valid acknowledgement comes in, the sender fetches the next message, and changes the control bit, otherwise a duplicate of the old frame is sent.

$$
\begin{aligned}
S &= RH^0 \\
RH^n &= \sum_{d \in D} r1(d) \cdot SF^{dn} \\
SF^{dn} &= s3(dn) \cdot ST^{dn} \\
ST^{dn} &= s7(st) \cdot WS^{dn} \\
WS^{dn} &= r5(ac) \cdot RH^{1-n} + (r5(ce) + r7(to)) \cdot SF^{dn}
\end{aligned}
$$

TABLE 7. Specification of the sender process S

3.1.4. The timer. The timer process T is very simple (see Table 8). There are two states: the initial (stop-) state and the (run-) state in which the timer is running. In both states the timer can be started, but only in the running state a time out can be generated.

$$
\begin{aligned}
T &= r7(st) \cdot T^r \\
T^r &= r7(st) \cdot T^r + s7(to) \cdot T
\end{aligned}
$$

TABLE 8. Specification of the timer process T

3.1.5. The receiver. For the specification of the receiver process R (see Table 9) we use formal variables WF^n, SA^n, SH^{dn} $(d \in D, n \in B)$:

$WF=$ Wait for the arrival of a Frame at port 4.
$SA=$ Send an Acknowledgement at port 6.
$SH=$ Send a message to the Host at port 2. In general the host of the receiver will of course be different than the host of the sender.

$$
\begin{aligned}
R &= WF^0 \\
WF^n &= r4(ce)\cdot WF^n + \sum_{d\in D} r4(d(1-n))\cdot SA^n + \sum_{d\in D} r4(dn)\cdot SH^{dn} \\
SA^n &= s6(ac)\cdot WF^n \\
SH^{dn} &= s2(d)\cdot SA^{1-n}
\end{aligned}
$$

TABLE 9. Specification of the receiver process R

When a valid frame arrives at the receiver, its control bit is checked to see if it is a duplicate. If not, it is accepted, the message contained in it is written at port 2, and an acknowledgement is generated. Duplicates and damaged frames are not written at port 2.

3.1.6. Premature time outs. We define

$$H = \{sp(g),\ rp(g) \mid p \in \{3,4,5,6,7\}, g \in \mathbb{D}\}$$

and consider the expression

$$\partial_H(S \| T \| K \| R \| L).$$

Each time after a frame is sent, the sender S starts the timer. An unpleasant property of the PAR-protocol is that a premature time out can disturb the functioning of the protocol. If the sender times out too early, while the acknowledgement is still on the way, it will send a duplicate. When the previous acknowledgement finally arrives, the sender will mistakenly think that the just sent frame is the one being acknowledged and will not realise that there is potentially another acknowledgement somewhere in the channel. If the next frame sent is lost completely, but the additional acknowledgement arrives correctly, the sender will not attempt to retransmit the lost frame, and the protocol will fail.

An important observation is that in our modelling 'too early' corresponds exactly to the availability of an alternative action. Thus we can express the desired behaviour of the timer by giving the action $c7(to)$ a *lower priority* then every other atomic action. In the next section we will elaborate on this idea.

4. PRIORITIES

The axiom system ACP_θ, introduced in [2], consists of the operators and axioms of ACP, extended with a unary *priority* operator θ, an auxiliary binary operator $\lhd$ *(unless)* and some defining axioms for these operators. We use θ to model priorities. Parameter of θ is a partial order $<$ on the atomic actions. So for $a,b,c \in A$ we have

$$\neg(a<a) \qquad \text{and} \qquad a<b \,\&\, b<c \Rightarrow a<c.$$

The constant δ can be incorporated in this ordering as a minimal element. We then have $\delta < a$ for all $a \in A$. Consider, as an example, the following partial order on atomic actions a,b and c:

$$b < a \text{ and } c < a$$

Relative to this ordering the operator θ will forbid in a sum-context all actions that are majorated by one of the other actions in that sum-context. So we have for instance:
(i) $\theta(a + b) = a$, $\theta(a + c) = a$ but
(ii) $\theta(b + c) = b + c$.
Operator θ is axiomatised in the system ACP_θ (see Table 10).

EXAMPLE. Let $b < a$ and $c < a$. Then:
(i) $\theta(a + b) = \theta(a) \lhd b + \theta(b) \lhd a = a \lhd b + b \lhd a = a + \delta = a$,
(ii) $\theta(b + c) = \theta(b) \lhd c + \theta(c) \lhd b = b \lhd c + c \lhd b = b + c$,
(iii) $\theta(b(a + c)) = \theta(b) \cdot \theta(a + c) = b \cdot (\theta(a) \lhd c + \theta(c) \lhd a) = b(a \lhd c + c \lhd a) = b(a + \delta) = ba$.

In [2] the proof can be found of the following theorem:

4.1. THEOREM.
i) *for each recursion-free closed* ACP_θ-term s *there is a basic term* t *such that* $ACP_\theta \vdash s = t$,
ii) ACP_θ *is a conservative extension of* ACP, *i.e. for all recursion-free* ACP-*terms* s,t *we have*: $ACP_\theta \vdash s = t \Rightarrow ACP \vdash s = t$.

There are some nontrivial aspects about θ which are not adressed in [2] nor in any other paper (up to now). Below we point out what the problems are and sketch the easiest way out. We think that θ is an interesting operator that deserves more attention. This paper however is not the appropriate place for large numbers of technical theorems about θ.

Firstly, it is not so clear what to think of θ operationally in a setting with recursion. Let $b < a$, $f(b) = a$ and consider the recursively defined process X:

$$X = \theta(b + \rho_f(X)).$$

It appears that X can do a b-action iff it cannot do a b-action. There are various ways to resolve this paradox. Our 'solution' is that we do not allow θ inside recursive definitions.

Although it seems possible to combine ACP_θ and ACP_τ into a system $ACP_{\tau\theta}$, the combination has not been worked out at this moment. If we require (and this seems reasonable) that $ACP_{\tau\theta}$ is a conservative extension of ACP_θ and of ACP_τ, then ($b < a$):

$$\theta(a \cdot (b + \tau \cdot (b + a))) = \theta(a \cdot (b + a)) = a \cdot a.$$

So in the scope of θ, $b + \tau(b + a)$ should not be able to do a b-step. It seems that there are two ways to achieve this operationally. The first way is to make

ACP_θ

$x+y = y+x$	A1	$a \lhd b = a$ if $\neg(a<b)$	P1
$x+(y+z) = (x+y)+z$	A2	$a \lhd b = \delta$ if $a<b$	P2
$x+x = x$	A3	$x \lhd yz = x \lhd y$	P3
$(x+y)z = xz+yz$	A4	$x \lhd (y+z) = (x \lhd y) \lhd z$	P4
$(xy)z = x(yz)$	A5	$xy \lhd z = (x \lhd z)y$	P5
$x+\delta = x$	A6	$(x+y) \lhd z = x \lhd z + y \lhd z$	P6
$\delta x = \delta$	A7		
$a \mid b = \gamma(a,b)$	CF		
$x \| y = x \mathbin{\llcorner} y + y \mathbin{\llcorner} x + x \mid y$	CM1	$\theta(a) = a$	TH1
$a \mathbin{\llcorner} x = ax$	CM2	$\theta(xy) = \theta(x) \cdot \theta(y)$	TH2
$ax \mathbin{\llcorner} y = a(x \| y)$	CM3	$\theta(x+y) = \theta(x) \lhd y + \theta(y) \lhd x$	TH3
$(x+y) \mathbin{\llcorner} z = x \mathbin{\llcorner} z + y \mathbin{\llcorner} z$	CM4		
$(ax) \mid b = (a \mid b)x$	CM5		
$a \mid (bx) = (a \mid b)x$	CM6		
$(ax) \mid (by) = (a \mid b)(x \| y)$	CM7		
$(x+y) \mid z = x \mid z + y \mid z$	CM8		
$x \mid (y+z) = x \mid y + x \mid z$	CM9		
$\partial_H(a) = a$ if $a \notin H$	D1		
$\partial_H(a) = \delta$ if $a \in H$	D2		
$\partial_H(x+y) = \partial_H(x) + \partial_H(y)$	D3		
$\partial_H(xy) = \partial_H(x) \cdot \partial_H(y)$	D4		

TABLE 10.

τ's 'transparent'. This means that in state $b+\tau(b+a)$, the process can 'see' that, after doing a τ, an a is possible. And because a has priority over b, no b will occur in state $b+\tau(b+a)$. This option is not so attractive because it means that processes have a 'look-ahead' and can see into the future. This is rather counter-intuitive.

The other possibility is to give τ priority over all other actions (in particular over b). This seems to work, even though intuition is a bit lacking. However, still there are some complications. For instance, Koomen's Fair Abstraction Rule (KFAR) is no longer valid. We have:

$$a \cdot b = \theta(a \cdot b) = \theta(a \cdot \tau_{\{i\}}(<X \mid X = i \cdot X + b>)) = a \cdot \theta(\tau_{\{i\}}(<X \mid X = i \cdot X + b>)).$$

It is not clear how, in a setting where τ has priority over all other actions, $\theta(\tau_{\{i\}}(<X \mid X = i \cdot X + b>))$ can do a b-step. Although KFAR becomes problematic, we think that $KFAR^-$ (see [9]), a variant of KFAR which allows for fair abstraction of *unstable divergence*, is valid in the new model. $KFAR^-$ is

still powerful enough for the verifications in this paper. We will not elaborate here further on models that incorporate τ and θ in a single sort. There are so many details to be worked out that it is better to have a separate paper on the topic. There is a simpler way to combine τ and θ.

Consider the model $\mathcal{A}$ for ACP of countably branching process graphs modulo strong bisimulation ([7]). Let $\mathcal{A}_\theta$ be the model obtained by adding the operators θ and $\lhd$ to $\mathcal{A}$ in the obvious way (if g is a process graph, then $\theta(g)$ is the process graph obtained from g by pruning in each node all transitions that have a label that is dominated by the label of another transition starting in that node, $\lhd$ is defined similarly). Then $\mathcal{A}_\theta$ is an *expansion* of $\mathcal{A}$, i.e. $\mathcal{A}$ is enriched with new function symbols but the domain is invariant. This means that all axioms that are valid in $\mathcal{A}$ are also valid in $\mathcal{A}_\theta$; in particular we have body replacement (REC) for all recursive specifications in the syntax of ACP. By construction recursive constants $<X|E>$ where E is a recursive specification containing θ, are not in the algebra $\mathcal{A}_\theta$. It is easily checked that the laws of ACP_θ are valid in $\mathcal{A}_\theta$ too. Next consider the model $\mathcal{A}_\tau$ for ACP_τ of countably branching process graphs modulo rooted-τ-bisimulation ([3]). Our way to combine θ and τ is that we work with a two-sorted algebra: there is a sort of ACP_τ-processes and a sort of ACP_θ-processes: we place the algebras $\mathcal{A}_\tau$ and $\mathcal{A}_\theta$ next to each other, obtaining an algebra $\mathcal{A}_{\tau\theta}$. Next we expand this algebra to an algebra $\mathcal{A}_{\tau\theta h}$ by adding a mapping h from the domain of $\mathcal{A}_\theta$ to the domain of $\mathcal{A}_\tau$ which is just the obvious isomorphic embedding of $\mathcal{A}$ into $\mathcal{A}_\tau$ (h maps the bisimulation equivalence class of a graph g to the rooted-τ-bisimulation equivalence class of g). All the laws that are valid in $\mathcal{A}_\tau$, in particular KFAR, are also valid for the sort of ACP_τ-processes of $\mathcal{A}_{\tau\theta h}$. Further, all laws valid in $\mathcal{A}_\theta$ are valid for the sort of ACP_θ-processes of $\mathcal{A}_{\tau\theta h}$. Finally we have the obvious laws stating that h gives an isomorphic embedding of $\mathcal{A}$ into $\mathcal{A}_\tau$ ($h(x)=h(y) \Rightarrow x=y$, $h(x\|y)=h(x)\|h(y)$, etc.).

Below we will not spend much effort on telling what processes are ACP_τ-processes and what processes are ACP_θ-processes. This should be clear from the context.

5. The PAR protocol (part 2)

Returning to the specification of the PAR protocol we define operator θ on the basis of the following partial ordering $<$ on A:

(1) $a<c7(st)$ for $a\in A-\{c7(st)\}$

(2) $c7(to)<a$ for $a\in A-\{c7(to)\}$

The reason for giving action $c7(to)$ a lower priority than the other actions has already been given in Section 3.1.6. In addition we have given action $c7(st)$ a higher priority than the other actions in order to express that immediately after sending a message the timer is started. This assumption is not essential for the correctness of the protocol. The system as a whole is now described by

$$\theta\circ\partial_H(S\|T\|K\|R\|L).$$

The fact that in the scope of a priority operator no τ's are allowed explains the use of i and j actions in the specification of components K and L. We are

only interested in the actions taking place at ports 1 and 2. The other actions cannot be observed.

$$I = \{cp(g) | p \in \{3,4,5,6,7\},\ g \in \mathbb{D}\} \cup \{i,j\}$$

The PAR protocol can now be described by:

$$\boxed{PAR = \tau_I \circ h \circ \theta \circ \partial_H(S \| T \| K \| R \| L)}$$

For verification of the protocol it is sufficient to prove the following theorem.

5.1. THEOREM.

$$PAR = \sum_{d \in D} r1(d) \cdot s2(d) \cdot PAR$$

PROOF. Let $I' = \{cp(g) | p \in \{4,5,7\}, g \in \mathbb{D}\} \cup \{i,j\}$. We use $[x]$ as notation for $\tau_{I'} \circ h \circ \theta \circ \partial_H(x)$. Since $I' \subseteq I$ we can apply axiom CA6:

$$PAR = \tau_I([S \| T \| K \| R \| L]).$$

In the first part of the proof we will derive a guarded system of recursion equations for the process expression $[S \| T \| K \| R \| L]$ in which only the operators $+$ and $\cdot$ occur. Thereafter, in the second part, we will abstract from the other internal actions using CFAR. Throughout the proof d ranges over D and n ranges over B. The transition diagram of $[S \| T \| K \| R \| L]$ is depicted in Figure 11.

$$[S \| T \| K \| R \| L] = [RH^0 \| T \| K \| WF^0 \| L] \tag{0}$$

$$\begin{aligned}[RH^n \| T \| K \| WF^n \| L] &= \tau_{I'} \circ h \circ \theta \circ \partial_H(RH^n \| T \| K \| WF^n \| L) = \\ &= \tau_{I'} \circ h \circ \theta(\sum_{d \in D} r1(d) \cdot \partial_H(SF^{dn} \| T \| K \| WF^n \| L)) = \\ &= \tau_{I'} \circ h(\sum_{d \in D} r1(d) \cdot \theta \circ \partial_H(SF^{dn} \| T \| K \| WF^n \| L)) = \\ &= \tau_{I'}(\sum_{d \in D} r1(d) \cdot h \circ \theta \circ \partial_H(SF^{dn} \| T \| K \| WF^n \| L)) = \\ &= \sum_{d \in D} r1(d) \cdot \tau_{I'} \circ h \circ \theta \circ \partial_H(SF^{dn} \| T \| K \| WF^n \| L) = \\ &= \sum_{d \in D} r1(d) \cdot [SF^{dn} \| T \| K \| WF^n \| L]\end{aligned} \tag{1}$$

$$\begin{aligned}[SF^{dn} \| T \| K \| WF^n \| L] &= c3(dn) \cdot [ST^{dn} \| T \| K^{dn} \| WF^n \| L] = \\ &= c3(dn) \cdot [WS^{dn} \| T^r \| K^{dn} \| WF^n \| L]\end{aligned} \tag{2}$$

(Here we used that the action $c7(st)$ has higher priority than the other

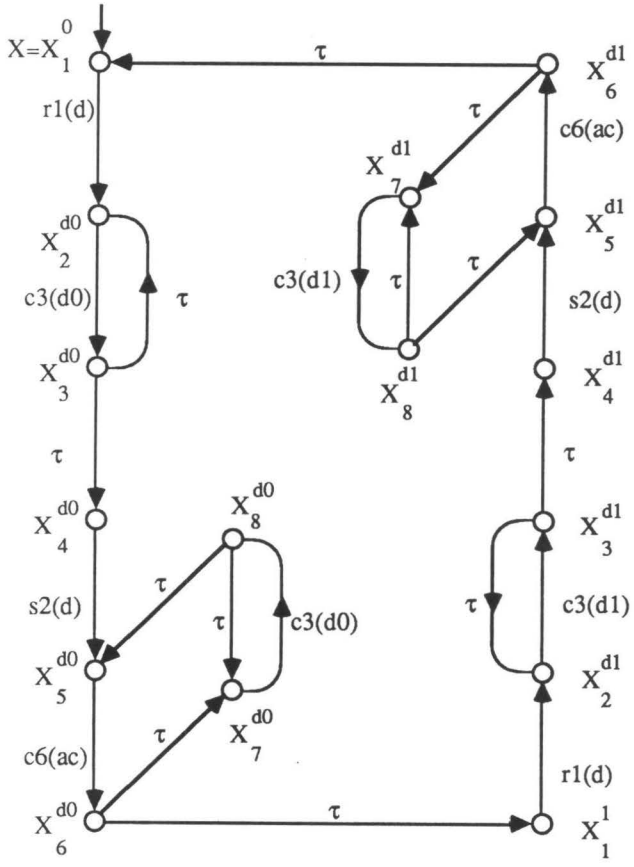

FIGURE 11

actions.)

$$[WS^{dn}\|T^r\|K^{dn}\|WF^n\|L] = \tau_{I'}\circ h\circ\theta(c7(to)\cdot\partial_H(SF^{dn}\|T\|K^{dn}\|WF^n\|L) + \quad (3)$$
$$+ i\cdot\partial_H(WS^{dn}\|T^r\|s4(dn)\cdot K\|WF^n\|L) +$$
$$+ i\cdot\partial_H(WS^{dn}\|T^r\|s4(ce)\cdot K\|WF^n\|L) +$$
$$+ i\cdot\partial_H(WS^{dn}\|T^r\|K\|WF^n\|L)\ ) =$$

(Action $c7(to)$ has lower priority than the other actions.)

$$= \tau\cdot[WS^{dn}\|T^r\|K\|SH^{dn}\|L] +$$
$$+ \tau\cdot[SF^{dn}\|T\|K\|WF^n\|L]$$

(If the message is damaged, the resulting state is the same as in the case in which the message gets lost. In both cases a time out event occurs.)

$$[WS^{dn}\|T^r\|K\|SH^{dn}\|L] = s\,2(d)\cdot[WS^{dn}\|T^r\|K\|SA^{1-n}\|L] \tag{4}$$

$$[WS^{dn}\|T^r\|K\|SA^{1-n}\|L] = c\,6(ac)\cdot[WS^{dn}\|T^r\|K\|WF^{1-n}\|L^{ac}] \tag{5}$$

$$\begin{aligned}[WS^{dn}\|T^r\|K\|WF^{1-n}\|L^{ac}] = {} & \tau\cdot[RH^{1-n}\|T^r\|K\|WF^{1-n}\|L] + \\ & + \tau\cdot[SF^{dn}\|T^r\|K\|WF^{1-n}\|L] + \\ & + \tau\cdot[SF^{dn}\|T\|K\|WF^{1-n}\|L]\end{aligned} \tag{6}$$

$$[RH^n\|T^r\|K\|WF^n\|L] = \sum_{d\in D} r\,1(d)\cdot[SF^{dn}\|T^r\|K\|WF^n\|L] \tag{1a}$$

$$[SF^{dn}\|T^r\|K\|WF^n\|L] = c\,3(dn)\cdot[WS^{dn}\|T^r\|K^{dn}\|WF^n\|L] \tag{2a}$$

$$[SF^{dn}\|T^r\|K\|WF^{1-n}\|L] = c\,3(dn)\cdot[WS^{dn}\|T^r\|K^{dn}\|WF^{1-n}\|L] \tag{7}$$

$$[SF^{dn}\|T\|K\|WF^{1-n}\|L] = c\,3(dn)\cdot[WS^{dn}\|T^r\|K^{dn}\|WF^{1-n}\|L] \tag{7a}$$

$$\begin{aligned}[WS^{dn}\|T^r\|K^{dn}\|WF^{1-n}\|L] = {} & \tau\cdot[SF^{dn}\|T^r\|K\|WF^{1-n}\|L] + \\ & + \tau\cdot[WS^{dn}\|T^r\|K\|SA^{1-n}\|L]\end{aligned} \tag{8}$$

Now observe that the processes of equations 1 and 1a, 2 and 2a, and 7 and 7a are identical. This means we have derived that X (= $[S\|T\|K\|R\|L]$) satisfies the system of recursion equations in Table 11.

(0) $X = X_1^0$	
(1) $X_1^n = \sum_{d\in D} r\,1(d)\cdot X_2^{dn}$	(5) $X_5^{dn} = c\,6(ac)\cdot X_6^{dn}$
(2) $X_2^{dn} = c\,3(dn)\cdot X_3^{dn}$	(6) $X_6^{dn} = \tau\cdot X_1^{1-n} + \tau\cdot X_7^{dn}$
(3) $X_3^{dn} = \tau\cdot X_2^{dn} + \tau\cdot X_4^{dn}$	(7) $X_7^{dn} = c\,3(dn)\cdot X_8^{dn}$
(4) $X_4^{dn} = s\,2(d)\cdot X_5^{dn}$	(8) $X_8^{dn} = \tau\cdot X_5^{dn} + \tau\cdot X_7^{dn}$

TABLE 11. Recursion equations for X

This finishes the first part of the proof. In the second part we will abstract from the communications at ports 3 and 6. Because $PAR = \tau_I(X) = \tau_I(X_1^0)$, it is enough to show that

$$\tau_I(X_1^0) = \sum_{d\in D} r\,1(d)\cdot s\,2(d)\cdot\tau_I(X_1^0).$$

For d and n fixed, variables X_2^{dn} and X_3^{dn} form a conservative cluster from I. Hence we can apply CFAR.

$$\tau_I(X_2^{dn}) = \tau\cdot\tau_I(X_4^{dn}).$$

Variables X_5^{dn}, X_6^{dn}, X_7^{dn} and X_8^{dn} (d and n fixed) also form a conservative cluster from I. CFAR gives:

$$\tau_I(X_5^{dn}) = \tau \cdot \tau_I(X_1^{1-n}).$$

We use these two results in the following derivation:

$$\begin{aligned}\tau_I(X_1^n) &= \sum_{d \in D} r1(d) \cdot \tau_I(X_2^{dn}) = \\ &= \sum_{d \in D} r1(d) \cdot \tau \cdot \tau_I(X_4^{dn}) = \\ &= \sum_{d \in D} r1(d) \cdot s2(d) \cdot \tau_I(X_5^{dn}) = \\ &= \sum_{d \in D} r1(d) \cdot s2(d) \cdot \tau_I(X_1^{1-n})\end{aligned}$$

Substituting this equation in itself gives:

$$\tau_I(X_1^0) = \sum_{d \in D} r1(d) \cdot s2(d) \cdot \sum_{e \in D} r1(e) \cdot s2(e) \cdot \tau_I(X_1^0) \quad \text{and}$$

$$\tau_I(X_1^1) = \sum_{d \in D} r1(d) \cdot s2(d) \cdot \sum_{e \in D} r1(e) \cdot s2(e) \cdot \tau_I(X_1^1).$$

Due to the Recursive Specification Principle we have:

$$\tau_I(X_1^0) = \tau_I(X_1^1).$$

Hence

$$\tau_I(X_1^0) = \sum_{d \in D} r1(d) \cdot s2(d) \cdot \tau_I(X_1^0),$$

which is the desired result. □

5.2. REMARK.
For the modelling of time outs in the PAR protocol the use of the priority operator is not essential. We sketch an alternative. If a frame gets lost in one of the channels then one can say that this event in a sense *causes* a time out. This causal relationship can be expressed in process algebra by means of a communication between the channel and the pair sender/timer. For channels K and L the specifications then become:

$$\begin{array}{|l|}\hline \overline{K} = \sum_{f \in D \times B} r3(f) \cdot \overline{K}^f \\ \overline{K}^f = (i \cdot s4(f) + i \cdot s4(ce) + i \cdot s7(to)) \cdot \overline{K} \\ \hline \overline{L} = r6(ac) \cdot \overline{L}^{ac} \\ \overline{L}^{ac} = (j \cdot s5(ac) + j \cdot s5(ce) + j \cdot s7(to)) \cdot \overline{L} \\ \hline \end{array}$$

TABLE 12. Specification for channels $\overline{K}$ and $\overline{L}$

In a time out event *three* processes participate: the timer, the sender and a channel. This means that when dealing with time outs we have ternary communication at port 7.

$$\gamma(s7(to), s7(to)) = ss7(to) \quad \gamma(s7(to), r7(to)) = sr7(to)$$
$$\gamma(s7(to), sr7(to)) = c7(to) \quad \gamma(r7(to), ss7(to)) = c7(to)$$

This leads to a slightly bigger set of unsuccessful communications:

$$\overline{H} = H \cup \{ss7(to), sr7(to)\}$$

The alternative specification of the PAR protocol now becomes:

$$\boxed{\overline{PAR} = \tau_I \circ \partial_{\overline{H}}(S \| \overline{K} \| R \| \overline{L})}$$

One can prove that $PAR = \overline{PAR}$. In [16], essentially the above idea is used to specify a simplified version of the PAR protocol.

5.3. Asymmetric communication.

Consider the situation where channel K contains a frame and the receiver is doing some other things and reads the datum from K only after a long time. Now one can consider it to be unnatural that during this whole period the datum keeps 'floating' in K and does not disappear. In a more realistic approach we would assume that if a datum is contained in channel K, either this is read by process R, or it gets lost if R is not willing to receive. Formally we can model this in process algebra by not encapsulating $s4(d)$ actions, but give them a lower priority than the corresponding $c4(d)$ actions. This mechanism is called *put mechanism* in [5]. One can prove that the ABP and the PAR protocol are correct as well if the put mechanism is used.

REFERENCES

[1] J.C.M. BAETEN (ED.) (1990): *Applications of process algebra*, to appear.
[2] J.C.M. BAETEN, J.A. BERGSTRA & J.W. KLOP (1986): *Syntax and defining*

equations for an interrupt mechanism in process algebra. Fund. Inf. IX(2), pp. 127-168.

[3] J.C.M. BAETEN, J.A. BERGSTRA & J.W. KLOP (1987): *On the consistency of Koomen's fair abstraction rule.* Theoretical Computer Science 51(1/2), pp. 129-176.

[4] K.A. BARTLETT, R.A. SCANTLEBURY & P.T. WILKINSON (1969): *A note on reliable full-duplex transmission over half-duplex links.* Communications of the ACM 12, pp. 260-261.

[5] J.A. BERGSTRA (1985): *Put and get, primitives for synchronous unreliable message passing.* Logic Group Preprint Series Nr. 3, CIF, State University of Utrecht.

[6] J.A. BERGSTRA & J.W. KLOP (1986): *Verification of an alternating bit protocol by means of process algebra.* In: Math. Methods of Spec. and Synthesis of Software Systems '85, Math. Research 31 (W. Bibel & K.P. Jantke, eds.), Akademie-Verlag, Berlin, pp. 9-23, first appeared as: Report CS-R8404, Centrum voor Wiskunde en Informatica, Amsterdam 1984.

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

[8] J.A. BERGSTRA & J.W. KLOP (1986): *Process algebra: specification and verification in bisimulation semantics.* In: Mathematics and Computer Science II, CWI Monograph 4 (M. Hazewinkel, J.K. Lenstra & L.G.L.T. Meertens, eds.), North-Holland, Amsterdam, pp. 61-94.

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

[10] R.J. VAN GLABBEEK (1987): *Bounded nondeterminism and the approximation induction principle in process algebra.* In: Proceedings STACS 87 (F.J. Brandenburg, G. Vidal-Naquet & M. Wirsing, eds.), LNCS 247, Springer-Verlag, pp. 336-347.

[11] R.J. VAN GLABBEEK & F.W. VAANDRAGER (1988): *Modular specifications in process algebra - with curious queues.* Report CS-R8821, Centrum voor Wiskunde en Informatica, Amsterdam, extended abstract appeared in: Algebraic Methods: Theory, Tools and Applications (M. Wirsing & J.A. Bergstra, eds.), LNCS 394, Springer-Verlag, pp. 465-506.

[12] R.A. GROENVELD (1987): *Verification of a sliding window protocol by means of process algebra.* Report P8701, Programming Research Group, University of Amsterdam.

[13] J.Y. HALPERN & L.D. ZUCK (1987): *A little knowledge goes a long way: simple knowledge-based derivations and correctness proofs for a family of protocols (extended abstract).* In: Proceedings 6^{th} ACM Symposium on Principles of Distributed Computing (PODC), Vancouver, Canada, August '87, pp. 269-280.

[14] C.P.J. KOYMANS & J.C. MULDER (1986): *A modular approach to protocol*

verification using process algebra. Logic Group Preprint Series Nr. 6, CIF, State University of Utrecht, to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 261-306.

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

[16] J. PARROW (1985): *Fairness properties in process algebra - with applications in communication protocol verification.* DoCS 85/03, Ph.D. Thesis, Department of Computer Systems, Uppsala University.

[17] T. STREICHER (1987): *A verification method for finite dataflow networks with constraints applied to the verification of the alternating bit protocol.* Report MIP-8706, Fakultät für Mathematik und Informatik, Universität Passau.

[18] A.S. TANENBAUM (1981): *Computer networks*, Prentice-Hall International.

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

[20] D. VERGAMINI (1986): *Verification by means of observational equivalence on automata.* Report 501, INRIA, Centre Sophia-Antipolis, Valbonne Cedex.

[21] W.P. WEIJLAND (1989): *Synchrony and asynchrony in process algebra.* Ph.D. Thesis, University of Amsterdam.

Some Observations on Redundancy in a Context

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

Let x be a process which can perform an action a when it is in state s. In this paper we consider the situation where x is placed in a context which blocks a whenever x is in s. The option of doing a in state s is *redundant* in such a context and x can be replaced by a process x' which is identical to x, except for the fact that x' cannot do a when it is in s (irrespective of the context). A simple, compositional proof technique is presented, which uses information about the traces of processes to detect redundancies in a process specification. As an illustration of the technique, a modular verification of a workcell architecture is presented.

Key Words & Phrases: process algebra, redundancy in a context, trace semantics, modularity, compositionality, computer integrated manufacturing.

Note. Without further reference, we will use in this paper the notation and axioms that are described in the second paper of this thesis (*Modular specifications in process algebra*).

1. Introduction

We are interested in the verification of distributed systems by means of algebraic manipulations. In process algebra, verifications often consist of a proof that the behaviour of an implementation $IMPL$ equals the behaviour of a specification $SPEC$, after abstraction from internal activity: $\tau_I(IMPL)=SPEC$.

The simplest strategy to prove such a statement is to derive first the transition system (process graph) for the process $IMPL$ with the expansion theorem, to apply an abstraction operator to this transition system, and then to simplify the resulting system to the system for $SPEC$ using the laws of (for instance) bisimulation semantics. This 'global' strategy however, is often not practical due to combinatorial state space explosion: the number of states of $IMPL$ can be of the same order as the product of the number of states of its components. Another serious problem with this strategy is that it provides almost no 'insight' in the structure of the system being verified. It is impossible to use the approach for the design of distributed systems, i.e. the stepwise construction of an implementation starting from a specification. This makes that there is a strong need for proof methods with a more *modular/compositional* character.

1.1. Modularity and compositionality. For the purpose of verification, we are interested in proof principles which transform a system locally, so that for a correctness proof of a local transformation one does not have to deal with the complexity of the system as a whole. A *modular* verification transforms an expression $\tau_I(IMPL)$ gradually into $SPEC$ by a sequence of local transformation steps. Consider, as an example, the case where $IMPL$ represents the parallel composition of components X_1, X_2 and X_3, where the actions in a set H have to synchronise: $IMPL = \partial_H(X_1 \| X_2 \| X_3)$. A possible step in a modular verification could be that X_1 and X_2 are replaced by Y_1 and Y_2. In that case one has to prove that:

$$\tau_I \circ \partial_H(X_1 \| X_2 \| X_3) = \tau_I \circ \partial_H(Y_1 \| Y_2 \| X_3).$$

It is sufficient to prove that $X_1 \| X_2 = Y_1 \| Y_2$. However, this will not be possible in general. It can be the case that processes $X_1 \| X_2$ and $Y_1 \| Y_2$ are only equal in the context $\tau_I \circ \partial_H(.. \| X_3)$. And even if the processes are equal, then still it is often not a good strategy to prove this. If one shows that two processes are equal, then one shows that they are interchangeable in any context, not only in the context in which they actually occur. In order to bring about successful substitutions, it is therefore desirable (or even necessary) to incorporate information about the context in which components are placed in correctness proofs of substitutions. A proof technique which allows one to do this to a sufficiently large degree is called modular. It is also possible to use a modular proof system the other way around. In that case one starts with a specification, which is refined to an implementation by a sequence of transformation steps.

A proof rule is called *compositional* if it helps to prove properties of the system as a whole from properties of the individual components. Compositional proof rules are essential for modular verifications.

In this paper we present a proof principle which can be used to enhance the modularity of verifications. We claim that the principle captures a simple intuition about the behaviour of concurrent systems, and moreover makes it possible to give short, modular proofs in quite a large number of situations.

1.2. EXAMPLE. We give a specification of a Dutch coffee machine similar to the one described in [12].

$$KM = \overline{30c} \cdot (\overline{kof} + \overline{choc}) \cdot zoem \cdot KM$$

After inserting 30 cents, the user may select 'koffie' or 'chocolade'. Dutch coffee machines make a humming sound ('zoemen') when they produce a drink. The behaviour of a typical Dutch user of such a machine can be described by the recursive equation below.

$$DU = (kof + 30c \cdot kof) \cdot talk \cdot DU$$

Dutch people are widely known for their thrift, and they will never spend 30 cents for a cup of coffee if they can get it for free.[1] Synchronisation of actions

1. Dutch users do not occur in [12]. In the modelling as presented here, the thrift of the Dutch user is not really taken into account: we can think of an environment where process DU performs

is given by: $\gamma(kof, \overline{kof}) = kof^*$, $\gamma(30c, \overline{30c}) = 30c^*$ and $\gamma(choc, \overline{choc}) = choc^*$. Let $H = \{kof, \overline{kof}, choc, \overline{choc}, 30c, \overline{30c}\}$. Consider the system $\partial_H(DU \| KM)$. It will be clear that in this environment the thrift of the Dutch user makes no sense. This behaviour is *redundant in the given context*. More 'realistic' is the behaviour $\overline{DU} = 30c \cdot kof \cdot talk \cdot \overline{DU}$, because $\partial_H(DU \| KM) = \partial_H(\overline{DU} \| KM)$.

1.3. Redundancy in a context. The example above is an instance of a situation which occurs very often: a process x has, in principle, the possibility to perform an action a when it is in state s, but is placed in an environment $\partial_H(.. \| y)$ which blocks a whenever the process is in s. In situations like this, the a-step from s is *redundant in the context* $\partial_H(.. \| y)$. We want to have the possibility to replace x by a component $\bar{x}$, that is identical to x except for the fact that $\bar{x}$ cannot do action a when it is in state s (irrespective of the context). For a compositional proof of the correctness of this type of substitutions new proof rules are needed. In this paper we will show that in most situations partial information about the *(finite, sequential) traces* of processes is sufficient to prove that a summand in a specification is redundant and can be omitted. The notion 'redundancy in a context' was introduced in [14]. The present paper can be viewed as a thorough revision of Section 6 from that paper.

1.4. Trace-specifications. It is argued by many authors (see for instance [5]), that if one is interested in program development by stepwise refinement, one needs to have the possibility of mixing programming notation with specification parts. A natural way to specify aspects of concurrent processes, advocated by [7, 12, 13, 15], is to give information about the traces, ready pairs and failure pairs of these processes. This leads to the notation

$$x \textbf{ sat } S$$

which expresses that process x satisfies property S. When we use the notation in this paper, S will always be a property of the traces of x. Without any problem we can also include other information in S but we don't need that here.

In recent years it has become abundantly clear that there are many notions of 'process'. For instance, the idea that a process, in general, *is* the set of its traces, ready pairs or failure pairs is just false, because these notions of process do not capture features like real-time and fairness. Therefore we are interested in proof rules which express 'universal' truths about processes, and which are not tied to some particular model.

The point which is new in this paper is that we use statements of the form x **sat** S, i.e. information about the traces of processes, in proofs that processes are equal in a sense different from (and finer than) trace equivalence. Thus we combine the advantages of a linear trace semantics with the distinctive power of finer equivalences.

an action $30c$ even though it has the possibility to perform an action *kof* instead. Preference of a process for certain actions can be modelled by means of the 'priority operator' of [2].

1.5. Workcell architecture. As an illustration of our technique, we present in Section 5 of this paper a specification and verification of a workcell architecture, i.e. a system consisting of a number of workcells which cooperate in order to manufacture a certain product. The verification is not only modular, but also short when compared with the non-modular verifications of the same system by BIEMANS & BLONK [4] and MAUW [11]. In the first steps of the verification we remove the redundant summands in the process specification of the workcell architecture. Often the information that some summand is redundant has some importance of its own. It allows one to replace one component by another which is simpler cq. cheaper. In our modular proof this information becomes available as a by-product.

1.6. Related work. This is not the first paper which is concerned with modular verification in the setting of process algebra. Work in this area has also been done by LARSEN & MILNER [9, 10] and KOYMANS & MULDER [8]. We think that our approach has basically two advantages when compared with this work. The first advantage is that our approach is technically speaking much simpler. People have strong intuitions concerning the trace behaviour of concurrent systems. Our proof rule makes it possible to use these intuitions quite directly in verifications. The intuitions behind the techniques of [8-10] are more involved and a lot of technical machinery is needed to formalise them. Our approach is probably less general than the approaches of [8-10], but we think that for many practical applications it can be used just as well.

The second advantage of our technique is that it is independent of the particular process semantics which is used. This in contrast to the work of [8-10], which is intrinsically tied to bisimulation semantics. In the discussion below we employ the laws of interleaved bisimulation semantics. However, we could just as well work with the laws of failure equivalence, ready equivalence or trace equivalence. Working with bisimulation semantics only makes our results stronger. We conjecture that the proof rule based on trace-specifications, as presented in this paper, also holds in partial order semantics (see [6]). Probably the correctness proof of the workcell architecture which is presented in Section 5, when reorganised a little bit, is also valid in partial order semantics. It is a potential topic for future research to substantiate these claims.

2. TRACES AND TRACE-SPECIFICATIONS

2. Traces and trace-specifications. A *trace* of a process is a finite sequence that gives a possible order in which atomic actions can be performed by that process. A trace can end with the symbol $\surd$ (pronounce 'tick'), to indicate that, after execution of the last atomic action, successful termination can occur. After some preliminary definitions we give, in Section 2.3, axioms that relate processes to trace sets.

2.1. DEFINITION.

1. For any alphabet Σ, we use Σ^* to denote the set of finite sequences over alphabet Σ. We write λ for the empty sequence and a for the sequence consisting of the single symbol $a\in\Sigma$. By $\sigma\star\sigma'$, often abbreviated as $\sigma\sigma'$, we denote the concatenation of sequences σ and σ'.
2. Let σ be a sequence and V be a set of sequences. We use the notation $\sigma\star V$ (or σV) for the set $\{\sigma\star\rho\,|\,\rho\in V\}$, and notation $V\star\sigma$ (or $V\sigma$) for the set $\{\rho\sigma\,|\,\rho\in V\}$.
3. By $\sharp\sigma$ we denote the length of a sequence σ.
4. On sequences we define a partial ordering $\leqslant$ (the *prefix ordering*) by: $\sigma\leqslant\rho$ iff, for some sequence σ', $\sigma\sigma'=\rho$. A set of sequences V is *closed under prefixing* if, for all $\sigma\leqslant\rho$, $\rho\in V$ implies that $\sigma\in V$.
5. $A_{\surd}=A\cup\{\surd\}$ is the set of atomic actions together with the termination symbol. Elements from $A^*\cup A^*\star\surd$ are called *traces* or *histories*. τ acts as the identity over $(A_{\surd})^*$ and is therefore replaced by λ when occurring in traces.
6. $\mathbf{T}$ is the set of nonempty, countable subsets of $T = A^*\cup A^*\star\surd$ which are closed under prefixing.

2.2. DEFINITION. Let $a,b\in A$, $V,W\in\mathbf{T}$, $\sigma,\sigma_1,\sigma_2\in T$. We define the following ACP-operators on trace sets:[1]

1. Sequential composition.

$$V\cdot W ::= (V\cap A^*)\cup\{\sigma_1\star\sigma_2\,|\,\sigma_1\surd\in V \text{ and } \sigma_2\in W\}.$$

2. Parallel composition. $V\|W ::= \{\sigma\,|\,\exists\sigma_1\in V,\ \sigma_2\in W:\sigma\in\sigma_1\|\sigma_2\}$. The set $\sigma_1\|\sigma_2$ of traces is defined inductively by:

$$a\sigma_1\|b\sigma_2 = \begin{cases} a(\sigma_1\|b\sigma_2)\cup b(a\sigma_1\|\sigma_2)\cup\gamma(a,b)(\sigma_1\|\sigma_2) & \text{if } \gamma(a,b)\in A \\ a(\sigma_1\|b\sigma_2)\cup b(a\sigma_1\|\sigma_2) & \text{otherwise} \end{cases}$$

$$\lambda\|a\sigma = a\sigma\|\lambda = a(\lambda\|\sigma),\quad \lambda\|\lambda = \{\lambda\},\quad \surd\|\sigma = \sigma\|\surd = \{\sigma\}.$$

   Here $\gamma:A_\delta\times A_\delta\to A_\delta$ is a given binary function which describes the synchronisation between atomic actions. γ is commutative, associative and has δ as its zero-element.
3. Encapsulation. Let $H\subseteq A$. $\partial_H(V) ::= V\cap(A_{\surd}-H)^*$.
4. Abstraction. Let $I\subseteq A$. $\tau_I(V) ::= \{\tau_I(\sigma)\,|\,\sigma\in V\}$. The function τ_I on traces is given by:

$$\tau_I(a\star\sigma) = \begin{cases} \tau_I(\sigma) & \text{if } a\in I \\ a\star\tau_I(\sigma) & \text{otherwise} \end{cases}$$

$$\tau_I(\lambda) = \lambda,\quad \tau_I(\surd) = \surd.$$

5. Renaming. Let $f:A_{\tau\delta}\to A_{\tau\delta}$ with $f(\tau)=\tau$ and $f(\delta)=\delta$. $\rho_f(V) ::=$

1. The auxiliary operator $\between$ cannot be defined on trace sets.

$\{\rho_f(\sigma) | \sigma \in V\}$. The function ρ_f on traces is given by:

$$\rho_f(a \star \sigma) = \begin{cases} f(a) \star \rho_f(\sigma) & \text{if } f(a) \neq \delta \\ \lambda & \text{otherwise} \end{cases}$$

$$\rho_f(\lambda) = \lambda, \quad \rho_f(\surd) = \surd.$$

6. Projection. Let $n \in \mathbb{N}$.

$$\pi_n(V) ::= \{\sigma \in V \cap A^* | \sharp\sigma \leqslant n\} \cup \{\sigma\surd \in V | \sharp\sigma \leqslant n\}.$$

7. Alphabets. $\alpha(V) ::= \{\alpha(\sigma) | \sigma \in V\}$. The function $\alpha: T \to Pow(A)$ is given by:

$$\alpha(a \star \sigma) = \{a\} \cup \alpha(\sigma), \quad \alpha(\lambda) = \alpha(\surd) = \varnothing.$$

2.3. The Trace Operator (TO). Let P be the sort of processes. The *trace operator* $tr: P \to \mathbf{T}$ relates to every process the set of traces that can be executed by that process. The operator satisfies the axioms of Table 1. ($a \in A$, $x,y \in P$, $H,I \subseteq A$, $f: A_{\tau\delta} \to A_{\tau\delta}$ with $f(\tau) = \tau$ and $f(\delta) = \delta$, and $n \in \mathbb{N}$)

$tr(\delta) = \{\lambda\}$	TO1	$tr(\partial_H(x)) = \partial_H(tr(x))$	TO7
$tr(\tau) = \{\lambda, \surd\}$	TO2	$tr(\tau_I(x)) = \tau_I(tr(x))$	TO8
$tr(a) = \{\lambda, a, a\surd\}$	TO3	$tr(\rho_f(x)) = \rho_f(tr(x))$	TO9
$tr(x+y) = tr(x) \cup tr(y)$	TO4	$tr(\pi_n(x)) = \pi_n(tr(x))$	TO10
$tr(x \cdot y) = tr(x) \cdot tr(y)$	TO5	$\alpha(x) = \alpha(tr(x))$	TO11
$tr(x \| y) = tr(x) \| tr(y)$	TO6		

TABLE 1. Axioms for the trace operator

When calculating with trace sets we implicitly use ZF. This means that the considerations of this paper are not of a completely algebraic nature. We restrict our attention to the models of the theory ACP_τ with recursion and auxiliary operators that can be mapped homomorphically to the trace algebra. This is no serious restriction because all 'interesting' process algebras are in this class. A similar approach is followed in [1].

2.3.1. EXAMPLES.

$$tr(x) = tr(\delta + x) = tr(\delta) \cup tr(x) = \{\lambda\} \cup tr(x). \tag{1}$$

So λ is member of the trace set of every process.

$$tr(ax) = tr(a) \cdot tr(x) = \{\lambda, a, a\surd\} \cdot tr(x) = \{\lambda, a\} \cup a \star tr(x) \overset{(1)}{=} \tag{2}$$
$$= \{\lambda\} \cup \{a\} \cup a \star (tr(x) \cup \{\lambda\}) = \{\lambda\} \cup a \star tr(x).$$

Let X be given by the recursive equation $X = aX$.

$$tr(X) = \bigcup_{n \geqslant 0} \pi_n(tr(X)) = \bigcup_{n \geqslant 0} tr(\pi_n(X)) = \bigcup_{n \geqslant 0} tr(a^n \cdot \delta) = \tag{3}$$

$$= \{\lambda\} \cup \{\lambda,a\} \cup \{\lambda,a,aa\} \cup \cdots = \{\lambda,a,aa, \cdots\}.$$

The first identity in derivation (3) follows from the structure of $\mathbf{T}$ and the definition of the π_n-operators on $\mathbf{T}$.

2.4. Trace-specifications. A *trace-specification* is a predicate. A trace-specification S describes the set of traces which, when assigned to free occurrences of a chosen variable σ of type trace in S, make the predicate true: $\{\sigma | S\}$. The syntax for trace-specifications we have in mind is a first-order language with integers, actions, traces, some simple functions like addition and multiplication, taking the i^{th} element of a trace, $\sharp\sigma$, $\rho_f(\sigma)$, equality predicates for the integers, actions and traces, and quantification over integers and traces. This syntax is almost equivalent to the syntax proposed in [12], except for the fact that we moreover have multiplication.

It is possible to define in the logic for each regular trace-language L a predicate S_L such that $L = \{\sigma | S_L\}$. In Section 4.5 it will be argued that such predicates are useful. All predicates that we will use in this paper are definable in terms of the syntax which is described informally above.

A process x *satisfies* a trace-specification S for trace variable σ, notation

$$x \ \mathbf{sat}_\sigma \ S,$$

if

$$\forall \sigma \in tr(x) : S.$$

Because in nearly all cases we will use a fixed trace-variable σ, we often omit the subscript σ and write x **sat** S. In this paper we regard x **sat** S merely as a notation. The proofs take place on the more elementary level of the *tr*-operator and trace sets. In [7] an elegant proof system is given which takes x **sat** S as a primitive notion. This system contains for instance rules like

$$\frac{x \ \mathbf{sat} \ S, \ x \ \mathbf{sat} \ S'}{x \ \mathbf{sat} \ S \wedge S'} \qquad \frac{x \ \mathbf{sat} \ S, \ S \Rightarrow S'}{x \ \mathbf{sat} \ S'}$$

2.4.1. Notation. Let $\sigma \in T$, $B \subseteq A$ and $a \in A$.

1. $\sigma \restriction B$ gives the *projection* of trace σ onto the actions of B: $\sigma \restriction B = \tau_{A-B}(\sigma)$.
2. $\sigma \downarrow a$ denotes the number of occurrences of a in σ:

$$\sigma \downarrow a = \begin{cases} \sharp(\sigma \restriction \{a\}) - 1 & \text{if } \sigma = \sigma' \surd \\ \sharp(\sigma \restriction \{a\}) & \text{otherwise} \end{cases}$$

3. Even though our trace-specification language contains no alphabet operator, we can talk about alphabets in predicates: $\alpha(\sigma) \subseteq B \Leftrightarrow \sigma \restriction B = \sigma$.

2.4.2. EXAMPLE. The coffee machine from Example 1.2 satisfies

$$KM \text{ sat } \alpha(\sigma) \subseteq \{\overline{kof}, \overline{choc}, \overline{30c}, zoem\} \wedge (\sigma\downarrow\overline{kof} \leqslant \sigma\downarrow\overline{30c}).$$

The number of cups of 'koffie' produced by the machine is always less or equal to the number of times 30 cents have been paid. The Dutch user however, takes care that never more than 30 cents are paid in advance:

$$DU \text{ sat } \alpha(\sigma) \subseteq \{kof, 30c, talk\} \wedge (\sigma\downarrow kof \geqslant (\sigma\downarrow 30c - 1)).$$

2.4.3. REMARK. Sometimes we write a specification as $S(\sigma)$, to indicate that the specification will normally contain σ as a free variable. In that case we use the notation $S(te)$ to denote the predicate obtained from $S(\sigma)$ by substituting all free occurrences of σ by an expression *te* of sort trace, perhaps after renaming of bound variables of S to avoid name clashes.

3. OBSERVABILITY AND LOCALISATION

The parallel combinator $\|$ is in some sense related to the cartesian product construction. In the graph model of [3], the set of nodes of a graph $g\|h$ is defined as the set of ordered pairs of the nodes of g and h. Still the $\|$-operator lacks an important property of cartesian products, namely the existence of projection operators. It is not possible in general to define operators l and r such that $l(x\|y)=x$ and $r(x\|y)=y$. In this section we show that, if we impose a number of constraints on the communication function, and on x and y, it becomes possible to define an operator which, given the alphabet of x, can recover x almost completely from $x\|y$:

$$\tau\cdot\rho_{H(\alpha(x))}(x\|y)\cdot\delta = \tau\cdot x\cdot\delta.$$

The conditions on x and y make that x is *observable*, the operator $\rho_{H(\alpha(x))}$ *localises* x in $x\|y$.

3.1. Communication. For the specification of distributed systems, we mostly use the read/send communication scheme, or communications of type $\gamma(kof, \overline{kof}) = kof^*$. Following [8], such communication functions will be characterised as *trijective*. The assumption that communication is trijective will simplify the discussion of this paper.

3.1.1. DEFINITION. A communication function γ is *trijective* if three pairwise disjoint subsets $R,S,C \subseteq A$ can be given, and bijections $^{\overline{}}: R\to S$ and $^{\circ}: R\to C$ such that for every $a,b,c \in A$:

$$\gamma(a,b)=c \Rightarrow (a\in R \wedge b=\overline{a} \wedge c=a^{\circ}) \vee (b\in R \wedge a=\overline{b} \wedge c=b^{\circ}).$$

In the rest of this paper we assume that communication is trijective.

3.1.2. REMARK. Observe that a trijective communication function γ satisfies the following three properties, and that each γ satisfying these properties is trijective ($a,b,c,d \in A$):

1. $\gamma(a,a)=\delta$,
2. if $\gamma(a,b)\neq\delta$ and $\gamma(a,c)\neq\delta$ then $b=c$ (γ is 'monogamous'),
3. if $\gamma(a,b)=\gamma(c,d)\neq\delta$ then $a=c$ or $a=d$ (γ is 'injective').

Observe in addition that a trijective γ satisfies $\gamma(\gamma(a,b),c)=\delta$ ('handshaking').

3.2. Observability. We are interested in the behaviour of a process x when it is placed in a context $..\|y$. In order to keep things simple, we will always choose x and y in such a way that x is observable in context with y: every action of $x\|y$ is either an action from x, or an action from y, or a synchronisation between x and y. In the last case we moreover know which action from x participates in the synchronisation. Below we give a formal definition of this notion of observability.

3.2.1. DEFINITION. Let $B \subseteq A$ be a set of atomic actions. B is called *observable* if for each triple $a,b,c \in A$ with $\gamma(a,b)=c$ at most one element of $\{a,b,c\}$ is a member of B.

Let for $A_1,A_2 \subseteq A$: $A_1|A_2 = \{\gamma(a_1,a_2) \in A \mid a_1 \in A_1, a_2 \in A_2\}$. From the fact that a set B of actions is observable, we can conclude that $B \cap B|A = \varnothing$. Because γ is injective, we know in addition that γ has an 'inverse' on $B|A$: for each $c \in B|A$, there is exactly one $b \in B$ such that an $a \in A$ exists with $\gamma(a,b) = c$. In this case we write $b = \gamma_B^{-1}(c)$.

3.2.2. DEFINITION. Let x,y be processes. Process x is called *observable in context* $..\|y$, if $\alpha(x)$ is observable, and $\alpha(y)$ is disjoint from $\alpha(x)$ and $\alpha(x)|A$.

If a process x is observable in a context $..\|y$, then one can tell for each action from $x\|y$ whether it is from x, from y, or from x and y together. In the last case one can also tell which action from x participates in the communication. Observe that the fact that x is observable in context $..\|y$ does not imply that y is observable in context $..\|x$.

3.3. Localisation. The 'localisation' of actions from x in a context $..\|y$ as described informally above, can be expressed formally by means of renaming operators. In the literature other definitions of the notions observability and localisation can be found (see [1] and [14]). In the choice of the definitions, there is a trade-off between the degree of generality (the capability of operators to localise actions) and the length of the definitions.

3.3.1. DEFINITION. Let $B \subseteq A$ be observable. The *localisation function* $\nu(B): A_{\tau\delta} \to A_{\tau\delta}$ is the renaming function defined by:

$$\nu(B)(a) = \begin{cases} a & \text{if } a \in B \cup \{\tau, \delta\} \\ \gamma_B^{-1}(a) & \text{if } a \in B \mid A \\ \tau & \text{otherwise} \end{cases}$$

3.3.2. EXAMPLE. The communication function in Example 1.2 is trijective. Furthermore $\alpha(DU) = \{kof, 30c, talk\}$ is observable. Process DU is observable in the context $..\|KM$. The expression

$$\rho_{\nu(\alpha(DU))} \circ \partial_H(DU \| KM)$$

denotes the process corresponding to the behaviour of the Dutch user in a context $\partial_H(..\|KM)$. We derive:

$$\begin{aligned} \rho_{\nu(\alpha(DU))} \circ \partial_H(DU \| KM) &= \\ &= \rho_{\nu(\alpha(DU))}(30c^* \cdot kof^* \cdot (talk \cdot zoem + zoem \cdot talk) \cdot \partial_H(DU \| KM)) = \\ &= 30c \cdot kof \cdot (talk \cdot \tau + \tau \cdot talk) \cdot \rho_{\nu(\alpha(DU))} \circ \partial_H(DU \| KM) = \\ &= 30c \cdot kof \cdot talk \cdot \rho_{\nu(\alpha(DU))} \circ \partial_H(DU \| KM) \end{aligned}$$

Hence $\rho_{\nu(\alpha(DU))} \circ \partial_H(DU \| KM)$ and $\overline{DU}$ satisfy the same guarded recursion equation. Application of the *Recursive Specification Principle (RSP)* now gives that both processes are equal.

3.3.3. REMARK. It may seem that one needs the τ-law T2 ($\tau x = \tau x + x$) in the verification above. Surprisingly we can perform the verification using only the τ-law T1 ($x\tau = x$):

$$kof \cdot (talk \cdot \tau + \tau \cdot talk) = kof \cdot (\tau \| talk) = kof \cdot \tau \between talk = kof \between talk = kof \cdot talk.$$

In fact we claim that all the verifications in this paper can be done using the τ-law T1 only. So we also do not need the law T3 ($a(\tau x + y) = a(\tau x + y) + ax$).

3.3.4. THEOREM. *Let p,q be closed terms with p observable in context $..\|q$. Then* $\mathrm{ACP}_\tau + \mathrm{RN} + \mathrm{AB} \vdash \tau \cdot \rho_{\nu(\alpha(p))}(p \| q) \cdot \delta = \tau \cdot p \cdot \delta$.
PROOF. Easy with induction on the structure of p and q. □

3.3.5. THEOREM. *Let x,y be processes, with x observable in context $..\|y$. Then we can prove using the axioms TO that: $tr(\rho_{\nu(\alpha(x))}(x \| y)) \subseteq tr(x)$.*
PROOF. Using the axioms from Table 1, we rewrite the statement we have to prove into:

$$\rho_{\nu(\alpha(tr(x)))}(tr(x) \| tr(y)) \subseteq tr(x).$$

Because $tr(x), tr(y) \in \mathbb{T}$, it is sufficient to prove that for every $V, W \in \mathbb{T}$ with $\alpha(V)$ observable and $\alpha(W)$ disjoint from $\alpha(V)$ and $\alpha(V) \mid A$:

$$\rho_{\nu(\alpha(V))}(V \| W) \subseteq V.$$

First we apply the definition of the merge-operator on trace sets:

$$\rho_{\nu(\alpha(V))}(V \| W) = \rho_{\nu(\alpha(V))}(\{\sigma \mid \exists v \in V,\ w \in W : \sigma \in v \| w\}).$$

The theorem is proved if we show for all $v \in V$ and $w \in W$ that:

$$\rho_{\nu(\alpha(V))}(v \| w) \subseteq V.$$

We prove a slightly stronger fact: Let $v = v_1 * v_2 \in V$ and let $w \in W$. Then:

$$v_1 * \rho_{\nu(\alpha(V))}(v_2 \| w) \subseteq V.$$

The proof goes by means of simultaneous induction on the structure of v_2 and w.

Case 1: $v_2 = \surd$

$$v_1 * \rho_{\nu(\alpha(V))}(\surd \| w) = v_1 * \rho_{\nu(\alpha(V))}(\{w\}) = v_1 * \{\rho_{\nu(\alpha(V))}(w)\} \subseteq v_1 * \{\lambda, \surd\} \subseteq V$$

Here we use that V is closed under prefixing.

Case 2: $w = \surd$

$$v_1 * \rho_{\nu(\alpha(V))}(v_2 \| \surd) = v_1 * \rho_{\nu(\alpha(V))}(\{v_2\}) = v_1 * \{\rho_{\nu(\alpha(V))}(v_2)\} = v_1 * \{v_2\} = \{v\} \subseteq V$$

Case 3.1: $v_2 = \lambda$ en $w \in A^*$

$$v_1 * \rho_{\nu(\alpha(V))}(\lambda \| w) = v_1 * \rho_{\nu(\alpha(V))}(\{w\}) = v_1 * \{\rho_{\nu(\alpha(V))}(w)\} = v_1 * \{\lambda\} = \{v\} \subseteq V$$

Case 3.2: $v_2 = \lambda$ en $w = w_1 \surd$

$$v_1 * \rho_{\nu(\alpha(V))}(\lambda \| w_1 \surd) = v_1 * \rho_{\nu(\alpha(V))}(\{w_1\}) = v_1 * \{\rho_{\nu(\alpha(V))}(w_1)\} = v_1 * \{\lambda\} = \{v\} \subseteq V$$

Case 4.1: $v_2 \in A^*$ en $w = \lambda$

$$v_1 * \rho_{\nu(\alpha(V))}(v_2 \| \lambda) = v_1 * \rho_{\nu(\alpha(V))}(\{v_2\}) = v_1 * \{\rho_{\nu(\alpha(V))}(v_2)\} = v_1 * \{v_2\} = \{v\} \subseteq V$$

Case 4.2: $v_2 = v_3 \surd$ en $w = \lambda$

$$v_1 * \rho_{\nu(\alpha(V))}(v_3 \surd \| \lambda) = v_1 * \rho_{\nu(\alpha(V))}(\{v_3\}) = v_1 * \{\rho_{\nu(\alpha(V))}(v_3)\} = v_1 * \{v_3\} = \{v_1 * v_3\} \subseteq V$$

(V is closed under prefixing)

Case 5.1: $v_2 = av_3$, $w = bw_1$ en $\gamma(a,b) = \delta$

$$\begin{aligned} v_1 * \rho_{\nu(\alpha(V))}(av_3 \| bw_1) &= v_1 * \rho_{\nu(\alpha(V))}(a(v_3 \| bw_1) \cup b(av_3 \| w_1)) = \\ &= v_1 * a * \rho_{\nu(\alpha(V))}(v_3 \| bw_1) \cup v_1 * \rho_{\nu(\alpha(V))}(av_3 \| w_1) \subseteq V \end{aligned}$$

(Apply induction hypothesis)

Case 5.2: $v_2 = av_3$, $w = bw_1$ en $\gamma(a,b) \in A$

$$\begin{aligned} v_1 * \rho_{\nu(\alpha(V))}(av_3 \| bw_1) &= v_1 * \rho_{\nu(\alpha(V))}(a(v_3 \| bw_1) \cup b(av_3 \| w_1) \cup \gamma(a,b)(v_3 \| w_1)) = \\ &= v_1 * a * \rho_{\nu(\alpha(V))}(v_3 \| bw_1) \cup v_1 * \rho_{\nu(\alpha(V))}(av_3 \| w_1) \cup \\ &\quad \cup v_1 * a * \rho_{\nu(\alpha(V))}(v_3 \| w_1) \subseteq V \end{aligned}$$

(Apply induction hypothesis) □

Notice that the $\subseteq$-sign in Theorem 3.3.5 cannot be changed into an $=$-sign. If $tr(y)$ contains no traces ending on $\surd$, then $tr(\rho_{\nu(\alpha(x))}(x\|y)$ will also contain no such traces, even if they are in $tr(x)$.

3.3.6. THEOREM. *Let x,y be processes, with x observable in context ..‖y, and let $H \subseteq A$. Then we can prove using the axioms TO that: $tr(\rho_{\nu(\alpha(x))} \circ \partial_H(x\|y)) \subseteq tr(x)$.*
PROOF. Just like we did in the proof of Theorem 3.3.5, we reformulate the statement. Let $V, W \in \mathbf{T}$ with $\alpha(V)$ observable, and $\alpha(W)$ disjoint from $\alpha(V)$ and $\alpha(V)|A$. We have to prove:

$$\rho_{\nu(\alpha(V))} \circ \partial_H(V\|W) \subseteq V.$$

For $X, Y \in \mathbf{T}$ we have that $\partial_H(X) \subseteq X$ and $X \subseteq Y \Rightarrow \rho_f(X) \subseteq \rho_f(Y)$. Hence

$$\rho_{\nu(\alpha(V))} \circ \partial_H(V\|W) \subseteq \rho_{\nu(\alpha(V))}(V\|W).$$

From the proof of Theorem 3.3.5 we conclude:

$$\rho_{\nu(\alpha(V))}(V\|W) \subseteq V. \qquad \square$$

The following corollary of Theorem 3.3.6 plays an important role in this paper because it allows us to derive a property of a system as a whole from a property of a component (this is the essence of compositionality).

3.3.7. COROLLARY. *Let x,y be processes, with x observable in context ..‖y, let $H \subseteq A$ and suppose $f = \nu(\alpha(x))$. If x* **sat** *$S(\sigma)$, then:*

$$\rho_f \circ \partial_H(x\|y) \textbf{ sat } S(\sigma)$$

and consequently

$$\partial_H(x\|y) \textbf{ sat } S(\rho_f(\sigma)).$$

3.4. REMARK. The formal definitions of the notions 'observable' and 'localisation' in this section are quite complex. The definitions are much simpler if one works with the synchronisation-merges $\|_A$ of OLDEROG & HOARE [13] instead of the parallel combinator $\|$ of ACP. In fact the whole discussion of this paper can be simplified considerably if one uses $\|_A$-combinators. The main reason for this is that these combinators corresponds quite directly with logical conjunction of trace-specifications (see [12]).

Still, one cannot say that $\|_A$ is a better operator than $\|$ in general. The synchronisation format of the $\|$-operator is very flexible and often allows for elegant specifications. An unpleasant property of the $\|_A$-operator is that it is not associative (in the sense that in some cases $(x\|_B y)\|_C z \neq x\|_B(y\|_C z)$). We think that the operators $\|$ and $\|_A$ are both very useful and that therefore notions like 'observable', 'localisation' and 'redundancy in context' should be worked out for both.

4. Redundancy in a context

We want to prove, in a compositional way, that in a given context a summand in a specification can be omitted. We will restrict ourselves in this paper to the case where the summand occurs in a 'linear' equation:

4.1. Definition. Let $E = \{X=t_X \mid X\in V_E\}$ be a recursive specification. A set $C\subseteq V_E$ of variables is called a *cluster* if for each $X\in C$, t_X is of the form:

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

for actions $a_k\in A_\tau$, variables $X_k\in C$ and $Y_l\in V_E-C$. Cluster C is called *isolated* if variables from C do not occur in the terms for the variables from V_E-C.

4.2. Definition. Let $E = \{X=t_X \mid X\in V_E\}$ be a recursive specification and let C be an isolated cluster in E. Let $X_0,X_1,X_2\in C$, $a\in A_\tau$ and let aX_2 be a summand of t_{X_1}. Let E' be obtained from E by replacing summand aX_2 in t_{X_1} by a 'fresh' atom t. Write $p\equiv\langle X_0\mid E\rangle$ and $p'\equiv\langle X_0\mid E'\rangle$. Let y be a process with p observable in context $..\|y$. Let $H\subseteq A$. The summand aX_2 of p *is redundant in the context* $\partial_H(..\|y)$ if:

$$tr(\rho_{\nu(\alpha(p))}\circ\partial_H(p\|y))\cap\{\sigma a\mid \sigma t\in tr(p')\}=\varnothing.$$

4.2.1. Comment. One can say that the set $\{\sigma a\mid \sigma t\in tr(p')\}$ is the contribution of summand aX_2 to $tr(p)$. Theorem 3.3.6 gives that $tr(\rho_{\nu(\alpha(p))}\circ\partial_H(p\|y))$ is also a subset of $tr(p)$. If summand aX_2 is redundant, this means that all behaviours of p of the form 'go from state X_1 with an a-step to state X_2' are not possible if p is placed in the context $\partial_H(..\|y)$.

We give an example which shows why we require in Definition 4.2 that cluster C is isolated. Assume a trijective communication function γ with $\gamma(a,\bar{a})=a^*$ and $\gamma(b,\bar{b})=b^*$. Assume further that $H=\{a,\bar{a},b,\bar{b}\}$ en $I=\{a^*,b^*\}$. Consider the following recursive specification E:

$$X_0 = aX_0 + X_1 \qquad X_1 = b\cdot\tau_I\circ\partial_H(X_0\|\bar{a}\cdot c).$$

In this system X_0 forms a cluster which is not isolated. We derive:

$$X_0 = aX_0 + b\cdot c\cdot\delta.$$

From this equation is is easy to see that X_0 is observable in context $..\|\bar{b}$. We have:

$$\rho_{\nu(\alpha(X_0))}\circ\partial_H(X_0\|\bar{b}) = b\cdot c\cdot\delta.$$

If the condition in Definition 4.2 that C is isolated would be absent, then the summand aX_0 would (by definition) be redundant in context $\partial_H(..\|\bar{b})$. However, the summand cannot be omitted: outside the cluster it plays an essential role!

We can now formulate the central proof principle of this paper:

A redundant summand can be omitted

Below we formally present this principle as a theorem.

4.3. THEOREM. *Let* $p \equiv \langle X_0 \mid E \rangle$ *and* $q \equiv \langle Y_0 \mid F \rangle$, *with* E *and* F *guarded recursive specifications, and* p *observable in context* $.. \| q$. *Let* $H \subseteq A$. *Let* C *be an isolated cluster in* E *with* $X_0, X_1, X_2 \in C$, $a \in A_\tau$ *and* aX_2 *a summand of* t_{X_1}. *Let* E' *and* $\overline{E}$ *be obtained from* E *by resp. replacing* aX_2 *by a fresh atom* t, *and omitting it. Let* $p' \equiv \langle X_0 \mid E' \rangle$ *and* $\overline{p} \equiv \langle X_0 \mid \overline{E} \rangle$. *Suppose that* $\mathrm{ACP}_\tau + \mathrm{REC} + \mathrm{RN} + \mathrm{PR} + \mathrm{TO}$ *proves that summand* aX_2 *is redundant. Then:* $\mathrm{ACP}_\tau + \mathrm{REC} + \mathrm{RN} + \mathrm{PR} + \mathrm{AIP}^- \vdash \partial_H(p \| q) = \partial_H(\overline{p} \| q)$.

PROOF (sketch). The proof uses a bisimulation model generated by Plotkin style action rules. It is proved that the (infinitary) axiom system $\mathrm{ACP}_\tau + \mathrm{REC} + \mathrm{RN} + \mathrm{PR} + \mathrm{AIP}^-$ is sound and complete for processes represented by a guarded specification. Consequently it is enough to prove that $\partial_H(p \| q)$ and $\partial_H(\overline{p} \| q)$ are bisimilar. The proof that the obvious candidate for a bisimulation between these processes indeed *is* a bisimulation uses the fact that every trace of actions in the transition system of an expression p is also a (provable) element of $tr(p)$. □

4.4. REMARK. A summand which can be omitted is in general not redundant. In every context the second summand of the equation

$$X = aX + aX$$

can be omitted, even if it is not redundant. At present we have no idea how a 'reversed version' of Theorem 4.3 would look like.

4.5. Proving redundancies. Now we know that a redundant summand can be omitted, it becomes of course interesting to look for techniques which allow us to prove that summands are redundant. The following strategy works in many cases.

Let E, C, X_0, etc., be as given in Definition 4.2. In order to prove that the summand is redundant, it is enough to show that for some predicate $S(\sigma)$:

$$p' \textbf{ sat } \forall \sigma' : \sigma = \sigma' t \Rightarrow S(\sigma' a) \quad \text{and}$$

$$\rho_{\nu(\alpha(p))} \circ \partial_H(p \| y) \textbf{ sat } \neg S(\sigma).$$

If the cluster C is finite, then $\{\sigma a \mid \sigma t \in tr(p')\}$ is a regular language and can be denoted by a predicate in the trace-specification language of Section 2.4. Consequently we can in such cases always express that a summand is redundant.

4.6. EXAMPLE. We return to Example 1.2 and show how the statement

$$\partial_H(DU\|KM) = \partial_H(\overline{DU}\|KM)$$

can be proved with the notions presented in this section. KM is observable in context $DU\|..$, and DU is observable in context $..\|KM$. The specification of DU contains no isolated clusters, but using RSP we can give an equivalent specification where the set of variables as a whole forms an isolated cluster ($DU = UD$):

$$\begin{array}{|lcl|}\hline UD & = & 30c\cdot UD_1 + kof\cdot UD_2 \\ UD_1 & = & kof\cdot UD_2 \\ UD_2 & = & talk\cdot UD \\ \hline\end{array}$$

TABLE 2. Specification of UD

In Example 2.4.2 we already observed that:

$$KM \textbf{ sat } \sigma\downarrow\overline{kof} \leqslant \sigma\downarrow\overline{30c}.$$

Because of Corollary 3.3.7 we also have:

$$\rho_{\mathcal{H}(\alpha(KM))}\circ\partial_H(UD\|KM) \textbf{ sat } \sigma\downarrow\overline{kof} \leqslant \sigma\downarrow\overline{30c}.$$

The alphabet of process $\partial_H(UD\|KM)$ contains no actions $\overline{kof}$ or $\overline{30c}$, because these actions are in H. This implies that occurrences of these actions in traces from $tr(\rho_{\mathcal{H}(\alpha(KM))}\circ\partial_H(UD\|KM))$ 'originated' (by renaming) from actions kof^* and $30c^*$. Hence:

$$\partial_H(UD\|KM) \textbf{ sat } \sigma\downarrow kof^* \leqslant \sigma\downarrow 30c^*.$$

But since the alphabet of $\partial_H(UD\|KM)$ contains no actions kof and $30c$, this implies:

$$\rho_{\mathcal{H}(\alpha(UD))}\circ\partial_H(UD\|KM) \textbf{ sat } \sigma\downarrow kof \leqslant \sigma\downarrow 30c.$$

Define UD' by:

$$\begin{array}{|lcl|}\hline UD' & = & 30c\cdot UD'_1 + t \\ UD'_1 & = & kof\cdot UD'_2 \\ UD'_2 & = & talk\cdot UD' \\ \hline\end{array}$$

Of course we have

$$UD' \textbf{ sat } \forall\sigma' : \sigma = \sigma' t \Rightarrow (\sigma' kof)\downarrow kof > (\sigma' kof)\downarrow 30c.$$

This shows that the second summand in the equation for UD is redundant. □

In the example above, we gave a long proof of a trivial fact. The nice thing

about the proof is however that it is compositional and only uses general properties of the separate components. This makes that the technique can be used also in less trivial situations where the number of states of the components is large.

In the sequel we will speak about redundant summands of equations which are not part of a cluster. What we mean in such a case is that the corresponding system of equations can be transformed into another system, that a certain summand in the new system is redundant, and that the system which results from omitting this summand is equivalent to the system obtained by omitting the summand in the original system that was called 'redundant'.

5. A WORKCELL ARCHITECTURE

In this section we present a modular verification of a small system which is described in [4, 11].

One can speak about *Computer Integrated Manufacturing (CIM)* if computers play a role in all phases of an industrial production process. In the CIM-philosophy one views a plant as a (possibly hierarchically organised) set of concurrently operating *workcells*. Each workcell is responsible for a well-defined part of the production process, for instance the filling and closing of bottles of milk.

In principle it is possible to specify the behaviour of individual workcells in process algebra. A composite workcell, or even a plant, can then be described as the parallel composition of a number of more elementary workcells. Proof techniques from process algebra can be applied to show that a composite workcell has the desired external behaviour.

In general, not all capabilities of a workcell which is part of a CIM-architecture will be used. A robot which can perform a multitude of tasks, can be part of an architecture where its only task is to fasten a bolt. Other possibilities of the robot will be used only when the architecture is changed. A large part of the behaviours of workcells will be redundant in the context of the CIM-architecture of which they are part. Therefore it can be expected that the notions which are presented in the previous sections of this paper, will be useful in the verification of such systems.

5.1. Specification.

5.1.1. The external behaviour. We want to construct a composite workcell which satisfies the following specification.

$$SPEC = \sum_{n=0}^{N} r\,1(n)\cdot SPEC^n\cdot SPEC$$

$$SPEC^0 = s\,0(r) \qquad SPEC^{n+1} = s\,10(proc(p\,1))\cdot SPEC^n$$

TABLE 3. Specification of a composite workcell *SPEC*

Via port 1, the workcell accepts an order to produce n products of type $proc(p\,1)$ and to deliver these products at port 10. Here $0 \leqslant n \leqslant N$ for a given upperbound $N > 0$. After execution of the order, the workcell gives a signal r at port 0, and returns to its initial state (r = ready).

5.1.2. Architecture. The architecture of the system that has to implement this specification is depicted in Figure 1.

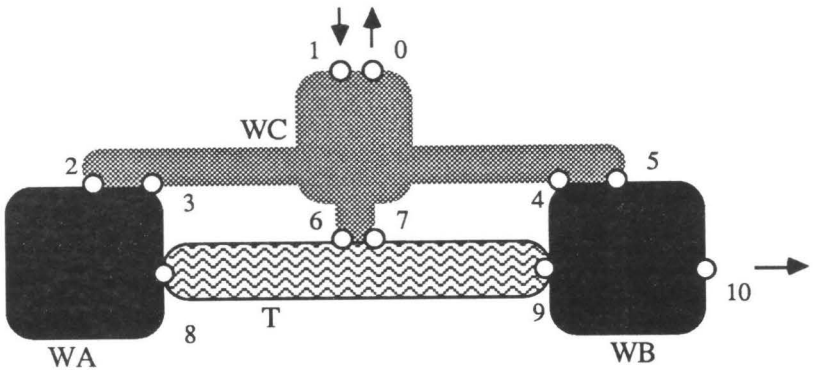

FIGURE 1

There are four components: workcell A (*WA*), workcell B (*WB*), the transport service *T*, and the workcell controller *WC*.

5.1.3. Workcell A. By means of a signal n at port 2, workcell A receives the order to produce n products of type $p\,1$. The cell performs the job and delivers the products to the transport service T at port 8. Thereafter a message r is sent at port 3, to indicate that a next order can be given.

$$WA = \sum_{n=0}^{N} r\,2(n)\cdot XA^n$$

$$XA^0 = s\,3(r)\cdot WA \qquad XA^{n+1} = s\,8(p\,1)\cdot XA^n$$

TABLE 4. Specification of workcell *A*

5.1.4. Workcell B. By means of a signal n at port 4, workcell B receives the order to process n products. B receives products from a set $PROD$ at port 9. An incoming product p is processed and the result $proc(p) \in PROD$ is delivered at port 10 ($proc$ = processed). Thereafter a message r is sent at port 5 and the workcell returns to its initial state. We assume that $p\,1 \in PROD$.

$$WB = \sum_{n=0}^{N} r4(n) \cdot XB^n$$

$$XB^0 = s5(r) \cdot WB \qquad XB^{n+1} = \sum_{p \in PROD} r9(p) \cdot s10(proc(p)) \cdot XB^n$$

TABLE 5. Specification of workcell B

5.1.5. Transport service T transports products in $PROD$ and behaves like a FIFO-queue. Products are accepted by T at port 8. Transport commands tc are given to T at port 6. The number of products accepted by the transport service should not exceed the number of transport commands which have been received by more than one. Each time a product leaves T at port 9, a signal $s7(ar)$ is given (ar = arrival). Variables in the specification below are indexed by the contents of the transport service: $\sigma \in PROD^*$ and $p,q \in PROD$.

$$T^{\lambda} = r6(tc) \cdot \Big(\sum_{p \in PROD} r8(p) \cdot T^{p} \Big) + \sum_{p \in PROD} r8(p) \cdot r6(tc) \cdot T^{p}$$

$$T^{\sigma q} = r6(tc) \cdot \Big(\sum_{p \in PROD} r8(p) \cdot T^{p \sigma q} \Big) + \sum_{p \in PROD} r8(p) \cdot r6(tc) \cdot T^{p \sigma q} + s9(q) \cdot s7(ar) \cdot T^{\sigma}$$

TABLE 6. Specification of transport service T

5.1.6. Workcell controller WC is the boss of components WA, T and WB. From its superiors (via port 1), WC can get the order to take care of the manufacturing of n products $proc(p\,1)$. In order to execute this order, WC sends a stream of commands to its subordinates, receiving progress reports from these subordinates in between. When the controller thinks that the task has been completed, it generates a signal $s0(r)$.

$$WC = \sum_{n=0}^{N} r1(n) \cdot s4(n) \cdot XC^n$$

$$XC^0 = r5(r) \cdot s0(r) \cdot WC \qquad XC^{n+1} = s2(1) \cdot r3(r) \cdot s6(tc) \cdot r7(ar) \cdot XC^n$$

TABLE 7. Specification of workcell controler WC

5.1.7. Sets. $\mathbb{D} = \{n \mid 0 \leqslant n \leqslant N\} \cup \{r,tc,ar\} \cup PROD$ is the set of objects which can be communicated in the system, and $\mathbb{P} = \{0,1,...,10\}$ is the set of port-names used. Communication takes place following the read/send-scheme:

$$\gamma(rp(d),sp(d)) = cp(d) \quad \text{for } p \in \mathbb{P}, d \in \mathbb{D}$$

and γ yields δ in all other cases. Important sets of actions are:

$$H = \{rp(d),sp(d) \mid 2 \leqslant p \leqslant 9 \text{ and } d \in \mathbb{D}\} \quad \text{and}$$

$$I = \{cp(d) \mid 2 \leqslant p \leqslant 9 \text{ and } d \in \mathbb{D}\}.$$

The implementation as a whole can now be described by:

$$\boxed{IMPL = \partial_H(WC \| WA \| T^{\lambda} \| WB)}$$

5.2. THEOREM *(correctness implementation).*
$ACP_\tau + SC + REC + PR + AIP^- + AB + CA \vdash \tau_I(IMPL) = SPEC.$

PROOF. In seven steps we transform $\tau_I(IMPL)$ to $SPEC$. Before we start with the 'real' calculations, we show in the first three steps that in the specifications of components WA, T and WB, a large number of summands can be omitted. Notice that communication is trijective and that each component of $IMPL$ is observable in context with the other components.

First we use that the only command which is given by the controller to workcell A is a request to produce a single product $p\,1$. This means that:

$$IMPL \textbf{ sat } \sigma\downarrow c\,2(n) = 0 \quad \text{for } n \neq 1.$$

Consequently

$$\rho_{\nu(\alpha(WA))}(IMPL) \textbf{ sat } \sigma\downarrow r\,2(n) = 0 \quad \text{for } n \neq 1.$$

Using the approach of Section 4.5, together with Theorem 4.3, we obtain that all the summands in the specification of WA which correspond to the acceptance of a command different from $r\,2(1)$ are redundant. We have

$$IMPL = \partial_H(WC \| \overline{WA} \| T^{\lambda} \| WB),$$

where $\overline{WA}$ is given by:

$$\boxed{\overline{WA} = r\,2(1) \cdot s\,8(p\,1) \cdot s\,3(r) \cdot \overline{WA}}$$

Hence:

$$\tau_I(IMPL) = \tau_I \circ \partial_H(WC \| \overline{WA} \| T^{\lambda} \| WB) \qquad \text{(step 1)}$$

Also component T^{λ} is clearly a candidate for simplification. With some simple trace-theoretic arguments we show that nearly all summands in the specification of T^{λ} are redundant.

The only product which is delivered by $\overline{WA}$ at port 8 is $p\,1$. This means

that:

$$IMPL \textbf{ sat } \sigma\downarrow c\,8(p) = 0 \quad \text{for } p \neq p\,1 \tag{1}$$

From the behaviour of component WC we conclude:

$$IMPL \textbf{ sat } \sigma\downarrow c\,6(tc) \leqslant \sigma\downarrow c\,3(r) \tag{2}$$

Further we deduce from the behaviour of $\overline{WA}$:

$$IMPL \textbf{ sat } \sigma\downarrow c\,3(r) \leqslant \sigma\downarrow c\,8(p\,1) \tag{3}$$

From (2) and (3) together we conclude that the number of transport commands at port 6 is less or equal to the number of products $p\,1$ that are handed to the transport service at port 8:

$$IMPL \textbf{ sat } \sigma\downarrow c\,6(tc) \leqslant \sigma\downarrow c\,8(p\,1) \tag{4}$$

From the specification of $\overline{WA}$ we learn that A does not deliver products without being asked for:

$$IMPL \textbf{ sat } \sigma\downarrow c\,8(p\,1) \leqslant \sigma\downarrow c\,2(1) \tag{5}$$

Further it follows from the specification of WC that the number of commands given to A by the controller, never exceeds the number of ar-signals with more than one:

$$IMPL \textbf{ sat } \sigma\downarrow c\,2(1) \leqslant \sigma\downarrow c\,7(ar) + 1 \tag{6}$$

From (5) and (6) together we conclude:

$$IMPL \textbf{ sat } \sigma\downarrow c\,8(p\,1) \leqslant \sigma\downarrow c\,7(ar) + 1 \tag{7}$$

From formulas (1), (4) and (7) it follows that nearly all summands in the specification of T^{λ} are redundant.

$$\tau_I \circ \partial_H(WC \| \overline{WA} \| T^{\lambda} \| WB) = \tau_I \circ \partial_H(WC \| \overline{WA} \| T \| WB) \qquad \text{(step 2)}$$

where T is given by:

$$\boxed{T = r\,8(p\,1) \cdot r\,6(tc) \cdot s\,9(p\,1) \cdot s\,7(ar) \cdot T}$$

The transport service delivers at port 9 only products of type $p\,1$. Therefore all summands in the specification of WB which correspond to the acceptance of another product, are redundant.

$$\tau_I \circ \partial_H(WC \| \overline{WA} \| T \| WB) = \tau_I \circ \partial_H(WC \| \overline{WA} \| T \| \overline{WB}) \qquad \text{(step 3)}$$

where $\overline{WB}$ is given by:

$$\boxed{\begin{array}{ll} \overline{WB} = \sum_{n=0}^{N} r4(n)\cdot\overline{XB}^{n} & \\ \overline{XB}^{0} = s5(r)\cdot\overline{WB} & \overline{XB}^{n+1} = r9(p\,1)\cdot s\,10(proc(p\,1))\cdot\overline{XB}^{n} \end{array}}$$

We will now 'zoom in' on components WC, $\overline{WA}$ and T. Define:

$$H' = \{rp(d), sp(d) \mid p \in \{2,3,6,7,8\} \text{ and } d \in \mathbb{D}\} \quad \text{and}$$

$$I' = \{cp(d) \mid p \in \{2,3,6,7,8\} \text{ and } d \in \mathbb{D}\}.$$

Application of the conditional axioms CA gives:

$$\tau_I \circ \partial_H(WC \| \overline{WA} \| T \| \overline{WB}) = \tau_I \circ \partial_H(\tau_{I'} \circ \partial_{H'}(WC \| \overline{WA} \| T) \| \overline{WB}) \quad \text{(step 4)}$$

Let W be given by:

$$\boxed{\begin{array}{ll} W = \sum_{n=0}^{N} r1(n)\cdot s4(n)\cdot W^{n} & \\ W^{0} = r5(r)\cdot s0(r)\cdot W & W^{n+1} = \tau\cdot s9(p\,1)\cdot W^{n} \end{array}}$$

We prove that $W = \tau_{I'} \circ \partial_{H'}(WC \| \overline{WA} \| T)$, by showing that process $\tau_{I'} \circ \partial_{H'}(WC \| \overline{WA} \| T)$ satisfies the defining equations of W.

$$\tau_{I'} \circ \partial_{H'}(WC \| \overline{WA} \| T) = \sum_{n=0}^{N} r1(n)\cdot s4(n)\cdot \tau_{I'} \circ \partial_{H'}(XC^{n} \| \overline{WA} \| T)$$

$$\tau_{I'} \circ \partial_{H'}(XC^{0} \| \overline{WA} \| T) = r5(r)\cdot s0(r)\cdot \tau_{I'} \circ \partial_{H'}(WC \| \overline{WA} \| T)$$

$$\tau_{I'} \circ \partial_{H'}(XC^{n+1} \| \overline{WA} \| T) =$$

$$= \tau_{I'}(c2(1)\cdot \partial_{H'}(s3(r)\cdot s6(tc)\cdot r7(ar)\cdot XC^{n} \| s8(p\,1)\cdot s3(r)\cdot \overline{WA} \| T)) =$$

$$= \tau\cdot\tau_{I'}(c8(p\,1)\cdot \partial_{H'}(s3(r)\cdot s6(tc)\cdot r7(ar)\cdot XC^{n} \| s3(r)\cdot \overline{WA} \| r6(tc)\cdot s9(p\,1)\cdot s7(ar)\cdot T)) =$$

$$= \tau\cdot\tau\cdot\tau_{I'}(c3(r)\cdot \partial_{H'}(s6(tc)\cdot r7(ar)\cdot XC^{n} \| \overline{WA} \| r6(tc)\cdot s9(p\,1)\cdot s7(ar)\cdot T)) =$$

$$= \tau\cdot\tau_{I'}(c6(tc)\cdot \partial_{H'}(r7(ar)\cdot XC^{n} \| \overline{WA} \| s9(p\,1)\cdot s7(ar)\cdot T)) =$$

$$= \tau\cdot\tau_{I'}(s9(p\,1)\cdot \partial_{H'}(r7(ar)\cdot XC^{n} \| \overline{WA} \| s7(ar)\cdot T)) =$$

$$= \tau\cdot s9(p\,1)\cdot\tau_{I'}(c7(ar)\cdot \partial_{H'}(XC^{n} \| \overline{WA} \| T)) =$$

$$= \tau\cdot s9(p\,1)\cdot\tau_{I'} \circ \partial_{H'}(XC^{n} \| \overline{WA} \| T)$$

We have now derived:

$$\tau_I \circ \partial_H(\tau_{I'} \circ \partial_{H'}(WC \| \overline{WA} \| T) \| \overline{WB}) = \tau_I \circ \partial_H(W \| \overline{WB}) \quad \text{(step 5)}$$

Let V be given by:

$$V = \sum_{n=0}^{N} r1(n)\cdot V^n$$
$$V^0 = \tau\cdot s0(r)\cdot V \qquad V^{n+1} = \tau\cdot s10(proc(p1))\cdot V^n$$

We show that $\tau_I\circ\partial_H(W\|\overline{WB})$ satisfies the defining equations of V.

$$\tau_I\circ\partial_H(W\|\overline{WB}) = \sum_{n=0}^{N} r1(n)\cdot\tau_I\circ\partial_H(s4(n)\cdot W^n\|(\sum_{m=0}^{N} r4(m)\cdot\overline{XB}^m)) =$$
$$= \sum_{n=0}^{N} r1(n)\cdot\tau_I(c4(n)\cdot\partial_H(W^n\|\overline{XB}^n)) = \sum_{n=0}^{N} r1(n)\cdot\tau\cdot\tau_I\circ\partial_H(W^n\|\overline{XB}^n)$$
$$\tau\cdot\tau_I\circ\partial_H(W^0\|\overline{XB}^0) = \tau\cdot\tau_I(c5(r)\cdot\partial_H(s0(r)\cdot W\|\overline{WB})) = \tau\cdot s0(r)\cdot\tau_I\circ\partial_H(W\|\overline{WB})$$
$$\tau\cdot\tau_I\circ\partial_H(W^{n+1}\|\overline{XB}^{n+1}) = \tau\cdot\tau_I(c9(p1)\cdot\partial_H(W^n\|s10(proc(p1))\cdot\overline{XB}^n)) =$$
$$= \tau\cdot s10(proc(p1))\cdot\tau_I\circ\partial_H(W^n\|\overline{XB}^n)$$

(here we use that $\tau(\tau x\|y)=\tau\tau x\|\_y=\tau x\|\_y=\tau(x\|y)$). From the above derivation it follows that:

$$\tau_I\circ\partial_H(W\|\overline{WB}) = V \qquad \text{(step 6)}$$

We show that $SPEC$ satisfies the defining equations of V.

$$SPEC = \sum_{n=0}^{N} r1(n)\cdot(\tau\cdot SPEC^n\cdot SPEC)$$
$$\tau\cdot SPEC^0\cdot SPEC = \tau\cdot s0(r)\cdot SPEC$$
$$\tau\cdot SPEC^{n+1}\cdot SPEC = \tau\cdot s10(proc(p1))\cdot(\tau\cdot SPEC^n\cdot SPEC)$$

Hence:

$$V = SPEC \qquad \text{(step 7)}$$

□

This example shows that a combination of trace-theoretic arguments and the use of alphabet calculus makes it possible to verify simple systems in a compositional and modular way.

ACKNOWLEDGEMENTS

I would like to thank the participants of the PAM seminar, especially Jos Baeten, Jan Bergstra and Hans Mulder, for their comments on earlier versions of this paper.

REFERENCES

[1] J.C.M. BAETEN & J.A. BERGSTRA (1988): *Global renaming operators in*

concrete process algebra. I&C 78(3), pp. 205-245.

[2] J.C.M. BAETEN, J.A. BERGSTRA & J.W. KLOP (1986): *Syntax and defining equations for an interrupt mechanism in process algebra.* Fund. Inf. IX(2), pp. 127-168.

[3] J.C.M. BAETEN, J.A. BERGSTRA & J.W. KLOP (1987): *On the consistency of Koomen's fair abstraction rule.* Theoretical Computer Science 51(1/2), pp. 129-176.

[4] F. BIEMANS & P. BLONK (1986): *On the formal specification and verification of CIM architectures using LOTOS.* Computers in Industry 7(6), pp. 491-504.

[5] E.W. DIJKSTRA (1976): *A discipline of programming*, Prentice-Hall, Englewood Cliff.

[6] R.J. VAN GLABBEEK & F.W. VAANDRAGER (1987): *Petri net models for algebraic theories of concurrency.* In: Proceedings PARLE conference, Eindhoven, Vol. II (Parallel Languages) (J.W. de Bakker, A.J. Nijman & P.C. Treleaven, eds.), LNCS 259, Springer-Verlag, pp. 224-242.

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

[8] C.P.J. KOYMANS & J.C. MULDER (1986): *A modular approach to protocol verification using process algebra.* Logic Group Preprint Series Nr. 6, CIF, State University of Utrecht, to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 261-306.

[9] K.G. LARSEN (1986): *Context-dependent bisimulation between processes.* Ph.D. Thesis, Department of Computer Science, University of Edinburgh.

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

[11] S. MAUW (1987): *Process algebra as a tool for the specification and verification of CIM-architectures.* Report P8708, Programming Research Group, University of Amsterdam, to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 53-80.

[12] E.-R. OLDEROG (1986): *Process theory: semantics, specification and verification.* In: Current Trends in Concurrency (J.W. de Bakker, W.-P. de Roever & G. Rozenberg, eds.), LNCS 224, Springer-Verlag, pp. 442-509.

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

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

[15] J. ZWIERS (1989): *Compositionality, concurrency and partial correctness*, LNCS 321, Springer-Verlag.

Process Algebra Semantics of POOL

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

In this paper we describe a translation of the Parallel Object-Oriented Language POOL to the language of ACP, the Algebra of Communicating Processes. This translation provides us with a large number of semantics for POOL. It is argued that an optimal semantics for POOL does not exist: what is optimal depends on the application domain one has in mind. We show that the select statement in POOL makes a semantical description of POOL with handshaking communication between objects incompatible with a description level where message queues are used. Attention is paid to the question how fairness and successful termination can be included in the semantics.

Key Words & Phrases: process algebra, concurrency, object-oriented programming, semantics of programming languages, attribute grammars, correctness of programming language implementations, fairness.

Notes. This paper is a revised version of [25]. Section 6 has been left out. Without further reference, we will use the notation and axioms that are described in the second paper of this thesis (*Modular specifications in process algebra*).

1. Introduction

At this moment there are a lot of programming languages which offer facilities for concurrent programming. The basic notions of some of these languages, for example CSP [18], occam [19] and LOTOS [20], are rather close to the basic notions in ACP, and it is not very difficult to give semantics to these languages in the framework of ACP. Milner [23] showed how a simple high level concurrent language can be translated into CCS. However, it is not obvious at first sight how to give process algebra semantics of more complex concurrent programming languages like Ada [6], Pascal-Plus [13] or POOL [1-3]. This is an important problem because of the simple fact that many concurrent systems are specified in terms of these languages. In this paper we will tackle the problem, and give process algebra semantics of the language POOL.

In order to modularize the problems we first give, in Section 2, a translation to process algebra of a simple sequential programming language: with each element of the language a process is associated, specified in terms of the operators $\cdot$, $+$, $\ggg$ (sequential and alternative composition, and chaining).

In Section 3, we give process algebra semantics of a representative subset of the programming language POOL-T (see [1]). POOL is an acronym for 'Parallel Object-Oriented Language'. It stands for a family of languages designed at

Philips Research Laboratories in Eindhoven. The 'T' in POOL-T stands for 'Target'. POOL is a language that permits the programming of systems with a large amount of parallelism, using object-oriented programming. In [4] an operational semantics is given of a language from the POOL-family. Our semantics of POOL is to a large extent inspired by this paper. A denotational semantics of POOL is presented in [5].

In order to deal with the complexity of POOL (compared to the toy language of Section 2) we make use of attribute grammars. We associate with each (abstract) POOL program a process specified in the signature of ACP together with some additional operators. As soon as the translation of a programming language into the signature of ACP (+additional operators) is accomplished, the whole range of process algebras becomes available as possible semantics of the language. We think this is a major advantage of our approach. Especially when dealing with concurrent programming languages, the answer to the question what is to be considered as the optimal semantics, is heavily influenced by the application one has in mind: if the system that executes the program is placed in a glass box and does not communicate with the external world, one can work with a more identifying semantics (allowing for simpler proofs) than in the case in which the system is part of a network and communicates with the external world. Issues like fairness and the presence of interrupt mechanisms are also relevant in the choice of the optimal semantics.

The process algebra semantics are very operational: we can define a term rewriting machine that executes the process algebra specification we relate to a program. Interestingly, the semantics are also (to a large extent) compositional: the value denoted by a construct is specified in terms of the values denoted by its syntactic subcomponents.

A good theory of semantics of programming languages is a method which makes it possible to predict the behaviour of a computer that executes a program. Furthermore a good theory assists people in building new predictable computers. This implies that a theory of semantics of programming languages should provide tools which make it possible to substantiate the claim that the mathematical models in which the language constructs are interpreted indeed model reality. In our framework such a tool is the abstraction operator τ_I. This operator makes it possible to prove that the semantics of POOL as presented in Section 3 has a common abstraction with a number of other semantics of the language, which are closer to implementation.

In an implementation of the language POOL there will be message queues in which the incoming messages for an object are stored. On the conceptual level, there are no queues and we have handshaking communication between the objects. In Section 4 an example is presented which shows that these two views are in contradiction with each other. The problem is due to the so-called 'select statement', which is part of the language POOL-T. A minor change in the definition of the select statement is proposed in order to remove this difficulty.[1] However, it is shown that even with the new language

1. In a more recent offspring of the POOL-family of languages, called POOL2 (see [3]), the select statement has been removed altogether. Instead this language contains a 'conditional answer state-

definition the two descriptions are different in bisimulation semantics. Although we conjecture that the two views of a POOL system are equivalent in failure semantics, we have not been able to prove this.

In Section 5 we discuss a trace semantics of the language POOL. Many things can be proved with more ease in this semantics, but we show that this semantics does not describe deadlock behaviour in a situation in which the POOL system interacts with the environment. We also pay some attention to the question how issues like fairness and successful termination can be included in a semantical description of POOL.

Section 6 contains a number of conclusions.

2. A SIMPLE SEQUENTIAL PROGRAMMING LANGUAGE

In this section we will give process algebra semantics to a simple programming language that is described in [9]. In the definition below we use the BNF-format.

2.1. DEFINITION (syntax of *Iexp*, *Bexp* and *Stat*). Let *Ivar*, with typical elements v,w,u, ..., and *Icon*, with typical elements α, ..., be given, finite sets of symbols.

a. The class *Iexp* of *integer expressions*, with typical elements s,t, ..., is defined by

$$s ::= v \mid \alpha \mid s_1 + s_2 \mid \cdots \mid \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}$$

(Expressions such as $s_1 - s_2$, $s_1 \times s_2$, ... may be added at the position of the ..., if desired.)

b. The class *Bexp* of *boolean expressions*, with typical elements b, ..., is defined by

$$b ::= \textbf{true} \mid \textbf{false} \mid s_1 = s_2 \mid \cdots \mid \neg b \mid b_1 \supset b_2$$

(Expressions such as $s_1 < s_2$, ... may be added at the position of the ..., if desired.)

c. The class *Stat* of *statements*, with typical elements S, ..., is defined by

$$S ::= v := s \mid S_1 ; S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{while } b \textbf{ do } S \textbf{ od}$$

2.2. Note. In contrast to [9], we require the sets *Ivar* and *Icon* to be finite. If we would allow them to be infinite this would lead to infinite sums in our process algebra specifications. It is trivial to add an infinite sum operator to, for example, the term model defined in [16]. However, the combination of such an operator and the abstraction operators τ_I leads to a number of non-trivial questions that are worth separate investigation. For this reason we will confine ourselves to the finite case in this paper.

ment'. It seems that this construct does not lead to semantical problems like the select statement.

2.3. Semantics of the toy language. We will now relate to each element of the language defined in Section 2.1, a recursive process specification. Besides the actions that will be described below, this specification may contain operators $\cdot$, $+$ and $\ggg$. The value domain D of the chaining operator is

$$D = (Ivar \to Icon) \cup Icon \cup \{\textbf{true}, \textbf{false}\}.$$

Here $Ivar \to Icon$ is the set of all functions from variables to their values. Elements of $Ivar \to Icon$ are called *states*. As usual with the chaining operator, we have actions $\uparrow d$, $\downarrow d$, $c(d)$, $r(d)$ and $s(d)$ for each $d \in D$.

2.4.1. Notation. Let $\sigma \in Ivar \to Icon$, $v \in Ivar$ and $\alpha \in Icon$. We use the well known notation $\sigma\{\alpha/v\}$ to denote the element of $Ivar \to Icon$ that satisfies for each $v' \in Ivar$

$$\sigma\{\alpha/v\}(v') = \begin{cases} \alpha & \text{if } v' = v \\ \sigma(v') & \text{otherwise} \end{cases}$$

2.4.2. Notation. For the term

$$x \ggg (\sum_{d_1 \in D_1} \downarrow d_1 \cdot \cdots \cdot \sum_{d_n \in D_n} \downarrow d_n \cdot y_{d_1,\ldots,d_n})$$

(where $D_1,\ldots,D_n \subseteq D$) we write

$$x \ggg_{d_1,\ldots,d_n} y_{d_1,\ldots,d_n}$$

In all applications it will be clear from the context what $D_1,\ldots,D_n$ are. A similar notation is used for the $\gg$-operator.

2.5. Translation to process algebra. Below we give a number of process algebra equations. The variables in these equations are elements of the toy language with semantical brackets ('$[\![$' and '$]\!]$') placed around them, often sub- and super-scripted with elements of D. The process corresponding to execution of language element $w \in Iexp \cup Bexp \cup Stat$, with an initial memory configuration $\sigma \in Ivar \to Icon$, is the solution of this system, with

$$[\![w]\!]^\sigma$$

taken as root variable. Throughout the rest of this section $\alpha, \alpha' \in Icon$, $\beta, \beta' \in \{\textbf{true}, \textbf{false}\}$ and $\sigma, \sigma' \in Ivar \to Icon$.

2.6. The class *Iexp*

$$[\![v]\!]^\sigma = \uparrow\sigma(v)$$

$$[\![\alpha]\!]^\sigma = \uparrow\alpha$$

$$[\![s_1 + s_2]\!]^\sigma = [\![s_1]\!]^\sigma \cdot [\![s_2]\!]^\sigma \ggg_{\alpha,\alpha'} \uparrow sum(\alpha,\alpha')$$

$$[\![\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}]\!]^\sigma = [\![b]\!]^\sigma \ggg (\downarrow\textbf{true} \cdot [\![s_1]\!]^\sigma + \downarrow\textbf{false} \cdot [\![s_2]\!]^\sigma)$$

2.7. The class *Bexp*

$$[\![\mathbf{true}]\!]^\sigma = \uparrow\mathbf{true}$$

$$[\![\mathbf{false}]\!]^\sigma = \uparrow\mathbf{false}$$

$$[\![s_1 = s_2]\!]^\sigma = [\![s_1]\!]^\sigma \cdot [\![s_2]\!]^\sigma \ggg_{\alpha,\alpha'} [\![=]\!]_{\alpha,\alpha'}$$

$$[\![=]\!]_{\alpha,\alpha'} = \begin{cases} \uparrow\mathbf{true} & \text{if } \alpha = \alpha' \\ \uparrow\mathbf{false} & \text{otherwise} \end{cases}$$

$$[\![\neg b]\!]^\sigma = [\![b]\!]^\sigma \ggg (\downarrow\mathbf{true}\cdot\uparrow\mathbf{false} + \downarrow\mathbf{false}\cdot\uparrow\mathbf{true})$$

$$[\![b_1 \supset b_2]\!]^\sigma = [\![b_1]\!]^\sigma \ggg (\downarrow\mathbf{true}\cdot[\![b_2]\!]^\sigma + \downarrow\mathbf{false}\cdot\uparrow\mathbf{true})$$

2.8. The class *Stat*

$$[\![v := s]\!]^\sigma = [\![s]\!]^\sigma \ggg_\alpha \uparrow\sigma\{\alpha/v\}$$

$$[\![S_1 ; S_2]\!]^\sigma = [\![S_1]\!]^\sigma \ggg_{\sigma'} [\![S_2]\!]^{\sigma'}$$

$$[\![\mathbf{if}\, b\, \mathbf{then}\, S_1\, \mathbf{else}\, S_2\, \mathbf{fi}]\!]^\sigma = [\![b]\!]^\sigma \ggg (\downarrow\mathbf{true}\cdot[\![S_1]\!]^\sigma + \downarrow\mathbf{false}\cdot[\![S_2]\!]^\sigma)$$

$$[\![\mathbf{while}\, b\, \mathbf{do}\, S\, \mathbf{od}]\!]^\sigma = [\![b]\!]^\sigma \ggg$$

$$(\downarrow\mathbf{true}\cdot([\![S]\!]^\sigma \ggg_{\sigma'} [\![\mathbf{while}\, b\, \mathbf{do}\, S\, \mathbf{od}]\!]^{\sigma'}) + \downarrow\mathbf{false}\cdot\uparrow\sigma)$$

The following theorem shows that the specification presented above singles out a unique process.

2.9. THEOREM. *The specification defined in 2.6-2.8 is guarded.*

PROOF. Recall the definition of relation $\xrightarrow{u}$ on Ξ:

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

It is enough to show that the relation $\xrightarrow{u}$ is well founded (i.e. there is no infinite sequence $X_1 \xrightarrow{u} X_2 \xrightarrow{u} X_3 \cdots$). This can be done by defining a function $m : \Xi \to \mathbb{N}$ such that for $X, Y \in \Xi$

$$X \xrightarrow{u} Y \quad \Rightarrow \quad m(Y) < m(X)$$

The definition goes by induction on the complexity of the language elements in the variables. We give only a very small part of it. This should convince the reader that it is possible to give a complete definition, which has the desired property.

$$m([\![v]\!]^\sigma) = 1$$

$$m([\![\alpha]\!]^\sigma) = 1$$

$$m([\![s_1 + s_2]\!]^\sigma) = m([\![s_1]\!]^\sigma) + m([\![s_2]\!]^\sigma)$$

etc. □

2.10. Note. As a direct consequence of the associativity of the chaining operator we have that ';' is associative:

$$[\![(S_1;S_2);S_3]\!]^\sigma = [\![S_1;(S_2;S_3)]\!]^\sigma$$

2.11. REMARK. In the equation for $[\![s_1+s_2]\!]^\sigma$ we say that, in order to evaluate s_1+s_2, we first have to evaluate s_1 and thereafter s_2. Other possibilities would have been

$$[\![s_1+s_2]\!]^\sigma = [\![s_2]\!]^\sigma \cdot [\![s_1]\!]^\sigma \ggg_{\alpha,\alpha'} \uparrow sum(\alpha,\alpha')$$

(evaluation in the reverse order), or

$$[\![s_1+s_2]\!]^\sigma = ([\![s_1]\!]^\sigma \| [\![s_2]\!]^\sigma) \ggg_{\alpha,\alpha'} \uparrow sum(\alpha,\alpha')$$

(evaluation in parallel). The three resulting semantics are all different. One can prove however that they are identical after appropriate abstraction.

2.12. REMARK. It is easy to define a term rewriting system which, for a given guarded specification $E = \{X = t_X \mid X \in \Xi\}$, rewrites a given term t in the signature of $ACP_\tau + RN + CH$ with variables in Ξ, into a term of the form $\sum a_i \cdot t_i + \sum b_j$. Now the simple data flow network of Figure 2.1 represents a machine that 'executes' specification E. Here *TRS* is a component that implements the term rewriting system described above, and N is a nondeterministic device that for each input $\sum a_i \cdot t_i + \sum b_j$ chooses either one summand $a_i \cdot t_i$, and thereafter sends term t_i to the input port and atomic action a_i to the output port, or chooses one summand b_j and sends this to the output port.

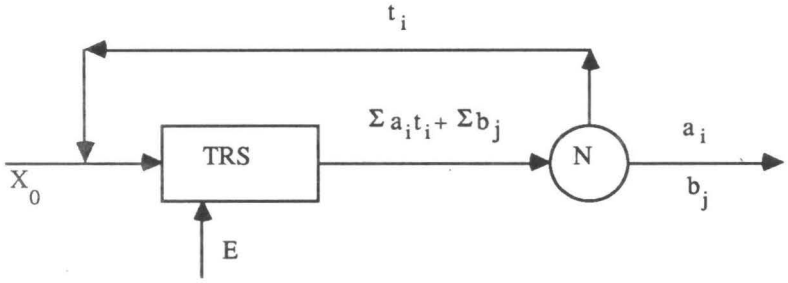

FIGURE 2.1

The following theorem says that the operators $+$ and $\ggg$ can be eliminated in favour of the sequential composition operator $\cdot$. This means that in the case of the toy language the nondeterministic device N of Section 2.12 never has a real choice.

2.13. THEOREM. *Let σ be a state. Using the axioms of* ACP+RN+CH+REC+PR+AIP$^-$ *we can prove:*

1. *Let s be an integer expression. Then for certain $d_1,...,d_n$ and α:*

$$[\![s]\!]^\sigma = c(d_1)\cdot\cdots\cdot c(d_n)\cdot\uparrow\alpha.$$

2. *Let b be a boolean expression. Then for certain $d_1,...,d_n$ and β:*

$$[\![b]\!]^\sigma = c(d_1)\cdot\cdots\cdot c(d_n)\cdot\uparrow\beta.$$

3. *Let S be a statement. Then either there exist $d_1,...,d_n$, σ' such that:*

$$[\![S]\!]^\sigma = c(d_1)\cdot\cdots\cdot c(d_n)\cdot\uparrow\sigma'$$

or there are $d_1,d_2,...$ such that:

$$[\![S]\!]^\sigma = c(d_1)\cdot c(d_2)\cdot\cdots$$

PROOF. By induction on the complexity of the language elements. □

2.14. REMARK. The reason why we used the operator $\ggg$ instead of operator $\gg$ in the definitions above is that the use of $\gg$ would lead to unguarded systems of equations. There exist models of ACP_τ (for example models based on Plotkin style action rules) in which we can relate to each specification (so also the unguarded ones) a special solution. If we would work in these models it would be possible to use the operator $\gg$ instead of the operator $\ggg$. But as stated before, we do not want to restrict ourselves to one single model. In the axiomatic framework the following approaches are available if one wants to obtain 'abstract' semantics:

1. *Partial Abstraction.* In the system of equations defining the semantics of the toy language (Sections 2.6-2.8) we can replace all occurrences of operator $\ggg$ in the equations for the classes *Iexp* and *Bexp* by an operator $\gg$. Using induction on the structure of the elements of *Iexp* and *Bexp* one can prove that the resulting system is still guarded. It is not possible to replace occurrences of $\ggg$ in the equations for elements of the class *Stat* by $\gg$. Consequently this approach will not lead to 'full abstractness'.
2. *Delayed Abstraction.* Let E be a guarded specification that contains no τ-steps or abstraction operator. For a language element w and a memory configuration σ,

$$[\![w]\!]^\sigma$$

is the formal variable that corresponds to execution of w with initial memory configuration σ. Now we extend specification E with variables $<w>^\sigma$ for which we have equations

$$<w>^\sigma = \tau_I([\![w]\!]^\sigma)$$

Here I is a set of 'unimportant' actions which we want to hide. Formal variable $<w>^\sigma$ corresponds to the execution of program w with initial memory state σ, in an environment where actions from I cannot be observed. Call the new system E_I. E_I has a unique solution because E

has one. Note that when we follow this approach we lose, to a certain extent, compositionality.
3. Combination of 1 and 2.

3. Translation of POOL to process algebra

3.1. In this section we give a translation to process algebra of a (representative) subset of the programming language POOL-T. Below we give, by means of a context-free grammar, the definition of a language POOL-⊥-CF. This language is a subset of the context free syntax of POOL-T, as presented in [1].[1] In this section we will give process algebra semantics of a language POOL-⊥, defined by:

$$\text{POOL-}\bot = \text{POOL-T} \cap \text{POOL-}\bot\text{-CF}.$$

By giving a definition in this way we do not have to give an exhaustive enumeration of all the context conditions. Because most of the context conditions in POOL are rather obvious ('all instance variables are declared in the current class definition', etc.), this is not a serious omission. Moreover, we will mention context conditions whenever we need them.

First we will define a mapping $SPEC_C$ that relates a recursion construct $<X_w \mid E_w>$ to each element w of the language POOL-⊥. The subscript C indicates that the resulting expression is in the signature of concrete process algebra, as opposed to the translation that we will present in Section 3.11, which leads to expressions that contain an abstraction operator.

3.2. Context-free languages. Although the notions of a context-free grammar and the language generated by it will be commonly known, we give a formal definition, because we will need this later on.

3.2.1. Definition. A *context-free grammar* is a 4-tuple $G = (T,N,S,P)$, where T and N are finite sets of *terminal* resp. *nonterminal symbols*; $V = T \cup N$ is called the *vocabulary* of symbols; $S \in N$ is the *start symbol*, and P is a finite set of *production rules* of the form $X_0 \rightarrow X_1 \cdots X_n$ with $X_0 \in N$, $n > 0$, and $X_1,\ldots,X_n \in V - \{S\}$.

3.2.2. Definition. Let $G = (T,N,S,P)$ be a context-free grammar, and let $V = T \cup N$. Let $\mathfrak{N} = (\mathbb{N} - \{0\})^*$ be the set of sequences of positive natural numbers. We write ϵ for the empty string, and if $\sigma \in \mathfrak{N}$, we use $\sigma.0$ as a notation for σ. A *derivation tree* of G is a 2-tuple $t = (nodes(t), label(t))$, where $nodes(t)$ is a nonempty finite subset of $\mathfrak{N}$ such that for all $\sigma \in \mathfrak{N}$ and $m,n \in \mathbb{N} - \{0\}$:
1. $\sigma.n \in nodes(t) \Rightarrow \sigma \in nodes(t)$

1. Except for the fact that the expression denoting the destination object in a send-expression can be **nil** in POOL-⊥-CF, which is not the case in the context-free syntax of POOL-T.

2. $\sigma.n \in nodes(t) \wedge m < n \Rightarrow \sigma.m \in nodes(t)$

and $label(t)$ is a function from $nodes(t)$ into V such that if $\sigma.n \in nodes(t)$ and $\sigma.(n+1) \notin nodes(t)$, and $label(t)(\sigma.j) = X_j$ for $0 \leqslant j \leqslant n$, then production $(X_0 \rightarrow X_1 \cdots X_n)$ is in P. $(X_0 \rightarrow X_1 \cdots X_n)$ is called the production *applied at* σ. An element $\sigma \in nodes(t)$ is called a *leaf* if $\sigma.1 \notin nodes(t)$. A derivation tree is called *complete* if the labels of all the leaves are in T. Let $\sigma_1 \cdots \sigma_n$ be the sequence consisting of all the leaves of t, ordered lexicographically. Now $yield(t)$ is the sequence $label(\sigma_1) \cdots label(\sigma_n)$.

3.2.3. DEFINITION. Let $G = (T,N,S,P)$ be a context-free grammar. The *language* $L(G)$ generated by G is the set

$$L(G) = \{yield(t) \mid t \text{ is a complete derivation tree of } G \text{ and } label(t)(\epsilon) = S\}.$$

3.3. Objects in POOL. A system that executes a POOL-program can be decomposed into *objects*. An object possesses some internal *data*, and also a *process*, that has the ability to act on these data. Each object has a clear separation between its inside and its outside: the data of an object cannot be accessed directly by (the process part of) other objects.

Interaction between objects takes place in the form of so-called *method-calls*. One object can send a message to another object, requesting it to perform a certain *method* (a kind of procedure). The result of the method execution is sent back to the sender. In this way one object can access the data of another object. However, because the object that receives a method call decides whether and when to execute this method, every object has its own responsibility of keeping its internal data in a consistent state.

The programs of POOL are called *units*. A unit consists of a number of *class definitions*. A *class* is a description of the behaviour of a set of objects. All objects in one class (the *instances* of that class) have the same data domain, the same methods for answering messages, and the same local process (called the object's *body*).

If a unit is to be executed, a new instance of the last class defined in the unit is created and its body is started. The body of an object can contain instructions for the creation of new objects. This makes it possible for the first object to start up the whole system.

When several objects have been created, their bodies may execute in parallel, thus introducing parallelism into the language. However, the sender of a message always waits until the destination object has returned its answer (this mechanism is known as *rendez-vous* message passing).

A number of standard classes are already predefined in the language (e.g. *Integer* and *Boolean*). They can be used in any program without defining them, but they also cannot be redefined.

The symbol **nil** denotes for each class a special object present in the system. Sending a message to such an object will always result in an error. The initial value of variables that are not parameters of a procedure is **nil**.

Because numbers are also objects, the addition of 3 and 4 is indicated in

POOL by sending a message with method name *add* and parameter 4 to the object 3.

We first give, in Section 3.4, the formal definition of POOL-⊥-CF. Section 3.5 contains some remarks concerning this definition, and the relation with POOL-T and POOL-⊥.

3.4. DEFINITION (POOL-⊥-CF). We assume that two finite sets, *LId* and *UId*, of syntactic elements are given. These sets correspond to the lower-identifiers resp. upper-identifiers in POOL-T. Elements of *LId* are strings starting with a lower case letter, elements of *UId* start with an upper case letter. We define: $Id = LId \cup UId$. Let $N_0 \in \mathbb{N}$ be given. The set *Int* of integers in POOL-⊥ is

$$Int = \{-N_0,...,-1,0,1,...,N_0\}.$$

N_0 can not be ω because that would lead to infinite sums and infinite merges. The set *Bool* of booleans is

$$Bool = \{\textbf{true},\textbf{false}\}.$$

Now the context-free grammar G, which defines POOL-⊥-CF, is

$$G = (T,N,U,P)$$

where

$T = Id \cup Int \cup Bool \cup \{$**root, unit, class, var, body, end, method, routine, local, in, nil return, post, if, then, else, fi, do, od, sel, les, or, answer, self, new**, ; , · , ← , ! , , , : $\}$

$N = \{U,RU,CDL,CD,MDL,MD,RDL,RD,PD,VDL,VD,SS,S,SE,$
$GCL,GC,AN,MIL,E,CO,SN,RC,MC,EL,CI,MI,RI,VI\}$

P : see Table 3.1

In Table 3.1, optional syntactical elements are enclosed in square brackets ('[' and ']').

Syntax of POOL-⊥

No	*Description*	*Syntactic Rule*
1	unit	$U \rightarrow RU$
2	root unit	$RU \rightarrow$ **root unit** CDL
3	class definition list	$CDL \rightarrow CD\,[\,,CDL\,]$
4	class definition	$CD \rightarrow$ **class** $CI\,[$ **var** $VDL\,][\,RDL\,][\,MDL\,]$

body *SS* **end** *CI*

5	method definition list	*MDL*→*MD* [*MDL*]
6	method definition	*MD*→**method** *MI PD* **end** *MI*
7	routine definition list	*RDL*→*RD* [*RDL*]
8	routine definition	*RD*→**routine** *RI PD* **end** *RI*
9	procedure denotation	*PD*→([*VDL*]) *CI* : [**local** *VDL* **in**] [*SS*] **return** *E* [**post** *SS*]
10	variable declaration list	*VDL*→*VD* [, *VDL*]
11	variable declaration	*VD*→*VI* : *CI*
12	statement sequence	*SS*→*S* [; *SS*]
13	statement	*S*→ *VI*←*E* | *AN* | **if** *E* **then** *SS* [**else** *SS*]**fi** | **do** *E* **then** *SS* **od** | *SE* | *SN* | *MC* | *RC*
14	select statement	*SE*→**sel** *GCL* **les**
15	guarded command list	*GCL*→*GC* [**or** *GCL*]
16	guarded command	*GC*→ *E* [*AN*]**then** *SS*
17	answer statement	*AN*→**answer** (*MIL*)
18	method identifier list	*MIL*→*MI* [, *MIL*]
19	expression	*E*→ *VI* | **self** | *CO* | **new** | *SN* | *MC* | *RC* | **nil**

20	constant	$CO \rightarrow c$ (for $c \in Bool \cup Int$)
21	send expression	$SN \rightarrow E\,!\,MI([EL])$
22	method call	$MC \rightarrow MI([EL])$
23	routine call	$RC \rightarrow CI \cdot RI([EL])$
24	expression list	$EL \rightarrow E[,EL]$
25	class identifier	$CI \rightarrow C$ (for $C \in UId$)
26	method identifier	$MI \rightarrow m$ (for $m \in LId$)
27	routine identifier	$RI \rightarrow r$ (for $r \in LId$)
28	variable identifier	$VI \rightarrow v$ (for $v \in LId$)

TABLE 3.1

3.5. REMARKS. (numbers refer to productions)

(1) In POOL-T a unit can, besides being a root unit, also be a specification unit or an implementation unit. This makes it possible to group a set of class definitions together into a logically coherent collection and to specify a clear interface with other units.

(2) The names of the classes defined in a unit must be different (similar context conditions in (5), (7), (9) and (10)). There are 4 standard classes: *Integer*, *Boolean*, *Read_File* and *Write_File*. The definitions of these classes can be found in Section 3.9.3. The standard classes can be used in any program without defining them, but they also cannot be redefined. Elements of *Int* are instances of class *Integer* and elements of *Bool* are instances of class *Boolean*.

(4) The class identifier following the **end** must be identical to the initial class identifier (similar context conditions in (6) and (8)).

(8) Routines are procedural abstractions related to a **class**, rather than to an individual object. They can be called also by objects from another class. Two objects can call and execute a routine concurrently as though each has its own version of the routine.

(9) The first variable declaration list is the formal parameter list, the second one contains the local variables of the method or routine. Only in the case of a method, a post-processing section may be present. The type of the return expression must be the class identifier in the procedure denotation.

(11) A strong typing mechanism is included in the language: each variable is associated to a class (its *type*) and may contain the names of objects of that class only.

(13) The statement *VI*←*E* is called an assignment and executed as follows: First the expression on the right hand side is evaluated and its result (a reference to an object) is determined. Then the variable is made to contain this reference.
The statement **do** *E* **then** *SS* **od** is the classical while statement.
A send expression, a method call and a routine call can occur as statement as well as expression. If they occur as statement, the corresponding expression is evaluated, and its result is discarded. So only the side-effects of the evaluation are important.

(14) The select statement is the most complicated construct in the language. It specifies the conditional answering of messages. A select statement is executed as follows:
- All the expressions (called: guards) of the guarded commands are evaluated in the order in which they occur in the text. If any of them results in **nil**, an error occurs.
- The guarded commands whose expressions result in **false** are discarded, they do not play a role in the remainder of the execution of the select statement. Only the ones with **true** (the open guarded commands) remain. If there are no open guarded commands, an error occurs.
- Now the object may choose to execute the (textually) first open guarded command without an answer statement, or it may choose to answer a message with a method identifier which occurs in one of the answer statements of an open guarded command that has no open guarded command without an answer statement before it. In the last case it must select the first open guarded command in which the method identifier of the chosen message occurs.
- If the object has chosen to answer a message, this is done.
- After that in either case the statement after **then** is executed, and the select statement terminates.

(17) An object executing an answer statement waits for a message with a method name that is present in the list. Then it executes the method (after initializing parameters). The result is sent back to the sender of the message, and the answer statement terminates.

(19) The symbol **self** always denotes the object that is executing the expression itself.
The expression **new** may only occur in a routine. When a **new** expression is evaluated, a new object of the class where the routine is defined, is created, and execution of its body is started. The result of the **new** expression is a reference to that new object.

(21) When a send expression is evaluated, first the expression before the '!' is evaluated. The result will be the destination for the message. Then

the expressions in the expression list are evaluated from left to right. The resulting objects will be the parameters of the message. Thereafter the message, consisting of the indicated method identifier and the parameters, is sent to the destination object. The answer of the destination object is the result of the send expression.

(22) An object may not send a message to itself. If an object wants to invoke one of its own methods, this can be done by means of a method call. A method call may not occur in a routine.

3.6. Attribute grammars. The complexity of the language POOL does not allow for a translation into process algebra which is as straightforward as in the case of the toy language of Section 2. Several problems arise, e.g. how to establish the relation between a method call and the corresponding method declaration, the semantics of a **new** expression, etc..

The main tool we will use in order to manage this complexity is the formalism of *attribute grammars*. This is not the place to give an extensive introduction into the theory of attribute grammars. For this we refer to e.g. [12, 14, 21].

Informally an attribute grammar is a context-free grammar in which we add to each nonterminal a finite number of *attributes*. For each occurrence of a nonterminal in a derivation tree these attributes have a *value*. With each production rule of the context-free grammar we associate a number of *semantic rules*. These rules define the values of the attributes. Some of the attributes are based on the attributes of the descendants of the nonterminal symbol. These are called *synthesized* attributes. Other attributes, called *inherited* attributes, are based on the attributes of the ancestors.

In the theory of abstract data types one presents specifications of the stack, Petri net people model the producer/consumer problem, and in the field of communication protocols one verifies the alternating bit protocol. The example one always encounters in an introduction into the theory of attribute grammars is the one, first presented in [21], in which the binary notation for numbers is defined. We do not want to break with this tradition, and will also give the famous example.

3.6.1. EXAMPLE. We start with a context-free grammar that generates binary notations for numbers: the terminal symbols are ·, 0, 1; the nonterminal symbols are B, L and N, standing respectively for bit, list of bits, and number; the starting symbol is N; and the productions are

$$B \to 0 \mid 1$$

$$L \to B \mid LB$$

$$N \to L \mid L \cdot L$$

Strings in the corresponding language are for instance '0', '010', '0.010' and '1010.101'. Now we introduce the following attributes

1. Each B has a 'value' $v(B)$ which is a rational number.
2. Each B has a 'scale' $s(B)$ which is an integer.

3. Each L has a 'value' $v(L)$ which is a rational number.
4. Each L has a 'length' $l(L)$ which is an integer.
5. Each L has a 'scale' $s(L)$ which is an integer.
6. Each N has a 'value' $v(N)$ which is a rational number.

These attributes can be defined as follows:

Syntactic Rules	Semantic Rules
$B \rightarrow 0$	$v(B) = 0$
$B \rightarrow 1$	$v(B) = 2^{s(B)}$
$L \rightarrow B$	$v(L) = v(B); s(B) = s(L); l(L) = 1$
$L_1 \rightarrow L_2 B$	$v(L_1) = v(L_2) + v(B); s(B) = s(L_1);$ $s(L_2) = s(L_1) + 1; l(L_1) = l(L_2) + 1$
$N \rightarrow L$	$v(N) = v(L); s(L) = 0$
$N \rightarrow L_1 \cdot L_2$	$v(N) = v(L_1) + v(L_2); s(L_1) = 0;$ $s(L_2) = -l(L_2)$

TABLE 3.2

(In the fourth and sixth rules subscripts have been used to distinguish between occurrences of like nonterminals.) If one looks for some time at this equations, one sees (hopefully) that for each complete derivation tree t with $label(t)(\epsilon) = N$ there is a unique valuation of the attributes such that the semantic rules hold. First one can compute the values of the attribute l, starting from the leaves of the tree (l is a *synthesized* attribute). Next one can compute the attribute s starting from root (s is an *inherited* attribute). Finally one computes, starting from the leaves, the synthesized attribute v. The v attribute of the root nonterminal gives the value of the string generated by the tree.

Below we give a formal definition of an attribute grammar. There are many (often essentially different) definitions possible. The following one is a simplified version of the definition presented in [14].

3.6.2. DEFINITION. The elements of an *attribute grammar G* are:

1. A context-free grammar $G_0 = (T,N,S_0,P)$.
2. A *semantic domain* (or set of data types) $D = <\Omega,\Phi>$, where Ω is a finite set of sets and Φ is a set of functions of type $V_1 \times \cdots \times V_m \rightarrow V_{m+1}$ for $m \geqslant 0$ and $V_i \in \Omega$. In the case $m = 0$, Φ can contain elements of V (for $V \in \Omega$). We demand that for each $V \in \Omega$ there is a $v \in V$ with $v \in \Phi$.
3. An *attribute description* consisting of
   a. Two finite disjoint sets *S-Att* and *I-Att* of *synthesized* or *s-attributes* resp. *inherited* or *i-attributes*; $Att = S\text{-}Att \cup I\text{-}Att$ is the set of *attributes*.
   b. For $X \in N$, $S(X)$ and $I(X)$ are subsets of *S-Att* resp. *I-Att*; $A(X) = S(X) \cup I(X)$ is the set of *attributes* of X. We demand $I(S_0) = \varnothing$.
   c. For each $\alpha \in Att$, $V(\alpha) \in \Omega$ is the (possibly infinite) set of *attribute values* of α.
4. First some intermediate terminology:
   For each production rule $p : X_0 \rightarrow X_1 \cdots X_n$, we define the set $A(p)$ of *attributes* of p, by

$$A(p) = \{<\alpha,j> \mid 0 \leqslant j \leqslant n, \alpha \in A(X_j)\}$$

   Intuitively $<\alpha,j>$ is an attribute of the occurrence of X_j on the j^{th} position in p. Furthermore the sets $INT(p)$ and $EXT(p)$ of *internal* resp. *external* attributes of p are defined by

$$INT(p) = \{<\alpha,j> \mid (j = 0 \wedge \alpha \in S(X_0)) \vee (1 \leqslant j \leqslant n \wedge \alpha \in I(X_j))\}$$

$$EXT(p) = \{<\alpha,j> \mid (j = 0 \wedge \alpha \in I(X_0)) \vee (1 \leqslant j \leqslant n \wedge \alpha \in S(X_j))\}$$

   A *semantic rule* for p is a string of the form

$$<\alpha,j> = f(<\alpha_1,k_1>,\ldots,<\alpha_m,k_m>) \qquad (*)$$

   with $<\alpha,j> \in INT(p)$, $m \geqslant 0$, $<\alpha_i,k_i> \in EXT(p)$ for $1 \leqslant i \leqslant m$, and $f \in \Phi$ is a function from $V(\alpha_1) \times \cdots \times V(\alpha_m)$ into $V(\alpha)$.
   Now we continue the definition:
   For each $p \in P$, $R(p)$ is a finite set of semantic rules for p. We demand that for each $p \in P$ and $<\alpha,j> \in INT(p)$, $R(p)$ contains exactly one semantic rule.

The definition above gives the 'syntax' of attribute grammars. To define the 'semantics' of an attribute grammar, we need again some terminology:

3.6.3. DEFINITION. Let G be an attribute grammar. Let t be a derivation tree of the corresponding context-free grammar. The *attributes* of t are defined by

$$A(t) = \{<\alpha,\sigma> \mid \sigma \in nodes(t), \alpha \in A(label(t)(\sigma))\}$$

(the notation $A(.)$ is clearly overloaded, but always means 'attributes of ... ')
A *decoration* of t is a function

$$val : A(t) \rightarrow \{v \mid \exists \alpha \in A(t) : v \in V(\alpha)\}$$

such that for each $<\alpha,\sigma> \in A(t)$, $val(\alpha,\sigma) \in V(\alpha)$.
Suppose $\sigma \in nodes(t)$ and $p : X_0 \rightarrow X_1 \cdots X_n$ is a production applied at σ. If $R(p)$ contains a semantic rule (*) (see Definition 3.6.2), then the string

$$<\alpha,\sigma.j> = f(<\alpha_1,\sigma.k_1>,...,<\alpha_m,\sigma.k_m>) \qquad (**)$$

is called a *semantic instruction* of t.

3.6.4. DEFINITION. A decoration *val* of t is called a *correct decoration* if for each semantic instruction (**) of t

$$val(\alpha,\sigma.j) = f(val(\alpha_1,\sigma.k_1),...,val(\alpha_m,\sigma.k_m))$$

(this is a serious equality, not a string!)

3.6.5. It follows from the Definitions 3.6.2 and 3.6.3, that for each attribute $<\alpha,\sigma>$ there is exactly one semantic instruction in $R(t)$ of the form $<\alpha,\sigma> = \cdots$. This means that each attribute of t is defined by exactly one equation in the system of equations $R(t)$. A sufficient condition to solve this system is that the system of equations contains no circularities. In [21], an algorithm is given which detects for an arbitrary attribute grammar whether or not the semantic rules can possibly lead to circular definition of some attributes. All the attribute grammars we will employ, contain no circularities, and therefore there is for each complete derivation tree precisely one correct decoration. This decoration can be computed if the functions which occur in the semantic rules are computable.

3.7. State Operator (SO). In [8], state operators λ_σ^m are introduced. Here m is member of a set M, the set of objects. These objects are very much like the objects in POOL: they posses some internal data, and there is a local process which can act upon these data. The object can block actions of the process, or rename then, depending on the data. $\lambda_\sigma^m(x)$ is a process corresponding to object m in state σ, executing process x. We can visualize this as in Figure 3.1.

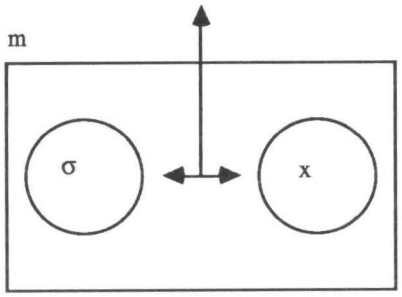

FIGURE 3.1

Below we give the formal definition of the state operators.

3.7.1. DEFINITION. Let M and Σ be two given sets. Elements of M are called *objects*, elements of Σ are called *states*. Suppose two functions *act* and *eff* are given

$$act: A \times M \times \Sigma \to A_{\tau\delta} \quad \text{(action function)}$$

$$eff: A \times M \times \Sigma \to \Sigma \quad \text{(effect function)}$$

Now we extend the signature with operators

$$\lambda_\sigma^m : P \to P \quad (\text{for } m \in M,\ \sigma \in \Sigma)$$

and extend the set of axioms by ($a \in A$; $x,y \in P$; $m \in M$; $\sigma \in \Sigma$)

$\lambda_\sigma^m(\delta) = \delta$	SO1
$\lambda_\sigma^m(\tau) = \tau$	SO2
$\lambda_\sigma^m(ax) = act(a,m,\sigma) \cdot \lambda_{eff(a,m,\sigma)}^m(x)$	SO3
$\lambda_\sigma^m(\tau x) = \tau \cdot \lambda_\sigma^m(x)$	SO4
$\lambda_\sigma^m(x+y) = \lambda_\sigma^m(x) + \lambda_\sigma^m(y)$	SO5

TABLE 3.3

The state operators can be defined in terms of the operators and constants of $ACP_\tau + RN$ (see [25]).

3.8. Parameters of the axiom system. We will relate to POOL-$\perp$ programs specifications in the signature of ACP + RN + CH + SO. The first thing we have to do is to specify the parameters of the axiom system. We will not give a complete list of all the atomic actions. The alphabet A of atomic actions simply consists of all the atomic actions we mention.

3.8.1. Objects. Let N_1 be a fixed natural number. N_1 gives an upperbound on the number of active (or non-standard) POOL objects which can be created during the execution of a POOL-$\perp$ program. The set *AObj* contains references to these potential objects.

$$AObj = \{\hat{0}, \hat{1}, ..., \hat{N}_1\}$$

The hats are needed to distinguish between the names of the non-standard objects and the names of the standard objects which are always present in the system:

$$SObj = Int \cup Bool \cup \{\textbf{nil}\} \cup \{\textbf{input}, \textbf{output}\}.$$

The set $Obj = SObj \cup AObj$ gives the domain of values of variables in POOL-$\perp$ programs. It is also the value domain of the chaining operator we will

employ; this means that the alphabet contains actions $\uparrow\alpha, \downarrow\alpha$, etc. for $\alpha \in Obj$).

3.8.2. Communication. Objects in POOL communicate by sending *frames* to each other. These frames are built up as follows

destination	type of message	message	sender

The field 'sender' contains a reference to the object which sends the message; the field 'destination' contains a reference to the object which reads the message. There are two types of messages:

mc: The sender asks the destination to perform a method-call. The field 'message' contains the name of the method together with the actual parameters. So an *mc*-frame looks as follows

$$(\alpha, mc, m(\alpha_1,\ldots,\alpha_n), \beta) \tag{3.8.2.1}$$

an: After an object has executed a method call, an *an*-frame is sent back to the object which originated the method call. The field 'message' contains the answer (a reference to an object):

$$(\beta, an, \gamma, \alpha) \tag{3.8.2.2}$$

Let N_2 be a fixed natural number. N_2 gives an upperbound on the length of a variable declaration list of a procedure denotation. The set $\mathfrak{M}$ of messages that occurs in a method call frame is:

$$\mathfrak{M} = \{m(\alpha_1,\ldots,\alpha_n) \mid m \in LId, 0 \leqslant n \leqslant N_2, \alpha_1,\ldots,\alpha_n \in Obj\} \tag{3.8.2.3}$$

and the set $\mathfrak{F}$ of frames is:

$$\mathfrak{F} = \{(\alpha, mc, d, \beta) \mid \alpha, \beta \in Obj, d \in \mathfrak{M}\} \cup \{(\beta, an, \gamma, \alpha) \mid \alpha, \beta, \gamma \in Obj\} \tag{3.8.2.4}$$

For each frame $f \in \mathfrak{F}$, we have atomic actions $read(f)$, $send(f)$ and $comm(f)$. The communication function on these actions is given by

$$\gamma(read(f), send(f)) = comm(f) \quad \text{for } f \in \mathfrak{F} \tag{3.8.2.5}$$

The set J of forbidden actions that will be encapsulated is

$$J = \{read(f), send(f) \mid f \in \mathfrak{F}\} \tag{3.8.2.6}$$

3.8.3. Renamings. A POOL object is fully determined by its class and its name. For each class we will specify a process that gives the *general* behaviour of the instances (the objects) of that class. Now the only thing we have to do in order to define the process corresponding to a *specific* object, is to give a renaming function which renames the actions of the process which is related to the class of that object. This renaming function gives the object its identity, a name. The frames which are sent and received by an object, contain the name of that object. But since at the level of a class this name is not known, the process related to a class contains 'incomplete' *read* and *send* actions: actions

$rd(if)$ and $sn(if)$, where if is an incomplete frame in which the field that gives the identity of the object is absent. Actions of the form $rd(if)$ and $sn(if)$ do not communicate.

For each $\alpha \in Obj$ we define a renaming function f_α by:

$$f_\alpha(sn(\beta,mc,m(\alpha_1,...,\alpha_n))) = send(\beta,mc,m(\alpha_1,...,\alpha_n),\alpha) \quad (3.8.3.1)$$

$$f_\alpha(rd(mc,m(\alpha_1,...,\alpha_n),\beta)) = read(\alpha,mc,m(\alpha_1,...,\alpha_n),\beta) \quad (3.8.3.2)$$

$$f_\alpha(sn(\beta,an,\gamma)) = send(\beta,an,\gamma,\alpha) \quad (3.8.3.3)$$

$$f_\alpha(rd(an,\beta,\gamma)) = read(\alpha,an,\beta,\gamma) \quad (3.8.3.4)$$

If an object α executes a **self** expression, the corresponding process on class level contains an alternative composition of actions $eqs(\beta)$ for $\beta \in Obj$. The following equations for the renaming functions make that, for a specific instance of the class, the action which will be actually performed is the right one.

$$f_\alpha(eqs(\beta)) = \begin{cases} skip & \text{if } \beta=\alpha \\ \delta & \text{otherwise} \end{cases} \quad (3.8.3.5)$$

If an object α answers a method call, the result of the return expression in the procedure denotation has to be sent back to the sender of the method call. To model this we introduce renaming functions g_α. The function g_α interprets a $\uparrow\beta$ action as a $sn(\alpha,an,\beta)$ action:

$$g_\alpha(\uparrow\beta) = sn(\alpha,an,\beta) \quad (3.8.3.6)$$

3.8.4. Process Creation. For $d \in \mathfrak{N} \times AObj$ we introduce atomic actions $create(d)$, $create^*(d)$ and $\overline{create}(d)$. $create(d)$ stands for: ask for the creation of a process on basis of initial information d. $create^*(d)$ means: receive a request for creation. $\overline{create}(d)$ indicates that process creation has taken place.

Elements of $\mathfrak{N}$ (see Definition 3.2.2) play the role of formal variables in the process algebra specification that we will construct in order to give the semantics of POOL-$\perp$. In general the process denoted by the first parameter of a create action will give the behaviours of a certain class, and the second parameter gives the name of the instance of that class to be created.

We extend the communication function by

$$\gamma(create(d), create^*(d)) = \overline{create}(d) \quad (3.8.4.1)$$

Create actions are not involved in any other proper communication. Let

$$K = \{create(d), create^*(d) \mid d \in \mathfrak{N} \times AObj\} \quad (3.8.4.2)$$

Actions from K will be encapsulated.

Our way of dealing with process creation in POOL is inspired by the mechanism described in [10]. We have chosen however not to use the process creation operator E_ϕ presented there, because of the lack of proof rules for this operator.

3.8.5. State Operator. In the semantical description of the toy language of Section 2 the state of the memory was a parameter of the formal variables in the specification. In principle this approach can also be followed in the case of the language POOL-$\perp$. But since in POOL objects of a different class have, in general, different variables; and the language contains recursion, which leads to the creation of new instances of variables, the memory state of a POOL object can become rather complicated. For this reason we prefer to keep track of the memory state in a different way: namely by means of a state operator. For each variable $v \in LId$ and value $\alpha \in Obj$, λ_α^v represents a memory cell with name v in state α. A value β can be assigned to variable v by means of an atomic action $ass(v,\beta)$:

$$\lambda_\alpha^v(ass(v,\beta)\cdot x) = skip \cdot \lambda_\beta^v(x) \tag{3.8.5.1}$$

If in the evaluation of an expression the value of a variable v is needed, this can be expressed at the level of process algebra by means of an alternative composition of actions $eqv(v,\beta)$. The following equation makes that in an environment with variable cell v, the correct action will be actually performed:

$$\lambda_\alpha^v(eqv(v,\beta)\cdot x) = \begin{cases} skip \cdot \lambda_\alpha^v(x) & \text{if } \alpha = \beta \\ \delta & \text{otherwise} \end{cases} \tag{3.8.5.2}$$

Notice that in the case of nested λ_α^v operators, actions $ass(v,\beta)$ and $eqv(v,\beta)$ interact with the innermost λ_α^v operator. This is relevant for nested method calls, etc..

The initial object, which starts up the system, has name $\hat{0}$. An object *counter* counts the number of objects which have been created. It also provides an environment in which new objects obtain new names. An error occurs when more than N_1 objects have been created. For $n \in \mathbb{N}$ we have

$$\lambda_n^{counter}(\overline{create}(X,\alpha)\cdot x) = \begin{cases} skip \cdot \lambda_{n+1}^{counter}(x) & \text{if } \alpha = \hat{n} \wedge n < N_1 \\ error \cdot \lambda_{n+1}^{counter}(x) & \text{if } n = N_1 \\ \delta & \text{otherwise} \end{cases} \tag{3.8.5.3}$$

3.8.6. Formal Variables. The set Ξ of formal variables of the process algebra specifications related to POOL-$\perp$ consists of the elements of $\mathfrak{N}$ (as defined in Section 3.2.2), possibly sub- and superscripted with elements of LId and Obj^*. Formally we have:

$$\Xi = \mathfrak{N} \cup \mathfrak{N} \times LId \cup \mathfrak{N} \times (Obj^*) \cup \mathfrak{N} \times LId \times (Obj^*) \tag{3.8.6.1}$$

We define $node : \Xi \to \mathfrak{N}$ to be the projection function which relates to each variable the corresponding element of $\mathfrak{N}$.

3.8.7. Note. From now on, when we speak about a POOL-⊥ program, what we mean is an extended program, in which the class definition list begins with the class definitions of the standard classes (see Section 3.9.3).

3.9. Attribute description. Table 3.4 contains a list of all the attributes we will employ for the semantical description of POOL-⊥. In Section 3.9.1 we give a detailed description of these attributes. Section 3.9.2 contains all the semantical rules which were not already given in Section 3.9.1, and in Section 3.9.3 the standard classes are defined.

Name attr.	*i/s*	*Description*	*Attribute type*	*Nonterminals*
$[\![\cdot]\!]$	i	Key variable	$\mathfrak{N}$	N-{U}
id	s	Identifier	*LId*	{VI,RI,MI,CI, VD,RD,MD,CD}
vd	s	Variable declarations	LId^*	{VDL}
pd	s	Procedure declaration	$\mathfrak{N}\times\mathbb{N}$	{PD,RD,MD}
rd	s	Routine declarations	$LId\to\mathfrak{N}\times\mathbb{N}$	{RDL,CD}
md	s	Method declarations	$LId\to\mathfrak{N}\times\mathbb{N}$	{MDL,CD}
cd	s	Class declarations	$UId\to\mathfrak{N}$	{CDL}
rdc	s	Routine decl. of a CDL	$UId\times LId\to\mathfrak{N}\times\mathbb{N}$	{CDL}
mdc	s	Method decl. of a CDL	$UId\times LId\to\mathfrak{N}\times\mathbb{N}$	{CDL}
cdf	i	Class definitions	$UId\to\mathfrak{N}$	N-{U,RU}
rdf	i	Routine definitions	$UId\times LId\to\mathfrak{N}\times\mathbb{N}$	N-{U,RU}
mdf	i	Method definitions	$UId\times LId\to\mathfrak{N}\times\mathbb{N}$	N-{U,RU}
class	i	Class	*UId*	N-{U,RU,CDL,CD}
l	s	Length	$\mathbb{N}$	{EL}
mis	s	Method ident. set	$Pow(LId)$	{MIL,AN,GC}
misl	s	Method ident. set list	$(Pow(LId))^*$	{GCL}
peq	s	Process equations	Sets of eq. over ACP+RN+CH+SO with variables in Ξ	N-{U,VI,RI,MI,CI, VD,VDL,RD,RDL, MD,MDL}
spec	s	Specification	Sets of eq. over ACP+RN+CH+SO with variables in Ξ	N-{VI,RI,MI,CI, VD,VDL,RD,RDL, MD,MDL}

TABLE 3.4

3.9.1. REMARKS.

1. We make the names of the nodes in a derivation tree explicit by means of an inherited attribute $[\![.]\!]$. With each node in a derivation tree we will relate a number of process algebra equations with variables in Ξ. The values of the attribute $[\![.]\!]$ (which are elements of Ξ) will be the 'most important' or 'key' variables in this specification. The semantic rules for this attribute are as follows
   - For production $U \rightarrow RU$ the rule is $[\![RU]\!] = 1$
   - If $X_0 \rightarrow X_1 \cdots X_n$ is a production with $X_0 \neq U$, and if $X_i \in N$ for certain $1 \leqslant i \leqslant n$ then we have the rule $[\![X_i]\!] = [\![X_0]\!].i$.
2. The value of synthesized attribute *id* is (one of) the identifier(s) generated by the corresponding nonterminal.
3. Attribute *vd* collects variables declared in a variable declaration list.
4. Attribute *pd* gives the information concerning a procedure declaration that we need: a formal variable denoting the process related to the procedure, and the number of parameters of the procedure.
5. The attribute *rd* gives for each routine in a routine definition list the essential information: a process variable and the number of parameters. The value of *rd* is arbitrary for elements of *LId* which are not the name of a routine.
6. The meaning of attribute *md* is similar to the meaning of *rd*.
7. The attribute *cd* gives the essential information for each class definition in a class definition list: the process corresponding to the general behaviour of that class. The value of *cd* is arbitrary for elements of *UId* which are not present in the class definition list.
8. Attribute *rdc* is like *rd* but now for a list of class definitions.
9. Attribute *mdc* is like *md* but now for a list of class definitions.
10. All the information that is gathered in the s-attribute *cd* is distributed over the parse tree by means of the i-attribute *cdf*:
    - For production $RU \rightarrow$ **root unit** CDL we have the rule $cdf(CDL) = cd(CDL)$.
    - If $X_0 \rightarrow X_1 \cdots X_n$ is a production $(X_0 \neq U, RU)$, and if $X_i \in N$ for certain $1 \leqslant i \leqslant n$, then $cdf(X_i) = cdf(X_0)$.
11. Attribute *rdf* is like attribute *cdf*.
12. Attribute *mdf* is like attribute *cdf*.
13. In order to define the semantics of, for example, a new expression, we need to know in which class definition this expression occurs. Therefore we define an i-attribute *class* with domain *UId*:
    - For production

      $CD \rightarrow$ **class** CI_1 [**var** VDL][RDL][MDL] **body** SS **end** CI_2

      we have rules

      $[class(VDL)=]\,[class(RDL)=]\,[class(MDL)=]\,class(SS)=id(CI_1)$
    - If $X_0 \rightarrow X_1 \cdots X_n$ is a production $(X_0 \neq U, RU, CDL, CD)$, and if $X_i \in N$ for certain $1 \leqslant i \leqslant n$, then $class(X_i) = class(X_0)$.

14. In the semantic rules for the send expression we need information about the length of the expression list. This information is contained in attribute *l*.
15. The attribute *mis* gives the method identifiers which occur in the method identifier list of an answer statement. The attribute is used to define the semantics of the select statement.
16. The attribute *misl* gives a list of the method identifier sets which occur in the answer statements in a guarded command list.
17. The value of the attribute *peq* is a set of equations in the signature of ACP+RN+CH+SO with variables in Ξ. We will define the attribute in such a way that for each nonterminal X:

$$(Y = t_Y) \in peq(X) \Rightarrow node(Y) = [\![X]\!].$$

Furthermore we take care that for each nonterminal X, $peq(X)$ never contains two equations for the same variable. These conditions make that the union for all the nodes in a derivation tree of the values of attribute *peq* never contains two equations for the same variable.

18. The s-attribute *spec* collects the values of attribute *peq*. If w is a POOL-program, then the value of the attribute *spec* belonging to the root of the derivation tree of w (which has label U) is the process specification that we relate to w. The process expression associated by mapping $SPECS_C$ to w is $<[\![RU]\!] \mid spec(U)>$. We have the following semantic rules:
    - Let $X_0 \rightarrow X_1 \cdots X_n$ be a production such that $X_0 \neq U$ has attribute *spec*. Let $S \subseteq \{1,...,n\}$ be the set of indices i for which X_i has an attribute *spec*. Then:

$$spec(X_0) = peq(X_0) \cup \bigcup_{i \in S} spec(X_i)$$

    - For production $U \rightarrow RU$ we have:

$$spec(U) = spec(RU) \cup$$
$$\cup \{(X = \delta) \mid X \in \Xi \text{ and there is no equation for } X \text{ in } spec(RU)\}.$$

3.9.2. Semantic rules. In case a production contains an optional syntactical element, we will often use a fraction notation in the semantic rules: the numerator corresponds to the semantic rule for the production *with* the optional element, the denominator corresponds to the production *without* the optional element. **In case of a semantic rule** $peq(X) = \{E_1, E_2, ...\}$, **we only write down the equations** $E_1, E_2, ...$**!!!** Numbers refer to the numbering of productions in Table 3.1.

$VI \rightarrow v \quad (v \in LId)$ (28)

$$id(VI) = v$$

$RI \rightarrow r \quad (r \in LId)$ (27)

$$id(RI) = r$$

$MI \rightarrow m \quad (m \in LId)$ (26)

$$id(MI) = m$$

$CI \rightarrow C \quad (C \in UId)$ (25)

$$id(CI) = C$$

$EL_0 \rightarrow E[, EL_1]$ (24)

$$l(EL_0) = 1[+ l(EL_1)]$$

$$[\![EL_0]\!] = [\![E]\!][\cdot [\![EL_1]\!]]$$

○ We state again that the equation for $[\![EL_0]\!]$ is not to be considered as a semantic rule defining attribute $[\![.]\!]$, but as an element of the set defining attribute *peq*. The equation says that execution of an expression list consists of sequential execution of all the expressions from left to right.

$RC \rightarrow CI \cdot RI()$ (23.1)

Let

$$rdf(RC)(id(CI), id(RI)) = (X, n)$$

then

$$[\![RC]\!] = skip \cdot X$$

○ Process X corresponds to execution of routine $id(RI)$ of class $id(CI)$. In a correct POOL-⊥ program n will be 0. The *skip* action is needed in order to keep the specification guarded.

$RC \rightarrow CI \cdot RI(EL)$ (23.2)

Let

$$rdf(RC)(id(CI), id(RI)) = (X, n)$$

then

$$[\![RC]\!] = [\![EL]\!] \ggg_{\alpha_1,\ldots,\alpha_n} X_{\alpha_1,\ldots,\alpha_n}$$

○ First the expressions of the parameter list are evaluated. Thereafter the routine call is executed, with the actual parameters instantiated. Process $X_{\alpha_1,\ldots,\alpha_n}$ corresponds to execution of routine $id(RI)$ of class $id(CI)$ with actual parameters $\alpha_1,\ldots,\alpha_n$. In a correct program the number of actual parameters equals the number of formal parameters: $l(EL) = n$.

$MC \rightarrow MI()$ (22.1)

Let

$$mdf(MC)(class(MC), id(MI)) = (X, n)$$

then

$$[\![MC]\!] = skip \cdot X$$

○ Method calls are treated in exactly the same way as routine calls.

$MC \rightarrow MI(EL)$ (22.2)

Let

$$mdf(MC)(class(MC), id(MI)) = (X, n)$$

then

$$[\![MC]\!] = [\![EL]\!] \ggg_{\alpha_1,\ldots,\alpha_n} X_{\alpha_1,\ldots,\alpha_n}$$

$SN \rightarrow E\,!\,MI()$ (21.1)

Let

$$id(MI) = m$$

then

$$[\![SN]\!] = [\![E]\!] \ggg_{\alpha} [\![SN]\!]_{\alpha}$$

$$[\![SN]\!]_{\alpha} = \begin{cases} error & \text{if } \alpha = \textbf{nil} \\ sn(\alpha, mc, m()) \cdot \sum_{\beta \in Obj} rd(an, \beta, \alpha) \cdot \uparrow\beta & \text{otherwise} \end{cases}$$

○ First the expression on the left is evaluated. If the result is **nil** an error occurs. Otherwise the result of the expression is the destination of the message. Now the message is sent and the answer awaited. This answer (if it comes) is the result of the send expression. In a correct POOL program the type of expression E will be a class that contains a method m without parameters.

$SN \rightarrow E\,!\,MI\,(EL\,)$ (21.2)

Let

$$id(MI) = m$$

$$l(EL) = n$$

then

$$[\![SN]\!] = [\![E]\!] \ggg_{\alpha} [\![SN]\!]_{\alpha}$$

$$[\![SN]\!]_{\textbf{nil}} = error$$

and for $\alpha \neq \textbf{nil}$:

$$[\![SN]\!]_{\alpha} = [\![EL]\!] \ggg_{\alpha_1,\ldots,\alpha_n} sn(\alpha, mc, m(\alpha_1,\ldots,\alpha_n)) \cdot \sum_{\beta \in Obj} rd(an, \beta, \alpha) \cdot \uparrow\beta$$

○ Like 21.1 but now with parameters.

$CO \rightarrow c \quad (c \in Bool \cup Int)$ (20)

$$[\![CO]\!] = \uparrow c$$

$E \rightarrow VI$ (19.1)

Let

$$id(VI) = v$$

then

$$[\![E]\!] = \sum_{\alpha \in Obj} eqv(v, \alpha) \cdot \uparrow \alpha$$

○ Cf. Equation 3.8.5.2.

$E \rightarrow$ **self** (19.2)

$$[\![E]\!] = \sum_{\alpha \in Obj} eqs(\alpha) \cdot \uparrow \alpha$$

○ Cf. Equation 3.8.3.5.

$E \rightarrow CO$ (19.3)

$$[\![E]\!] = [\![CO]\!]$$

$E \rightarrow$ **new** (19.4)

Let

$$cdf(E)(class(E)) = X$$

then

$$[\![E]\!] = \sum_{\alpha \in AObj} create(X, \alpha) \cdot \uparrow \alpha$$

○ Process creation takes place in an environment (cf. Equation 3.8.5.3) that takes care of the naming of new objects, and always allows only one of the actions $create(X, \alpha)$ to occur. See also the definition of attribute $peq(RU)$ at the end of this section.

$E \rightarrow SN$ (19.5)

$$[\![E]\!] = [\![SN]\!]$$

$E \rightarrow MC$ (19.6)

$$[\![E]\!] = [\![MC]\!]$$

$E \rightarrow RC$ (19.7)

$$[\![E]\!] = [\![RC]\!]$$

$E \rightarrow$ **nil** (19.8)

$$[\![E]\!] = \uparrow \mathbf{nil}$$

$MIL_0 \rightarrow MI\,[\,,\,MIL_1\,]$ (18)

Let

$$id(MI) = m$$

$$mdf(MIL_0)(class(MIL_0),m) = (X\ n)$$

then

$$mis(MIL_0) = \{m\}[\,\cup mis(MIL_1)\,]$$

$$[\![MIL_0]\!]_m = \sum_{\alpha_1,\ldots,\alpha_n,\alpha \in Obj} rd(mc,m(\alpha_1,\ldots,\alpha_n),\alpha)\cdot \rho_{g_\alpha}(X_{\alpha_1,\ldots,\alpha_n})$$

$$[\![MIL_0]\!]_{\bar{m}} = \frac{[\![MIL_1]\!]_{\bar{m}}}{\delta} \quad \text{if } \bar{m} \neq m$$

○ For the m which occur in the method identifier list, $[\![MIL_0]\!]_m$ gives the process that describes the answering of a message m: first a method call with identifier m is read, then the method is executed, and the result is returned to the sender (cf. Equation 3.8.3.6). For m not in MIL_0, $[\![MIL_0]\!]_m = \delta$.

$AN \rightarrow$ **answer** (MIL) (17)

$$mis(AN) = mis(MIL)$$

$$[\![AN]\!]_m = [\![MIL]\!]_m$$

$$[\![AN]\!] = \sum_{m \in LId} [\![MIL]\!]_m$$

○ The variables $[\![AN]\!]_m$ will be needed for the description of the select statement.

The semantic rules for the nonterminals MIL, AN, GC, GCL and SE are rather complicated. This is because the semantics of the select statement is to a large extent not compositional: it is not defined in terms of the semantics of the answer statements which occur in the guarded commands, but in terms of the individual method identifiers of these answer statements. The formalism of attribute grammars has difficulties in dealing with such a case. It seems that the 'conditional answer statement', which replaces the select statement in a more recent version of the POOL language, does have a compositional semantics. Moreover the semantical description of this construct will be much shorter than the one of the select statement.

$GC \rightarrow E$ **then** SS (16.1)

$$mis(GC) = \varnothing$$

$$[\![GC]\!] = [\![E]\!]$$

$$[\![GC]\!]_\epsilon = skip \cdot [\![SS]\!]$$

○ The prefix *skip* in the equation for variable $[\![GC]\!]_\epsilon$ is needed because we want to give a different semantics to the following two select statements:

sel

true answer(m_1) **then** $x \leftarrow 1$ **or**

true answer(m_2) **then** $x \leftarrow 2$

les

and

sel

true answer(m_1) **then** $x \leftarrow 1$ **or**

true then answer(m_2) ; $x \leftarrow 2$

les

If the environment offers a method call with method identifier m_1, but no method call with method identifier m_2, then the first select statement will answer m_1. The second select statement however may choose to execute the second guarded command, which will result in a deadlock.

$GC \rightarrow E\ AN$ **then** SS (16.2)

$$mis(GC) = mis(AN)$$

$$[\![GC]\!] = [\![E]\!]$$

$$[\![GC]\!]_m = [\![AN]\!]_m \cdot [\![SS]\!]$$

$GCL \rightarrow GC$ (15.1)

Let

$$mis(GC) = M$$

then

$$misl(GCL) = (M)$$

$$[\![GCL]\!] = [\![GC]\!]$$

$$[\![GCL]\!]^{\alpha}_{\epsilon} = \begin{cases} [\![GC]\!]_{\epsilon} & \text{if } \alpha = \textbf{true} \wedge M = \varnothing \\ \delta & \text{otherwise} \end{cases}$$

$$[\![GCL]\!]^{\alpha}_{m} = \begin{cases} [\![GC]\!]_{\epsilon} & \text{if } \alpha = \textbf{true} \wedge M = \varnothing \\ [\![GC]\!]_{m} & \text{if } \alpha = \textbf{true} \wedge m \in M \\ \delta & \text{otherwise} \end{cases}$$

○ See remark about production 14.

$GCL_0 \rightarrow GC$ **or** GCL_1 (15.2)

Let

$$mis(GC) = M_0$$

$$misl(GCL_1) = (M_1,...,M_n)$$

then

$$misl(GCL_0) = (M_0,M_1,...,M_n)$$

$$[\![GCL_0]\!] = [\![GC]\!] \cdot [\![GCL_1]\!]$$

$$[\![GCL_0]\!]_\epsilon^{\alpha_0,...,\alpha_n} = \begin{cases} [\![GC]\!]_\epsilon & \text{if } \alpha_0 = \textbf{true} \wedge M_0 = \varnothing \\ [\![GCL_1]\!]_\epsilon^{\alpha_1,...,\alpha_n} & \text{otherwise} \end{cases}$$

$$[\![GCL_0]\!]_m^{\alpha_0,...,\alpha_n} = \begin{cases} [\![GC]\!]_\epsilon & \text{if } \alpha_0 = \textbf{true} \wedge M_0 = \varnothing \\ [\![GC]\!]_m & \text{if } \alpha_0 = \textbf{true} \wedge m \in M_0 \\ [\![GCL_1]\!]_m^{\alpha_1,...,\alpha_n} & \text{otherwise} \end{cases}$$

○ See remark about production 14.

SE→**sel** *GCL* **les** (14)

Let

$$misl(GCL) = (M_1,...,M_n)$$

then

$$[\![SE]\!] = [\![GCL]\!] \ggg_{\alpha_1,...,\alpha_n} [\![SE]\!]_{\alpha_1,...,\alpha_n}$$

$$[\![SE]\!]_{\alpha_1,...,\alpha_n} = error \qquad \text{if } (\exists i : \alpha_i = \textbf{nil}) \vee (\forall i : \alpha_i = \textbf{false})$$

$$[\![SE]\!]_{\alpha_1,...,\alpha_n} = \sum_{m \in LId \cup \{\epsilon\}} [\![GCL]\!]_m^{\alpha_1,...,\alpha_n} \qquad \text{otherwise}$$

○ Execution of a select statement starts with evaluation of the expressions in the guarded commands. If one expression yields **nil** or all expressions yields **false** an error occurs. The intuitive meaning of variable

$$[\![GCL]\!]_\epsilon^{\alpha_1,...,\alpha_n}$$

is: Execute the first open guarded command without an answer statement, assuming that evaluation of the expressions yields values $\alpha_1,...,\alpha_n$. If there is no open guarded command without an answer statement the result is δ. Analogously, for $m \in LId$, the intuitive meaning of variable

$$[\![GCL]\!]_m^{\alpha_1,...,\alpha_n}$$

is: Execute the first open guarded command without an answer statement or with m in the method identifier list of the answer statement.

S→ *VI*←*E* (13.1)

Let

$$id(VI) = v$$

then

$$[\![S]\!] = [\![E]\!] \gg\!\gg_\alpha ass(v, \alpha)$$

○ Cf. Equation 3.8.5.1.

$S \rightarrow AN$ (13.2)

$$[\![S]\!] = [\![AN]\!]$$

$S \rightarrow$ **if** E **then** SS_1 [**else** SS_2] **fi** (13.3)

$$[\![S]\!] = [\![E]\!] \gg\!\gg_\alpha [\![S]\!]_\alpha$$

$$[\![S]\!]_\alpha = \begin{cases} [\![SS_1]\!] & \text{if } \alpha = \textbf{true} \\ \dfrac{[\![SS_2]\!]}{skip} & \text{if } \alpha = \textbf{false} \\ error & \text{otherwise} \end{cases}$$

$S \rightarrow$ **do** E **then** SS **od** (13.4)

$$[\![S]\!] = [\![E]\!] \gg\!\gg_\alpha [\![S]\!]_\alpha$$

$$[\![S]\!]_\alpha = \begin{cases} [\![SS]\!] \cdot [\![S]\!] & \text{if } \alpha = \textbf{true} \\ skip & \text{if } \alpha = \textbf{false} \\ error & \text{otherwise} \end{cases}$$

$S \rightarrow SE$ (13.5)

$$[\![S]\!] = [\![SE]\!]$$

$S \rightarrow SN$ (13.6)

$$[\![S]\!] = [\![SN]\!] \gg\!\gg (\sum_{\alpha \in Obj} \downarrow\alpha)$$

○ The send expression is evaluated and afterwards the result is discarded.

$S \rightarrow MC$ (13.7)

$$[\![S]\!] = [\![MC]\!] \gg\!\gg (\sum_{\alpha \in Obj} \downarrow\alpha)$$

$S \rightarrow RC$ (13.8)

$$[\![S]\!] = [\![RC]\!] \gg\!\gg (\sum_{\alpha \in Obj} \downarrow\alpha)$$

$SS_0 \rightarrow S$ [; SS_1] (12)

$$[\![SS_0]\!] = [\![S]\!] [\cdot [\![SS_1]\!]]$$

$VD \rightarrow VI : CI$ (11)

$$id(VD) = id(VI)$$

$VDL_0 \rightarrow VD\ [\ ,\ VDL_1\ ]$ (10)

$$vd(VDL_0) = (id(VD))[\star vd(VDL_1)]$$

○ Here $\star$ denotes concatenation of lists.

$PD \rightarrow ([\ VDL_1\ ])\ CI : [\ \textbf{local}\ VDL_2\ \textbf{in}\ ]\ [\ SS_1\ ]\ \textbf{return}\ E\ [\ \textbf{post}\ SS_2\ ]$ (9)

Let

$$vd(VDL_1) = (v_1,\ldots,v_n)$$

$$vd(VDL_2) = (w_1,\ldots,w_k)$$

($n = 0$ or $k = 0$ if there is no VDL_1 resp. VDL_2)
then

$$pd(PD) = ([\![PD]\!]\ n)$$

$$[\![PD]\!]_{\alpha_1,\ldots,\alpha_n} = \lambda^{v_1}_{\alpha_1} \circ \cdots \circ \lambda^{v_n}_{\alpha_n} \circ \lambda^{w_1}_{\mathbf{nil}} \circ \cdots \circ \lambda^{w_k}_{\mathbf{nil}} ([\![SS_1]\!] \cdot][\![E]\!][\cdot [\![SS_2]\!]])$$

○ Process $[\![PD]\!]_{\alpha_1,\ldots,\alpha_n}$ corresponds to execution of the procedure with parameters $\alpha_1,\ldots,\alpha_n$.

$RD \rightarrow \textbf{routine}\ RI_1\ PD\ \textbf{end}\ RI_2$ (8)

$$id(RD) = id(RI_1)$$

$$pd(RD) = pd(PD)$$

$RDL_0 \rightarrow RD\ [\ RDL_1\ ]$ (7)

$$rd(RDL_0) = \frac{rd(RDL_1)}{rd_0}\{pd(RD)/id(RD)\}$$

○ We use the notation for function modification of Section 2.4.1. rd_0 is an arbitrarily chosen element out of the domain of attribute *rd*. We use similar conventions in the semantic rules for productions 5,4 and 3.

$MD \rightarrow \textbf{method}\ MI_1\ PD\ \textbf{end}\ MI_2$ (6)

$$id(MD) = id(MI_1)$$

$$pd(MD) = pd(PD)$$

$MDL_0 \rightarrow MD\ [\ MDL_1\ ]$ (5)

$$md(MDL_0) = \frac{md(MDL_1)}{md_0}\{pd(MD)/id(MD)\}$$

$CD \rightarrow \textbf{class}\ CI_1\ [\ \textbf{var}\ VDL\ ]\ [\ RDL\ ]\ [\ MDL\ ]\ \textbf{body}\ SS\ \textbf{end}\ CI_2$ (4)

Let

$$vd(VDL) = (v_1,\ldots,v_n)$$

then

$$id(CD) = id(CI_1)$$

$$md(CD) = \frac{md(MDL)}{md_0}$$

$$rd(CD) = \frac{rd(RDL)}{rd_0}$$

$$[\![CD]\!] = \lambda^{v_1}_{\mathbf{nil}} \circ \cdots \circ \lambda^{v_n}_{\mathbf{nil}}([\![SS]\!])$$

$CDL_0 \rightarrow CD[\ , CDL_1\ ]$ (3)

$$cd(CDL_0) = \frac{cd(CDL_1)}{cd_0}\{[\![CD]\!]/id(CD)\}$$

$$mdc(CDL_0) = \frac{mdc(CDL_1)}{mdc_0}\{md(CD)/id(CD)\}$$

$$rdc(CDL_0) = \frac{rdc(CDL_1)}{rdc_0}\{rd(CD)/id(CD)\}$$

$$[\![CDL_0]\!] = \frac{[\![CDL_1]\!]}{[\![CD]\!]}$$

○ Process $[\![CDL_0]\!]$ gives the behaviour of the last class defined in CDL_0.

$RU \rightarrow$ **root unit** CDL (2)

Let

$$cd(CDL)(Integer) = I$$

$$cd(CDL)(Boolean) = B$$

$$cd(CDL)(Read\_File) = R$$

$$cd(CDL)(Write\_File) = W$$

$$\mathcal{C} = \{cd(CDL)(C) \mid C \in UId\}$$

$$ACTIVE = \|_{\alpha \in AObj} (\sum_{X \in \mathcal{C}} create^*(X, \alpha) \cdot \rho_{f_\alpha}(X))$$

$$STANDARD = (\|_{\alpha \in Int} \rho_{f_\alpha}(I))\|\rho_{f_{\mathbf{bool}}}(B)\|\rho_{f_{\mathbf{false}}}(B)\|\rho_{f_{\mathbf{true}}}(B)\|\rho_{f_{\mathbf{read}}}(R)\|\rho_{f_{\mathbf{write}}}(W)$$

then

$$[\![RU]\!] = \lambda_0^{counter} \circ \partial_J \circ \partial_K(create([\![CDL]\!], \hat{0})\|ACTIVE\|STANDARD)$$

○ The environment in which a POOL-⊥ unit is to be executed consists of encapsulation operators ∂_J and ∂_K (cf. Equations 3.8.2.6 and 3.8.4.2), and the object *counter* (cf. Equation 3.8.5.3). In the scope of these operators we have the 'sleeping' active objects and the standard objects (except for **nil**, which is in our semantics a kind of virtual object). Now execution of a POOL-⊥ unit

starts with an action that orders for the creation of an instance of the last class defined in the unit.

3.9.3. Standard classes. In POOL-T there are a number of classes that are predefined. Four of them, the classes *Integer*, *Boolean*, *Read_File* and *Write_File*, are, although in simplified form, also present in POOL-⊥. The standard classes can, to a large extent, be defined in terms of POOL-⊥. To make a complete definition possible, we extend the language POOL-⊥ with a new construct:

$$E \rightarrow \textbf{acp}\ t\ \textbf{pca}$$

for each closed term t in the signature of ACP. The corresponding semantic rule is

$$peq(E) = \{[\![E]\!] = t\}$$

The standard classes are described by the following class definitions:

3.9.3.1. The Booleans. This is a class with as only objects **true**, **false** and the virtual **nil**. The methods of the class generate an error if a parameter is **nil**. Surprisingly, we can describe this class completely in terms of POOL itself.

class *Boolean*

var *result* : *Boolean*

method *or* (*b* : *Boolean*) *Boolean* :

if self then

if *b* **then** *result*←**true else** *result*←**true fi else**

if *b* **then** *result*←**true else** *result*←**false fi**

fi

return *result*

end *or*

method *and*(*b* : *Boolean*) *Boolean* :

if self then

if *b* **then** *result*←**true else** *result*←**false fi else**

if *b* **then** *result*←**false else** *result*←**false fi**

fi

return *result*

end and

method *not* () *Boolean* :

if self then *result*←**false else** *result*←**true fi**

return *result*

end *not*

method *equal* (*b* : *Boolean*) *Boolean* :

if self then

if *b* **then** *result*←**true else** *result*←**false fi else**

if *b* **then** *result*←**false else** *result*←**true fi**

fi

return *result*

end *equal*

body do true then answer (*or*,*and*,*not*,*equal*) **od**

end *Boolean*

3.9.3.2. The Integers. This class contains all the integers from *Int* (plus **nil**). The methods of the class generate an error if the parameter is **nil**. In case of overflow the result of a method call is **nil** (so, for example $sum(N_0,N_0) = \textbf{nil}$). Another option would have been to generate an error. We only give the definition of the method *add*. The other method definitions are similar.

class *Integer*

method *add* (*i* : *Integer*) *Integer* :

return acp $\sum_{\alpha \in Int} eqs(\alpha)(\sum_{\beta \in Int} eqv(i,\beta) \cdot \uparrow sum(\alpha,\beta) + eqv(i,\textbf{nil}) \cdot error)$ **pca**

end *add*

etc., etc.

body do true then answer(*add*,, *mul*,*div*,*mod*,*power*,*minus*,

less,*less_or_equal*,*equal*,*greater*,*greater_or_equal*) **od**

end *Integer*

3.9.3.3. The classes Read_File and Write_File. In POOL-T it is possible to open new input and output files. These options are not present in POOL-⊥: there is only one object of class *Read_File* (the object **input**), and one object of class *Write_File* (the object **output**). These objects communicate with the external world by means of actions *input* (*d*) and *output* (*d*), for $d \in Int \cup Bool$.

class *Read_File*

routine *standard_in* () *Read_File* :

return acp $\uparrow$**input pca**

end *standard_in*

method *read_int* () *Integer* :

return acp $\sum_{\alpha \in Int} input(\alpha) \cdot \uparrow\alpha$ **pca**

end *read_int*

method *read_bool* () *Boolean* :

return acp $\sum_{\beta \in Bool} input(\beta) \cdot \uparrow\beta$ **pca**

end *read_bool*

body do true then answer(*read_int*,*read_bool*) **od**

end *Read_File*

class *Write_File*

routine *standard_out* () *Write_File* :

return acp $\uparrow$**output pca**

end *standard_out*

method *write_int* (*i* : *Integer*) *Write_File* :

return acp $\sum_{\alpha \in Int} eqv(i,\alpha) \cdot output(\alpha) \cdot \uparrow$**output** $+ eqv(i,$ **nil**$) \cdot error$ **pca**

end *write_int*

method *write_bool* (*b* : *Boolean*) *Write_File* :

return acp $\sum_{\beta \in Bool} eqv(b,\beta) \cdot output(\beta) \cdot \uparrow$**output** $+ eqv(b,$**nil**$) \cdot error$ **pca**

end *write_bool*

body do true then answer(*write_int*,*write_bool*) **od**

end *Write_File*

3.10. THEOREM. *Let* $w \in$ POOL-$\perp$ *and let* $SPEC_C(w) = <X|E>$. *Then E is guarded.*

PROOF. Introduce a new s-attribute *height* for those nonterminals which have attribute *peq*. Let the value domain of this new attribute be the set $\mathbb{N}$ of natural numbers. Let $X_0 \rightarrow X_1 \cdots X_n$ be a production where X_0 has attribute *height*. Then the semantic rule for the attribute *height* is:

$height(X_0) = \max(\{0\} \cup \{height(X_i) \mid 1 \leqslant i \leqslant n \text{ and } X_i \text{ has attribute } height\}) + 1$

Using the same technique as in the proof of Theorem 2.9, the proof that for each POOL-$\perp$ program the corresponding specification is guarded can now be given by means of straightforward induction on the value of attribute *height*.

□

3.11. Abstraction. Most of the atomic actions which were used in the description of the semantics for POOL will be invisible in an actual implementation of the language. If one looks at a computer executing a POOL program, one most likely cannot observe that one object sends a message to another object. In general the only visible actions will be the actions by means of which the POOL system communicates with the external world: the *error* action and the actions $input(d)$ and $output(d)$ $(d \in Int \cup Bool)$ as defined in Section 3.9.3.3. Let

$$I = \{c(d) \mid d \in D\} \cup \{comm(f) \mid f \in \mathfrak{F}\} \cup \{skip\} \tag{3.11.1}$$

and let w be a POOL program. Then

$$SPECS_A(w) = \tau_I(SPECS_C(w)). \tag{3.11.2}$$

$SPEC_A$ gives the abstract behaviour of a POOL system executing a given unit.

3.12. Models. A large variety of semantics (models, Σ-algebras) have been given of the signature that is used in this section. For each of these models M, there exists a mapping INT_M that maps process expressions in M. As examples of models we mention the semantics $\mathcal{A}(BS)$ of terms modulo bisimulation equivalence presented in [16], the semantics $\mathcal{A}(FS)$ of process graphs modulo failure equivalence described in [11], and the trace model that is for instance presented in [27].

4. Message queues

In the description of POOL as presented in the previous section, communication between objects takes place by means of handshaking. However, in the official language definition (see [1]) communication is described differently: All messages sent to a certain object will be stored there in a queue in the order in which they arrive. When that object executes an answer statement, the first message in the queue which name occurs in the method identifier list of the answer statement will be answered. Below we present a modified process algebra description of POOL, in which each object has its own message queue. This description, which, due to the select statement, turns out to be rather complicated, corresponds to the language definition in [1]. We call the new translation function $SPEC_{AQ}$. Thereafter, in Section 4.5, we discuss the important question for which models M the mappings $SPEC_A$ and $SPEC_{AQ}$ are semantically the same.

4.1. New channels. If we view the field 'type of message' of a frame (cf. Section 3.8.2) as the name of a channel, then we can depict the situation in which there are two objects α and β, connected by channel *mc*, 'classically' as follows:

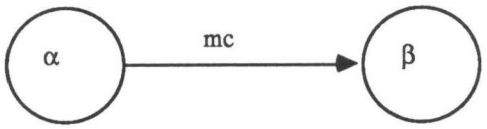

FIGURE 4.1

In this section we introduce for each object β a message queue $\rho_{f_\beta}(Q)$. Furthermore we have new channels (message types) *iq*, *om* and *fm*. The modified version of Figure 4.1 becomes:

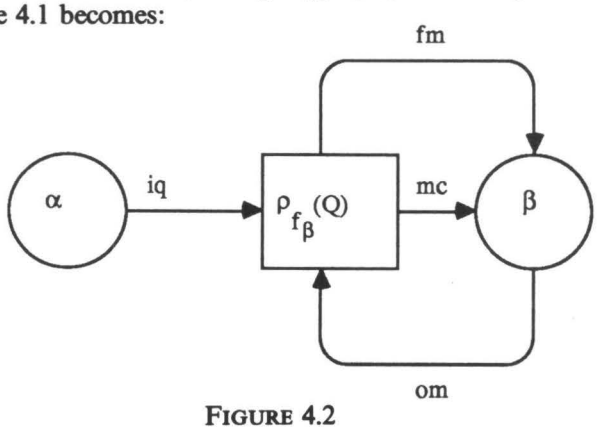

FIGURE 4.2

First we discuss the new message types.

iq: (in queue). If object α wants to send a message to object β, it must send this message by channel *iq* to the queue of object β. We have the following new semantic rules for the send expression:

$$SN \rightarrow E\,!\,MI() \tag{21.1}$$

Let

$$id(MI) = m$$

then

$$[\![SN]\!] = [\![E]\!] \ggg_\alpha [\![SN]\!]_\alpha$$

$$[\![SN]\!]_\alpha = \begin{cases} error & \text{if } \alpha = \textbf{nil} \\ sn(\alpha, iq, m()) \cdot \sum_{\beta \in Obj} rd(an, \beta, \alpha) \cdot \uparrow\beta & \text{otherwise} \end{cases}$$

(production 21.2 is changed analogously).

om: (order message). Let $L \subseteq LId$. By sending message L along channel *om* to

its queue, object β orders the queue to deliver the first message with a message identifier in L. The message type *om* occurs in the new semantic rules for the answer statement:

$AN \rightarrow$ **answer** (MIL) (17)

Let

$$M = mis(MIL)$$

then

$$mis(AN) = M$$

$$[\![AN]\!]_m = sn(om, \{m\}) \cdot [\![MIL]\!]_m$$

$$[\![AN]\!] = sn(om, M) \cdot \sum_{m \in M} [\![MIL]\!]_m$$

fm: (first method). During the execution of a select statement object β sometimes needs to know, for a given $L \subseteq LId$, if there is a message in its queue with a method identifier in L, and if so, what is the method identifier of the first one. This information is passed along channel *fm* (the negative answer is coded as ϵ). The new semantic rules for the select statement are:

$SE \rightarrow$ **sel** GCL **les** (14)

Let

$$misl(GCL) = (M_1, \ldots, M_n)$$

$$M_{\alpha_1, \ldots, \alpha_n} = \bigcup_{\{i \mid \alpha_i = \textbf{true}\}} M_i$$

then

$$[\![SE]\!] = [\![GCL]\!] \ggg_{\alpha_1, \ldots, \alpha_n} [\![SE]\!]_{\alpha_1, \ldots, \alpha_n}$$

$$[\![SE]\!]_{\alpha_1, \ldots, \alpha_n} = error$$

if $(\exists i : \alpha_i = \textbf{nil}) \vee (\forall i : \alpha_i = \textbf{false})$,

$$[\![SE]\!]_{\alpha_1, \ldots, \alpha_n} = \sum_{m \in LId} rd(fm, (M_{\alpha_1, \ldots, \alpha_n}, m)) \cdot [\![GCL]\!]_m^{\alpha_1, \ldots, \alpha_n}$$

if $\forall i : \alpha_i = \textbf{true} \Rightarrow M_i \neq \emptyset$, and

$$[\![SE]\!]_{\alpha_1, \ldots, \alpha_n} = \sum_{m \in LId \cup \{\epsilon\}} rd(fm, (M_{\alpha_1, \ldots, \alpha_n}, m)) \cdot [\![GCL]\!]_m^{\alpha_1, \ldots, \alpha_n}$$

otherwise.

○ $M_{\alpha_1, \ldots, \alpha_n}$ is the set of all method identifiers occurring in the answer statement of an open guarded command. If there is no message in the queue whose method identifier is in $M_{\alpha_1, \ldots, \alpha_n}$, and there are open guarded

commands without an answer statement ($M_i = \varnothing$ for some i), then the (textually) first of them is selected. If there is no message in the queue whose method identifier is in $M_{\alpha_1,\ldots,\alpha_n}$, and there is no open guarded command without an answer statement, the object waits until a message that belongs to $M_{\alpha_1,\ldots,\alpha_n}$ arrives, and then proceeds with this message. This waiting may last forever. If there is a message in the queue with method identifier in $M_{\alpha_1,\ldots,\alpha_n}$ this message is selected. The first guarded command is chosen that has either no answer statement or whose answer statement contains the method named in the message.

4.2. The process Q. We introduce a new object q as parameter of the state operator. The state of this object (the content of the queue) will be an element of $(\mathfrak{N}\times Obj)^*$ (for definition $\mathfrak{N}$, see Equation 3.8.2.3): a list of pairs of method calls and references to the senders of these calls. We need four fresh formal variables Q, R, S and A. The process Q gives the behaviour of an 'unfinished' queue, a queue that is not yet associated with one specific object. We have the following equation:

$$Q = \lambda_\epsilon^q(R \| S \| A) \tag{4.2.1}$$

Q consists of the merge of three processes, R, S and A, which operate in an environment in which the content of the queue is known. The job of process R is to read messages in the queue:

$$R = \sum_{d\in\mathfrak{N}} \sum_{\alpha\in Obj} rd(iq,d,\alpha)\cdot R \tag{4.2.2}$$

The relevant equation for the state operator is:

$$\lambda_\sigma^q(rd(iq,d,\alpha)\cdot x) = rd(iq,d,\alpha)\cdot\lambda_{(d,\alpha)*\sigma}^q(x) \tag{4.2.3}$$

The process S first waits for an order to deliver a message with method identifier in a certain set L, and thereafter delivers the first message in the queue with this property. When such a message is not in the queue, process S waits until it arrives.

$$S = \sum_{L\subseteq LId} rd(om,L)\cdot sn(mc,L)\cdot S \tag{4.2.4}$$

In order to define the interaction between actions $sn(mc,L)$ and operator λ_σ^q we need three auxiliary functions. The function $mf(L,\sigma)$ picks the first message in σ with a method identifier in L, and returns ϵ if there is no such message. The function is recursively defined by:

$$mf(L,\epsilon) = \epsilon \tag{4.2.5}$$

$$mf(L,\sigma*(m(\alpha_1,\ldots,\alpha_n),\alpha)) = \begin{cases} m(\alpha_1,\ldots,\alpha_n) & \text{if } m\in L \\ mf(L,\sigma) & \text{otherwise} \end{cases} \tag{4.2.6}$$

The function $sf(L,\sigma)$ returns the sender of the first message in σ with method identifier in L, or returns ϵ.

$$sf(L,\epsilon) = \epsilon \tag{4.2.6}$$

$$sf(L,\sigma\star(m(\alpha_1,\ldots,\alpha_n),\alpha)) = \begin{cases}\alpha & \text{if } m\in L\\ sf(L,\sigma) & \text{otherwise}\end{cases} \tag{4.2.7}$$

The function $of(L,\sigma)$ omits the first element of σ with method identifier in L.

$$of(L,\epsilon) = \epsilon \tag{4.2.8}$$

$$of(L,\sigma\star(m(\alpha_1,\ldots,\alpha_n),\alpha)) = \begin{cases}\sigma & \text{if } m\in L\\ of(L,\sigma)\star(m(\alpha_1,\ldots,\alpha_n),\alpha)) & \text{otherwise}\end{cases} \tag{4.2.9}$$

Now we can define:

$$\lambda_\sigma(sn(mc,L)\cdot x) = \begin{cases}sn(mc,mf(L,\sigma),sf(L,\sigma))\cdot\lambda^q_{of(L,\sigma)}(x) & \text{if } mf(L,\sigma)\neq\epsilon\\ \delta & \text{otherwise}\end{cases} \tag{4.2.10}$$

The process A gives an answer to questions of the form: 'Is there a message in the queue with method identifier in a set L, and if so, what is the method identifier of the first one?'.

$$A = \sum_{L\subseteq LId}\ \sum_{m\in LId\cup\{\epsilon\}} sn(fm,(L,m))\cdot A \tag{4.2.11}$$

Again we need an auxiliary function: $if(L,\sigma)$ gives the identifier of the first message in σ with identifier in L.

$$if(L,\epsilon) = \epsilon \tag{4.2.12}$$

$$if(L,\sigma\star(m(\alpha_1,\ldots,\alpha_n),\alpha)) = \begin{cases}m & \text{if } m\in L\\ if(L,\sigma) & \text{otherwise}\end{cases} \tag{4.2.13}$$

The relevant equation for the state operator is:

$$\lambda^q_\sigma(sn(fm,(L,m))\cdot x) = \begin{cases}sn(fm,(L,m))\cdot\lambda^q_\sigma(x) & \text{if } if(L,\sigma) = m\\ \delta & \text{otherwise}\end{cases} \tag{4.2.14}$$

4.3. Extensions. We add the new frames which were introduced in the previous section to the set $\mathcal{F}$ of frames (see Equation 3.8.2.4), we introduce actions $rd(f)$, $sn(f)$, $read(f)$, $send(f)$ and $comm(f)$ for the new frames, and extend the communication function in the obvious way. Furthermore the set J of encapsulated actions (see Equation 3.8.2.4) is extended. For the new atoms the renaming functions f_α are defined by:

$$f_\alpha(sn(\beta,iq,d)) = send(\beta,iq,d,\alpha) \tag{4.3.1}$$

$$f_\alpha(rd(iq,d,\beta)) = read(\alpha,iq,d,\beta) \tag{4.3.2}$$

$$f_\alpha(sn(om,M)) = send(\alpha,om,M,\alpha) \tag{4.3.3}$$

$$f_\alpha(rd(om,M)) = read(\alpha,om,M,\alpha) \tag{4.3.4}$$

$$f_\alpha(sn(fm,(M,m))) = send(\alpha,fm,(M,m),\alpha) \tag{4.3.5}$$

$$f_\alpha(rd(fm,(M,m))) = read(\alpha,fm,(M,m),\alpha) \tag{4.3.6}$$

4.4. Root unit. Now we change the semantic rule for the root unit as follows:

$RU \rightarrow$ **root unit** CDL (2)

Let

$$cd(CDL)(Integer) = I$$

$$cd(CDL)(Boolean) = B$$

$$cd(CDL)(Read\_File) = R$$

$$cd(CDL)(Write\_File) = W$$

$$\mathcal{C} = \{cd(CDL)(C) \mid C \in UId\}$$

$$ACTIVE = \|_{\alpha \in AObj} (\sum_{X \in \mathcal{C}} create^*(X,\alpha) \cdot \rho_{f_\alpha}(X))$$

$$STANDARD = (\|_{\alpha \in Int} \rho_{f_\alpha}(I)) \| \rho_{f_{new}}(B) \| \rho_{f_{bool}}(B) \| \rho_{f_{read}}(R) \| \rho_{f_{write}}(W)$$

$$QUEUE = \|_{\alpha \in Obj} (\rho_{f_\alpha}(Q))$$

then

$$[\![RU]\!] = \lambda_0^{counter} \circ \partial_J \circ \partial_K(create([\![CDL]\!],\hat{0}) \| ACTIVE \| STANDARD \| QUEUE)$$

4.5. The incompatibility of $SPEC_A$ and $SPEC_{AQ}$. Clearly the mapping $SPEC_{AQ}$ is much more complicated than the mapping $SPEC_A$. Therefore we would like to work with $SPEC_A$ instead of $SPEC_{AQ}$. But since $SPEC_{AQ}$ corresponds to the official language definition in [1] and $SPEC_A$ does not, we first have to show that the two mappings lead to the same semantics of POOL. Unfortunately this is not possible as we will demonstrate below: for any model M of ACP_τ which preserves fairness and liveness properties we have

$$M \not\models SPEC_A = SPEC_{AQ}.$$

Stated informally, the fairness we require of the models is that (1) all processes that become permanently enabled, must execute infinitely often, and (2) two processes that can communicate infinitely often will do so infinitely often. These fairness requirements correspond to the fairness requirements formulated in [1]. The issue of fairness is discussed in more detail in Section 5.4.

The notions of safety and liveness are frequently used in the literature. Roughly, safety means that something bad cannot happen, while liveness means that something good will eventually happen. In the context of POOL, liveness implies that a program that will certainly perform a certain action is different from a program which may not do this.

Now consider the situation in which an object executes the following piece of POOL text:

```
b←true ;
do b then sel
        true answer(m1) then b←false or
        true then b←b or
        true answer(m2) then b←false or
les od ;
Write_File . standard_out() ! write_bool(b)
```

Suppose the object operates in a system with message queues, and that at the moment at which the object starts execution of the POOL text, the message queue of the object contains two messages: first a message with method identifier m_2, and after that a message with method identifier m_1. Now execution of the POOL text takes place as follows: first b is set to **true**, then the object enters the do-loop and the select statement is executed. The set of method identifiers occurring in an open guarded command is $\{m_1,m_2\}$. The first message in the queue with a method identifier in this set is m_2. Now the first guarded command is chosen that has either no answer statement or whose answer statement contains m_2. In our case this is the second guarded command. The trivial statement part of this guarded command is executed, and the select statement terminates. But since variable b is still equal to **true**, the select statement is immediately executed for the second time. Again b remains **true**. It will be clear that the select statement never terminates.

However, if the object operates in a system without message queues, the select statement *will* terminate! In the situation with handshaking communication there is one object that wants to send a message with identifier m_1, and one object that wants to send a message with identifier m_2. Due to the fairness requirement communication of the message with identifier m_1 will eventually take place, b is set to **false**, the do-loop terminates, and **false** is printed. This means that there is a difference with respect to liveness between the situation with, and the situation without message queues.

A good semantics of POOL should preserve fairness and liveness properties. The example presented above shows that in a semantical description that uses message queues between objects instead of handshaking communication, liveness properties get lost almost inevitably.

4.6. In this section we propose a minor change in the language definition of POOL, which removes the difficulty of Section 4.5. In the example of Section 4.5 it is clear from the beginning that the third guarded command will never be chosen. But instead of leaving the turmoil of battle, the third guarded command starts helping his neighbour, the second guarded command. Because of this the competition between the first and the second guarded command is not fair and the second guarded command always wins. The modification of the language definition we propose consists of the removal of all open guarded

commands in a select statement which have an open guarded command without an answer statement before them. Formally this means that we replace the definition of sets $M_{\alpha_1,\ldots,\alpha_n}$ in the semantic rules for the select statement in Section 4.1 by:

$$M_{\alpha_1,\ldots,\alpha_n} = \{m \mid \exists i : m \in M_i \wedge \alpha_i = \mathbf{true} \wedge (\forall j < i : \alpha_j = \mathbf{true} \Rightarrow M_j \neq \emptyset)\}$$

The modified version of $SPEC_{AQ}$ is called $SPEC_{AQ'}$.

4.7. Even after modification of the language definition, the semantical description with handshaking communication is not equivalent to the description using message queues. The following theorem shows that it is impossible to prove equivalence if one only uses the axioms presented thus far. However, whereas the difficulty of Section 4.5 was a *general* difficulty, present in all semantical descriptions employing handshaking communication between the objects, the difficulty pointed out in the following theorem is *specific*, and only present in bisimulation semantics and other semantics which distinguish processes that cannot be distinguished by observation.

4.7.1. THEOREM. $\mathcal{C}(BS) \nvdash SPEC_A = SPEC_{AQ'}$.
PROOF. Below we present a POOL-$\perp$ unit u with the property that in the models based on bisimulation $SPEC_A(u)$ and $SPEC_{AQ'}(u)$ are different. The program is a very simple one: the initial object of class *Root* creates 3 objects of class *Number* and these three objects ask the standard output object to print resp. numbers 1, 2 and 3.

root unit

class *Number*

var *m* : *Integer*

routine *new* () *Number* :

return new

end *new*

method *init* (*n* : *Integer*) *Number* :

m←*n* **return self**

end *init*

body *answer*(*init*) ; *Write_File* . *standard_out* () ! *write_int*(*m*)

end *Number*,

class *Root*

body *Number* . *new* () ! *init* (1) ; *Number* . *new* () ! *init* (2) ; *Number* . *new* () ! *init* (3)

end *Root*

Writing down $SPEC_A(u)$ and $SPEC_{AQ'}(u)$ is a long and tedious job which we happily leave to the reader. However, it is easy to see that the process graphs that correspond to these specifications can not be bisimilar. If there is a message queue before the standard output object, it is possible that at a certain moment during execution of the program the three method calls of the three objects of class *Number* are waiting in the queue. Because, for a given method, an object answers the methods calls in the queue in the order in which they have arrived, the order in which the actions *output*(1), *output*(2) and *output*(3) will be performed, is completely determined in such a state. However, in the case where there are no message queues there is no state in which no output action has taken place but still the order in which the output actions will occur is known. Therefore the process graphs corresponding to $SPEC_A(u)$ and $SPEC_{AQ'}(u')$ are not bisimilar. □

What we learn from Theorem 4.7.1 is that we can either do bisimulation semantics based on a translation of units in which we use queues (this leads to very long and complicated proofs), or add some axioms to our theory in such a way that we can prove equivalence of $SPEC_A$ and $SPEC_{AQ'}$. We conjecture that

$$\mathcal{A}(FS) \vdash SPEC_A = SPEC_{AQ'}$$

and that equivalence can be proved if we add to our theory the axioms of failure semantics as presented in [11]. The proof however will be long and complicated, and we do not even try to give it in this paper.

5. Trace semantics, fairness and successful termination

5.1. A trace model as presented for instance in [27], is not a good semantic domain for POOL in the sense that it identifies too much and does not describe deadlock behaviour. In $\mathcal{A}(TR_{den})$ we have for example:

$$output(0) = output(0) + \tau \cdot \delta.$$

We do not want to identify these processes because the first one will definitely output a 0, whereas the second one may not.

5.2. It is well known that it is not possible to give a trace model of ACP in which one looks at the terminating (and infinite) traces, and the trace sets do not have to be prefix closed. In such a model $a(b+c)$ and $ab+ac$ would be identical. This is problematic since $\partial_{\{c\}}(a(b+c)) = ab$ and $\partial_{\{c\}}(ab+ac) = ab+a\delta$ are different.

5.3. However, there exist some interesting semantics of POOL based on trace sets. The basic idea of the approach which is, although in a different setting, followed in [4], is that one first interprets a specification in a domain in which not very many processes are identified (the domain of transition systems, the model $\mathcal{A}(BS)$) and then takes the set of terminating (and infinite) traces of this process. In this approach one typically looks at

$$YIELD \circ INT_{\mathcal{A}(BS)} \circ SPEC_A(u)$$

where *YIELD* is a function that gives the set of terminating (and infinite) traces of elements of $\mathcal{A}(BS)$. The resulting semantic domain is not a model of ACP but for most applications that does not matter. An advantage of the approach is that it allows for simple solutions to a number of problems.

5.4. Fairness. The fairness problem for example can be solved easily. In [1] a fairness condition concerning POOL is formulated by stating that the execution 'speed' of any object is arbitrary but positive. Whenever an object can proceed with its execution without having to wait for a message or a message result, it will eventually do so. A second fairness requirement on the execution of a POOL program is the condition that all messages sent to a certain object will be stored there in one queue in the order in which they arrive. In process algebra we have deliberately chosen to ignore the exact timing of occurrences of events. Fortunately the fairness requirements concerning POOL can be defined without referring to timing aspects. The first fairness requirement is called *weak process fairness* or *justice* in the literature:

All processes that become permanently enabled, must execute infinitely often

The second requirement is called *strong channel fairness:*

Two processes that can communicate infinitely often will do so infinitely often

For reviews of the literature on fairness we refer to [15, 24]. We think that the Petri net model for ACP based on occurrence nets, which is presented in [17], preserves enough information for a description of the fairness requirements of POOL. More research is needed to make this explicit. In the trace set approach the solution is very simple: one omits all the unfair traces and looks at:

$$YIELD_F \circ INT_{\mathcal{A}(BS)} \circ SPEC_C(u)$$

where $YIELD_F$ gives the set of fair terminating and infinite traces of elements of $\mathcal{A}(BS)$.

5.4.1. Fair abstraction. If we work with 'abstract' translation functions like $SPEC_A$ and $SPEC_{AQ}$, then it is possible to give a 'more or less' fair semantics of POOL without using a $YIELD_F$ function. This employs the fact that Koomen's Fair Abstraction Rule (KFAR) is valid in (for example) the model $\mathcal{A}(BS)$. Consider the following unit *f*:

root unit

class *Out*

routine *new*() *Out* :

return new

end *new*

body *Write_File* . *standard_out* ! *write_int*(0)

end *Out*,

class *Chatter*

var *x* : *Integer*

body *Out* . *new*() ; **do true then** *x*←1 **od**

end *Chatter*

It can be proved that for any model M in which KFAR holds:

$$M \models SPEC_A(f) = \tau \cdot output(0) \cdot \delta.$$

This means that the object of class *Out* will make progress despite the infinite chatter of the object of class *Chatter*. Note that KFAR equates infinite chatter and deadlock.

5.4.2. KFAR is too fair. We give an example which shows that sometimes KFAR is too fair. Consider the architecture of Figure 5.1.

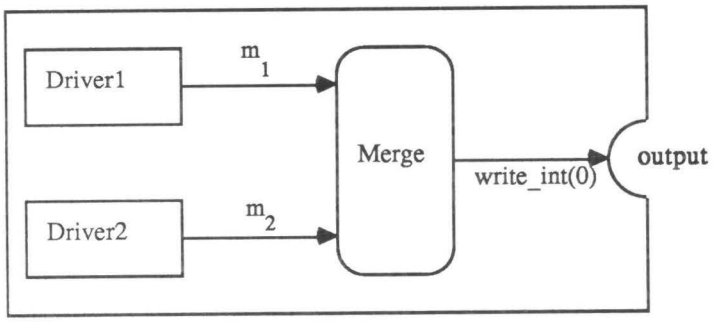

FIGURE 5.1

There are two objects *Driver1* and *Driver2*. The only thing these objects do is sending method calls to an object *Merge*. *Driver1* all the time asks *Merge* to perform method m_1 and analogously *Driver2* asks *Merge* to perform method m_2. The object *Merge* has the task to perform statement **answer**(m_1,m_2) until doomsday. Every time when it has answered method m_1 two times consecutively, the object *Merge* asks the object **output** to print a 0. We leave it to the reader to write down the corresponding POOL program.

The point we want to make is this. According to the language definition in

[1], the execution where object *Merge* answers messages of *Driver1* and *Driver2* in turn ($m_1, m_2, m_1, m_2,...$) will be fair. Hence it is possible that *Merge* never orders to print a 0. However, in a semantics where KFAR holds, a 0 will be printed: the only way for the system to get out of the 'cluster' of internal actions is to perform an action *output*(0). This action is always possible during execution of the program. KFAR says that therefore it will occur. Again we leave it to the reader to fill in the formal details.

5.4.3. Failure semantics. In [11] it is shown that KFAR is not valid in the model $\mathcal{A}(FS)$. Nevertheless the model admits a restricted rule KFAR$^-$ for the fair abstraction of so-called unstable divergence:

$$(\text{KFAR}^-) \qquad \frac{x = ix + \tau y}{\tau_{\{i\}}(x) = \tau \cdot \tau_{\{i\}}(\tau y)}$$

KFAR$^-$ turns out to be sufficient for the protocol verifications in [22, 26, 28]. However, for our purposes KFAR$^-$ is not what we want. Like KFAR, the rule is too fair for some applications. But in addition there are applications where KFAR$^-$ is not fair enough. KFAR$^-$ does not allow for a proof that the object of class *Out* in the example of Section 5.4.1 will make progress. We even have:

$$\mathcal{A}(FS) \not\models \pi_1(SPEC_A(f)) = \tau \cdot output(0) \cdot \delta.$$

This is a crucial observation. Failure semantics - being a linear semantics - often yields simpler proofs than bisimulation semantics which preserves the full branching structure of processes. Although the notion of full abstractness still has to be defined for the language POOL, it is clear that failure semantics is closer to full abstractness than bisimulation semantics. Furthermore, as pointed out in Section 4, failure semantics will supposedly allow for a proof that the communication between objects can be implemented by means of message queues. Thus failure semantics seems to be ideal for POOL. But now it turns out that the combination of failure semantics and weak process fairness is problematic. At present we do not know if it is possible to give a semantics of POOL which is 'fully abstract' and also 'fair'.

5.5. Deadlock behaviour. A limit on the applicability of the trace approach sketched in Section 5.3 is that it only describes the behaviour of a POOL system in situations in which this system is placed in a 'glass' box, and does not communicate with the environment. Below we present two POOL-$\perp$ units $u1$ and $u2$ with the property that

$$YIELD \circ INT_{\mathcal{A}(BS)} \circ SPEC_A(u1) = YIELD \circ INT_{\mathcal{A}(BS)} \circ SPEC_A(u2)$$

although

$$\mathcal{A}(BS) \not\models SPEC_A(u1) = SPEC_A(u2)$$

(we even have $\mathcal{A}(FS) \not\models SPEC_A(u1) = SPEC_A(u2)$).
The root object of unit $u1$ creates an object that performs the job of outputting a 0. After ordering for the creation, the root object inputs a value.

root unit

class *Out*

routine *new*() *Out* :

return new

end *new*

body *Write_File* . *standard_out* ! *write_int* (0)

end *Out*,

class *In*

body *Out* . *new*() ; *Read_File* . *standard_in*() ! *read_int*()

end *In*

In unit $u2$ the root object of class *Semaphore* creates two objects: one object has to output a 0, and the other object inputs a value. But before the I/O actions can take place the objects have to decrease a semaphore. After an object has decreased a semaphore, it can perform the I/O action. After that, it increases the semaphore again. If during execution of $u2$ the input actions are blocked (the enemy has bombed the input device), it can happen (if the object that has to input a value is the first one to decrease the semaphore) that the output action will not take place. In this respect $u2$ differs from $u1$: if during execution of $u1$ the input actions are blocked, the output action will still happen.

root unit

class *Out*

var *sem* : *Semaphore*

routine *new*() *Out* :

return new

end *new*

method *init* (*s* : *Semaphore*) *Out* :

sem←*s* **return self**

end *init*

body

```
            answer(init);
            sem ! down();
            Write_File . standard_out() ! write_int(0);
            sem ! up()
end Out,

class In
var sem : Semaphore
routine new() In :
            return new
end new
method init (s : Semaphore) In :
            sem←s  return self
end init
body
            answer;
            sem ! down();
            Read_File . standard_in() ! read_int();
            sem ! up()
end In,

class Semaphore
method down () Semaphore :
            return self
end down
method up () Semaphore :
            return self
end up
body
            Out . new () ! init (self);
            In . new () ! init (self);
```

do true then answer(*down*); **answer** (*up*) **od**

end *Semaphore*

We can prove in the theory that:
(1) The following x_1 equals $SPEC_A(u1)$:

$$x_1 = \tau \cdot (output(0) \| \sum_{\alpha \in Int} input(\alpha)) \cdot \delta$$

(2) The following x_2 equals $SPEC_A(u2)$:

$$x_2 = \tau \cdot (\tau \cdot output(0) \cdot (\sum_{\alpha \in Int} input(\alpha)) + \tau \cdot (\sum_{\alpha \in Int} input(\alpha)) \cdot output(0)) \cdot \delta$$

Let $B = \{input(\alpha) | \alpha \in Int\}$ be the set of blocked actions. Then

$$\partial_B(x_1) = \tau \cdot output(0) \cdot \delta$$

$$\partial_B(x_2) = \tau \cdot (\tau \cdot output(0) \cdot \delta + \tau \cdot \delta)$$

Thus units $u1$ and $u2$ behave differently in an environment which does not offer certain actions: in environment $\partial_B(.)$ $u1$ will certainly output a 0, whereas $u2$ may not do this.

5.6. Successful termination. For arbitrary POOL units u_1 and u_2, and for an arbitrary model M we have that:

$$M \models SPEC_A(u_1) \cdot \circ SPEC_A(u_2) = SPEC_A(u_1).$$

This is because the process corresponding to a unit is infinite or ends in a deadlock. If one wants to describe a situation where after execution of a POOL unit, something else can be done, one has to change the semantics. In the trace set approach of the previous section this is simple: one simply defines the operation sequential composition in the obvious way. In the axiomatic approach things are not that easy. We propose (but do not work out) a solution in the spirit of [7]: one defines a program transformation that transforms the original program (in the case of POOL also the definitions of the standard classes have to be transformed). The transformation introduces a number of new program variables and statements in such a way that the resulting program can terminate successfully. In this approach it is possible to differentiate between various ways in which a unit can terminate: one option is that a unit terminates successfully if all active objects have finished execution of their body; another option says that a unit terminates successfully if there is no object (or pair of objects) that can do a step.

6. Conclusions

1. In this paper we have shown that it is possible to give semantics of a realistic concurrent programming language by means of process algebra. The translation of POOL programs into process algebra is complicated, but this is mainly caused by the complexity of POOL, in particular by the complexity of the select statement. The attribute grammar which we used

for the translation made it possible to give the semantics in a modular way.

2. This paper contains an application of ACP where the sequential composition operator is used in full generality. It would have been more involved to give semantics of POOL in a signature containing prefixing (an operator $A \times P \rightarrow P$) instead of sequential composition. Three auxiliary operators, the renaming operator, the chaining operator and the state operator, turned out to be useful.
3. Because we have no infinite sum and infinite merge operators in the signature, we had to choose the value domain of POOL variables finite. Furthermore the number of objects which can be created during execution of a POOL unit is finite. Although it would be useful to have these infinitary operators available, we do not think that their absence in the present paper is a real deficiency: the memory of each computer is finite, and no computer will function eternally.
4. The approach followed in this paper can also be used to give semantics of other concurrent programming languages. From the point of view of process algebra we see no fundamental difference between the object-oriented approach from POOL, and the imperative or functional approaches followed in other languages. However, at present it is difficult to give process algebra semantics of a language in which real-time aspects play a role.
5. KFAR does not completely capture the notion of fairness in POOL. In Section 5.4.3 we pointed out that combination of failure semantics and weak process fairness is especially problematic. An open question is whether or not the two concepts can be combined in a consistent manner.
6. There is not one single 'optimal' semantics of POOL. Depending on the application domain one has in mind one can try to find an optimum. There are several features which can be included in the semantical description of the language: infinite domains of variables, fairness, error behaviour, termination behaviour, etc.. An important parameter in the choice of a semantics is the type of interaction between the environment and the POOL system. In case one wants to use the semantics to build an executable prototype, the semantics has to be operational. In case the semantics is used for the construction of proof systems or for the correctness proof of implementations, one requires abstractness and compositionality. It might be the case that the combination of all these requirements leads to inconsistencies.
7. The translation of POOL into process algebra can be used for prototyping of the language. The shortest route seems to be a translation into an algebraic specification formalism. The attribute grammar which we used can be specified algebraically in a straightforward way. The process algebra part is already specified algebraically but some work has to be done in order to deal with a number of notational conventions, for example the sum operator and the numerous '...' occurring in the equations. There are several alternatives for transforming algebraic specifications into executable prototypes, for example by means of a transformation into a

complete (conditional) term rewriting system and execution by means of an existing rewrite rule interpreter, or by means of a transformation into a set of Horn clauses and using an existing Prolog system for their execution.

8. A semantical description of POOL with handshaking communication between the objects is incompatible with the description in [1], where message queues are used. A minor change in the language definition is proposed in order to remove this difficulty. In our opinion this result shows that, when dealing with concurrent programming languages, questions like: 'Is this semantical description in accordance with the language definition?' and 'Is this a correct implementation of the language?' are highly relevant.
9. An important problem to be solved is in our view the development of techniques which make it possible to prove that two semantics of POOL have a common abstraction. In [25] we gave a sketch of such a proof, showing that the Integers and Booleans can be implemented in different ways. In Section 4 we discussed the question whether or not the communication between objects can be implemented by message queues. We showed that, even after modification of the language definition, this is not possible in bisimulation semantics. An open question is the equivalence in failure semantics.

ACKNOWLEDGEMENTS

I would like to thank Pierre America, Joost Kok, Jan Rutten and all the participants of the PAM seminar for their valuable criticism and many inspiring discussions.

REFERENCES

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

[2] P. AMERICA (1986): *Rationale for the design of POOL.* ESPRIT project 415, Doc. Nr. 53, Philips Research Laboratories, Eindhoven.

[3] P. AMERICA (1987): *A sketch for POOL2.* ESPRIT project 415, Doc. Nr. 240, Philips Research Laboratories, Eindhoven.

[4] P. AMERICA, J.W. DE BAKKER, J.N. KOK & J.J.M.M. RUTTEN (1986): *Operational semantics of a parallel object-oriented language.* In: Conference Record of the 13th ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida, pp. 194-208.

[5] P. AMERICA, J.W. DE BAKKER, J.N. KOK & J.J.M.M. RUTTEN (1986): *A denotational semantics of a parallel object-oriented language.* Report CS-R8626, Centrum voor Wiskunde en Informatica, Amsterdam, to appear in I&C.

[6] ANSI (1983): *Reference manual for the Ada programming language.* ANSI/MIL-STD 1815 A, United States Department of Defense, Washington D.C..

[7] K.R. APT & N. FRANCEZ (1984): *Modelling the distributed termination convention of CSP*. TOPLAS 6(3), pp. 370-379.

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

[9] J.W. DE BAKKER (1980): *Mathematical theory of program correctness*, Prentice-Hall International.

[10] J.A. BERGSTRA (1985): *A process creation mechanism in process algebra*. Logic Group Preprint Series Nr. 2, CIF, State University of Utrecht, to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 81-88.

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

[12] G.V. BOCHMAN (1976): *Semantic evaluation from left to right*. Communications of the ACM 19(2), pp. 55-62.

[13] D.W. BUSTARD (1980): *An introduction to Pascal-Plus*. In: On the construction of programs - an advanced course (R.M. McKeag & A.M. Macnaghten, eds.), Cambridge University Press, pp. 1-57.

[14] J. ENGELFRIET (1984): *Formele talen en automaten 2*, Department of Computer Science, State University of Leiden, lecture notes (in Dutch).

[15] N. FRANCEZ (1986): *Fairness*, Springer-Verlag, Berlin.

[16] R.J. VAN GLABBEEK (1987): *Bounded nondeterminism and the approximation induction principle in process algebra*. In: Proceedings STACS 87 (F.J. Brandenburg, G. Vidal-Naquet & M. Wirsing, eds.), LNCS 247, Springer-Verlag, pp. 336-347.

[17] R.J. VAN GLABBEEK & F.W. VAANDRAGER (1987): *Petri net models for algebraic theories of concurrency*. In: Proceedings PARLE conference, Eindhoven, Vol. II (Parallel Languages) (J.W. de Bakker, A.J. Nijman & P.C. Treleaven, eds.), LNCS 259, Springer-Verlag, pp. 224-242.

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

[19] INMOS, LTD. (1984): *The occam programming manual*, Prentice-Hall International.

[20] ISO (1987): *Information processing systems - open systems interconnection - LOTOS - a formal description technique based on the temporal ordering of observational behaviour*. ISO/TC 97/SC 21 N DIS8807.

[21] D.E. KNUTH (1968): *Semantics of context-free languages*. Mathematical Systems Theory 2, pp. 127-145, Correction: Mathematical Systems Theory 5, 1971, pp. 95-96.

[22] C.P.J. KOYMANS & J.C. MULDER (1986): *A modular approach to protocol verification using process algebra*. Logic Group Preprint Series Nr. 6, CIF, State University of Utrecht, to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 261-306.

[23] R. MILNER (1980): *A Calculus of Communicating Systems*, LNCS

92, Springer-Verlag.

[24] J. PARROW (1985): *Fairness properties in process algebra - with applications in communication protocol verification.* DoCS 85/03, Ph.D. Thesis, Department of Computer Systems, Uppsala University.

[25] F.W. VAANDRAGER (1986): *Process algebra semantics of POOL.* Report CS-R8629, Centrum voor Wiskunde en Informatica, Amsterdam.

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

[27] F.W. VAANDRAGER (1988): *Some observations on redundancy in a context.* Report CS-R8812, Centrum voor Wiskunde en Informatica, Amsterdam, to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 237-260.

[28] F.W. VAANDRAGER (1989): *Two simple protocols,* to appear in: Applications of process algebra, (J.C.M. Baeten, ed.), 1990, pp. 23-44.

Determinism → (Event Structure Isomorphism = Step Sequence Equivalence)

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

A concurrent system S is called *deterministic* if for all states s of S we have that whenever S can evolve from state s into states s' and s'' by doing an action a, it must be the case that s' equals s''. It is well known that for deterministic concurrent systems, most of the interleaved equivalences (bisimulation-, failure-, trace-equivalence) coincide. In this paper we prove in the setting of event structures that also most of the non-interleaved equivalences coincide (with each other) on this domain. In the last section of the paper we show that, as a consequence of our result, the causal structure of a deterministic concurrent system can be unravelled by observers who are capable to observe the beginning and termination of events.

Key Words & Phrases: event structures, determinism, sequence/trace semantics, bisimulation semantics, step semantics, partial order semantics, pomset semantics, action refinement, split semantics.

1. Introduction

A (discrete) concurrent system generates events as it evolves in time. At any moment a set of events will have occurred and these will be ordered 'in time' or by 'causal precedence'. This order may be partial. When modelling concurrent systems and reasoning about their behaviour, it is often useful to consider different events as occurrences of the same action. This may indicate that certain events are produced by the same physical resource or that they cannot be distinguished by an observer. The relation between events and actions can be expressed by a labelling function $l:E\rightarrow A$ that relates an action to each event. Different approaches to the modelling of concurrent systems can be classified by looking at the types of labelling functions they allow for. For instance, if one models a concurrent system with an elementary net system [24], then it can never be the case that in some behaviour two events with the same label are concurrent (i.e. not related by the ordering). If we consider the usual semantics for process algebra languages like CCS [17], TCSP [14], ACP [4] and Meije [3], then it turns out that these languages are very liberal wrt labellings of events: there is (almost) no restriction at all. There exists a very rich theory of 'comparative concurrency semantics' relating the interleaved semantics for CCS-like languages, i.e. those semantics which do not treat concurrency as a primitive notion. Now a well-known result says that almost all

these equivalences (bisimulation equivalence, trace equivalence and everything in between) coincide for *deterministic* systems (see for instance ENGELFRIET [9]). A concurrent system S is called deterministic if for all states s of S we have that whenever S can evolve from state s into states s' and s'' by doing an action a, it must be the case that s' equals s''.

Recently, many equivalences have been proposed that do consider concurrency as a primitive notion. Besides the event structure equivalence and the step sequence equivalence that will be discussed in this paper, we have for instance the occurrence net equivalence of NIELSEN, PLOTKIN & WINSKEL [18], the NMS equivalence of DEGANO, DE NICOLA & MONTENARI [8], the BS bisimulation of TRAKHTENBROT, RABINOVICH & HIRSHFELD [27], the step failure semantics of TAUBNER & VOGLER [26], the step bisimulation semantics of NIELSEN & THIAGARAJAN [19], the pomset semantics of PRATT [22], the pomset bisimulation semantics of BOUDOL & CASTELLANI [6], the generalised pomset bisimulation and the ST-bisimulation of VAN GLABBEEK & VAANDRAGER [11], the split sequence equivalence which we present at the end of this paper, etc, etc.

Now one can ask the obvious question what happens with all these equivalences if we restrict ourselves to the domain of deterministic systems. The main result of this paper is that almost all non-interleaved equivalences coincide (with each other) for deterministic systems. More specifically, we will show that step sequence equivalence and event structure isomorphism agree on this domain. Of the equivalences mentioned above only occurrence net equivalence is not situated in between step sequence equivalence and event structure isomorphism.

Event structures. A natural domain for modelling concurrency is the class of *event structures*, which were introduced in NIELSEN, PLOTKIN & WINSKEL [18]. By now many different types of event structures have been defined. For an overview we refer to WINSKEL [28]. In our view an especially important class of event structures is the class of prime event structures. Prime event structures contain no *junk*: every event in the set of events of a prime event structure can occur in at least one behaviour. The event structures used in this paper are labelled prime event structures with binary conflict. Below we give a formal definition of this type of event structures, followed by some explanatory remarks. If one assumes binary conflict, then one can only express that *two* events exclude each other. Thus it is not possible to say that three or more events cannot occur in combination even though each proper subset can. For this one needs more general types of event structures. The assumption of binary conflict is not essential in the proof of the main theorem of this paper. Because most people will be more familiar with event structures with binary conflicts and because the main use we foresee of our theorem lies in the field of CCS-like languages (where conflict is always binary), we decided to present the theorem for the case with binary conflict only, and to leave the generalisation to the case with arbitrary conflict as a (simple) exercise to the reader.

Arbitrary interleaving versus True concurrency. In the last section of the paper some consequences will be discussed of our result for the issue of arbitrary interleaving versus 'True' concurrency. We introduce an operator which splits each event into a beginning and an end and show that the causal structure of a deterministic concurrent system can be unravelled by observers who are capable to observe these beginnings and ends.

Related work. One can view the main theorem of this paper as a *retrievability* result: given the step sequences of a deterministic event structure, we can retrieve this event structure up to isomorphism. Within the theory of concurrency there are quite a number of other retrievability results. BEST & DEVILLERS [5] prove various retrievability results for Petri nets. KIEHN [15] describes how the partial language of a p/t net can be recovered from the set of its step sequences. SHIELDS [25] considers a subclass of deterministic systems ('behaviour systems with conservative labelling') which makes it possible to lift concurrency up to a relation on labels, just like in MAZURKIEWICZ's trace theory [16]. In both cases the partial order structure of a system can be retrieved from firing sequences (or words) and the concurrency relation. In TRAKHTENBROT, RABINOVICH & HIRSHFELD [27], some retrievability results are proved for 'behaviour structures'.

In this paper we investigate the effect of assuming determinism on the lattice of equivalences in between sequence/trace equivalence and event structure isomorphism. In the course of the discussion we will sketch parts of this lattice: we will define a number of equivalences and establish their mutual relationships. Hence our paper can be viewed as a contribution to the research area of comparative concurrency semantics. Related work on this topic has been done by POMELLO [21], VAN GLABBEEK & VAANDRAGER [11] and ACETO, DE NICOLA & FANTECHI [1].

2. EVENT STRUCTURES

2.1. DEFINITION. A *(labelled) event structure (over an alphabet A)* is a 4-tuple $(E, \leqslant, \sharp, l)$, where

- E is a set of *events*;
- $\leqslant \subseteq E \times E$ is a partial order satisfying the principle of *finite causes*:

$$\{e' \in E \mid e' \leqslant e\} \text{ is finite for } e \in E;$$

- $\sharp \subseteq E \times E$ is an irreflexive, symmetric relation (the *conflict relation*) satisfying the principle of *conflict heredity*:

$$e_1 \sharp e_2 \leqslant e_3 \Rightarrow e_1 \sharp e_3;$$

- l: $E \to A$ is a *labelling function.*

As usual we write $e' < e$ for $e' \leqslant e \wedge e' \neq e$, $\geqslant$ for $\leqslant^{-1}$, and $>$ for $<^{-1}$. We use $\smile$ to denote the relation $E \times E - (\leqslant \cup \geqslant \cup \sharp)$. $\smile$ is called the *concurrency relation*. By definition $<$, $=$, $>$, $\sharp$ and $\smile$ form a partition of $E \times E$.

2.2. Note. The components of an event structure **E** will be denoted by respectively $E_{\mathbf{E}}$, $\leqslant_{\mathbf{E}}$, $\sharp_{\mathbf{E}}$ and $l_{\mathbf{E}}$. The derived relations will be denoted $\smile_{\mathbf{E}}$, $<_{\mathbf{E}}$, $>_{\mathbf{E}}$, $\geqslant_{\mathbf{E}}$. For $e \in E_{\mathbf{E}}$, $pre_{\mathbf{E}}(e)$ denotes the set of events which precede e in the ordering (so $pre_{\mathbf{E}}(e) = \{e' \in E_{\mathbf{E}} \mid e' \leqslant_{\mathbf{E}} e\}$).

2.3. Graphical representation. In the graphical representation we either depict the events or their labels, depending on what we want to illustrate. The partial order relation is indicated by arrows. The confict relation is denoted by means of dotted lines. If we draw no relation between events they are concurrent, unless, by means of the transitive and reflexive closure of the arrows, it can be deduced that they are ordered, or, by means of the principle of conflict heredity, it can be deduced that they are in conflict.

2.4. EXAMPLE. Let the event structure **E** be given by:

$$E_{\mathbf{E}} = \{e_1, e_2, e_3, e_4, e_5\}$$

$$\leqslant_{\mathbf{E}} = \{(e_1, e_2), (e_1, e_3), (e_2, e_3)\} \cup \{(e, e) \mid e \in E_{\mathbf{E}}\}$$

$$\sharp_{\mathbf{E}} = \{(x, e_4), (e_4, x) \mid x \in \{e_1, e_2, e_3\}\}$$

$$l_{\mathbf{E}}(e_i) = a_i$$

Graphically we can depict **E** as follows:

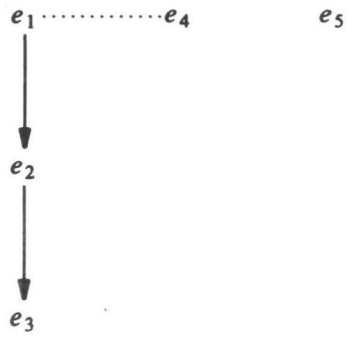

FIGURE 1

2.5. Operational meaning of event structures. The events in a event structure can be anything varying from a clock pulse in a computer, the printing of a file, my act of writing this article, your act of reading it, the next crash of Wall Street, etc.

The partial order relation expresses that some events are causally related to other events or that for all observers the occurrence of certain events will be seen to precede the occurrence of others. For instance, my act of writing this article will precede your act of reading it. On the other hand, your act of reading this article will probably not be causally related to the next crash of Wall

Street. The question what, in general, constitutes a causal link, is a metaphysical one and difficult to answer. However, in a lot of practical situations it is perfectly clear what we mean with causality and reasoning about the behaviour of concurrent systems in terms of causality is useful.

The principle of finite causes says that the systems we consider are discrete and that moreover we do not consider situation like

FIGURE 2

or

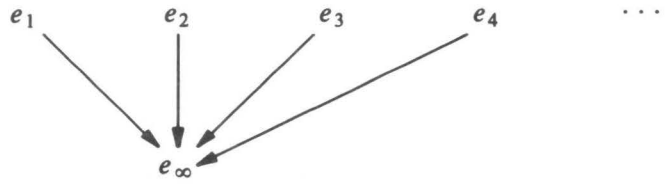

FIGURE 3

In the first situation it is not clear that any of the e_i can ever happen, in the second situation e_∞ can occur if execution of *all* events $e_1, e_2, \cdots$ finishes after a finite amount of time. Because we do not make any assumptions about the time it takes to perform an event, it is possible that e_1 takes 1 second, e_2 takes 2 seconds, etc. In that case e_∞ will never take place.

If two events are in conflict, then at most one of them can occur. As a consequence of the principle of conflict heredity we have that when an event occurs, all its 'causes' must have occurred before. So if two events e and e' are related in the ordering, say $e<e'$, then occurrence of e is a prerequisite for the occurrence of e'. In general it is not the case that after occurrence of e the occurrence of e' is inevitable. It would be possible to allow event structures where one event has two causes, which are in conflict:

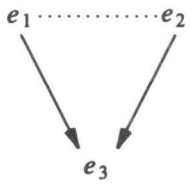

FIGURE 4

Two interpretations of the above event structure are possible: either one can say that e_3 will never occur because it is impossible that all its causes occur (in

that case one can just as well leave e_3 out of the event structure and adopt the principle of conflict heredity), or one can say that e_3 can occur if a maximal, conflict-free subset of its causes has occurred, so $\{e_1\}$ ór $\{e_2\}$.

There are no fundamental reasons to adopt the principles of finite causes and conflict heredity. We have included them in our definition of event structures because this makes an elegant formulation possible of the main result of this paper.

The operational intuitions that we presented in the discussion above, will be defined formally below.

2.6. DEFINITION. Let $\mathbf{E}$ be an event structure and let X be a subset of $E_{\mathbf{E}}$. We say that X is *left-closed* if

$$e \in X \wedge e' \leqslant_{\mathbf{E}} e \Rightarrow e' \in X.$$

X is *conflict-free* if X does not contain a pair of events which are in conflict, so if $\sharp_{\mathbf{E}} \cap (X \times X) = \emptyset$. $\mathbf{E}$ is *conflict-free* if $\sharp_{\mathbf{E}} = \emptyset$. A *configuration* of $\mathbf{E}$ is a finite,[1] left-closed, conflict-free subset of $E_{\mathbf{E}}$. With $\mathcal{C}(\mathbf{E})$ we denote the set of configurations of $\mathbf{E}$.

2.7. EXAMPLE. In Figure 5 below we have depicted all configurations of the event structure of Example 2.4. An arrow is drawn between two configurations if one can be obtained from the other by adding a single event.

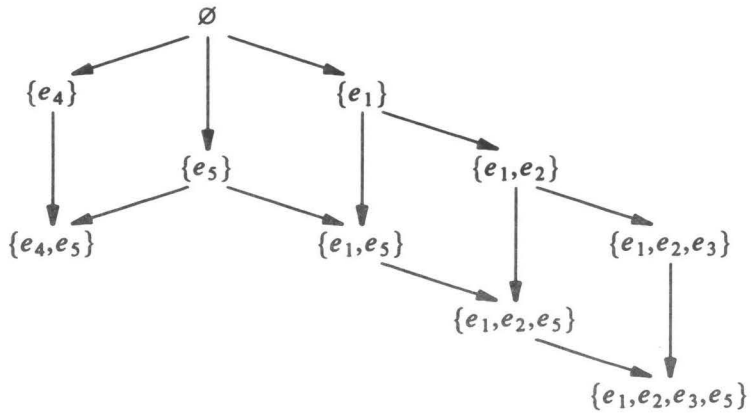

FIGURE 5

1. WINSKEL [28] does not require that configurations are finite.

2.8. DEFINITION. For any alphabet Σ, we use Σ^* to denote the set of finite sequences over alphabet Σ and Σ^+ to denote the set of finite nonempty sequences over this alphabet. We write λ for the empty sequence and a for the sequence consisting of the single symbol $a \in \Sigma$. By $\sigma * \sigma'$, sometimes abbreviated $\sigma\sigma'$, we denote the concatenation of sequences σ and σ'. On sequences we define a partial ordering $\leqslant$ (the *prefix ordering*) by: $\sigma \leqslant \rho$ iff, for some sequence σ', $\sigma\sigma' = \rho$. If $\sigma \leqslant \rho$ we say that σ is a *prefix* of ρ.

2.9. DEFINITION. Let $\mathbf{E}$ be an event structure and let X and Y be configurations of $\mathbf{E}$.

i) Let $a \in A$. We say that there is an *a-transition* from X to Y, notation $X \xrightarrow{a}_{\mathbf{E}} Y$, if $Y = X \cup \{e\}$ for some event $e \notin X$ with $l_{\mathbf{E}}(e) = a$.

ii) An action $a \in A$ is *enabled* in X, notation $X \xrightarrow{a}_{\mathbf{E}}$, if $X \xrightarrow{a}_{\mathbf{E}} X'$ for some configuration X'.

iii) A sequence of actions $\sigma = a_1 * \cdots * a_n \in A^*$ is *enabled* in X, notation $X \xrightarrow{\sigma}_{\mathbf{E}}$, if there exist configurations $X_0, \ldots, X_n$ such that $X = X_0$ and for $1 \leqslant i \leqslant n$: $X_{i-1} \xrightarrow{a_i}_{\mathbf{E}} X_i$. We say that X_n is *obtained from X by the occurrence of* σ, notation $X \xrightarrow{\sigma}_{\mathbf{E}} X_n$. We also say that σ is an *(action) sequence* of X.

iv) A sequence of events $\alpha = e_1 * \cdots * e_n \in E_{\mathbf{E}}^*$ is *enabled* in X, notation $X \xrightarrow{\alpha}_{\mathbf{E}}$, if there exist configurations $X_0, \ldots, X_n$ such that $X = X_0$ and for $1 \leqslant i \leqslant n$: $e_i \notin X_{i-1}$ and $X_i = X_{i-1} \cup \{e_i\}$. We say that α is an *(event) sequence* of X.

v) With $seq_{\mathbf{E}}(X)$ we denote the set of action sequences of X, so $seq_{\mathbf{E}}(X) = \{\sigma \in A^* \mid X \xrightarrow{\sigma}_{\mathbf{E}}\}$.

2.10. PROPOSITION *(no junk). Let* $\mathbf{E}$ *be an event structure and let* $e \in E_{\mathbf{E}}$. *Then there exists a configuration* X *of* $\mathbf{E}$ *with* $e \in X$.

PROOF. Take $X = pre_{\mathbf{E}}(e)$. Due to the principle of finite causes X is finite. From the fact that $\leqslant_{\mathbf{E}}$ is a partial order it follows that X is left-closed. X is conflict-free due to the principle of conflict heredity. Hence X is a configuration. Clearly $e \in X$. □

3. THREE BASIC EQUIVALENCES ON EVENT STRUCTURES

We will now define three equivalences on event structures which make increasingly more identifications.

3.1. DEFINITION. An *event structure isomorphism* between two even structures $\mathbf{E}$ and $\mathbf{F}$ is a bijective mapping $f: E_{\mathbf{E}} \to E_{\mathbf{F}}$ such that:

- $f(e) \leqslant_{\mathbf{F}} f(e') \Leftrightarrow e \leqslant_{\mathbf{E}} e'$,
- $f(e) \sharp_{\mathbf{F}} f(e') \Leftrightarrow e \sharp_{\mathbf{E}} e'$ and
- $l_{\mathbf{F}}(f(e)) = l_{\mathbf{E}}(e)$.

$\mathbf{E}$ and $\mathbf{F}$ are *isomorphic*, notation $\mathbf{E} \cong \mathbf{F}$, if there exists an event structure

isomorphism between them.

3.2. DEFINITION. Let **E**, **F** be two event structures. A relation $R \subseteq \mathcal{C}(\mathbf{E}) \times \mathcal{C}(\mathbf{F})$ is a *bisimulation* between **E** and **F** if:

1. $\varnothing R \varnothing$;
2. If $X R Y$ and $X \xrightarrow{a}_{\mathbf{E}} X'$ for some $a \in A$, then there exists a $Y' \in \mathcal{C}(\mathbf{F})$ such that $Y \xrightarrow{a}_{\mathbf{E}} Y'$ and $X' R Y'$;
3. As 2 but with the roles of X and Y reversed.

E and **F** are *bisimilar*, notation $\mathbf{E} \leftrightarrows \mathbf{F}$, if there exists a bisimulation between them.

3.3. DEFINITION. Two event structures **E** and **F** are *sequence equivalent*, notation $\mathbf{E} \equiv_{seq} \mathbf{F}$, if:

$$seq_{\mathbf{E}}(\varnothing) = seq_{\mathbf{F}}(\varnothing).$$

3.3.1. REMARK. The semantical notion of sequence equivalence, is usually called *trace* equivalence in the settings of process algebra and trace theory à la REM [23]. However, use of the word trace would be very confusing in a paper on event structures, since event structures are closely related to a completely different type of traces, namely those which are studied in trace theory à la MAZURKIEWICZ [16]. Therefore we have chosen to use the word 'sequence' to denote a finite string of symbols recording the actions in which a process has engaged up to some moment in time.

3.4. PROPOSITION. *$\cong$, $\leftrightarrows$ and $\equiv_{seq}$ are equivalence relations and their relations are as indicated below:*

$$\cong \Rightarrow \leftrightarrows \Rightarrow \equiv_{seq}.$$

PROOF. Standard. □

3.5. EXAMPLES. The event structures in Figure 6 show that $\cong$, $\leftrightarrows$ and $\equiv_{seq}$ are really different equivalences. In the graphical representations we have depicted the labels of the events and not the events themselves.

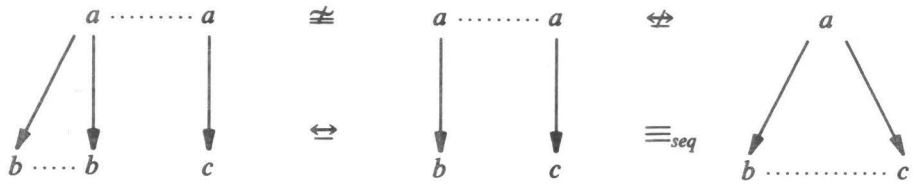

FIGURE 6

The following definition is central in this paper:

3.6. DEFINITION. Let $\mathbf{E}$ be an event structure. $\mathbf{E}$ is *deterministic* if for all configurations $X \in \mathcal{C}(\mathbf{E})$ we have that whenever $X \xrightarrow{a}_{\mathbf{E}} Y$ and $X \xrightarrow{a}_{\mathbf{E}} Y'$ for some $a \in A$ and $Y, Y' \in \mathcal{C}(\mathbf{E})$, we have that $Y = Y'$.

So an event structure is deterministic if it does not have a configuration with the property that two different events are enabled which have the same label.

3.7. DEFINITION. Let $\mathbf{E}$ be an event structure. Two events $e, e' \in E_{\mathbf{E}}$ are in *immediate conflict*, notation $e \sharp^1_{\mathbf{E}} e'$, if they are in conflict and furthermore:

$$e \geqslant_{\mathbf{E}} f \sharp_{\mathbf{E}} e' \Rightarrow e = f \quad \text{and} \quad e \sharp_{\mathbf{E}} f \leqslant_{\mathbf{E}} e' \Rightarrow f = e'.$$

Using the notion of immediate conflict we can give a 'less operational' characterization of deterministic event structures.

3.8. PROPOSITION. *Let* $\mathbf{E}$ *be an event structure. Then* $\mathbf{E}$ *is deterministic iff:*

$$e \smile_{\mathbf{E}} e' \text{ or } e \sharp^1_{\mathbf{E}} e' \Rightarrow l_{\mathbf{E}}(e) \neq l_{\mathbf{E}}(e').$$

PROOF. Easy. □

It is well-known that the linear time - branching time spectrum collapses for deterministic event structures.

3.9. PROPOSITION. *Let* $\mathbf{E}, \mathbf{F}$ *be deterministic event structures. Then:* $\mathbf{E} \leftrightarrows \mathbf{F} \Leftrightarrow \mathbf{E} \equiv_{seq} \mathbf{F}$.

PROOF. '$\Rightarrow$' follows from Proposition 3.4. In order to prove '$\Leftarrow$' define a relation $R \subseteq \mathcal{C}(\mathbf{E}) \times \mathcal{C}(\mathbf{F})$ by:

$$X \; R \; Y \Leftrightarrow seq_{\mathbf{E}}(X) = seq_{\mathbf{F}}(Y).$$

It is easy to show that R gives a bisimulation between $\mathbf{E}$ and $\mathbf{F}$. □

3.10. REMARK. In a dictionary ([20]) we found the following entry for the word 'determinism':

1. a doctrine that all phenomena are determined by preceding occurrences; *esp.* the doctrine that all human acts, choices etc are causally determined and that free will is illusory;
2. a belief in predestination.

One may think that the notion of determinism introduced in Definition 3.6 is in conflict with the above description. If one for instance considers the deterministic event structure containing two events labelled a and b which are in conflict, then one may argue that the choice between a and b is not causally determined, that the event structure 'has a free will' and 'may choose' whether to perform a or b. Therefore one may propose another definition of determinism for event structures which says that an event structure is deterministic iff it is conflict-free. In fact this definition occurs in ACETO, DE NICOLA & FANTECHI [1].

We however prefer our own definition because we like to view event structures as 'reactive systems'. An event structure model of a concurrent system describes how the system reacts to stimuli received from its environment. In the above example of the event structure with actions a and b, it is completely determined how a system modelled by this event structure will react to external stimuli: the system has no choice.

Now consider the following event structure:

FIGURE 7

This event structure is conflict-free and hence deterministic in the sense of [1]. However, if the environment offers an a, then there is a choice between the 'left' a and the 'right' a. Depending on how this choice is resolved by the system, it can engage in b *or* in c afterwards. Hence one can argue that the event structure exhibits nondeterministic behaviour.

4. NON-INTERLEAVED EQUIVALENCES

Many people think that bisimilation equivalence, and consequently also sequence equivalence, make too many identifications on event structures to be of use in general. In bisimulation semantics concurrency is not preserved, i.e. for each event structure we can give a bisimilar event structure with an empty concurrency relation. We elaborate on this below.

4.1.1. DEFINITION. The *sequentialisation* of an event structure $\mathbf{E}$, notation $\mathbb{S}(\mathbf{E})$, is the event structure $\mathbf{F}$ defined by:

- $E_{\mathbf{F}} = \{\alpha \in (E_{\mathbf{E}})^+ \mid \varnothing \xrightarrow{\alpha}_{\mathbf{E}}\}$;
- $\alpha \leqslant_{\mathbf{F}} \beta$ iff α is a prefix of β;
- $\sharp_{\mathbf{F}} = (E_{\mathbf{F}} \times E_{\mathbf{F}}) - (\leqslant_{\mathbf{F}} \cup \geqslant_{\mathbf{F}})$;
- $l_{\mathbf{F}}(\alpha * e) = l_{\mathbf{E}}(e)$.

4.1.2. PROPOSITION. *Let* $\mathbf{E}$ *be an event structure. Then:*

i) the concurrency relation of $\mathbb{S}(\mathbf{E})$ is empty,
ii) $\mathbf{E} \leftrightarrows \mathbb{S}(\mathbf{E})$,
iii) $\mathbb{S}(\mathbf{E}) \cong \mathbb{S}(\mathbb{S}(\mathbf{E}))$.

PROOF. Easy. □

4.2. Step semantics. Intuitively, one of the reasons why an event structure is in general different from its sequentialisation is that it sometimes has the possibility to do a number of events simultaneously in one 'step'. The notion of a 'step' immediately suggests refinements of sequence equivalence and bisimulation equivalence which do not disregard concurrency. These refinements will be called step sequence equivalence and step bisimulation equivalence respectively. Step sequences were defined already in [10]. Step bisimulations appear in [19]. In [11] they are called 'concurrent bisimulations'. Below we give the formal definitions of step sequence equivalence.

4.2.1. DEFINITION. Let $\mathbf{E}$ be an event structure and let X and Y be configurations of $\mathbf{E}$.

(i) Let U be a finite subset of $E_{\mathbf{E}}$. We say that Y *U-follows* X, notation $X[U\rangle Y$, if $X \cap U = \emptyset$, the elements of U are pairwise concurrent (so $\forall e,e' \in U: e \neq e' \Rightarrow e \frown_{\mathbf{E}} e'$) and $Y = X \cup U$.

(ii) Let $U \subseteq E_{\mathbf{E}}$. We say that U is *enabled* in X (U is a *step* from X), notation $X[U\rangle_{\mathbf{E}}$, if $X[U\rangle_{\mathbf{E}} X'$ for some configuration X' of $\mathbf{E}$.

(iii) A sequence $\alpha = U_1 \star \cdots \star U_n \in (Pow(E_{\mathbf{E}}))^*$ is *enabled* in X, notation $X[\alpha\rangle_{\mathbf{E}}$, if there exist configurations $X_0,\ldots,X_n$ such that $X = X_0$ and for $1 \leqslant i \leqslant n$: $X_{i-1}[U_i\rangle_{\mathbf{E}} X_i$. We say that X_n *is obtained from* X *by the occurrence of* α, notation $X[\alpha\rangle_{\mathbf{E}} X_n$. We also say that α is an *(event) step sequence of* X.

(iv) Let $\alpha = U_1 \star \cdots \star U_n \in (Pow(E_{\mathbf{E}}))^*$ such that $X[\alpha\rangle_{\mathbf{E}} Y$. Let σ be the sequence $l_{\mathbf{E}}(U_1) \star \cdots \star l_{\mathbf{E}}(U_n)$ where $l_{\mathbf{E}}(U_i)$ denotes the multiset of labels of events in U_i. We say that σ is *enabled* in X, notation $X[\sigma\rangle_{\mathbf{E}}$. We also say that σ is an *(action) step sequence* of X, and that Y is *obtained from* X *by the occurrence of* σ, notation $X[\sigma\rangle_{\mathbf{E}} Y$.

(v) With $step_{\mathbf{E}}(X)$ we denote the set of action step sequences of X, so $step_{\mathbf{E}}(X) = \{\sigma \in (Mul(A))^* \mid X[\sigma\rangle_{\mathbf{E}}\}$.

4.2.2. DEFINITION. Two event structures $\mathbf{E}$ and $\mathbf{F}$ are *step sequence equivalent*, notation $\mathbf{E} \equiv_{step} \mathbf{F}$, if:

$$step_{\mathbf{E}}(\emptyset) = step_{\mathbf{F}}(\emptyset).$$

4.2.3. PROPOSITION. *$\equiv_{step}$ is an equivalence relation. The following relations hold between the equivalences presented thus far:*

$$\begin{array}{ccc} \cong & \Rightarrow & \leftrightarrows \\ \Downarrow & & \Downarrow \\ \equiv_{step} & \Rightarrow & \equiv_{seq} \end{array}$$

PROOF. Easy. □

4.2.4. EXAMPLES. We give some examples which show that the diagram above gives *all* relations between the equivalences. Our first example shows that step semantics (at least sometimes) takes concurrency as a primitive notion.

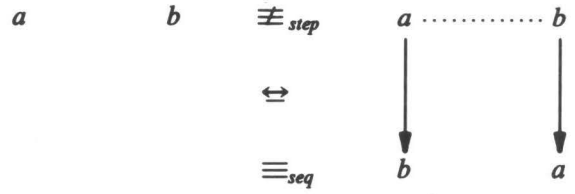

FIGURE 8

The two leftmost event structures in Figure 6 are not isomorphic but they are step sequence equivalent. This follows from the observation that on the domain of event structures with empty concurrency relation step sequence equivalence and sequence equivalence coincide.

The two rightmost event structures in Figure 6 are not bisimilar, but they are step sequence equivalent.

4.3. Partial order semantics. An *A-labelled partially ordered set* is a triple $(X,\leqslant,l)$ with X a set, $\leqslant$ a partial order on X, and $l: X\to A$ a labelling function. Two such sets $(X_0,\leqslant_0,l_0)$ and $(X_1,\leqslant_1,l_1)$ are *isomorphic* if there exists a bijective mapping $f: X_0\to X_1$ such that $f(x)\leqslant_1 f(y) \Leftrightarrow x\leqslant_0 y$ and $l_1(f(x))=l_0(x)$. A *partially ordered multiset (pomset)* is an isomorphism class of labelled partially ordered sets. As usual, pomsets can be made setlike by requiring that the events in the partial orders should be chosen from a given set. Below we will view equivalence classes of conflict-free event structures as pomsets.

4.3.1. DEFINITION. The *restriction* of an event structure $\mathbf{E}$ to a set $X\subseteq E_{\mathbf{E}}$ of events is the event structure $\mathbf{E}\restriction X = (X,\ \leqslant_{\mathbf{E}}\cap(X\times X),\ \sharp_{\mathbf{E}}\cap(X\times X),\ l_{\mathbf{E}}\restriction X)$.

4.3.2. DEFINITION. Let $\mathbf{E}$ be an event structure and let X be a configuration of $\mathbf{E}$. The set of *pomsets* of X, notation $pom_{\mathbf{E}}(X)$, is defined by:

$$pom_{\mathbf{E}}(X) = \{(\mathbf{E}\restriction(X'-X))/_{\simeq} \mid X\subseteq X'\in\mathcal{C}(\mathbf{E})\}.$$

4.3.3. DEFINITION. Two event structures $\mathbf{E}$ and $\mathbf{F}$ are *pomset equivalent*, notation $\mathbf{E}\equiv_{pom}\mathbf{F}$, if:

$$pom_{\mathbf{E}}(\varnothing) = pom_{\mathbf{F}}(\varnothing).$$

The first systematic study of pomsets is by GRABOWSKI [12], who called them *partial words*. Pomset semantics is advocated by PRATT [22].

4.3.4. PROPOSITION. $\equiv_{pom}$ *is an equivalence relation. It fits in our semantical lattice as follows:*

$$\begin{array}{ccccc} \cong & & \Rightarrow & & \leftrightarrows \\ \Downarrow & & & & \Downarrow \\ \equiv_{pom} & \Rightarrow & \equiv_{step} & \Rightarrow & \equiv_{seq} \end{array}$$

4.3.5. EXAMPLES. The two rightmost event structures in Figure 6 provide an example of two event structures which are identified in pomset semantics, but distinguished in bisimulation semantics. The remaining examples distinguishing pomset equivalence and the other equivalences are displayed in Figure 9 below. The example of Figure 10 is interesting because it only contains conflict-free event structures. The example disproves Theorem 3.5 of ACETO, DE NICOLA & FANTECHI [1].

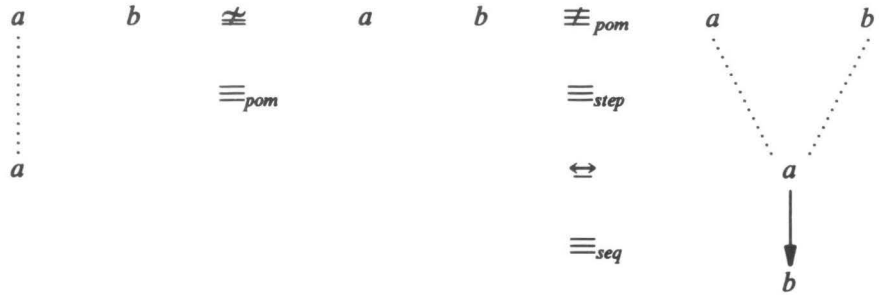

FIGURE 9

FIGURE 10

Notice that all these examples contain non-deterministic event structures.

5. DETERMINISM → (EVENT STRUCTURE ISOMORPHISM = STEP SEQUENCE EQUIVALENCE)

Proposition 3.9 stated that bisimulation equivalence and sequence equivalence coincide on the domain of deterministic event structures. Surprisingly, most of the non-interleaved semantics which have been proposed in the literature, also coincide on this domain.

In the introduction of this paper we mentioned a large number of

equivalences which are situated in between event structure isomorphism and step sequence equivalence. As a consequence of the following result *all* these equivalences (except for occurrence net equivalence) coincide with event structure isomorphism on the domain of deterministic event structures.

5.1. THEOREM. *Let* $\mathbf{E},\mathbf{F}$ *be deterministic event structures. Then:* $\mathbf{E} \cong \mathbf{F} \Leftrightarrow \mathbf{E} \equiv_{step} \mathbf{F}$.

5.2. LEMMA. *Let* $\mathbf{E}$ *be a deterministic event structure and let* X,Y *be configurations of* $\mathbf{E}$ *such that* $\mathbf{E}\restriction X \cong \mathbf{E}\restriction Y$. *Then:* $X=Y$.
PROOF. Induction on the size of X. If X is the empty set, then Y must be empty too and we are done. Suppose X is nonempty. Let e be a maximal element of X and let $X'=X-\{e\}$. Now we use that there exists an event structure isomorphism f between $\mathbf{E}\restriction X$ and $\mathbf{E}\restriction Y$: we have $\mathbf{E}\restriction X' \cong \mathbf{E}\restriction Y'$ for $Y'=Y-\{f(e)\}$ and furthermore X' and Y' are configurations. Applying the induction hypothesis gives $X'=Y'$. Let $a=l_{\mathbf{E}}(e)=l_{\mathbf{E}}(f(e))$. We have that $X' \xrightarrow{a}_{\mathbf{E}} X$ but also $X' \xrightarrow{a}_{\mathbf{E}} Y$. Now use that $\mathbf{E}$ is deterministic to obtain that $X=Y$. □

5.3. LEMMA. *Let* $\mathbf{E}$ *and* $\mathbf{F}$ *be deterministic event structures. Then:* $\mathbf{E} \equiv_{pom} \mathbf{F} \Leftrightarrow \mathbf{E} \cong \mathbf{F}$.
PROOF. '$\Leftarrow$' is trivial, so the interesting direction is '$\Rightarrow$'. Define relation $\sim \subseteq E_{\mathbf{E}} \times E_{\mathbf{F}}$ by:

$$e_0 \sim e_1 \Leftrightarrow_{def} \mathbf{E}\restriction pre_{\mathbf{E}}(e_0) \cong \mathbf{F}\restriction pre_{\mathbf{F}}(e_1).$$

We claim that $\sim$ gives a bijective mapping between $E_{\mathbf{E}}$ and $E_{\mathbf{F}}$. Because $\mathbf{E} \equiv_{pom} \mathbf{F}$, it is obvious that $dom(\sim)=E_{\mathbf{E}}$ and $range(\sim)=E_{\mathbf{F}}$. Suppose that $e_0 \sim e_1$ and $e_0 \sim e_1'$. We show that $e_1=e_1'$. By definition we have $\mathbf{E}\restriction pre_{\mathbf{E}}(e_0) \cong \mathbf{F}\restriction pre_{\mathbf{F}}(e_1) \cong \mathbf{F}\restriction pre_{\mathbf{F}}(e_1')$. Application of the previous lemma gives $pre_{\mathbf{F}}(e_1)=pre_{\mathbf{F}}(e_1')$. Since both sets have a unique maximal element, these maximal elements must be identical: $e_1=e_1'$. In the same way we can prove that if $e_0 \sim e_1$ and $e_0' \sim e_1$, this implies $e_0=e_0'$. Hence $\sim$ gives a bijection between $E_{\mathbf{E}}$ and $E_{\mathbf{F}}$. It is not hard to see that this bijection is in fact an event structure isomorphism. □

Proof of Theorem 5.1. From the previous results it follows that in order to prove Theorem 5.1 it is enough to show that for deterministic event structures $\mathbf{E},\mathbf{F}$:

$$\mathbf{E} \equiv_{step} \mathbf{F} \Rightarrow \mathbf{E} \equiv_{pom} \mathbf{F}.$$

By definition this is equivalent to:

$$step_{\mathbf{E}}(\emptyset) = step_{\mathbf{F}}(\emptyset) \Rightarrow pom_{\mathbf{E}}(\emptyset) = pom_{\mathbf{F}}(\emptyset).$$

We will prove a slightly stronger statement, namely:

$$\forall X \in \mathcal{C}(\mathbf{E})\ \forall Y \in \mathcal{C}(\mathbf{F}) : step_{\mathbf{E}}(X) = step_{\mathbf{F}}(Y) \Rightarrow pom_{\mathbf{E}}(X) = pom_{\mathbf{F}}(Y).$$

Let $X \in \mathcal{C}(\mathbf{E})$, $Y \in \mathcal{C}(\mathbf{F})$ with $step_{\mathbf{E}}(X) = step_{\mathbf{F}}(Y)$. Let X' be a configuration of $\mathbf{E}$ with $X \subseteq X'$. Let $\alpha_0 = \{e_1\}\{e_2\} \cdots \{e_n\}$ be a sequence of singleton steps such that $X[\alpha_0 >_{\mathbf{E}} X'$ and $X' - X = \{e_1,...,e_n\}$. Let $\alpha_1 = \{e_1'\}\{e_2'\} \cdots \{e_n'\}$ be a step sequence such that $Y[\alpha_1 >_{\mathbf{F}}$ and $l_{\mathbf{E}}(e_i) = l_{\mathbf{F}}(e_i')$ for $1 \leqslant i \leqslant n$ (due to the fact that X and Y have the same step sequences, such a sequence will always exist). Let $Y' = Y \cup \{e_1',...,e_n'\}$. We claim that the function which maps e_i to e_i' is an event structure isomorphism between $\mathbf{E}\restriction(X'-X)$ and $\mathbf{F}\restriction(Y'-Y)$. For reasons of symmetry we have proved the theorem if we have shown this.

The proof goes by induction to n. The case with $n=0$ is trivial. Now suppose $n>0$. Due to the fact that X and Y have the same step sequences and due to the determinism of $\mathbf{E}$ and $\mathbf{F}$, we have:

$$step_{\mathbf{E}}(X \cup \{e_1\}) = step_{\mathbf{F}}(Y \cup \{e_1'\}).$$

Since

$$X \cup \{e_1\}\,[\{e_2\} \cdots \{e_n\} >_{\mathbf{E}} X' \quad \text{and}$$
$$Y \cup \{e_1'\}\,[\{e_2'\} \cdots \{e_n'\} >_{\mathbf{E}} Y',$$

we can now apply the induction hypothesis which gives:

$$\mathbf{E}\restriction(X'-(X \cup \{e_1\})) \cong \mathbf{F}\restriction(Y'-(Y \cup \{e_1'\})).$$

In order to prove the induction step it is enough to show that for $2 \leqslant i \leqslant n$: $e_1 <_{\mathbf{E}} e_i \Leftrightarrow e_1' <_{\mathbf{F}} e_i'$. If $n=1$ we are done, so assume $n \geqslant 2$. Let for some i, e_i be minimal in $\{e_2,...,e_n\}$. Then e_i' is minimal in $\{e_2',...,e_n'\}$. We claim that $e_1 <_{\mathbf{E}} e_i \Leftrightarrow e_1' <_{\mathbf{F}} e_i'$. Suppose $e_1 <_{\mathbf{E}} e_i$ but not $e_1' <_{\mathbf{F}} e_i'$. If we show that this leads to a contradiction we have proved the claim because the remaining case is symmetric. If it is not the case that $e_1' <_{\mathbf{F}} e_i'$ then $e_1' \smile_{\mathbf{F}} e_i'$. Due to the minimality of e_i' we have that $Y[\{e_1', e_i'\} >_{\mathbf{F}}$. Now we use that X and Y have the same step sequences and the fact that $\mathbf{E}$ is deterministic. There must be some f such that $X[\{e_1,f\} >_{\mathbf{E}}$ and $l_{\mathbf{E}}(f) = l_{\mathbf{F}}(e_i') = l_{\mathbf{E}}(e_i)$. Because $e_1 <_{\mathbf{E}} e_i$, $f \neq e_i$. But now there is a contradiction since we can go from configuration $X \cup \{e_1\}$ with an $l_{\mathbf{E}}(f)$-transition to $X \cup \{e_1,f\}$ as well as $X \cup \{e_1,e_i\}$.

Now we have proved that for e_i, which are minimal in $\{e_2,...,e_n\}$, $e_1 <_{\mathbf{E}} e_i \Leftrightarrow e_1' <_{\mathbf{F}} e_i'$. In order to prove this fact also for e_i which are not minimal, we distinguish between two cases.

1. For all e_i which are minimal in $\{e_2,...,e_n\}$, we have that $e_1 <_{\mathbf{E}} e_i$. This implies that $e_1 <_{\mathbf{E}} e_l$ for $2 \leqslant l \leqslant n$. Further we have that for all e_i' which are minimal in $\{e_2',...,e_n'\}$, $e_1' <_{\mathbf{F}} e_i'$. Consequently $e_1' <_{\mathbf{F}} e_l'$ for $2 \leqslant l \leqslant n$, and we are done.
2. There is an e_i which is minimal in $\{e_2,...,e_n\}$ such that $e_1 \smile_{\mathbf{E}} e_i$. This means that $e_1' \smile_{\mathbf{F}} e_i'$. We now have the following situation:

$$X \cup \{e_i\}\,[\{e_1\} \cdots \{e_{i-1}\}\{e_{i+1}\} \cdots \{e_n\} >_{\mathbf{E}} X' \quad \text{and}$$
$$Y \cup \{e_i'\}\,[\{e_1'\} \cdots \{e_{i-1}'\}\{e_{i+1}'\} \cdots \{e_n'\} >_{\mathbf{F}} Y'.$$

Of course $X \cup \{e_i\}$ and $Y \cup \{e_i'\}$ have the same step sequences. Application of the induction hypothesis gives:

$$\mathbf{E} \restriction \{e_1, e_2, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n\} \cong \mathbf{F} \restriction \{e_1', e_2', \ldots, e_i', e_{i+1}', \ldots, e_n'\}.$$

Consequently $e_1 <_{\mathbf{E}} e_l \Leftrightarrow e_1' <_{\mathbf{F}} e_l'$ for $2 \leqslant l \leqslant n$. □

Observe that in the proof of Theorem 5.1 we only use that **E** and **F** have the same sequences of steps containing at most two events.

5.4. The diagram below presents the relations between the equivalences presented thus far when restricted to the domain of deterministic event structures.

$$\begin{array}{ccccc} \cong & \Leftrightarrow & \equiv_{pom} & \Leftrightarrow & \equiv_{step} \\ \Downarrow & & & & \\ \leftrightarrows & \Leftrightarrow & \equiv_{seq} & & \end{array}$$

The example of Figure 8 shows that even for deterministic systems there is a difference between arbitrary interleaving and partial order semantics.

6. Arbitrary interleaving versus True concurrency

One can consider event structures up to step sequence equivalence as an interleaving semantics if one is willing to view a multiset of actions as an action again. In the process algebra languages Meije and ACP this idea can be implemented by working for instance with an action structure which is the product of a free commutative monoid and a free commutative group. Under this interpretation one can say that for deterministic systems there is no difference between arbitrary interleaving and True concurrency.

Now one can ask the question to what extent a multiset of more than one action can be considered as something which is observable. In a synchronous system like a systolic architecture there is certainly no problem. After each clock tick one can just stop the system and examine which 'cells' have performed an action. The multiset (or set if the system is deterministic) of actions performed by the separate cells gives the step which is performed by the synchronous system. It is much harder to imagine how a 'step' can be observed in an asynchronous system. The only thing I can come up with is that some observer notices the beginning of one action before another action has been finished. In such a situation the observer can conclude that the two actions occur concurrently.

Below, this way of observing concurrent processes is formally implemented by means of an operator *split* on event structures that splits any event e into events e^+ and e^-, which are ordered. One may think of e^+ as the beginning of e and of e^- as the end of e.

6.1.1. DEFINITION. Let **E** be an event structure over some alphabet A. Let $A^+ = \{a^+ \mid a \in A\}$ and $A^- = \{a^- \mid a \in A\}$ be two disjoint copies of A. The event structure $\mathbf{F} = split(\mathbf{E})$ over alphabet $A^+ \cup A^-$ is given by:

$$E_{\mathbf{F}} = \{e^+, e^- \mid e \in E_{\mathbf{E}}\}$$

$$<_{\mathbf{F}} = \{(e^x, f^y) \mid x,y \in \{+,-\} \text{ and } e <_{\mathbf{E}} f\} \cup \{(e^+, e^-) \mid e \in E_{\mathbf{E}}\}$$

$$\#_{\mathbf{F}} = \{(e^x, f^y) \mid x,y \in \{+,-\} \text{ and } e \#_{\mathbf{E}} f\}$$

$$l_{\mathbf{F}}(e^+) = (l_{\mathbf{E}}(e))^+$$

$$l_{\mathbf{F}}(e^-) = (l_{\mathbf{E}}(e))^-$$

6.1.2. REMARK.

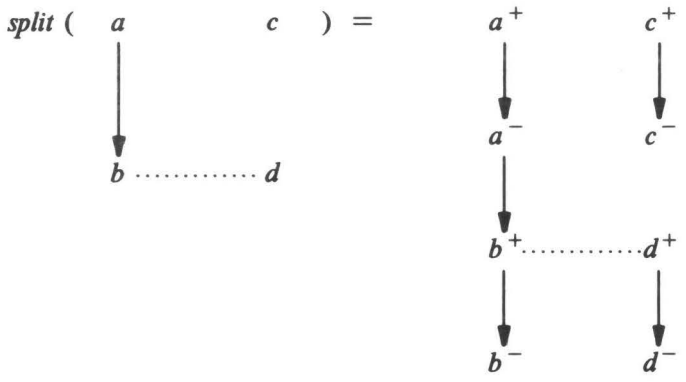

FIGURE 11

6.1.3. DEFINITION. Two event structures **E** and **F** are *split sequence equivalent*, notation $\mathbf{E} \equiv_{split} \mathbf{F}$, if: $split(\mathbf{E}) \equiv_{seq} split(\mathbf{F})$.

Split sequence equivalence is closely related to ST-bisimulation semantics as presented in VAN GLABBEEK & VAANDRAGER [11] on the domain of Petri nets, but there are some differences. Besides the fact that split sequence equivalence does not respect branching time it is also not real time consistent in the sense of [11]. The idea of splitting actions into a beginning and an end is, on a different and more restricted domain, also described by HENNESSY [13]. Our *split*-operator can be viewed as a special case of *action refinement* as described by CASTELLANO, DE MICHELIS & POMELLO [7] and ACETO & HENNESSY [2].

6.2. LEMMA. *Let* **E** *and* **F** *be two event structures. Then:*

$$\mathbf{E} \equiv_{pom} \mathbf{F} \Rightarrow split(\mathbf{E}) \equiv_{pom} split(\mathbf{F}).$$

PROOF. The main idea of the proof occurs already in [7].
Let **E** and **F** be event structures with $\mathbf{E} \equiv_{pom} \mathbf{F}$. Choose a configuration $X \in \mathcal{C}(split(\mathbf{E}))$. We must show that there exists a configuration $Y \in \mathcal{C}(split(\mathbf{F}))$ such that:

$$split(\mathbf{E}) \restriction X \cong split(\mathbf{F}) \restriction Y.$$

By symmetry it follows that we are ready if we have proved this. Define the sets $X^{\pm}, X^{+} \subseteq E_{\mathbf{E}}$ by:

$$X^{\pm} = \{e \in E_{\mathbf{E}} \mid e^{+} \in X \text{ and } e^{-} \in X\},$$

$$X^{+} = \{e \in E_{\mathbf{E}} \mid e^{+} \in X \text{ and } e^{-} \notin X\},$$

One can easily check that $X^{\pm} \cup X^{+}$ is a configuration of **E**. Since $\mathbf{E} \equiv_{pom} \mathbf{F}$, there is a configuration $Y \in \mathcal{C}(\mathbf{F})$ and a bijection $f: X^{\pm} \cup X^{+} \to Y$ which gives an event structure isomorphism between $\mathbf{E} \restriction (X^{\pm} \cup X^{+})$ and $\mathbf{F} \restriction Y$. Define $Y^{split} \subseteq E_{split(\mathbf{F})}$ by:

$$Y^{split} = \{(f(e))^{+}, (f(e))^{-} \mid e \in X^{\pm}\} \cup \{(f(e))^{+} \mid e \in X^{+}\}.$$

It is not hard to see that Y^{split} is a configuration of $split(\mathbf{F})$. Now define a mapping $f^{split}: X \to Y^{split}$ by:

$$f^{split}(e^{+}) = (f(e))^{+} \quad \text{for } e^{+} \in X,$$

$$f^{split}(e^{-}) = (f(e))^{-} \quad \text{for } e^{-} \in X.$$

We claim that f^{split} is an event structure isomorphism between $split(\mathbf{E}) \restriction X$ and $split(\mathbf{F}) \restriction Y^{split}$. A simple argument gives that f^{split} is a bijection. Clearly f^{split} preserves labels. Finally we have that if two events in X are ordered their images under f^{split} are also ordered, and if two events in X are concurrent their images under f^{split} are concurrent too. □

6.3. PROPOSITION. *Let* **E** *and* **F** *be two event structures. Then:*

$$\mathbf{E} \equiv_{pom} \mathbf{F} \Rightarrow \mathbf{E} \equiv_{split} \mathbf{F}.$$

PROOF. $\mathbf{E} \equiv_{pom} \mathbf{F} \Rightarrow split(\mathbf{E}) \equiv_{pom} split(\mathbf{F}) \Rightarrow split(\mathbf{E}) \equiv_{seq} split(\mathbf{F}) \Rightarrow \mathbf{E} \equiv_{split} \mathbf{F}$. □

6.4. PROPOSITION. *Let* **E** *and* **F** *be two event structures. Then:* $\mathbf{E} \equiv_{split} \mathbf{F} \Rightarrow \mathbf{E} \equiv_{step} \mathbf{F}$.
PROOF. Let **E** and **F** be two event structures with $\mathbf{E} \equiv_{split} \mathbf{F}$. Let $\sigma = A_1 \cdots A_m \in (Mul(A))^{*}$ with $A_i = \{a_{i1}, \ldots, a_{in_i}\}$ be an action step sequence of **E**. We must show that σ is also an action step sequence of **F**. By symmetry we are ready if we have proved this. The following sequence ρ is an action sequence of $split(\mathbf{E})$:

$$\rho = a_{11}^{+}\ a_{12}^{+} \cdots a_{1n_1}^{+}\ a_{11}^{-}\ a_{12}^{-} \cdots a_{1n_1}^{-}\ a_{21}^{+} \cdots a_{m1}^{+} \cdots a_{mn_m}^{+}\ a_{m1}^{-} \cdots a_{mn_m}^{-}$$

Since $\mathbf{E} \equiv_{split} \mathbf{F}$, ρ is also an action sequence of *split*(**F**). Hence *split*(**F**) has some event sequence α with the property that, if we replace the events in α by their labels, we obtain ρ. Let this α be given by:

$$\alpha = e_{11}^{+}\ e_{12}^{+} \cdots e_{1n_1}^{+}\ f_{11}^{-}\ f_{12}^{-} \cdots f_{1n_1}^{-}\ e_{21}^{+} \cdots e_{m1}^{+} \cdots e_{mn_m}^{+}\ f_{m1}^{-} \cdots f_{mn_m}^{-}$$

Note that in general e_{ij} may be different from f_{ij}. However, we do have that $\{e_{i1},\ldots,e_{in_i}\}$ equals $\{f_{i1},\ldots,f_{in_i}\}$.
From the fact that α is an event sequence of *split*(**F**) it follows that **F** has the event step sequence:

$$\{e_{11},\ldots,e_{1n_1}\} \cdots \{e_{m1},\ldots,e_{mn_m}\}.$$

Hence σ is an action step sequence of **F**. □

6.5. As a consequence of Propositions 6.3 and 6.4, split sequence equivalence can be located in our semantical lattice as follows:

$$\begin{array}{ccccccc} \cong & & \Rightarrow & & \leftrightarrows & & \\ \Downarrow & & & & \Downarrow & & \\ \equiv_{pom} & \Rightarrow & \equiv_{split} & \Rightarrow & \equiv_{step} & \Rightarrow & \equiv_{seq} \end{array}$$

6.6. EXAMPLES. The following examples show that all equivalences in 6.5 are different.

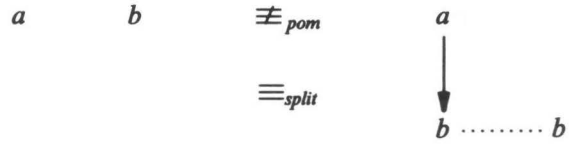

FIGURE 12

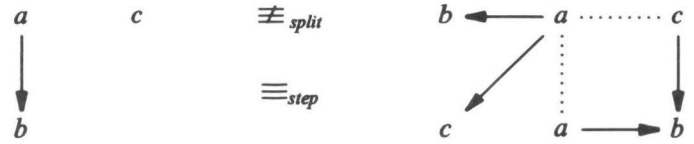

FIGURE 13

Due to Theorem 5.1 and the position of $\equiv_{split}$ in the semantical lattice we have that for deterministic event structures, split bisimulation equivalence and event structure isomorphism coincide:

6.7. PROPOSITION. *Let* **E,F** *be deterministic event structures. Then:* $\mathbf{E} \cong \mathbf{F} \Leftrightarrow \mathbf{E} \equiv_{split} \mathbf{F}$.

Thus the causal structure of a deterministic concurrent system can be unravelled by observers who are capable to observe the beginning and termination of events.

ACKNOWLEDGEMENTS

The author would like to thank Rob van Glabbeek for many stimulating discussions and careful proofreading, Henk Goeman for some useful comments on an earlier version, and Alex Rabinovich for pointing out that the assumption of binary conflict is not essential for the results of this paper.

REFERENCES

[1] L. ACETO, R. DE NICOLA & A. FANTECHI (1987): *Testing equivalences for event structures*. In: Proceedings Advanced School on Mathematical Models for the Semantics of Parallelism, 1986 (M. Venturini Zilli, ed.), LNCS 280, Springer-Verlag, pp. 1-20.

[2] L. ACETO & M. HENNESSY (1989): *Towards action-refinement in process algebras*. In: Proceedings 4th Annual Symposium on Logic in Computer Science (LICS), Asilomar, California, IEEE Computer Society Press, Washington, pp. 138-145.

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

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

[5] E. BEST & R. DEVILLERS (1987): *Sequential and concurrent behavior in Petri net theory*. Theoretical Computer Science 55(1), pp. 87-136.

[6] G. BOUDOL & I. CASTELLANI (1988): *Concurrency and atomicity*. Theoretical Computer Science 59(1/2), pp. 25-84.

[7] L. CASTELLANO, G. DE MICHELIS & L. POMELLO (1987): *Concurrency vs Interleaving: an instructive example*. Bulletin of the EATCS 31, pp. 12-15.

[8] P. DEGANO, R. DE NICOLA & U. MONTANARI (1987): *Observational equivalences for concurrency models*. In: Formal Description of Programming Concepts - III, Proceedings of the third IFIP WG 2.2 working conference, Ebberup 1986 (M. Wirsing, ed.), Elsevier Science Publishers B.V. (North Holland), pp. 105-129.

[9] J. ENGELFRIET (1985): *Determinacy → (observation equivalence = trace equivalence)*. Theoretical Computer Science 36(1), pp. 21-25.

[10] H.J. GENRICH & E. STANKIEWICZ-WIECHNO (1980): *A dictionary of some basic notions of Petri nets*. In: Advanced course on general net theory of processes and systems, Hamburg 1979 (W. Brauer, ed.), LNCS 84, Springer-Verlag.

[11] R.J. VAN GLABBEEK & F.W. VAANDRAGER (1987): *Petri net models for algebraic theories of concurrency*. In: Proceedings PARLE conference,

Eindhoven, Vol. II (Parallel Languages) (J.W. de Bakker, A.J. Nijman & P.C. Treleaven, eds.), LNCS 259, Springer-Verlag, pp. 224-242.

[12] J. GRABOWSKI (1981): *On partial languages*. Fundamenta Informaticae IV(2), pp. 427-498.

[13] M. HENNESSY (1988): *Axiomatising finite concurrent processes*. SIAM Journal on Computing 17(5), pp. 997-1017.

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

[15] A. KIEHN (1988): *On the interrelation between synchronized and non-synchronized behaviour of Petri nets*. J. Inf. Process. Cybern. EIK 24(1/2), pp. 3-18.

[16] A. MAZURKIEWICZ (1987): *Trace theory*. In: Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course, Bad Honnef, September 1986 (W. Brauer, W. Reisig & G. Rozenberg, eds.), LNCS 255, Springer-Verlag, pp. 279-324.

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

[18] M. NIELSEN, G.D. PLOTKIN & G. WINSKEL (1981): *Petri nets, event structures and domains, part I*. Theoretical Computer Science 13(1), pp. 85-108.

[19] M. NIELSEN & P.S. THIAGARAJAN (1984): *Degrees of non-determinism and concurrency: a Petri net view*. In: Proceedings of the 5th Conf. on Found. of Softw. Techn. and Theor. Comp. Sci. (M. Joseph & R. Shyamasundar, eds.), LNCS 181, Springer-Verlag, pp. 89-118.

[20] PENGUIN (1986): *The new penguin english dictionary*, Penguin Books.

[21] L. POMELLO (1986): *Some equivalence notions for concurrent systems. An overview*. In: Advances in Petri Nets 1985 (G. Rozenberg, ed.), LNCS 222, Springer-Verlag, pp. 381-400.

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

[23] M. REM (1987): *Trace theory and systolic computations*. In: Proceedings PARLE conference, Eindhoven, Vol. I (Parallel Architectures) (J.W. de Bakker, A.J. Nijman & P.C. Treleaven, eds.), LNCS 258, Springer-Verlag, pp. 14-33.

[24] G. ROZENBERG & P.S. THIAGARAJAN (1986): *Petri nets: basic notions, structure, behaviour*. In: Current trends in concurrency (J.W. de Bakker, W.-P. de Roever & G. Rozenberg, eds.), LNCS 224, Springer-Verlag, pp. 585-668.

[25] M.W. SHIELDS (1982): *Non sequential behaviour: 1*. Internal Report CSR-120-82, Department of Computer Science, University of Edinburgh.

[26] D.A. TAUBNER & W. VOGLER (1987): *The step failure semantics*. In: Proceedings STACS 87 (F.J. Brandenburg, G. Vidal-Naquet & M. Wirsing, eds.), LNCS 247, Springer-Verlag, pp. 348-359.

[27] B.A. TRAKHTENBROT, A. RABINOVICH & J. HIRSHFELD (1988): *Discerning causality in the behaviour of automata*. Technical Report 104/88, Tel Aviv University.

[28] G. WINSKEL (1987): *Event structures.* In: Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course, Bad Honnef, September 1986 (W. Brauer, W. Reisig & G. Rozenberg, eds.), LNCS 255, Springer-Verlag, pp. 325-392.

Samenvatting:

Algebraïsche Technieken voor Parallelle, Communicerende Processen en hun Toepassing

Dit proefschrift bestaat, naast een inleiding, uit zes artikelen op het gebied van de semantiek van programmeer- en specificatietalen voor parallelle en gedistribueerde computersystemen.

Het eerste artikel behandelt Plotkin's methode om operationele semantiek te geven aan dergelijke talen. Betoogt wordt dat allerlei resultaten die tot nu toe bewezen werden voor een gegeven taal en semantiek afzonderlijk, ook bewezen kunnen worden voor grote klassen van talen en semantieken tegelijkertijd. Eigenschappen waar naar gekeken is zijn onder meer de vraag of bisimulatie equivalentie een congruentie is, de mate waarin een semantiek robuust is onder uitbreidingen van de taal, en de aard van de volledig abstracte semantiek die hoort bij een gegeven taal en observatie-criterium.

De laatste jaren is er veel onderzoek verricht op het gebied van de procesalgebra. Het betreft hier kleine taaltjes, die gegenereerd worden door een beperkt aantal fundamentele taalconstructies zoals parallelle, sequentiële en alternatieve compositie. Het idee is dat een grondige studie van de semantiek van dergelijke kleine talen een gedegen ondergrond verschaft voor het geven van semantiek aan de vaak uiterst complexe talen die in de praktijk gebruikt worden. Het blijkt mogelijk en tevens verhelderend om de diverse semantieken van de basistalen (bijvoorbeeld de semantieken die onder gebruikmaking van Plotkin's methode zijn gedefinieerd) algebraïsch te karakteriseren. In het tweede artikel van dit proefschrift wordt een algemeen raamwerk gepresenteerd om de algebraïsche wetten te groeperen en te structureren. Dit raamwerk wordt dan toegepast op ACP (der Algebra der Communicerende Processen), een in Amsterdam ontwikkelde familie van basistalen.

In het derde artikel worden twee eenvoudige communicatieprotocollen gespecificeerd in de taal van ACP en correct bewezen met behulp van de axioma's die in het tweede artikel zijn gepresenteerd. Dit ondersteunt de claim

dat een volledig algebraïsche verificatie uitvoerbaar is voor kleine systemen (in een infinitaire, conditionele equationele eensoortige logica).

Voor grotere systemen ligt dit anders. In theorie is algebraïsche verificatie nog steeds mogelijk maar in de praktijk blijkt dat correctheidsbewijzen onnodig lang worden en dat het niet mogelijk is om bepaalde elementaire inzichten over het gedrag van een systeem te laten corresponderen met een korte afleiding van een algebraïsche identiteit. Daarom wordt in het vierde artikel het axiomatisch raamwerk ACP uitgebreid met de zogenaamde 'trace logica'. Deze logica laat toe om eigenschappen te formuleren van de rijtjes van acties die door een systeem kunnen worden uitgevoerd. Het artikel laat zien hoe soms de lengte van verificaties drastisch kan worden ingekort door gebruik te maken van trace logica.

Het vijfde artikel behandelt de vertaling naar ACP van de parallelle, object-georiënteerde programmertaal POOL. Dit is een door Philips ontwikkelde taal, bedoeld voor het programmeren van een parallelle computer. Omdat er voor ACP vele semantieken beschikbaar zijn, kan de vertaling van POOL naar ACP gezien worden als een manier om semantiek aan POOL te geven. Verder kan de veelheid aan theorie die voor ACP beschikbaar is gebruikt worden om feiten te bewijzen over POOL. In het artikel wordt in het bijzonder ingegaan op de vraag of bepaalde implementaties van POOL correct zijn. Deze analyse heeft geleid tot de ontdekking van een kleine fout in het taalontwerp van POOL.

Het laatste artikel in dit proefschrift handelt over 'gebeurtenisstructuren' (event structures). Gebeurtenisstructuren vormen de basis voor een serie van nieuwe modellen voor parallelle systemen die verfijnder zijn dan de gangbare transitiesysteemmodellen omdat ze parallellisme, het onafhankelijk en tegelijkertijd plaatsvinden van gebeurtenissen, als primitieve notie incorporeren. Het belangrijkste resultaat van het artikel is dat voor de klasse van 'deterministische' gebeurtenisstructuren vrijwel alle in de literatuur voorkomende noties van equivalentie samenvallen.

STELLINGEN

1. Voor parallelle, op hun omgeving reagerende systemen bestaat er niet een kanonieke notie van waarneembaar gedrag. In plaats daarvan is er een veelheid van dergelijke noties die aanleiding geeft tot een veelheid van procesequivalenties. Het succes van de algebraïsche, axiomatische aanpak van de theorie der communicerende processen wordt voor een belangrijk deel verklaard door deze situatie.

2. Beschouw het domein $\mathcal{G}/\leftrightarrows$ van eindigsplitsende procesgrafen modulo sterke bisimulatie-equivalentie. Zij $[g],[h] \in \mathcal{G}/\leftrightarrows$. Defineer:

$$d([g],[h]) = 2^{-\sup\{n \mid g \text{ en } h \text{ zijn } n\text{-genest simulatie-equivalent}\}},$$

met de conventie dat $2^{-\infty} = 0$. Dan is d een ultrametriek en alle operatoren op $\mathcal{G}/\leftrightarrows$ die definieerbaar zijn via welgefundeerde *tyft/tyxt* regels zijn continu ten opzichte van d.

Zie: het eerste artikel in dit proefschrift. De bovenstaande stelling volgt onder gebruikmaking van Stelling 8.5.5 en Lemma 8.5.7 uit dit artikel.

3. Zij $\mathcal{K}$ een Kripke-structuur met n toestanden en m transities. Er bestaat een $O(m \cdot n)$ algorithme voor het beslissen van de zogeheten stotter-equivalentie op $\mathcal{K}$. Aangezien $m \leqslant n^2$, betekent dit dat het vermoeden van Browne, Clarke & Grümberg dat hun $O(n^5)$ algorithme verbeterd kon worden, juist was.

Zie: M.C. Browne, E.M. Clarke & O. Grümberg, *Characterizing finite Kripke structures in propositional temporal logic.* Theoretical Computer Science 59(1,2), 115-131, 1988.
J.F. Groote & F.W. Vaandrager, *An efficient algorithm for branching bisimulation and stuttering equivalence.* (In voorbereiding)

4. Een semantische beschrijving van de programmeertaal POOL-T die gebaseerd is op *handshaking* communicatie tussen objecten is niet verenigbaar met een implementatie van deze communicatie waarbij gebruik wordt gemaakt van wachtrijen voor boodschappen. Dit probleem wordt veroorzaakt door het *select statement* in POOL-T en het is daarom een goede zaak dat deze constructie niet meer deel uitmaakt van de meer recente versies van de taal.

Zie: het vijfde artikel in dit proefschrift.

5. Het is een opmerkelijk feit dat, bij aanname van maximaal parallellisme, pomset-equivalentie niet en ST-bisimulatie-equivalentie wel real-time consistent is.

   Zie: R.J. van Glabbeek & F.W. Vaandrager, *Petri net models for algebraic theories of concurrency*. In: Proceedings PARLE conference, Eindhoven, Vol. II (Parallel Languages), LNCS 259, Springer-Verlag, 224-242, 1987.

6. Indien men in de definitie van Milner's *observation equivalence*, een zwakke bisimulatie definieert als een relatie tussen paden in plaats van tussen toestanden, en indien men verder naast de gebruikelijke 'voorwaartse' conditie ook de conditie toevoegt dat indien twee paden gerelateerd zijn en men langs het ene pad een stap achterwaarts kan doen dit langs het andere pad geïmiteerd moet kunnen worden, dan valt de resulterende equivalentie samen met de *branching bisimulation equivalence* van Van Glabbeek & Weijland.

   Zie: R. De Nicola, U. Montanari & F.W. Vaandrager, *Back and forth bisimulations*. (In voorbereiding)

7. Er bestaat een eenvoudige constructie die aan een procesgraaf g een procesgraaf $tr(g)$ toevoegt waarvan de knopen gelabeld zijn op zodanige wijze dat (1) wanneer g en $tr(g)$ worden uitgerold tot bomen, deze bomen (afgezien van knooplabels) isomorf zijn, en verder (2) twee grafen g_1 en g_2 vertakkend bisimulatie-equivalent zijn precies dan wanneer $tr(g_1)$ en $tr(g_2)$ dezelfde CTL*-formules zonder *nexttime*-operator waar maken in een interpretatie waarbij gequantificeerd wordt over alle paden en niet alleen de maximale.

   Zie: R. De Nicola & F.W. Vaandrager, *Three logics for branching bisimulation*. (In voorbereiding)

8. In de geometrische opbouw van het fresco *De dood van Adam* van Piero della Francesca is het linkeroog van Adam het belangrijkste punt. Vermoedelijk met opzet heeft de schilder dit punt een fractie ter linkerzijde van de gulden snedelijn geplaatst.

9. Ook al neemt men het matige peil van veel wetenschappelijke presentaties in aanmerking, dan nog bestaat er een opvallend verschil tussen de staande ovaties die musici en toneelspelers vrijwel altijd ten deel vallen en het lauwe handgeklap waarmee wetenschappers het in het algemeen moeten doen.

10. In tegenstelling met de kroketten in het restaurant,
    zijn die in de kantine juist aan de grote kant.

    Vgl.: C.B. Vaandrager, *Vaandrager's totale poëzie*, De Bezige Bij, Amsterdam, 1981.