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 article 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. Finally it is shown that integers and booleans in POOL can be implemented in various ways.

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 of 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 a lot of concurrent systems are specified in terms of these languages. In this article 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

Partial support received from the European Community under ESPRIT project no. 432, An Integrated Formal Approach to Industrial Software Development (METEOR).

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 does communicate with the external world. Issues like fairness and the presence of interrupt mechanism are also relevant in the choice of the optimal semantics. The axioms we will give correspond to bisimulation semantics. In this semantics relatively few processes are identified, and therefore all the results we will prove are also valid in a large number of other 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 definition the two descriptions are different in bisimulation semantics. Although we think that the two views of a POOL system are equivalent in failure semantics, we have not proved this.

A similar question is dealt with in Section 6: on the conceptual level each integer and boolean in POOL is an object which has a data part and a process part. In an implementation this is of course not the case. Instead, an implementation will contain some special circuits for arithmetical and logical operations. We prove that these views of the system have a common abstraction.

In Section 5 we discuss a trace semantics of the language POOL. A lot of 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 7 contains a number of conclusions.

At the end of this introduction we give the definition of the *renaming operators* and *chaining operators*. These operators will pay an important role in the rest of the paper, but are not described in the introduction of this volume.

1.1. Renaming operators (RN)

For every function $f: A_{\tau\delta} \rightarrow A_{\tau\delta}$ with the property that $f(\delta)=\delta$ and $f(\tau)=\tau$, we define an operator $\rho_f: P \rightarrow P$. Axioms for ρ_f are given in Table 1.1. (Here $a \in A_{\tau\delta}$.)

$\rho_f(a) = f(a)$	RN1
$\rho_f(x+y) = \rho_f(x)+\rho_f(y)$	RN2
$\rho_f(xy) = \rho_f(x)\cdot\rho_f(y)$	RN3

TABLE 1.1

For $t \in A_{\tau\delta}$, and $H \subseteq A$ we define the function $r_{t,H} : A_{\tau\delta} \rightarrow A_{\tau\delta}$ by:

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

We use t_H as a notation for the operator $\rho_{r_{t,H}}$. The operators ∂_H and δ_H are considered to be equal.

[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 statement'. It seems that this construct does not lead to semantical problems like the select statement.

1.2. Chaining operators (CH)

A basic situation we will encounter is one in which there are processes which input and output values in a domain D. Often we want to 'chain' two processes in such a way that the output of the first one becomes the input of the second. In order to describe this, we define *chaining* operators $\ggg$ and $\gg$. In the process $x \ggg y$ the output of process x serves as input of process y. Operator $\gg$ is identical to operator $\ggg$, but hides in addition the communications that take place at the internal communication port. The reason for introducing two operators is a technical one: the operator $\gg$ (in which we are interested most) often leads to *unguarded recursion*. We will define the chaining operators in terms of the operators of $\mathrm{ACP}_\tau + \mathrm{RN}$. In this way we obtain a finite axiomatisation of the operator (if the alphabet of atomic actions is finite).

First we make a number of assumptions about the alphabet A and the communication function γ. Let for $d \in D$, $\downarrow d$ be the action of reading d, and $\uparrow d$ be the action of sending d. Let A' be the following set

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

We assume $A' \subseteq A$ and furthermore that for $a,b \in A - A'$: $\gamma(a,b) \notin A'$. On A' communication is defined by

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

and all other communications give δ. Define $H_{CH} = \{s(d), r(d) \mid d \in D\}$. The renaming functions f and g are defined by

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

and $f(a) = g(a) = a$ 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

$$\boxed{x \ggg y = \partial_{H_{CH}}(\rho_f(x) \| \rho_g(y)) \quad \text{CHC}}$$

Figure 1.1 contains a graphical display of the construction.

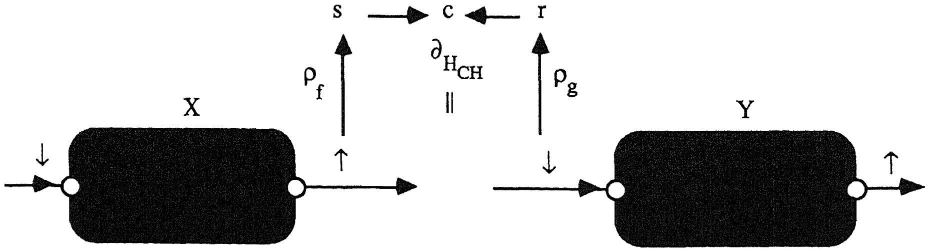

FIGURE 1.1

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

$$\boxed{x \gg y = \tau_{I_{CH}}(x \ggg y) \quad \text{CHA}}$$

One of the properties of the chaining operators we use most is that they are associative (under some very weak assumptions). The conditional axioms below state that the chaining operators are associative if the actions of H_{CH} do not occur in the alphabets of the components. In [26] it is shown that, if we add some natural axioms about alphabets to the axiom system, these two axioms become derivable.

$$\boxed{\begin{array}{ll} \dfrac{\alpha(x) \cap H_{CH} = \alpha(y) \cap H_{CH} = \alpha(z) \cap H_{CH} = \varnothing}{(x \ggg y) \ggg z = x \ggg (y \ggg z)} & \text{CC1} \\ \dfrac{\alpha(x) \cap H_{CH} = \alpha(y) \cap H_{CH} = \alpha(z) \cap H_{CH} = \varnothing}{(x \gg y) \gg z = x \gg (y \gg z)} & \text{CC2} \end{array}}$$

The module consisting of axioms CHC, CHA, CC1 and CC2 is denoted CH.

1.2.1. Notation. For the term

$$x \ggg (\sum_{d_1 \in D_1} \downarrow d_1 \cdots \sum_{d_n \in D_n} \downarrow d_n \cdot y_{d_1, \ldots, d_n})$$

(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. A SIMPLE SEQUENTIAL PROGRAMMING LANGUAGE

The following definition of a simple programming language is adopted from [9]. In the definition a choice between different versions of a rule is indicated by a vertical bar ('|').

2.1. DEFINITION (syntax of *Iexp*, *Bexp* and *Stat*). Let *Ivar*, with typical elements $v, w, u, \ldots$, and *Icon*, with typical elements $\alpha, \ldots$, be given finite sets of symbols.

a. The class *Iexp*, with typical elements $s, t, \ldots$, 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*, 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*, 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 article.

2.3. Semantics of the toy language. We will now relate to each element of the language defined in Section 2.1, a recursive specification in the signature of the operators $\cdot$, $+$ and $\ggg$. The first thing we have to do is to give the parameters of ACP: the alphabet A and the communication function. The value domain D of the chaining operator is

$$D = (Ivar \rightarrow Icon) \cup Icon \cup \{\textbf{true}, \textbf{false}\}.$$

Here $Ivar \rightarrow Icon$ is the set of all functions from variables to their values. The set A of atomic actions is the set A' as described in Section 1.2. Communication on A' is also as described in Section 1.2.

2.4. Notation. Let $\sigma \in Ivar \rightarrow Icon$, $v \in Ivar$ and $\alpha \in Icon$. We use the well-known notation $\sigma\{\alpha/v\}$ to denote the element of $Ivar \rightarrow Icon$ that satisfies for each $v' \in Ivar$

$$\sigma\{\alpha/v\}(v') = \begin{cases} \alpha & \text{if } v'=v \\ \sigma(v') & \text{otherwise} \end{cases}$$

2.5. Below we give a number of process algebra equations. The variables in these equations are elements of the toy language with semantical brackets ('$\llbracket$' and '$\rrbracket$') 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 $\sigma \in Ivar \rightarrow Icon$, is the solution of this system, with

$$\llbracket w \rrbracket^{\sigma}$$

taken as root variable. Throughout the rest of this section $\alpha,\alpha' \in Icon$, $\beta,\beta' \in \{\mathbf{true},\mathbf{false}\}$ and $\sigma,\sigma' \in Ivar \rightarrow Icon$.

2.6. The class *Iexp*

$$\llbracket v \rrbracket^{\sigma} = \uparrow\sigma(v)$$

$$\llbracket \alpha \rrbracket^{\sigma} = \uparrow\alpha$$

$$\llbracket s_1 + s_2 \rrbracket^{\sigma} = \llbracket s_1 \rrbracket^{\sigma} \cdot \llbracket s_2 \rrbracket^{\sigma} \ggg_{\alpha,\alpha'} \uparrow sum(\alpha,\alpha')$$

$$\llbracket \mathbf{if}\, b\, \mathbf{then}\; s_1\; \mathbf{else}\; s_2\; \mathbf{fi} \rrbracket^{\sigma} = \llbracket b \rrbracket^{\sigma} \ggg (\downarrow\mathbf{true}\cdot\llbracket s_1 \rrbracket^{\sigma} + \downarrow\mathbf{false}\cdot\llbracket s_2 \rrbracket^{\sigma})$$

2.7. The class *Bexp*

$$\llbracket \mathbf{true} \rrbracket^{\sigma} = \uparrow\mathbf{true}$$

$$\llbracket \mathbf{false} \rrbracket^{\sigma} = \uparrow\mathbf{false}$$

$$\llbracket s_1 = s_2 \rrbracket^{\sigma} = \llbracket s_1 \rrbracket^{\sigma} \cdot \llbracket s_2 \rrbracket^{\sigma} \ggg_{\alpha,\alpha'} \llbracket = \rrbracket_{\alpha,\alpha'}$$

$$\llbracket = \rrbracket_{\alpha,\alpha'} = \begin{cases} \uparrow\mathbf{true} & \text{if } \alpha=\alpha' \\ \uparrow\mathbf{false} & \text{otherwise} \end{cases}$$

$$\llbracket \neg b \rrbracket^{\sigma} = \llbracket b \rrbracket^{\sigma} \ggg (\downarrow\mathbf{true}\cdot\uparrow\mathbf{false} + \downarrow\mathbf{false}\cdot\uparrow\mathbf{true})$$

$$\llbracket b_1 \supset b_2 \rrbracket^{\sigma} = \llbracket b_1 \rrbracket^{\sigma} \ggg (\downarrow\mathbf{true}\cdot\llbracket b_2 \rrbracket^{\sigma} + \downarrow\mathbf{false}\cdot\uparrow\mathbf{true})$$

2.8. The class *Stat*

$$\llbracket v := s \rrbracket^{\sigma} = \llbracket s \rrbracket^{\sigma} \ggg_{\alpha} \uparrow\sigma\{\alpha/v\}$$

$$\llbracket S_1 ; S_2 \rrbracket^{\sigma} = \llbracket S_1 \rrbracket^{\sigma} \ggg_{\sigma'} \llbracket S_2 \rrbracket^{\sigma'}$$

$$\llbracket \mathbf{if}\, b\, \mathbf{then}\; S_1\; \mathbf{else}\; S_2\; \mathbf{fi} \rrbracket^{\sigma} = \llbracket b \rrbracket^{\sigma} \ggg (\downarrow\mathbf{true}\cdot\llbracket S_1 \rrbracket^{\sigma} + \downarrow\mathbf{false}\cdot\llbracket S_2 \rrbracket^{\sigma})$$

$$\llbracket \mathbf{while}\, b\, \mathbf{do}\, S\, \mathbf{od} \rrbracket^{\sigma} = \llbracket b \rrbracket^{\sigma} \ggg$$

$$(\downarrow\mathbf{true}\cdot(\llbracket S \rrbracket^{\sigma} \ggg_{\sigma'} \llbracket \mathbf{while}\, b\, \mathbf{do}\, S\, \mathbf{od} \rrbracket^{\sigma'}) + \downarrow\mathbf{false}\cdot\uparrow\sigma)$$

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. Define a relation $\xrightarrow{u}$ between elements of Ξ by

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

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 $m : \Xi \rightarrow \mathbb{N}$ 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 axiom CC1 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 given guarded specification $E = \{X = t_X \mid X \in \Xi\}$, rewrites a given term t in the signature of $\mathrm{ACP}_\tau + \mathrm{RN} + \mathrm{CH}$ with variables in Ξ, into a term of the form $\Sigma a_i \cdot t_i + \Sigma b_j$. 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 $\Sigma a_i \cdot t_i + \Sigma b_j$ 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.

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.

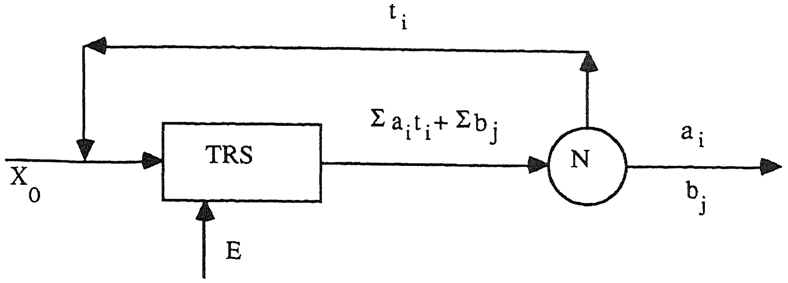

FIGURE 2.1

2.13. THEOREM. *Using the axioms of ACP+RN+CH+RDP+PR+AIP⁻ we can prove:*

(1) $\forall s \in Iexp\ \forall \sigma \in (Ivar \rightarrow Icon)\ \exists d_1, ..., d_n \in D\ \exists \alpha \in Icon:$

$$[\![s]\!]^\sigma = c(d_1) \cdots c(d_n) \cdot \uparrow \alpha$$

(2) $\forall b \in Bexp\ \forall \sigma \in (Ivar \rightarrow Icon)\ \exists d_1, ..., d_n \in D\ \exists \beta \in \{\mathbf{true}, \mathbf{false}\}:$

$$[\![b]\!]^\sigma = c(d_1) \cdots c(d_n) \cdot \uparrow \beta$$

(3) $\forall S \in Stat\ \forall \sigma \in (Ivar \rightarrow Icon):$
$(\exists d_1, ..., d_n \in D\ \exists \sigma' \in (Ivar \rightarrow Icon): [\![S]\!]^\sigma = c(d_1) \cdots c(d_n) \cdot \uparrow \sigma')$
$\vee (\exists d_1, d_2, ... : [\![S]\!]^\sigma = c(d_1) \cdot c(d_2) \cdots)$

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$_\tau$ (for example the term model discussed in [16]) 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 $\langle w\rangle^{\sigma}$ for which we have equations

$$\langle w\rangle^{\sigma} = \tau_I([\![w]\!]^{\sigma}).$$

Here I is a set of 'unimportant' actions which we want to hide. Formal variable $\langle w\rangle^{\sigma}$ 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. 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'. 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 process algebra specification to each element of the language POOL-⊥. The subscript C indicates that the resulting specification is in the signature of concrete process algebra, as opposed to the specification we will present in Section 3.11, which contains 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.

[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.

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 $X_1, ..., X_n \in V - \{S\}$.

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 use $\sigma.0$ as a notation for sequence 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. $\sigma.n \in nodes(t) \Rightarrow \sigma \in nodes(t)$
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 the whole system up.

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-$\perp$-CF. Section 3.5 contains some remarks concerning this definition, and the relation with POOL-T and POOL-$\perp$.

3.4. DEFINITION (POOL-$\perp$-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-$\perp$ 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

$$Bool = \{\mathbf{true},\mathbf{false}\}.$$

Now the context-free grammar *G*, which defines POOL-$\perp$-CF, is

$$G = (T,N,U,P)$$

where

$$T = Id \cup Int \cup Bool \cup \{\mathbf{root},\mathbf{unit},\mathbf{class},\mathbf{var},\mathbf{body},\mathbf{end},\mathbf{method},\mathbf{routine},\mathbf{local},\mathbf{in},\mathbf{nil}$$
$$\mathbf{return},\mathbf{post},\mathbf{if},\mathbf{then},\mathbf{else},\mathbf{fi},\mathbf{do},\mathbf{od},\mathbf{sel},\mathbf{les},\mathbf{or},\mathbf{answer},\mathbf{self},\mathbf{new}, ;, \cdot, \leftarrow, !, ,, : \}$$

$$N = \{U,RU,CDL,CD,MDL,MD,RDL,RD,PD,VDL,VD,SS,S,SE,$$
$$GCL,GC,AN,MIL,E,CO,SN,RC,MC,EL,CI,MI,RI,VI\}$$

P : see Table 3.1

In Table 3.1, optional syntactical elements are enclosed in square brackets ('[' and ']').

Syntax of POOL-$\perp$

No	*Description*	*Syntactic Rule*
1	unit	$U \rightarrow RU$
2	root unit	$RU \rightarrow \textbf{root unit}\ CDL$
3	class definition list	$CDL \rightarrow CD\,[\,,CDL\,]$
4	class definition	$CD \rightarrow \textbf{class}\ CI\,[\,\textbf{var}\,VDL\,][\,RDL\,][\,MDL\,]$ $\textbf{body}\ SS\ \textbf{end}\ CI$
5	method definition list	$MDL \rightarrow MD\,[\,MDL\,]$
6	method definition	$MD \rightarrow \textbf{method}\ MI\ PD\ \textbf{end}\ MI$
7	routine definition list	$RDL \rightarrow RD\,[\,RDL\,]$
8	routine definition	$RD \rightarrow \textbf{routine}\ RI\ PD\ \textbf{end}\ RI$
9	procedure denotation	$PD \rightarrow (\,[\,VDL\,]\,)\,CI : [\,\textbf{local}\ VDL\ \textbf{in}\,]\,[\,SS\,]$ $\textbf{return}\ E\,[\,\textbf{post}\ SS\,]$
10	variable declaration list	$VDL \rightarrow VD\,[\,,VDL\,]$
11	variable declaration	$VD \rightarrow VI : CI$
12	statement sequence	$SS \rightarrow S\,[\,;SS\,]$
13	statement	$S \rightarrow\ VI \leftarrow E$ $\mid\ AN$ $\mid\ \textbf{if}\ E\ \textbf{then}\ SS\,[\,\textbf{else}\ SS\,]\,\textbf{fi}$ $\mid\ \textbf{do}\ E\ \textbf{then}\ SS\ \textbf{od}$ $\mid\ SE$ $\mid\ SN$ $\mid\ MC$ $\mid\ RC$
14	select statement	$SE \rightarrow \textbf{sel}\ GCL\ \textbf{les}$
15	guarded command list	$GCL \rightarrow GC\,[\,\textbf{or}\ GCL\,]$
16	guarded command	$GC \rightarrow\ E\,[\,AN\,]\,\textbf{then}\ SS$

17	answer statement	$AN \rightarrow \mathbf{answer}(MIL)$
18	method identifier list	$MIL \rightarrow MI[,MIL]$
19	expression	$E \rightarrow VI$ $\mid$ **self** $\mid CO$ $\mid$ **new** $\mid SN$ $\mid MC$ $\mid RC$ $\mid$ **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 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 same as 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 $VI \leftarrow E$ 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 rest 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 $\cdot$, 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
$B \to 0$	$v(B) = 0$
$B \to 1$	$v(B) = 2^{s(B)}$
$L \to B$	$v(L) = v(B); s(B) = s(L); l(L) = 1$
$L_1 \to L_2 B$	$v(L_1) = v(L_2) + v(B); s(B) = s(L_1);$
	$s(L_2) = s(L_1) + 1; l(L_1) = l(L_2) + 1$
$N \to L$	$v(N) = v(L); s(L) = 0$
$N \to L_1 \cdot L_2$	$v(N) = v(L_1) + v(L_2); s(L_1) = 0;$
	$s(L_2) = -l(L_2)$

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.

Furthermore 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) $D = \langle \Omega,\Phi \rangle$, 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 $m \geq 0$ 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

   $$A(p) = \{\langle \alpha,j \rangle \mid 0 \leq j \leq n,\ \alpha \in A(X_j)\}.$$

   Intuitively $\langle \alpha,j \rangle$ 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

   $$INT(p) = \{\langle \alpha,j \rangle \mid (j=0 \wedge \alpha \in S(X_0)) \vee (1 \leq j \leq n \wedge \alpha \in I(X_j))\}$$
   $$EXT(p) = \{\langle \alpha,j \rangle \mid (j=0 \wedge \alpha \in I(X_0)) \vee (1 \leq j \leq n \wedge \alpha \in S(X_j))\}$$

   A *semantic rule* for p is a string of the form

   $$\langle \alpha,j \rangle = f(\langle \alpha_1,k_1 \rangle, \ldots, \langle \alpha_m,k_m \rangle) \qquad (*)$$

   with $\langle \alpha,j \rangle \in INT(p)$, $m \geq 0$, $\langle \alpha_i,k_i \rangle \in EXT(p)$ for $1 \leq i \leq m$, 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 $\langle \alpha,j \rangle \in INT(p)$, $R(p)$ 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

$$A(t) = \{\langle\alpha,\sigma\rangle \mid \sigma\in nodes(t),\ \alpha\in A(label(t)(\sigma))\}$$

(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 $\langle\alpha,\sigma\rangle\in A(t)$, $val(\alpha,\sigma)\in V(\alpha)$.
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

$$\langle\alpha,\sigma.j\rangle = f(\langle\alpha_1,\sigma.k_1\rangle, ..., \langle\alpha_m,\sigma.k_m\rangle) \qquad (**)$$

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 $\langle\alpha,\sigma\rangle$ there is exactly one semantic instruction in $R(t)$ of the form $\langle\alpha,\sigma\rangle = ...$ 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 as in Figure 3.1.

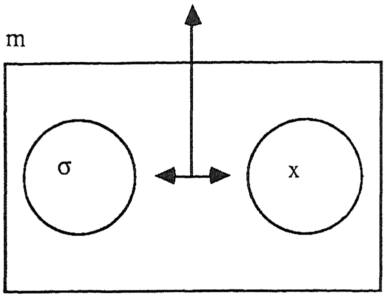

FIGURE 3.1

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

$$act: A \times M \times \Sigma \rightarrow A_{\tau\delta} \quad \text{(action function)}$$

$$eff: A \times M \times \Sigma \rightarrow \Sigma \quad \text{(effect function)}$$

Now we extend the signature with operators

$$\lambda_\sigma^m : P \rightarrow P \quad (\text{for } m \in M,\ \sigma \in \Sigma)$$

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 [26]).

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:

$$SObj = Int \cup Bool \cup \{\mathbf{nil}\} \cup \{\mathbf{input}, \mathbf{output}\}.$$

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 a *mc*-frame looks as follows

$$(\alpha, mc, m(\alpha_1, ..., \alpha_n), \beta) \tag{3.8.2.1}$$

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:

$$\mathcal{M} = \{m(\alpha_1, ..., \alpha_n) \mid m \in LId, 0 \leqslant n \leqslant N_2, \alpha_1, ..., \alpha_n \in Obj\} \qquad (3.8.2.3)$$

and the set $\mathcal{F}$ of frames is:

$$\mathcal{F} = \{(\alpha, mc, d, \beta) \mid \alpha, \beta \in Obj, d \in \mathcal{M}\} \cup \{(\beta, an, \gamma, \alpha) \mid \alpha, \beta, \gamma \in Obj\}. \qquad (3.8.2.4)$$

For each frame $f \in \mathcal{F}$, we have atomic actions $read(f)$, $send(f)$ and $comm(f)$. The communication function on these actions is given by

$$read(f) \mid send(f) = comm(f) \quad \text{for } f \in \mathcal{F}. \qquad (3.8.2.5)$$

The set J of forbidden actions that will be encapsulated is

$$J = \{read(f), send(f) \mid f \in \mathcal{F}\}. \qquad (3.8.2.6)$$

3.8.3. Renamings. A POOL object is fully determined by its class and by 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 on the level of a class this name is not known, the process related to a class contains 'unfinished' *read* and *send* actions: actions $rd(uf)$ and $sn(uf)$, where uf is an unfinished frame in which the field that gives the identity of the object is absent. Actions of the form $rd(uf)$ and $sn(uf)$ do not communicate.

For each $\alpha \in Obj$ we define a renaming function f_α by:

$$f_\alpha(sn(\beta, mc, m(\alpha_1, ..., \alpha_n))) = send(\beta, mc, m(\alpha_1, ..., \alpha_n), \alpha) \qquad (3.8.3.1)$$

$$f_\alpha(rd(mc, m(\alpha_1, ..., \alpha_n), \beta)) = read(\alpha, mc, m(\alpha_1, ..., \alpha_n), \beta) \qquad (3.8.3.2)$$

$$f_\alpha(sn(\beta, an, \gamma)) = send(\beta, an, \gamma, \alpha) \qquad (3.8.3.3)$$

$$f_\alpha(rd(an, \beta, \gamma)) = read(\alpha, an, \beta, \gamma) \qquad (3.8.3.4)$$

If an object executes a **self** expression, the corresponding process on class level contains a non-deterministic choice between 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.

$$f_\alpha(eqs(\beta)) = \begin{cases} skip & \text{if } \beta = \alpha \\ \delta & \text{otherwise} \end{cases} \qquad (3.8.3.5)$$

If an object 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:

$$g_\alpha(\uparrow\beta) = sn(\alpha, an, \beta). \qquad (3.8.3.6)$$

3.8.4. Process creation. For $d \in \mathcal{N} \times AObj$ 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 $\mathcal{N}$ (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-⊥. 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

$$create(d) \mid create^*(d) = \overline{create}(d). \tag{3.8.4.1}$$

Create actions are not involved in any other proper communication. Let

$$K = \{create(d), create^*(d) \mid d \in \mathcal{N} \times AObj\}. \tag{3.8.4.2}$$

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-⊥. 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^v_α 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 $ass(v,\beta)$:

$$\lambda^v_\alpha(ass(v,\beta)\cdot x) = skip\cdot\lambda^v_\beta(x). \tag{3.8.5.1}$$

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:

$$\lambda^v_\alpha(eqv(v,\beta)\cdot x) = \begin{cases} skip\cdot\lambda^v_\alpha(x) & \text{if } \alpha=\beta \\ \delta & \text{otherwise} \end{cases} \tag{3.8.5.2}$$

Notice that in the case of nested λ^v_α operators, actions $ass(v,\beta)$ and $eqv(v,\beta)$ interact with the innermost λ^v_α operator. This is relevant for nested method calls, etc.

The initial object, which starts the system up, 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

$$\lambda_n^{counter}(\overline{create}(X,\alpha)\cdot x) = \begin{cases} skip \cdot \lambda_{n+1}^{counter}(x) & \text{if } \alpha = \hat{n} \wedge n < N_1 \\ error \cdot \lambda_{n+1}^{counter}(x) & \text{if } n = N_1 \\ \delta & \text{otherwise} \end{cases} \qquad (3.8.5.3)$$

3.8.6. Formal variables. The set Ξ of formal variables of the process algebra specification related to POOL-$\perp$ consists of the elements of $\mathfrak{N}$, possibly sub- and superscripted with elements of *LId* and *Obj**. Formally we have:

$$\Xi = \mathfrak{N} \cup \mathfrak{N} \times LId \cup \mathfrak{N} \times (Obj^*) \cup \mathfrak{N} \times LId \times (Obj^*). \qquad (3.8.6.1)$$

We define *node* : $\Xi \rightarrow \mathfrak{N}$ 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-$\perp$ 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-$\perp$. 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.

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.

Name attr.	*i/s*	*Description*	*Attribute value*	*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	$LId\rightarrow\mathfrak{N}\times\mathbb{N}$	{RDL,CD}
md	s	Method declarations	$LId\rightarrow\mathfrak{N}\times\mathbb{N}$	{MDL,CD}
cd	s	Class declarations	$UId\rightarrow\mathfrak{N}$	{CDL}
rdc	s	Routine decl. of a CDL	$UId\times LId\rightarrow\mathfrak{N}\times\mathbb{N}$	{CDL}
mdc	s	Method decl. of a CDL	$UId\times LId\rightarrow\mathfrak{N}\times\mathbb{N}$	{CDL}
cdf	i	Class definitions	$UId\rightarrow\mathfrak{N}$	N-{U,RU}
rdf	i	Routine definitions	$UId\times LId\rightarrow\mathfrak{N}\times\mathbb{N}$	N-{U,RU}
mdf	i	Method definitions	$UId\times LId\rightarrow\mathfrak{N}\times\mathbb{N}$	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

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

      $$CD \rightarrow \mathbf{class}\ CI_1\ [\mathbf{var} VDL\ ][\ RDL\ ][\ MDL\ ]\mathbf{body}\ SS\ \mathbf{end}\ CI_2$$

      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*. The value of the attribute *spec* belonging to the root of the derivation tree (which has label U) is the specification we relate to the parse tree. 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 $S \subseteq \{1, \ldots, n\}$ 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** $peq(X) = \{E_1, E_2, \ldots\}$**, we only write down the equations** $E_1, E_2, \ldots$**!!!** 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)

$$l(EL_0) = 1[+l(EL_1)]$$

$$\llbracket EL_0 \rrbracket = \llbracket E \rrbracket [\cdot \llbracket EL_1 \rrbracket]$$

○ We state again that the equation for $\llbracket EL_0 \rrbracket$ is not to be considered as a semantic rule defining attribute $\llbracket . \rrbracket$, 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

$$\llbracket RC \rrbracket = skip \cdot X_\epsilon$$

○ 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

$$\llbracket RC \rrbracket = \llbracket EL \rrbracket \gg_{\alpha_1, \ldots, \alpha_n} X_{\alpha_1, \ldots, \alpha_n}$$

○ First the expressions of the parameter list are evaluated. Thereafter the routine call is executed, with the actual parameters instantiated. 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

$$[\![MC]\!] = skip \cdot X_{\epsilon}$$

$MC \rightarrow MI(EL)$ (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

$$[\![SN]\!] = [\![E]\!] \ggg_{\alpha} [\![SN]\!]_{\alpha}$$

$$[\![SN]\!]_{\mathbf{nil}} = error$$

and for $\alpha{\neq}\mathbf{nil}$:

$$[\![SN]\!]_\alpha = [\![EL]\!] \ggg_{\alpha_1,\ \ldots,\alpha_n} sn(\alpha,mc,m(\alpha_1,\ \ldots,\alpha_n))\cdot \sum_{\beta\in Obj} rd(an,\beta,\alpha)\cdot\uparrow\beta$$

○ 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)

$$[\![E]\!] = \sum_{\alpha\in Obj} eqv(id(VI),\alpha)\cdot\uparrow\alpha$$

○ Cf. equation 3.8.5.2.

$E{\rightarrow}\mathbf{self}$ (19.2)

$$[\![E]\!] = \sum_{\alpha\in Obj} eqs(\alpha)\cdot\uparrow\alpha$$

○ Cf. equation 3.8.3.5.

$E{\rightarrow}CO$ (19.3)

$$[\![E]\!] = [\![CO]\!]$$

$E{\rightarrow}\mathbf{new}$ (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.

$E{\rightarrow}SN$ (19.5)

$$[\![E]\!] = [\![SN]\!]$$

$E{\rightarrow}MC$ (19.6)

$$[\![E]\!] = [\![MC]\!]$$

$E{\rightarrow}RC$ (19.7)

$$[\![E]\!] = [\![RC]\!]$$

$E \rightarrow$ **nil** (19.8)

$$\llbracket E \rrbracket = \uparrow \mathbf{nil}$$

$MIL_0 \rightarrow MI\,[\,, MIL_1\,]$ (18)

Let

$$id(MI) = m$$

$$mdf(MIL_0)(class(MIL_0), m) = (X\ n)$$

then

$$mis(MIL_0) = \{m\}\,[\,\cup\, mis(MIL_1)\,]$$

$$\llbracket MIL_0 \rrbracket_m = \sum_{\alpha_1, \ldots, \alpha_n, \alpha \in Obj} rd(mc, m(\alpha_1, \ldots, \alpha_n), \alpha) \cdot \rho_{g_\alpha}(X_{\alpha_1, \ldots, \alpha_n})$$

$$\llbracket MIL_0 \rrbracket_{\overline{m}} = \frac{\llbracket MIL_1 \rrbracket_{\overline{m}}}{\delta} \quad \text{if } \overline{m} \neq m$$

○ For the m which occur in the method identifier list, $\llbracket MIL_0 \rrbracket_m$ 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, $\llbracket MIL_0 \rrbracket_m = \delta$.

$AN \rightarrow$ **answer** (MIL) (17)

$$mis(AN) = mis(MIL)$$

$$\llbracket AN \rrbracket_m = \llbracket MIL \rrbracket_m$$

$$\llbracket AN \rrbracket = \sum_{m \in LId} \llbracket MIL \rrbracket_m$$

○ The variables $\llbracket AN \rrbracket_m$ 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.

$GC \rightarrow E$ **then** SS (16.1)

$$mis(GC) = \emptyset$$

$$\llbracket GC \rrbracket = \llbracket E \rrbracket$$

$$\llbracket GC \rrbracket_\epsilon = skip \cdot \llbracket SS \rrbracket$$

○ The prefix *skip* in the equation for variable $\llbracket GC \rrbracket_\epsilon$ is needed because we want to give a different semantics to the following two select statements:

```
sel
        true answer(m₁) then x←1 or
        true answer(m₂) then x←2
les
```

and

```
sel
        true answer(m₁) then x←1 or
        true then answer(m₂) ; x←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

$$misl(GCL) = (M)$$

$$[\![GCL]\!] = [\![GC]\!]$$

$$[\![GCL]\!]^{\alpha}_{\epsilon} = \begin{cases} [\![GC]\!]_{\epsilon} & \text{if } \alpha = \mathbf{true} \wedge M = \emptyset \\ \delta & \text{otherwise} \end{cases}$$

$$[\![GCL]\!]^{\alpha}_{m} = \begin{cases} [\![GC]\!]_{\epsilon} & \text{if } \alpha = \mathbf{true} \wedge M = \emptyset \\ [\![GC]\!]_{m} & \text{if } \alpha = \mathbf{true} \wedge m \in M \\ \delta & \text{otherwise} \end{cases}$$

○ 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

$$misl(GCL_0) = (M_0, M_1, ..., M_n)$$

$$[\![GCL_0]\!] = [\![GC]\!] \cdot [\![GCL_1]\!]$$

$$[\![GCL_0]\!]_\epsilon^{\alpha_0, ..., \alpha_n} = \begin{cases} [\![GC]\!]_\epsilon & \text{if } \alpha_0 = \mathbf{true} \wedge M_0 = \varnothing \\ [\![GCL_1]\!]_\epsilon^{\alpha_1, ..., \alpha_n} & \text{otherwise} \end{cases}$$

$$[\![GCL_0]\!]_m^{\alpha_0, ..., \alpha_n} = \begin{cases} [\![GC]\!]_\epsilon & \text{if } \alpha_0 = \mathbf{true} \wedge M_0 = \varnothing \\ [\![GC]\!]_m & \text{if } \alpha_0 = \mathbf{true} \wedge m \in M_0 \\ [\![GCL_1]\!]_m^{\alpha_1, ..., \alpha_n} & \text{otherwise} \end{cases}$$

○ See remark about production 14.

$SE \rightarrow$ **sel** GCL **les** (14)

Let

$$misl(GCL) = (M_1, ..., M_n)$$

then

$$[\![SE]\!] = [\![GCL]\!] \ggg_{\alpha_1, ..., \alpha_n} [\![SE]\!]_{\alpha_1, ..., \alpha_n}$$

$$[\![SE]\!]_{\alpha_1, ..., \alpha_n} = error \text{ if } (\exists i : \alpha_i = \mathbf{nil}) \vee (\forall i : \alpha_i = \mathbf{false})$$

$$[\![SE]\!]_{\alpha_1, ..., \alpha_n} = \sum_{m \in LId \cup \{\epsilon\}} [\![GCL]\!]_m^{\alpha_1, ..., \alpha_n} \text{ otherwise}$$

○ 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.

$S \rightarrow VI \leftarrow E$ (13.1)

$$[\![S]\!] = [\![E]\!] \ggg_\alpha ass(id(VI), \alpha)$$

○ Cf. equation 3.8.5.1.

$S \rightarrow AN$ (13.2)

$$\llbracket S \rrbracket = \llbracket AN \rrbracket$$

$S \rightarrow$ **if** E **then** SS_1 [**else** SS_2] **fi** (13.3)

$$\llbracket S \rrbracket = \llbracket E \rrbracket \ggg_{\alpha} \llbracket S \rrbracket_{\alpha}$$

$$\llbracket S \rrbracket_{\alpha} = \begin{cases} \llbracket SS_1 \rrbracket & \text{if } \alpha = \textbf{true} \\ \dfrac{\llbracket SS_2 \rrbracket}{skip} & \text{if } \alpha = \textbf{false} \\ error & \text{otherwise} \end{cases}$$

$S \rightarrow$ **do** E **then** SS **od** (13.4)

$$\llbracket S \rrbracket = \llbracket E \rrbracket \ggg_{\alpha} \llbracket S \rrbracket_{\alpha}$$

$$\llbracket S \rrbracket_{\alpha} = \begin{cases} \llbracket SS \rrbracket \cdot \llbracket S \rrbracket & \text{if } \alpha = \textbf{true} \\ skip & \text{if } \alpha = \textbf{false} \\ error & \text{otherwise} \end{cases}$$

$S \rightarrow SE$ (13.5)

$$\llbracket S \rrbracket = \llbracket SE \rrbracket$$

$S \rightarrow SN$ (13.6)

$$\llbracket S \rrbracket = \llbracket SN \rrbracket \ggg (\sum_{\alpha \in Obj} \downarrow \alpha)$$

○ The send expression is evaluated and afterwards the result is discarded.

$S \rightarrow MC$ (13.7)

$$\llbracket S \rrbracket = \llbracket MC \rrbracket \ggg (\sum_{\alpha \in Obj} \downarrow \alpha)$$

$S \rightarrow RC$ (13.8)

$$\llbracket S \rrbracket = \llbracket RC \rrbracket \ggg (\sum_{\alpha \in Obj} \downarrow \alpha)$$

$SS_0 \rightarrow S\,[\,; SS_1\,]$ (12)

$$\llbracket SS_0 \rrbracket = \llbracket S \rrbracket [\cdot \llbracket SS_1 \rrbracket]$$

$VD \rightarrow VI : CI$ (11)

$$id(VD) = id(VI)$$

$VDL_0 \rightarrow VD\ [\ ,\ VDL_1\ ]$ (10)

$$vd(VDL_0) = (id(VD))[\star vd(VDL_1)]$$

○ The function $\star$ 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

$$vd(VDL_1) = (v_1, ..., v_n)$$

$$vd(VDL_2) = (w_1, ..., w_k)$$

($n = 0$ or $k = 0$ if there is no VDL_1 resp. VDL_2)
then

$$pd(PD) = ([\![PD]\!]\ n)$$

$$[\![PD]\!]_{\alpha_1, ..., \alpha_n} = \lambda_{\alpha_1}^{v_1} \circ \cdots \circ \lambda_{\alpha_n}^{v_n} \circ \lambda_{\mathbf{nil}}^{w_1} \circ \cdots \circ \lambda_{\mathbf{nil}}^{w_k}([\,[\![SS_1]\!] \cdot \,]\,[\![E]\!]\,[\, \cdot [\![SS_2]\!]\,])$$

○ Process $[\![PD]\!]_{\alpha_1, ..., \alpha_n}$ corresponds to execution of the procedure with parameters $\alpha_1, ..., \alpha_n$.

$RD \rightarrow \textbf{routine}\, RI_1\ PD\ \textbf{end}\, RI_2$ (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. 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

$$vd(VDL) = (v_1, ..., v_n)$$

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

$$cd(CDL)(Integer) = I$$

$$cd(CDL)(Boolean) = B$$

$$cd(CDL)(Read\_File) = R$$

$$cd(CDL)(Write\_File) = W$$

$$\mathcal{C} = \{cd(CDL)(C) \mid C \in UId\}$$

$$ACTIVE = \mathop{\|}_{\alpha \in AObj}\Big(\sum_{X \in \mathcal{C}} create^*(X,\alpha) \cdot \rho_{f_\alpha}(X)\Big)$$

$$STANDARD = \Big(\mathop{\|}_{\alpha \in Int} \rho_{f_\alpha}(I)\Big) \| \rho_{f_{\mathrm{true}}}(B) \| \rho_{f_{\mathrm{false}}}(B) \| \rho_{f_{\mathrm{input}}}(R) \| \rho_{f_{\mathrm{output}}}(W)$$

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:

$$E \rightarrow \mathbf{acp}\ t\ \mathbf{pca}$$

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 $sum(N_0,N_0)$ = **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 $\sum_{\alpha\in Int} eqs(\alpha)(\sum_{\beta\in Int} eqv(i,\beta)\cdot\uparrow sum(\alpha,\beta) + eqv(i,\mathbf{nil})\cdot error)$ pca
end add
etc., etc.
body do true then answer(add, sub, 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-$\perp$: 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 ↑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 $\sum_{\alpha \in Int} eqv(i, \alpha) \cdot output(\alpha) \cdot \uparrow\mathbf{output} + eqv(i, \mathbf{nil}) \cdot error$ pca
end write_int
method write_bool ( b : Boolean ) Write_File :
        return acp $\sum_{\beta \in Bool} eqv(b, \beta) \cdot output(\beta) \cdot \uparrow\mathbf{output} + eqv(b, \mathbf{nil}) \cdot error$ pca
end write_bool
body do true then answer(write_int, write_bool) od
end Write_File
```

3.10. THEOREM. *For each program $w \in$ POOL-$\perp$ the specification $SPEC_C(w)$ 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:

$$height(X_0) = \max(\{0\} \cup \{height(X_i) \mid 1 \leq i \leq n \text{ and } X_i \text{ has attribute } height\}) + 1$$

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. Therefore we define:

$$I = \{c(d) \mid d \in D\} \cup \{comm(f) \mid f \in \mathcal{F}\} \cup \{skip\} \qquad (3.11.1)$$

and introduce a new formal variable *ROOT*, which will be the root variable of the specification corresponding to a given POOL-$\perp$ unit. The equation for *ROOT* is:

$$ROOT = \tau_I([\![RU]\!]). \qquad (3.11.2)$$

ROOT gives the abstract behaviour of a POOL system executing a given unit. We call the corresponding function from POOL units to process algebra expressions $SPEC_A$.

3.12. Models. A lot of semantics (models, Σ-algebras) have been given of the signature that is used in this section. In this article we are only interested in models where the principles RDP and RSP are valid. For each of these models M, there exists a mapping INT_M that relates to every guarded specification E the unique solution of this system in the model. 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 presented in [28].

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 whose 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 $INT_M \circ SPEC_A$ and $INT_M \circ SPEC_{AQ}$ are identical.

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:

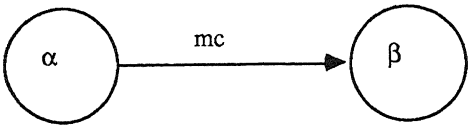

FIGURE 4.1

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 new version of Figure 4.1 becomes:

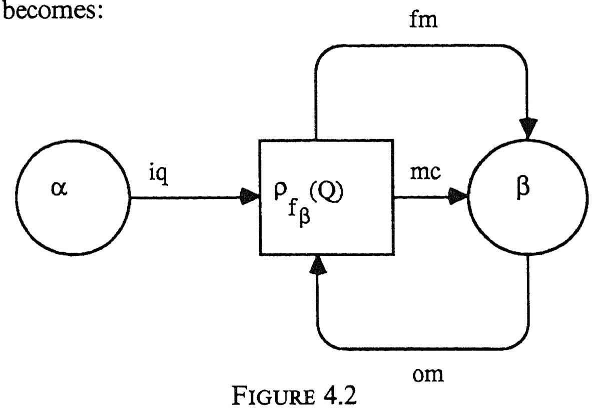

FIGURE 4.2

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:

$SN \rightarrow E\,!\,MI(\,)$ (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 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

$$misl(GCL) = (M_1, ..., M_n)$$

$$M_{\alpha_1, ..., \alpha_n} = \bigcup_{\{i \mid \alpha_i = \mathbf{true}\}} M_i$$

then

$$[\![SE]\!] = [\![GCL]\!] \ggg_{\alpha_1, ..., \alpha_n} [\![SE]\!]_{\alpha_1, ..., \alpha_n}$$

$$[\![SE]\!]_{\alpha_1, ..., \alpha_n} = error$$

if $(\exists i : \alpha_i = \mathbf{nil}) \vee (\forall i : \alpha_i = \mathbf{false})$,

$$[\![SE]\!]_{\alpha_1, ..., \alpha_n} = \sum_{m \in LId} rd(fm, (M_{\alpha_1, ..., \alpha_n}, m)) \cdot [\![GCL]\!]_m^{\alpha_1, ..., \alpha_n}$$

if $\forall i : \alpha_i = \mathbf{true} \Rightarrow M_i \neq \varnothing$, and

$$[\![SE]\!]_{\alpha_1, ..., \alpha_n} = \sum_{m \in LId \cup \{\epsilon\}} rd(fm, (M_{\alpha_1, ..., \alpha_n}, m)) \cdot [\![GCL]\!]_m^{\alpha_1, ..., \alpha_n}$$

otherwise.

○ $M_{\alpha_1, ..., \alpha_n}$ 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 $(\mathcal{M} \times Obj)^*$ (for definition $\mathcal{M}$, 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:

$$Q = \lambda^q_\epsilon(R \| S \| A). \tag{4.2.1}$$

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:

$$R = \sum_{d \in \mathcal{M}} \sum_{\alpha \in Obj} rd(iq,d,\alpha) \cdot R. \tag{4.2.2}$$

The relevant equation for the state operator is:

$$\lambda^q_\sigma(rd(iq,d,\alpha) \cdot x) = rd(iq,d,\alpha) \cdot \lambda^q_{(d,\alpha)\star\sigma}(x). \tag{4.2.3}$$

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^q_σ 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:

$$mf(L,\epsilon) = \epsilon \tag{4.2.5}$$

$$mf(L,\sigma\star(m(\alpha_1, \ldots,\alpha_n),\alpha)) = \begin{cases} m(\alpha_1, \ldots,\alpha_n) & \text{if } m \in L \\ mf(L,\sigma) & \text{otherwise} \end{cases} \tag{4.2.6}$$

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:

$$f_\alpha(sn(\beta,iq,d)) = send(\beta,iq,d,\alpha) \tag{4.3.1}$$

$$f_\alpha(rd(iq,d,\beta)) = read(\alpha,iq,d,\beta) \tag{4.3.2}$$

$$f_\alpha(sn(om,M)) = send(\alpha,om,M,\alpha) \tag{4.3.3}$$

$$f_\alpha(rd(om,M)) = read(\alpha,om,M,\alpha) \tag{4.3.4}$$

$$f_\alpha(sn(fm, (M,m))) = send(\alpha, fm, (M,m), \alpha) \qquad (4.3.5)$$

$$f_\alpha(rd(fm, (M,m))) = read(\alpha, fm, (M,m), \alpha) \qquad (4.3.6)$$

4.4. Root unit. Now we change the semantic rule for the root unit as follows:

$RU \rightarrow$**root unit** CDL (2)

Let

$$cd(CDL)(Integer) = I$$

$$cd(CDL)(Boolean) = B$$

$$cd(CDL)(Read\_File) = R$$

$$cd(CDL)(Write\_File) = W$$

$$\mathcal{C} = \{cd(CDL)(C) \mid C \in UId\}$$

$$ACTIVE = \mathop{\|}_{\alpha \in AObj} (\sum_{X \in \mathcal{C}} create^*(X, \alpha) \cdot \rho_{f_\alpha}(X))$$

$$STANDARD = (\mathop{\|}_{\alpha \in Int} \rho_{f_\alpha}(I)) \| \rho_{f_{\mathrm{true}}}(B) \| \rho_{f_{\mathrm{false}}}(B) \| \rho_{f_{\mathrm{input}}}(R) \| \rho_{f_{\mathrm{output}}}(W)$$

$$QUEUE = \mathop{\|}_{\alpha \in Obj} (\rho_{f_\alpha}(Q))$$

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: for any model M of ACP$_\tau$ which preserves fairness and liveness properties we have

$$INT_M \circ SPEC_A \neq INT_M \circ SPEC_{AQ}.$$

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 employing handshaking communication between the objects instead of communication by means of message queues, 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. $INT_{@(BS)} \circ SPEC_A \neq INT_{@(BS)} \circ SPEC_{AQ'}$.

PROOF. Below we present a POOL-⊥ unit u with the property that in the term model modulo bisimulation the unique solutions of specifications $SPEC_A(u)$ 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 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 $SPEC_A$ and $SPEC_{AQ'}$ 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

$$INT_{\mathcal{Q}(FS)}\circ SPEC_A \;=\; INT_{\mathcal{Q}(FS)}\circ SPEC_{AQ'}$$

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 give it in this article.

5. Trace semantics, fairness and successful termination

5.1. The trace model as presented in [28], is not a good semantic domain for POOL in the sense that it identifies too much and does not describe deadlock behaviour. In $\mathcal{Q}(TR_{den})$ 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 $\mathcal{Q}(BS)$) and then takes the set of terminating (and infinite) traces of this process. In this approach one typically looks at

$$YIELD \circ INT_{\mathcal{Q}(BS)} \circ SPEC_A(u)$$

where *YIELD* is a function that gives the set of terminating (and infinite) traces of elements of $\mathcal{Q}(BS)$. 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:

$$YIELD_F \circ INT_{\mathcal{Q}(BS)} \circ SPEC_C(u)$$

where $YIELD_F$ gives the set of fair terminating and infinite traces of elements of $\mathcal{Q}(BS)$.

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 $\mathcal{Q}(BS)$. 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 in any model M in which KFAR holds:

$$INT_M \circ SPEC_A(f) = \tau \cdot output(0) \cdot \delta.$$

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.

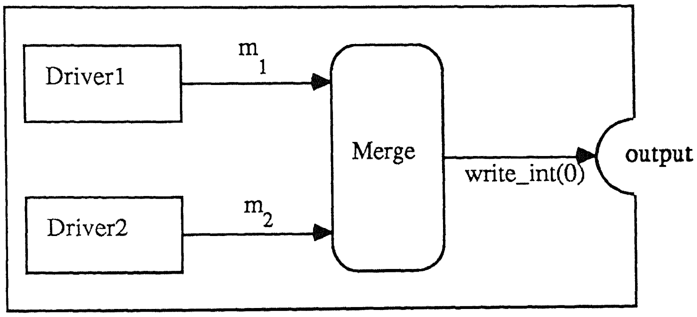

FIGURE 5.1

There are two objects *Driverl* and *Driver2.* The only thing these objects do is sending method calls to an object *Merge. Driverl* 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 ($m_1, m_2, m_1, m_2, \ldots$) 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:

$$\boxed{(\mathrm{KFAR}^-) \qquad \frac{x = ix + \tau y}{\tau_{\{i\}}(x) = \tau \cdot \tau_{\{i\}}(\tau y)}}$$

KFAR$^-$ turns out to be sufficient for the protocol verifications in [22, 25, 27]. 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:

$$\pi_1(INT_{\mathcal{A}(FS)} \circ SPEC_A(f)) \neq \tau \cdot output(0) \cdot \delta.$$

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

$$YIELD \circ INT_{@(BS)} \circ SPEC_A(u1) = YIELD \circ INT_{@(BS)} \circ SPEC_A(u2)$$

although

$$INT_{@(BS)} \circ SPEC_A(u1) \neq INT_{@(BS)} \circ SPEC_A(u2)$$

(we even have

$$INT_{@(FS)} \circ SPEC_A(u1) \neq INT_{@(FS)} \circ SPEC_A(u2)\ \ ).$$

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 is a solution of $SPEC_A(u1)$:

$$x_1 = \tau \cdot (output(0) \| \sum_{\alpha \in Int} input(\alpha)) \cdot \delta$$

(2) The following x_2 is a solution of $SPEC_A(u2)$:

$$x_2 = \tau \cdot (\tau \cdot output(0) \cdot (\sum_{\alpha \in Int} input(\alpha)) + \tau \cdot (\sum_{\alpha \in Int} input(\alpha)) \cdot output(0)) \cdot \delta$$

Let $B = \{input(\alpha) \mid \alpha \in Int\}$ 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 ∂_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:

$$INT_M \circ SPEC_A(u_1) \cdot INT_M \circ SPEC_A(u_2) = INT_M \circ SPEC_A(u_1).$$

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. Integers and booleans

On the conceptual level, each integer and each boolean is represented by a different object. In an implementation of the language it will of course not be possible to point at different processors saying: 'This is object **true**' or 'That processor over there implements object 4370578', etc. On the level of implementation integers and booleans certainly will not be objects. Instead an implementation will contain some special circuits for arithmetical and logical operations. The aim of this section is to make it plausible that, when speaking about integers and booleans, the conceptual and implementation view of the system are not in contradiction with each other (although there is a problem).

6.1. Simple expressions for integer and Boolean objects. The first two equations in a $SPEC_A$ specification have the form (cf. equation 3.11.2 and the semantic rule for production (2) in Section 3.9.2):

$$ROOT = \tau_I([\![RU]\!])$$

$$[\![RU]\!] = \lambda_0^{counter} \circ \partial_J \circ \partial_K(create([\![CDL]\!], \hat{0}) \| ACTIVE \| STANDARD)$$

Here $I = \{c(d) \mid d \in D\} \cup \{comm(f) \mid f \in \mathfrak{F}\} \cup \{skip\}$. If we define

$$I' = \{c(d) \mid d \in D\} \cup \{skip\}$$

then we can prove, using the conditional axioms, that this is equivalent to:

$$ROOT = \tau_I([\![RU]\!])$$

$$[\![RU]\!] = \lambda_0^{counter} \circ \partial_J \circ \partial_K(create([\![CDL]\!], \hat{0}) \| ACTIVE \| \tau_{I'}(STANDARD))$$

Applying the conditional axioms again gives that $\tau_{I'}(STANDARD)$ equals

$$(\mathop{\|}_{\alpha \in Int} \tau_{I'} \circ \rho_{f_\alpha}(I)) \| \tau_{I'} \circ \rho_{f_{\mathbf{true}}}(B) \| \tau_{I'} \circ \rho_{f_{\mathbf{false}}}(B) \| \tau_{I'} \circ \rho_{f_{\mathbf{input}}}(R) \| \tau_{I'} \circ \rho_{f_{\mathbf{output}}}(W).$$

The processes corresponding to the objects of class *Integer* and *Boolean* are very simple. For the object **true** we can derive:

$$\tau_{I'} \circ \rho_{f_{\mathbf{true}}}(B) =$$

$$= \tau \cdot (\sum_{\beta \in Bool} \sum_{\alpha \in Obj} read(\mathbf{true}, mc, or(\beta), \alpha) \cdot send(\alpha, an, \mathbf{true}, \mathbf{true}) \cdot \tau_{I'} \circ \rho_{f_{\mathbf{true}}}(B)$$

$$+ \sum_{\alpha \in Obj} read(\mathbf{true}, mc, or(\mathbf{nil}), \alpha) \cdot error \cdot \tau_{I'} \circ \rho_{f_{\mathbf{true}}}(B)$$

$$+ \sum_{\beta \in Obj - Bool - \{\mathbf{nil}\}} \sum_{\alpha \in Obj} read(\mathbf{true}, mc, or(\beta), \alpha) \cdot \delta + \cdots).$$

The dots at the end of the equation stand for similar summands corresponding to the other methods of class *Boolean*. In a correct POOL-⊥ program the parameter of a message with method identifier *or* will always be an element of $Bool \cup \{\mathbf{nil}\}$. Therefore the summand $\Sigma_{\beta \in Obj - Bool - \{\mathbf{nil}\}}(.)$ is redundant in the context in which it is placed, and we can omit it (the corresponding summands of the other methods can of course also be omitted). A formal proof of this obvious fact can be given using the theorems about the notion of 'redundancy in a context' of [28].

After this simplification the process that gives the behaviour of object **true** can be written into the following form:

$$X_{\mathbf{true}} = \tau \cdot (\textstyle\sum read(\mathbf{true}, mc, \ ..., \beta) \cdot send(\beta, an, \ ..., \mathbf{true})$$
$$+ \textstyle\sum read(\mathbf{true}, mc, \ ..., \beta) \cdot error) \cdot X_{\mathbf{true}}.$$

Using the identity $\tau x \| y = \tau(x \| y)$, we can replace the equation for variable $ROOT$ by:

$$ROOT = \tau \cdot \tau_I([\![RU]\!])$$

and omit the initial τ in the equation for $X_{\mathbf{true}}$:

$$X_{\mathbf{true}} = (\textstyle\sum read(\mathbf{true}, mc, \ ..., \beta) \cdot send(\beta, an, \ ..., \mathbf{true})$$
$$+ \textstyle\sum read(\mathbf{true}, mc, \ ..., \beta) \cdot error) \cdot X_{\mathbf{true}}.$$

We claim that all the processes corresponding to objects of class *Integer* and *Boolean* can be specified analogously. Let for $\alpha \in Int \cup Bool$, $\mathbb{S}_\alpha \subseteq \mathcal{F}$ be the set of frames that can be sent to object α:

$$\mathbb{S}_\alpha = \{(\alpha, mc, d, \beta) \mid d \in \mathcal{M}, \ \beta \in Obj \text{ and } d \text{ correct for } \alpha\}.$$

Message d is correct for object α if the method identifier of d occurs in the class description of α, the number of parameters is correct, and the parameters are of the right type. For each $\alpha \in Int \cup Bool$ process X_α is defined by:

$$X_\alpha = \sum_{f \in \mathbb{S}_\alpha} read(f) \cdot an_f \cdot X_\alpha.$$

Here an_f is an atomic action, the answer to the method call f. This can be a *send* action or the *error* action. For example:

$$an_{(1, mc, add(1), \hat{1})} = send(\hat{1}, an, 2, 1)$$

$$an_{(1, mc, minus(\mathbf{nil}), \hat{0})} = error$$

Now we define:

$$INT = \mathop{\|}_{\alpha \in Int} X_\alpha$$

$$BOOL = X_{\mathbf{true}} \| X_{\mathbf{false}}$$

$$I/O = \tau_{I'} \circ \rho_{f_{\mathbf{input}}}(R) \| \tau_{I'} \circ \rho_{f_{\mathbf{output}}}(W)$$

Let $SPEC_{AA}$ be the same function as $SPEC_A$ except for the fact that the term for variable $ROOT$ is prefixed with a τ and that in the equation for $[\![RU]\!]$ $STANDARD$ is replaced by $INT \| BOOL \| I/O$. We have for all models M:

$$INT_M \circ SPEC_A = INT_M \circ SPEC_{AA}.$$

6.2. Monadic objects. The processes $STANDARD$ and $INT \| BOOL \| I/O$ both consist of the merge of a number of objects. Each object answers all the messages for one integer or boolean, and different objects answer messages for different integers or booleans.

We now introduce processes INT_M and $BOOL_M$. These processes are composed of a huge amount of 'monadic' objects. For each frame there is a monadic object which has nothing else to do but answering that frame. There is for example a monadic object answering the message from object $\hat{0}$ to object 1 in which it asks to perform method *add* with parameter 3:

$$M_{(1,mc,add(3),\hat{0})} = read(1,mc,add(3),\hat{0}) \cdot send(\hat{0},an,4,1) \cdot M_{(1,mc,add(3),\hat{0})}.$$

Let $\mathcal{S}_{INT} = \bigcup_{\alpha \in Int} \mathcal{S}_\alpha$ and $\mathcal{S}_{BOOL} = \mathcal{S}_{\mathbf{true}} \cup \mathcal{S}_{\mathbf{false}}$. We define for $f \in \mathcal{S}_{INT} \cup \mathcal{S}_{BOOL}$ the process M_f by:

$$M_f = read(f) \cdot an_f \cdot M_f.$$

Processes INT_M and $BOOL_M$ are defined by:

$$INT_M = \mathop{\|}_{f \in \mathcal{S}_{INT}} M_f \quad \text{and} \quad BOOL_M = \mathop{\|}_{f \in \mathcal{S}_{BOOL}} M_f.$$

Let $SPEC_{AM}$ be the same as $SPEC_{AA}$ except for the fact that in the equation for $[\![RU]\!]$, $INT \| BOOL$ is replaced by $INT_M \| BOOL_M$.

6.3. The error action. We would like to prove for all models M:

$$INT_M \circ SPEC_{AA} = INT_M \circ SPEC_{AM}.$$

This would be a nice theorem because the same argument used to 'ungroup' the standard objects into monadic objects, can, when reversed, also be used to 'group' the monadic objects into a new configuration (a single object **integer** and a single object **boolean**, or separate objects for the various methods, etc.).

Unfortunately the two semantics are different. The problem, which has to do with the *error* action, is illustrated by the following POOL unit m:

```
root unit

class One_plus_one

var n : Integer

routine new( ) One_plus_one :

        return new

end new

body n←1 ! add(1) ; Write_File · standard_out( ) ! write_int(n)

end One_plus_one,
```

class *One_minus_nil*

body *One_plus_one* · *new*( ) ; 1 ! *minus*(**nil**)

end *One_minus_nil*

In the $SPEC_{AA}$ case where integers are objects, it can happen that object 1 first answers the method call *minus*(**nil**). This leads to a state in which no external action has been performed but the order in which the actions will be executed is fully determined, namely first the *error* action and then the action *output*(2). In the $SPEC_{AM}$ case such a state cannot be reached since there are different monadic objects for frames $(1,mc,minus(\mathbf{nil}),\hat{0})$ and $(1,mc,add(1),\hat{1})$, and these monadic objects work independently. If the *error* action is blocked it can happen in the $SPEC_{AA}$ case that the action *output*(2) will not be performed. In the $SPEC_{AM}$ case the *output*(2) action will always be performed in such a situation. As a result of this:

$$INT_{\mathfrak{A}(FS)} \circ SPEC_{AA}(m) \neq INT_{\mathfrak{A}(FS)} \circ SPEC_{AM}(m).$$

6.4. Ostrich policy. The problem is not typical for the 'monadic' implementation of the integers and booleans but arises in every implementation different from the one suggested by $SPEC_{AA}$. However, it has to be noticed that in the trace set approach of Section 5.3, $SPEC_{AA}$ and $SPEC_{AM}$ (and thereby all other implementations) lead to the same semantics. In case we do not want to describe the system in terms of trace semantics, the best solution seems to be to abstract from the *error* action. We replace the equation for variable $ROOT$ in $SPEC_{AA}$ and $SPEC_{AM}$ by

$$ROOT = \tau \cdot \tau_{I \cup \{error\}}([\![RU]\!]).$$

Call the new functions $SPEC_{AAO}$ and $SPEC_{AMO}$ (the 'O' from ostrich policy).

CLAIM. For all models M:

$$INT_M \circ SPEC_{AAO} = INT_M \circ SPEC_{AMO}.$$

We will not give a rigorous proof of this claim but confine ourselves to a sketch of it.

6.5. DEFINITION. A specification $E = \{X = t_X \mid X \in \Xi\}$ is called *strictly linear* if for every $X \in \Xi$:

$t_X = \tau$ or

$t_X = \delta$ or

$\exists m \geqslant 1$

$\exists a_1, ..., a_m \in A_\tau$

$\exists X_1, ..., X_m \in \Xi$ such that

$$t_X = \sum_{k=1}^{m} a_k \cdot X_k$$

6.6. THEOREM. *For every guarded specification E there exists a strictly linear guarded specification F with the same solution.*

6.7. *Structure of active objects.* Although a POOL system contains a large amount of parallelism, the individual objects work in a totally sequential way. The process algebra equations which define the behaviour of these objects contain chaining operators but, beside value passing, the process on the right hand side always starts after termination of the left hand side process. This observation (which of course can be expressed formally) motivates the following claim.

CLAIM. For every $\alpha \in AObj$ there exists a strictly linear guarded system of equations with root variable X_α such that

$$X_\alpha = \sum_{V \in \mathbb{C}} create^*(V, \alpha) \cdot \rho_{f_\alpha}(V)$$

and with the property (cf. semantic rules for production 21) that atomic actions $send(\alpha, mc, m(\alpha_1, ..., \alpha_n), \beta)$ only occur in equations of the form:

$$X = send(\alpha, mc, m(\alpha_1, ..., \alpha_n), \beta) \cdot Y$$

where Y is a variable for which we have an equation of the form:

$$Y = \sum_{\gamma \in Obj} read(\beta, an, \gamma, \alpha) \cdot Z_\gamma.$$

This means that every time when an active object performs an action $send(\alpha, mc, m(\alpha_1, ..., \alpha_n), \beta)$, the next action will be of the form $read(\beta, an, \gamma, \alpha)$.

6.8. We rewrite the equations for INT, $BOOL$, INT_M and $BOOL_M$ into the following form:

$$INT = \sum_{f \in \mathbb{S}_{INT}} read(f) \cdot INT^f$$

$$INT^f = (\mathop{\|}_{\beta \in Int - \{\alpha\}} X_\beta) \| an_f \cdot X_\alpha \quad \text{for } f \in \mathbb{S}_\alpha$$

$$BOOL = \sum_{f \in \mathbb{S}_{BOOL}} read(f) \cdot BOOL^f$$

$$BOOL^f = (\mathop{\|}_{\beta \in Bool - \{\alpha\}} X_\beta) \| an_f \cdot X_\alpha \quad \text{for } f \in \mathbb{S}_\alpha$$

$$INT_M = \sum_{f \in \mathbb{S}_{INT}} read(f) \cdot INT_M^f$$

$$INT_M^f = (\mathop{\|}_{g \in \mathbb{S}_{INT} - \{f\}} M_g) \| an_f \cdot M_f$$

$$BOOL_M = \sum_{f \in \mathbb{S}_{BOOL}} read(f) \cdot BOOL_M^f$$

$$BOOL_M^f = (\mathop{\|}_{g \in S_{BOOL} - \{f\}} M_g) \| an_f \cdot M_f$$

Define:

$$I'' = \{comm(\alpha, an, \beta, \gamma) \mid \alpha, \beta \in Obj;\ \gamma \in Int \cup Bool\} \cup \{error\}.$$

Application of the conditional axioms gives that, in order to prove the claim of Section 6.4, it is enough to show:

$$LHS = RHS$$

where

$$LHS = \tau_{I''} \circ \partial_J \circ \partial_K (create([\![CDL]\!], \hat{0}) \| (\mathop{\|}_{\alpha \in AObj} X_\alpha) \| INT \| BOOL \| I/O)$$

$$RHS = \tau_{I''} \circ \partial_J \circ \partial_K (create([\![CDL]\!], \hat{0}) \| (\mathop{\|}_{\alpha \in AObj} X_\alpha) \| INT_M \| BOOL_M \| I/O)$$

A quick inspection of the semantic rules defining $SPEC_{AAO}$ learns us that LHS is specifiable by means of guarded equations for all $n \in \mathbb{N}$. Therefore it is enough to show that for every $n \in \mathbb{N}$:

$$\pi_n(LHS) = \pi_n(RHS).$$

6.9. Definition. For X a variable and t a term, the relation X *occurs open in* t is defined inductively by:

1. X occurs open in X
2. if X occurs open in t then X occurs open in $t \cdot s$, $t \mathbin{\|\!\_} s$, $t+s$, $s+t$, $t \| s$, $s \| t$, $t \mid s$, $s \mid t$, $\partial_H(t)$, $\tau_I(t)$, $\rho_f(t)$ and $\pi_n(t)$.

An occurrence of a variable X in a term t is *needed* if t contains a subterm of the form $\pi_n(s)$ and X occurs open in s.

6.10. Definition. For given specification E, $\vec{E}$ is the term rewriting system consisting of the axioms from $\mathrm{ACP}_\tau + \mathrm{RN} + \mathrm{CH} + \mathrm{SO} + \mathrm{PR} + \mathrm{RC}\text{-}\mathrm{AT}$ together with the equations of E (read from left to right). Here RC is the rewrite rule:

$$a \mid b = \gamma(a, b)$$

that rewrites a term $a \mid b$ into the corresponding communication, and AT is the set of axioms consisting of A1, A2, C1-3 and T1-3.

6.11. Theorem. *Let E be a guarded specification with root variable X_0. Let $n \in \mathbb{N}$. Then the term $\pi_n(X_0)$ will be rewritten into a closed term if we apply the rewrite rules of $\vec{E}$, following the strategy that only needed occurrences of variables are replaced.*

6.12. Choose a $n \in \mathbb{N}$. We have to prove:

$$\pi_n(LHS) = \pi_n(RHS).$$

The specifications that specify LHS and RHS are almost the same. We relate variables LHS and RHS, INT and INT_M, INT^f and INT_M^f, $BOOL$ and $BOOL_M$, $BOOL^f$ and $BOOL_M^f$, and furthermore all variables with the same name. Now we start to rewrite the term $\pi_n(LHS)$ into a closed term. Simultaneously we start rewriting $\pi_n(RHS)$ *in exactly the same way.* If on the left hand side a variable is rewritten, then we also rewrite the corresponding variable on the right hand side, etc. The problem with this imitation game is of course that the equations for INT^f and INT_M^f, $BOOL^f$ and $BOOL_M^f$ are different. What we do in order to solve this problem is that, when during the rewrite process a variable INT^f or $BOOL^f$ becomes needed, we rewrite the left and right hand side in such a way that:

1. The new left and right hand side are equivalent modulo names of variables.
2. No variable INT^f or $BOOL^f$ occurs needed in the left hand side.
3. It is clear that this intermediate 'surgery' will not slow down the process of rewriting $\pi_n(LHS)$ into a closed term.

Using the imitation+surgery strategy we rewrite $\pi_n(LHS)$ and $\pi_n(RHS)$ into the same closed term. Because n was chosen arbitrarily that finishes the proof of the claim of Section 6.4.

6.13. Surgery. Let $\alpha \in Int$ and $f = (\alpha, mc, d, \beta) \in \mathbb{S}_\alpha$ (the boolean case can be dealt with analogously). Suppose that after some rewrite step variable INT^f becomes needed in the left hand side term. We claim that INT^f occurs in a subterm which can be brought into the form:

$$comm(f) \cdot \pi_m \circ \tau_{I''} \circ \partial_H \circ \partial_K(\cdots \| INT^f \| \sum_{\gamma \in Obj} read(\beta, an, \gamma, \alpha) \cdot Z_\gamma).$$

If we rewrite variable INT^f this becomes:

$$comm(f) \cdot \pi_m \circ \tau_{I''} \circ \partial_H \circ \partial_K(\cdots \| (\mathop{\|}_{\kappa \in Int - \{\alpha\}} X_\kappa) \| an_f \cdot X_\alpha \| \sum_{\gamma \in Obj} read(\beta, an, \gamma, \alpha) \cdot Z_\gamma).$$

The corresponding right hand side subterm can be brought into the form:

$$comm(f) \cdot \pi_m \circ \tau_{I''} \circ \partial_H \circ \partial_K(\cdots \| (\mathop{\|}_{g \in \mathbb{S}_{INT} - \{f\}} M_g) \| an_f \cdot M_f \| \sum_{\gamma \in Obj} read(\beta, an, \gamma, \alpha) \cdot Z_\gamma).$$

If $an_f = error$ we bring the ostrich policy into practice: because $error \in I''$ we can replace the *error* action by τ in both terms. The next step is to eliminate these τ's using the identity $\tau x \| y = \tau(x \| y)$. But then the subterm on the left contains the merge for all $\alpha \in Int$ of X_α. This is equal to INT. The subterm on the right contains the merge for all $f \in \mathbb{S}_{INT}$ of processes M_f, which is equal to INT_M. This finishes the surgery activities for the case $an_f = error$.

In the other case we have $an_f = send(\beta, an, \bar{\gamma}, \alpha)$ for some $\bar{\gamma} \in Obj$. Using the conditional axioms we can replace the left hand side subsubterm (excusez le mot):

$$an_f \cdot X_\alpha \| \sum_{\gamma \in Obj} read(\beta, an, \gamma, \alpha) \cdot Z_\gamma$$

by

$$\tau_{\{comm(\beta,an,\bar{\gamma},\alpha)\}} \circ \partial_{\{send(\beta,an,\bar{\gamma},\alpha)\}}(send(\beta,an,\bar{\gamma},\alpha) \cdot X_\alpha \| \sum_{\gamma \in Obj} read(\beta,an,\gamma,\alpha) \cdot Z_\gamma)$$

which is equal to

$$\tau \cdot \tau_{\{comm(\beta,an,\bar{\gamma},\alpha)\}} \circ \partial_{\{send(\beta,an,\bar{\gamma},\alpha)\}}(X_\alpha \| Z_{\bar{\gamma}}) +$$
$$+ \sum_{\gamma \in Obj} read(\beta,an,\gamma,\alpha) \cdot \tau_{\{comm(\beta,an,\bar{\gamma},\alpha)\}} \circ \partial_{\{send(\beta,an,\bar{\gamma},\alpha)\}}(send(\beta,an,\bar{\gamma},\alpha) \cdot X_\alpha \| Z_\gamma).$$

The second summand is redundant in the context in which it occurs and can be omitted. Using the conditional axioms again, together with identity $\tau x \| y = \tau(x \| y)$, yields that the term can be replaced by:

$$X_\alpha \| Z_{\bar{\gamma}}.$$

Now we have brought the left hand side subterm in a form which contains the merge for all $\alpha \in Int$ of X_α. This merge we can replace by INT. The same strategy that was used to rewrite the left hand side can be used to rewrite the right hand side. The result is the same term as obtained on the left hand side, except that we have variable INT_M instead of INT.

7. Conclusions

1. In this paper we showed 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, logic 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 a lot of 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. It would be interesting to construct a proof system, based on our process algebra semantics, which can be used to prove correctness of POOL programs.
9. 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.
10. 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 Section 6 we gave a sketch of such a proof, showing that the Integers and Booleans can be implemented in a lot of 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. *Conference Record of the 13th ACM Symposium on Principles of Programming Languages,* St. Petersburg, Florida, 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,* CWI Report CS-R8626, Centre for Mathematics and Computer Science, Amsterdam. To appear in *Information and Computation.*
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),* 370-379.
8. J.C.M. BAETEN, J.A. BERGSTRA (1988). Global renaming operators in concrete process algebra. *Information and Computation 78(3),* 205-245.
9. J.W. DE BAKKER (1980). *Mathematical Theory of Program Correctness,* Prentice-Hall.
10. J.A. BERGSTRA (1989). *A Process Creation Mechanism in Process Algebra.* This volume.
11. J.A. BERGSTRA, J.W. KLOP, E.-R. OLDEROG (1987). Failures without chaos: a new process semantics for fair abstraction. M. WIRSING (ed.). *Proc. IFIP Conf. on Formal Description of Programming Concepts - III,* Ebberup 1986, North-Holland, Amsterdam, 77-103.
12. G.V. BOCHMAN (1976). Semantic evaluation from left to right. *Communications of the ACM 19(2),* 55-62.
13. D.W. BUSTARD (1980). An introduction to Pascal-Plus. R.M. MCKEAG, A.M. MACNAGHTEN (eds.). *On the construction of programs - an advanced course,* Cambridge University Press, 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. F.J. BRANDENBURG, G. VIDAL-NAQUET, M. WIRSING (eds.). *Proc. STACS 87,* LNCS 247, Springer-Verlag, 336-347.
17. R.J. VAN GLABBEEK, F.W. VAANDRAGER (1987). Petri net models for algebraic theories of concurrency (extended abstract). J.W. DE BAKKER, A.J. NIJMAN, P.C. TRELEAVEN (eds.). *Proceedings PARLE conference, Eindhoven, Vol. II (Parallel Languages),* LNCS 259, Springer-Verlag, 224-242.
18. C.A.R. HOARE (1985). *Communicating Sequential Processes,* Prentice-Hall.
19. INMOS, LTD. (1984). *The Occam Programming Manual,* Prentice-Hall.
20. ISO (1987). *A Formal Description Technique.* ISO/TC97/SC21/WG16-1 DP8807.
21. D.E. KNUTH (1968). Semantics of context-free languages. *Mathematical Systems Theory, 2,* 127-145. Correction: *Mathematical Systems Theory 5,* 1971, 95-96.
22. C.P.J. KOYMANS, J.C. MULDER (1989). *A Modular Approach to Protocol Verification using Process Algebra,* This volume.
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). *Verification of Two Communication Protocols by means of Process Algebra,* CWI Report CS-R8608, Centre for Mathematics and Computer Science, Amsterdam.
26. F.W. VAANDRAGER (1986). *Process Algebra Semantics of POOL,* CWI Report CS-R8629, Centre for Mathematics and Computer Science, Amsterdam.
27. F.W. VAANDRAGER (1989). *Some Observations on Redundancy in a Context.* This volume.
28. F.W. VAANDRAGER (1989). *Two Simple Protocols,* This volume.