P. America, J.W. de Bakker, J.N. Kok, J. Rutten

# Operational semantics of a parallel object-oriented language

Department of Computer Science Report CS-R8515 August

# Operational Semantics of a Parallel Object-Oriented Language

Pierre America
*Philips Research Laboratories*
*P.O.Box 80000, 5600 JA Eindhoven, The Netherlands*

Jaco de Bakker

Joost N. Kok

Jan Rutten
*Centre for Mathematics and Computer Science*
*P.O. Box 4079, 1009 AB Amsterdam, The Netherlands*

In this paper the semantics of the programming language POOL is described. It is a language that integrates the object-oriented structure of languages like Smalltalk-80 with facilities for concurrency and communication like the ones in Ada. The semantics is described in an operational way: it is based on transition systems. By using a way of representing parallel processes that is different from the traditional one, it is possible to overcome some difficulties pertaining to the latter. The resulting semantics shows a close resemblance with the informal language description and at the same time there are good prospects that it can serve as a secure guide for the implementation of the language.

1980 *Mathematics Subject Classification*: 68B10, 68C01 69D13, 69D21, 69D41, 69F32
1982 CR Categories: D.1.3, D.2.1, D.3.1, F.3.2
*Key Words & Phrases: Object-Oriented Programming, Parallelism, Transition Systems, Language Design, Implementation, Fairness, Operational Semantics, Maximal Parallelism*

Note:
This work was carried out in the context of ESPRIT project 415: Parallel Architectures and Languages for AIP-a VLSI-directed approach.

Report CS-R8515
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

## 1. Introduction

POOL is a programming language that combines the structuring mechanisms of object-oriented programming with means to express parallelism. In the present paper we will give a formal semantics for it, using the technique of transition systems. The plan of the paper is as follows: In section 2 a general introduction is given to parallel object-oriented programming and to the language POOL in particular. In section 3 its syntax is presented, together with a more detailed explanation of the individual language constructs. Section 4 gives an introduction to transition systems in general and then it presents the axioms and rules of our particular system. Section 5 describes a variant of the formalism in which parallelism occurs more directly. Finally in the conclusions the advantages of the way in which we represented a system state are discussed, together with an overview of work that is still to be done in this area.

## 2. An informal introduction to the language

Programming concurrent systems is a difficult task in which it occurs very often that subtle errors disturb the correct functioning of a program in a most disastrous way. As the number of parallel processes increases, these difficulties tend also to increase dramatically. Here a clear way of structuring such systems is badly needed. Object-oriented programming (of which the language Smalltalk-80 [Goldberg & Robson 1983] is a representative example) offers a way to structure large systems. Originally it was only used in sequential systems, but it offers excellent possibilities for a very advantageous integration with parallelism. (This was already proposed in [Hewitt 1977], but our approach is different from his.)

POOL is an acronym for "Parallel Object-Oriented Language". It stands for a family of languages designed at Philips Research Laboratories in Eindhoven in the second half of 1984 and the first half of 1985. Of these languages, POOL-T [America 1985] is the latest, and the one that will be implemented. These languages all make use of the structuring mechanisms of object-oriented programming, integrated with the most important mechanisms for expressing concurrency from the language Ada [ANSI 1983]: processes and rendez-vous. This paper describes the semantics of a language that we will simply call POOL. It is undoubtedly a member of the above family, but it is slightly simplified (especially with respect to syntax) in order to facilitate the semantic description.

A POOL program describes the behaviour of a whole system in terms of its constituents, *objects*. Objects possess some internal data, and they have the ability to act on these data. They are entities of a very dynamic nature: they can be created dynamically, they can be modified, and they have even an internal activity of their own. At the same time they are units of protection: the internal data of one object are not directly accessible by other objects.

An object can have *variables* (also called instance variables) to store its internal data in. A variable can contain (a reference to) an object (another object, or, possibly, the object under consideration itself). Changing (assigning to) a variable means making it refer to a different object than before. The variables of one object cannot be accessed directly by other objects. They can only be read and assigned to by the object itself.

The objects may only interact via sending *messages* to each other. Each object states explicitly to which object it sends a certain message, and also when it is prepared to accept one. When an object sends a message, its activity is suspended until the result of that message arrives. When the receiver answers the message, it will execute a so-called *method* (a kind of procedure) and the result of this method execution will be sent back to the sender. The sender of the message indicates which method should be invoked. It can also pass some *parameters* to the method.

Such a method can access the variables of the receiving object. In addition it can have some local variables of its own. In addition to answering a message, an object can execute a method of its own simply by calling it. Because of this, and because answering a message within a method is also allowed, recursive invocations of methods are possible. Each of these invocations has its own set of parameters and local variables, of course.

When an object is created, a local process is started up: the object's *body*. When several objects have

been created, their bodies may execute in parallel, thus introducing parallelism into the language. Within a body an object states explicitly when it is prepared to answer certain messages. Having returned the answer, the execution of its body is resumed.
Objects are grouped into *classes*. All objects in one class (the *instances* of that class) have the same number and kind of variables, the same methods for answering messages, and the same body. In this way a class describes the behaviour of its instances.
There is a special object, *nil*, which can be considered to be an element of every class. If a message is sent to this object, an error occurs. Upon the creation of a new object, its instance variables are initialized to *nil*, and when a method is invoked, its local variables are also initialized to *nil*.
There are a few standard classes predefined in the language. In this semantic description we will only incorporate the classes Boolean and Integer. On these objects the usual operations can be performed, but they must be formulated by sending messages. For example, the addition 2+4 is indicated by sending the message with method name add and parameter 4 to the object 2.

## 3. Syntax

In this section the (abstract) syntax of the language is described. The syntax contains some elements that are not present in the original language, but which are included to facilitate the description of the operational semantics.
We assume that the following sets of syntactic elements are given:

| | | |
|---|---|---|
| *IVar* | (instance variables) | with typical elements: $x, y, \cdots$ , |
| *LVar* | (local variables) | with typical elements: $u, v, \cdots$ , |
| *CName* | (class names) | with typical elements: $C, D, \cdots$ , |
| *MName* | (method names) | with typical elements: $m, \cdots$ . |

We define the set *AObj* of active (i.e. non-standard) objects, with typical elements $\alpha, \cdots$ by taking a copy of the set $\mathbb{N}$ of natural numbers (with typical elements $n, \cdots$ ):

$$AObj = \{ \hat{n} \mid n \in \mathbb{N} \}$$

Together with the standard objects (integers, booleans and *nil*), this constitutes the set *Obj* of objects, with typical elements $\beta, \cdots$ :

$$Obj = AObj \cup \mathbb{Z} \cup \{tt, ff\} \cup \{nil\}$$

We now define the set *Exp* of expressions, with typical elements $e, \cdots$ :

$$\begin{array}{lll} e & ::= & x \\ & | & u \\ & | & e \,!\, m\,(e_1, \cdots, e_n) \\ & | & m\,(e_1, \cdots, e_n) \\ & | & \textbf{new}\,(C) \\ & | & s\,;\,e \\ & | & \textbf{self} \\ & | & \textbf{wait} \\ & | & \underline{\beta} \end{array}$$

The set *Stat* of statements, with typical elements $s, \cdots$ :

$$\begin{array}{lll} s & ::= & x \leftarrow e \\ & | & u \leftarrow e \\ & | & answer\,(m_1, \cdots, m_n) \\ & | & e \\ & | & s_1\,;\,s_2 \\ & | & \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi} \\ & | & \textbf{do}\ e\ \textbf{then}\ s\ \textbf{od} \end{array}$$

$|$ **sel** $g_1$ **or** $\cdots$ **or** $g_n$ **les**
$|$ $e \Rightarrow \underline{\alpha}$

The set *GCom* of guarded commands, with typical elements $g, \cdots$ :

$g ::= [\, e\, ]\ answer\ (m_1, \cdots, m_n)$ **then** $s$.

(Here the expression is optional. Note that $n = 0$ is allowed.)

The set *Unit* of units, with typical elements $U, \cdots$ :

$U ::= < C_1 \Leftarrow d_1 , \cdots , C_n \Leftarrow d_n > \qquad (n \geqslant 1)$.

The set *ClassDef* of class definitions, with typical elements $d, \cdots$ :

$d ::= < ( x_1 , \cdots , x_n ) , ( m_1 \Leftarrow \mu_1 , \cdots , m_k \Leftarrow \mu_k ) , s >$

And finally the set *MethDef* of method definitions, with typical elements $\mu, \cdots$ :

$\mu ::= < ( u_1 , \cdots , u_n ) , ( v_1 , \cdots , v_k ) , e >$.

### 3.1. Informal explanation

**Expressions**

An instance variable or a local variable used as an expression will yield as its value the object that is stored currently in that variable.
The next kind of expression is a send-expression. Here, $e$ is the destination object, to which the message will be sent, $m$ is the method to be invoked, and $e_1$ through $e_n$ are the parameters. When a send-expression is evaluated, first the destination expression is evaluated, then the parameters are evaluated from left to right and then the message is sent to the destination object. When this answers the message, the corresponding method is executed, that is, the parameters are initialized to the objects in the message, the local variables are initialized to *nil*, and the expression in the method definition is evaluated. The value which results from this evaluation is sent back to the sender and this will be the value of the send-expression.
The next kind is a method call. This means simply that the corresponding method is executed (after the evaluation of the parameters from left to right). The result of this execution will be the value of the method call.
Next we have a new-expression. This indicates that a new object is to be created, an instance of the indicated class. Of this object the instance variables are initialized to *nil* and the body starts executing in parallel with all other objects in the system. The result of the new-expression is (the name of) this newly created object.
An expression may also be preceded by a statement. In this case the statement is executed before the expression is evaluated.
The expression **self** always results in the object that is executing this expression.
The expression **wait** is not present in the original language. It is incorporated here to facilitate the semantic description. It indicates that a message is outstanding and that the result, when it is delivered, is to be inserted at the place of the **wait**.
Also the direct naming of objects, indicated by underlining them, is not present in the original language, except for the standard objects.

**Statements**

The first two kinds of statements are assignments, the first to an instance variable, the second to a local variable. An assignment is executed by first evaluating the expression on the right, and then

making the variable refer to the resulting object. Assignments to parameters of a method are not allowed.
The next kind of statement is an answer-statement. This indicates that a message is answered. The object executing the answer-statement waits until a message arrives with a method name that is present in the list. Then it executes the method (after initializing parameters and local variables). The result of the method is sent back to the sender of the message, and the answer-statement terminates.
Next it is indicated that every expression may also occur as a statement. Upon execution, the expression is evaluated and the result is discarded. So only the side effects are important.
Sequential composition, if-statements and do-statements have the usual meaning.
A select-statement is executed as follows: first all the expressions (called: guards) in the guarded commands are evaluated from left to right (if absent, they default to *tt*). They must all result in an object of class Boolean. The guarded commands where the result is *ff* are discarded (they do not play a role in the further execution of the statement). Now one of the (remaining) guarded commands is chosen. This must be either the first one in which the answer statement contains no method names, or there has arrived a message with its method name in the list, and there is no guarded command before this one with an empty method name list, or with the same method name in the list. (Otherwise stated, the object may choose the first guarded command for which it does not have to answer a message, or it may choose a to answer a message, in which case it must select the first guarded command in which that can be done.) If the selected guarded command has a non-empty answer list the above message is answered. After that in either case the statement after **then** is executed, and the select-statement is terminated.
The last kind of statement does not occur in the ordinary POOL syntax but is added to make the semantics easier to describe. It indicates that the expression is to be evaluated and the result sent to the specified object (this may be an object that sent a message and is suspended at a **wait** expression, or the current object itself, in case of a method call).

**Guarded commands**
These are sufficiently described in the treatment of the select-statement.

**Units**
These are the programs of POOL. If a unit is to be executed, a single new instance of the *last* class defined in the unit is created and its body is started. This object has the task to start the whole system up, by creating new objects and putting them to work.

**Class definitions**
A class definition describes how instances of the specified class behave. It indicates the instance variables, the methods and the body each instance of the class will have.

**Method definitions**
A method definition describes, of course, a method. $u_1$ through $u_n$ are the parameters, $v_1$ through $v_k$ the local variables, and $e$ is the expression to be evaluated when the method is invoked.

## 3.2. Context conditions
For a POOL program to be valid there are a few more conditions to be satisfied. In general, they amount roughly to the requirement that such a program be a translated version of a valid POOL-T program. We assume in the semantic treatment that the underlying program is valid.
These context conditions are the following:
- All class names in a unit are different.
- All instance variables in a class definition are different.
- All method names in a class definition are different.
- All parameters and local variables in a method definition are different.

- All instance variables are declared in the current class definition.
- All local variables are declared in the current method definition (they may not occur in a body).
- The class in a new-expression is defined in the current unit. (Here the standard classes are not allowed).
- For each method name in an answer-statement or method call there is a method definition in the current class definition.
- There should be a consistent typing possible for the whole program.

The last condition means that each (instance or local) variable, each parameter and each method should be assigned a class name, its *type*, in such a way that, if starting from these assigned types, the types of each expression is determined, the following conditions hold:
- The left hand side of an assignment has the same type as its right hand side.
- In a send-expression, the class of the destination has a method with the indicated name, and the number and types of the actual parameters agree with those of the formal parameters. The same holds for the parameters of a method call.
- The types of the expressions in if-statements, do-statements and guarded commands are Boolean.
- The type of the expression in a method definition agrees with the type of the method.

### 3.3. Auxiliary definitions

The set *LStat* of labeled statements, with typical elements $l, \cdots$, is defined by:

$$l ::= < \alpha , s >.$$

These will serve in the semantic description to indicate a statement together with the object that is going to execute it.

The following function will appear to be very handy :

$\mu : Unit \times CName \times MName \rightarrow MethDef$

defined by

$$\mu(U,C,m) = \mu$$

where

$$U = < \cdots ,C \Leftarrow d, \cdots >,$$

$$d = < (x_1, \cdots ,x_p) , (m_1 \Leftarrow \mu_1, \cdots ,m \Leftarrow \mu, \cdots ,m_r \Leftarrow \mu_r) , s >,$$

$$\mu = < (u_1, \cdots ,u_n) , (v_1, \cdots ,v_k) , e >.$$

## 4. Semantics

### 4.1. Introduction

Our operational semantics is based on a *transition system.* Transition systems were first used by Hennesy and Plotkin [Hennesy & Plotkin 1979],[Plotkin 1981, 1983]. They were also used by Apt in [Apt 1981, 1983]. In order to introduce this kind of system we first shall explain by a *transition* step. Therefore let us consider the set *Conf* of pairs

$$<s,I>$$

which we shall call *configurations.* (The actual configurations that we will use are more complicated; these pairs only serve as an illustration.) In $<s,I>$ $s$ is a statement and $I$ is some amount of information that has been collected up to now. Now a transition describes what a statement $s$ can do as its next step. The intuitive meaning of the transition:

$$<s_1,I_1> \rightarrow <s_2,I_2>$$

is: executing $s_1$ one step with information $I_1$ can lead to a new amount of information $I_2$ with $s_2$ being the remainder of $s_1$ still to be executed. Note that in general there are different transitions possible.
To define our operational semantics we use a transition system, which is a (syntax directed) deductive system for proving transitions. Such a transition system consists of *axioms* and *rules.*
The axioms tell us what we consider to be the basic transitions, e.g.:

for all integers $\alpha$, for all $I$ and $x$:

$$<x := \alpha \, , I> \rightarrow <\epsilon \, , I \cup \{ \text{ the value of } x \text{ is } \alpha\}>,$$

using a very informal notation. Here $\epsilon$ is an empty statement such that for all statements $s$: $s;\epsilon = \epsilon;s = s$.
The rules tell us how we can deduce new transitions from old ones, e.g.:

for all statements $s_1,s_2,s$ and for all $I_1,I_2$ :

$$\frac{<s_1 \, , I_1> \rightarrow <s_2 \, , I_2>}{< s_1;s \; , I_1> \rightarrow < s_2;s \; , I_2>},$$

meaning: if the upper transition holds, then the lower transition also holds. Rules and axioms together determine a *transition relation* $\rightarrow$ with

$$\rightarrow \; \subset Conf \times Conf,$$

namely the set of all transitions that are provable in the system. That is: the transitions that either are axioms or are deducible from the axioms using the rules.
Given the axiom and the rule mentioned above the following is an example of a deduction in our formal system:

$$\frac{<x \leftarrow 3 \, , I> \rightarrow <\epsilon \, , I \cup \{\text{the value of } x \text{ is } 3\}>}{< x \leftarrow 3 \, ; y \leftarrow 5 \; , I> \rightarrow <y \leftarrow 5 \, , I \cup \{\text{the value of } x \text{ is } 3\}>}.$$

This is an application of our rule, using an instance of our axiom as the premiss and resulting in a new transition as the conclusion of the rule.
Given a certain transition system we consider transition sequences

$$<s_1 \, , I_1> \rightarrow <s_2 \, , I_2> \rightarrow \; \cdots$$

such that for all $n > 0$

$$<s_n \, , I_n> \rightarrow <s_{n+1} \, , I_{n+1}>$$

holds. We shall give a simple example. Another instance of our axiom is :

$$<y \leftarrow 5 \, , I'> \rightarrow <\epsilon \, , I' \cup \{ \text{ the value of } y \text{ is } 5\}>$$

When we take

$$I' = I \cup \{ \text{ the value of } x \text{ is } 3\}$$

we have the following example of a transition sequence:

$$< x \leftarrow 3 \, ; y \leftarrow 5 \, ,I> \rightarrow$$

$$<y \leftarrow 5 \, , I \cup \{\text{the value of } x \text{ is } 3\}> \rightarrow$$

$$<\epsilon \, , I \cup \{\text{the value of } x \text{ is } 3\} \cup \{\text{the value of } y \text{ is } 5\}>.$$

We now are able to give meaning to a program P by defining its semantics as the set of all possible transition sequences

$$<s_P\ , I_0> \rightarrow \ \cdots$$

where $I_0$ is a basic amount of information, which is to be defined precisely and $s_P$ is the program text. ($I_0$ must contain at least the information given by the declarations of the program.)

## 4.2. Definitions

As already mentioned before, the configurations that we shall use will have a more complex structure than the configurations of the introduction. To be more specific, the information component will consist of three distinct parts: a unit, a state and a type function. We have seen in the syntax that a unit contains all necessary information about class definitions. Next we define states.
The set of states, with typical elements $\sigma, \ \cdots$ , is defined by:

$$\Sigma = (AObj \rightarrow (IVar \rightarrow Obj)) \times (AObj \rightarrow (LVar \rightarrow Obj)^*).$$

We denote the first and the second component of $\sigma$ by

$$\sigma = <\sigma_1, \sigma_2>.$$

The first component of a state is used for the values of the instance variables of each (active) object. The second component of a state is used for the values of the local (method) variables of each active method invocation of each object in order to handle recursion. For each state $\sigma$ and each object $\alpha$, $\sigma_2(\alpha)$ denotes an element of the set $(LVar \rightarrow Obj)^*$, which can be seen as a stack of frames. We shall need the following operations upon these stacks. For $f_1, \ \cdots \ ,f_n, f \in (LVar \rightarrow Obj)$, we define:

$$- \ pop(<f_1, \ \cdots \ ,f_n>) = \begin{cases} <f_1, \ \cdots \ ,f_{n-1}> & \text{if } n>1 \\ \epsilon & \text{if } n=1 \end{cases}$$

$$- \ push(f, <f_1, \ \cdots \ ,f_n>) = <f_1, \ \cdots \ ,f_n, f>$$

$$- \ top(<f_1, \ \cdots \ ,f_n>) = f_n \ .$$

The set of type functions, with typical elements $\tau, \ \cdots$ , is defined by

$$Type = AObj \rightarrow CName.$$

These type functions will be used to store for each object its class name, which indicates the class definition that has been used for its creation.

## 4.3. Configurations

We define the set of configurations by:

$$Conf = \mathcal{P}_{fin}(LStat) \times \Sigma \times Type \times Unit \cup \{ \ err \ \}.$$

Here *err* denotes an error configuration. Each other configuration is a quadruple

$$< X \ , \sigma \ , \tau \ , U >$$

consisting of a finite set of labeled statements, a state, a type function and a unit. Let $U$ be

$$U = <C_1 \Leftarrow d_1, \ \cdots \ ,C_n \Leftarrow d_n>$$

for the sequel of this paper. So from now on we suppose that we are studying a fixed unit $U$.
We have seen what kind of information is represented by states and type functions. That leaves us to explain the meaning of the finite set of labelled statements. In general $X$ has the following form:

$$X = \{ <\alpha_1,s_1>, \cdots ,<\alpha_n,s_n>\}.$$

This represents $n$ objects that are active in parallel. These objects have names: $\alpha_1, \cdots ,\alpha_n$. The statement $s_i$ still has to be executed by object $\alpha_i$. For the sets $X$ that we shall encounter in our transition system, the following equality holds:

$$\{\alpha_1, \cdots ,\alpha_n\} = \{\hat{1}, \cdots ,\hat{n}\}.$$

When one of the objects $\alpha_1, \cdots ,\alpha_n$ executes **new**($C_i$), $X$ will be expanded with a new element $<\widehat{n+1},s>$.

### 4.4. The transition system

Our transition relation

$$\rightarrow \ \subset Conf \times Conf$$

will be defined by the following axioms and rules.

#### 4.4.1. Axioms

Throughout the sections 4.4.1 and 4.4.2, whenever we write $X \cup <\alpha , s>$, we require that $<\alpha , s> \notin X$.

1. Creating a new object

$$< X \cup \{<\alpha,\mathbf{new}(C)>\} , \sigma , \tau , U > \rightarrow < X \cup \{<\alpha,\underline{\hat{n}}>,<\hat{n},s>\} , \sigma' , \tau' , U >,$$

where

$$U = < \cdots , C \Leftarrow d , \cdots >,$$
$$d = < (x_1, \cdots ,x_k) , (m_1 \Leftarrow \mu_1, \cdots ,m_l \Leftarrow \mu_l) , s >,$$
$$(\text{so } s = d_3,)$$
$$\sigma' = < \sigma_1\{ \lambda x.nil / \hat{n} \} , \sigma_2 >,$$
$$\tau' = \tau\{C / \hat{n}\},$$
$$n = |X| + 2.$$

● We make heavy use of the *variant notation* for functions. Let $f$ be a function that is defined in $x$. Then $f\{y / x\}$ is defined by

$$f\{y / x\}(p) = \begin{cases} f(p) & \text{if } p \neq x \\ y & \text{if } p = x. \end{cases}$$

When a new object is created the set $X$ is expanded with a new element $<\hat{n},s>$. Note that $n$ is the first number that is not in use for an active object. So the name $\hat{n}$ is new and it is the resulting value for the object $\alpha$, which has created the new object. The new object $\hat{n}$ is set to work and starts the execution of the statement $s$, which is the body of the class definition $d$. In the state $\sigma'$ the values of the instance variables of $\hat{n}$ are set to *nil*. Also the type function $\tau$ changes. The class name $C$ is stored as its value at object $\hat{n}$ . This value will be needed later when $\hat{n}$ wants to execute one of its methods.

2. Assignment

$$< X \cup \{<\alpha, x \leftarrow \underline{\beta}>\}\, , \sigma \, , \tau \, , U > \rightarrow < X \cup \{<\alpha, \underline{\beta}>\}\, , \sigma' \, , \tau \, , U >,$$

where

$$\sigma' = < \sigma_1\{\ (\sigma_1(\alpha)\{\beta / x\})\ / \alpha\ \}\, , \sigma_2 >.$$

● The value of $x$ is set to $\beta$ in $\sigma_1(\alpha)$.

$$< X \cup \{<\alpha, u \leftarrow \underline{\beta}>\}\, , \sigma \, , \tau \, , U > \rightarrow < X \cup \{<\alpha, \underline{\beta}>\}\, , \sigma' \, , \tau \, , U >,$$

where

$$\sigma' = < \sigma_1\, , \sigma_2\{S / \alpha\} >,$$
$$S = < f_1, \ \cdots \ , f_{n-1}, f_n\{\beta / u\} >,$$
$$\sigma_2(\alpha) = < f_1, \ \cdots \ , f_n >.$$

● The top frame $f_n$ of the stack $\sigma_2(\alpha)$ is changed. The value of the local variable $u$ is set to $\beta$.

3. Variables

$$< X \cup \{<\alpha, x >\}\, , \sigma \, , \tau \, , U > \rightarrow < X \cup \{<\alpha, \underline{\beta}>\}\, , \sigma \, , \tau \, , U >,$$

where

$$\beta = \sigma_1(\alpha)(x).$$

$$< X \cup \{<\alpha, u >\}\, , \sigma \, , \tau \, , U > \rightarrow < X \cup \{<\alpha, \underline{\beta}>\}\, , \sigma \, , \tau \, , U >,$$

where

$$\beta = top(\sigma_2(\alpha))(u).$$

$$< X \cup \{<\alpha, \mathbf{self} >\}\, , \sigma \, , \tau \, , U > \rightarrow < X \cup \{<\alpha, \underline{\alpha}>\}\, , \sigma \, , \tau \, , U >.$$

4. Discarding a value

$$< X \cup \{<\alpha, \underline{\beta}; s >\}\, , \sigma \, , \tau \, , U > \rightarrow < X \cup \{<\alpha, s >\}\, , \sigma \, , \tau \, , U >.$$

● A single value can be thrown away. For example when we evaluate $x \leftarrow \underline{\beta}$ the value of this expression is $\beta$. In general we shall want to discard this value.

## 5. Communication

(a) sending a message

$$< X \cup \{<\alpha_1,\textbf{answer}(\ \cdots\ ,m,\ \cdots\ )>,$$
$$<\alpha_2,\underline{\alpha_1}!m(\underline{\beta_1},\ \cdots\ ,\underline{\beta_k})>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow$$
$$< X \cup \{<\alpha_1,e \Rightarrow \underline{\alpha_2}>,<\alpha_2,\textbf{wait}>\}\ ,\ \sigma'\ ,\ \tau\ ,\ U >,$$

where

$$\mu(U,\tau(\alpha_1),m) = < (u_1,\ \cdots\ ,u_k)\ ,\ (v_1,\ \cdots\ ,v_r)\ ,\ e >,$$
$$(\text{so } e = (\mu(U,\tau(\alpha_1),m))_3)$$
$$\sigma' = < \sigma_1\ ,\ \sigma_2\{push\ [f,\sigma_2(\alpha_1)]\ /\alpha_1\} >,$$
$$f(u_i) = \beta_i \quad \text{for } i = 1,\ \cdots\ ,k,$$
$$f(v_i) = nil \quad \text{for } i = 1,\ \cdots\ ,r.$$

● Object $\alpha_2$ sends a message to object $\alpha_1$. The message is accepted and object $\alpha_2$ is set to **wait** until $\alpha_1$ has finished the execution of the method *m*. In order to know where the result of the method has to be returned to, $e \Rightarrow \alpha_2$is substituted in the statement of $\alpha_1$. We can read this as: execute *e* and send the resulting value to $\alpha_2$. Note that object $\alpha_1$ first has to execute the method before it can continue its body.
When the execution of the method *m* is started, a new frame *f* is created, with the parameters $u_1,\ \cdots\ ,u_k$ initialized to the objects $\beta_1,\ \cdots\ ,\beta_k$ in the message, and the local variables $v_1,\ \cdots\ ,v_r$ set to *nil*.This frame is pushed onto the stack $\sigma_2(\alpha_1)$. When the execution of *m* is finished, *f* will be popped.
Note that to apply this axiom, the destination and the parameters of the send expression must have been completely evaluated. For this evaluation the rules 1.(iii) and 1.(iv) can be used.
For the definition of the $\mu$ function, see section 3.3.

(b) returning the result

$$< X \cup \{<\alpha_1,\underline{\beta} \Rightarrow \underline{\alpha_2}>,<\alpha_2,\textbf{wait}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow$$
$$< X \cup \{<\alpha_1,\underline{nil}>,<\alpha_2,\underline{\beta}>\}\ ,\ \sigma'\ ,\ \tau\ ,\ U >,$$

where

$$\sigma' = < \sigma_1\ ,\ \sigma_2\{pop(\sigma_2(\alpha_1))\,/\,\alpha_1\} >.$$

● Object $\alpha_1$ has finished a method execution, the result of which is $\beta$. This value is returned to object $\alpha_2$. Because a method execution has been finished, the last frame must be popped off the stack $\sigma_2(\alpha_1)$.

## 6. Method call

(a) method invocation

$$< X \cup \{<\alpha_1,m(\underline{\beta_1},\ \cdots\ ,\underline{\beta_k})>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow < X \cup \{<\alpha_1,e \Rightarrow \underline{\alpha_1}>\}\ ,\ \sigma'\ ,\ \tau\ ,\ U >,$$

where

$\sigma', e$ as in 5.(a).

● In this case no one has to wait. When object $\alpha_1$ reaches a method invocation, the execution of its body is suspended until the execution of the method will have been finished.

(b) returning the result

$< X \cup \{ <\alpha, \underline{\beta} \Rightarrow \underline{\alpha} > \}, \sigma, \tau, U > \rightarrow < X \cup \{ <\alpha, \underline{\beta} > \}, \sigma', \tau, U >$,

where

$\sigma = < \sigma_1 , \sigma_2\{pop(\sigma_2(\alpha)) / \alpha\} >$.

7. Conditional

$$< X \cup \{ <\alpha, \mathbf{if}\ \underline{\beta}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi} > \}, \sigma, \tau, U > \rightarrow \begin{cases} < X \cup \{ <\alpha, s_1 > \}, \sigma, \tau, U > & \text{if } \beta = tt \\ < X \cup \{ <\alpha, s_2 > \}, \sigma, \tau, U > & \text{if } \beta = ff \\ err & \text{otherwise.} \end{cases}$$

8. Do-statement

$< X \cup \{ <\alpha, \mathbf{do}\ e\ \mathbf{then}\ s\ \mathbf{od} > \}, \sigma, \tau, U > \rightarrow$

$< X \cup \{ <\alpha, \mathbf{if}\ e\ \mathbf{then}\ (s\ ;\ \mathbf{do}\ e\ \mathbf{then}\ s\ \mathbf{od})\ \mathbf{else}\ nil\ \mathbf{fi} > \}, \sigma, \tau, U >$.

9. Axioms for the standard-classes Integer and Boolean

(a)

$$< X \cup \{ <\alpha, \underline{\beta}!op(\underline{\gamma}) > \}, \sigma, \tau, U > \rightarrow \begin{cases} < X \cup \{ <\alpha, (\underline{op(\beta,\gamma)}) > \}, \sigma, \tau, U > & \text{if } \beta,\gamma \in \mathbb{Z} \\ err & \text{otherwise,} \end{cases}$$

for $op$ = add, sub, div, mod, $\cdots$ .

(b)

$$< X \cup \{ <\alpha, \underline{\beta}!rel(\underline{\gamma}) > \}, \sigma, \tau, U > \rightarrow \begin{cases} < X \cup \{ <\alpha, \underline{tt} > \}, \sigma, \tau, U > & \text{if } \beta,\gamma \in \mathbb{Z},\ rel(\beta,\gamma) \\ < X \cup \{ <\alpha, \underline{ff} > \}, \sigma, \tau, U > & \text{if } \beta,\gamma \in \mathbb{Z},\ \neg\, rel(\beta,\gamma) \\ err & \text{otherwise,} \end{cases}$$

for *rel* = equal, less, $\cdots$ .

(c)

$$< X \cup \{<\alpha,\beta!op(\gamma)>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow$$
$$\begin{cases} < X \bar{\cup}\ \{\bar{<}\alpha,\underline{op(\ \beta\ ,\ \gamma\ )}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > & \text{if } \beta,\gamma \in \{tt,ff\} \\ err & \text{otherwise,} \end{cases}$$

for *op* = and, or, $\cdots$ .

(d)

$$< X \cup \{<\alpha,\beta!\mathrm{not}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow$$
$$\begin{cases} < X \bar{\cup}\ \{<\alpha,\ \underline{\neg\ \beta}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > & \text{if } \beta \in \{tt,ff\} \\ err & \text{otherwise.} \end{cases}$$

10. Select-statement

(a)

$$< X \cup \{<\alpha,\mathbf{sel}\ \cdots\ \mathbf{or}\ \underline{tt}\mathbf{answer}(m_1,\ \cdots\ ,m_k)\ \mathbf{then}\ s$$
$$\mathbf{or}\ \cdots\ \mathbf{les}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow$$
$$< X \cup \{<\alpha,\mathbf{sel}\ \cdots\ \mathbf{or\ answer}(m_1,\ \cdots\ ,m_k)\ \mathbf{then}\ s$$
$$\mathbf{or}\ \cdots\ \mathbf{les}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U >.$$

• When we have evaluated the guard of a guarded command (using rule 1.(vi)) and the result is 'true', we omit the syntactical expression *tt*. So a guarded command with no guard is tantamount to a guarded command with a *tt* as its guard.

(b)

$$< X \cup \{<\alpha,\mathbf{sel}\ \cdots\ \mathbf{or}\ g_{i-1}\ \mathbf{or}\ \underline{ff}\mathbf{answer}(m_1,\ \cdots\ ,m_k)\ \mathbf{then}\ s$$
$$\mathbf{or}\ g_{i+1}\ \mathbf{or}\ \cdots\ \mathbf{les}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow$$
$$< X \cup \{<\alpha,\mathbf{sel\ or}\ \cdots\ \mathbf{or}\ g_{i-1}\ \mathbf{or}\ g_{i+1}\ \mathbf{or}\ \cdots$$
$$\mathbf{les}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U >.$$

• When the result of the evaluation of the expression in a guarded command is 'false', we discard the whole guarded command.

(c)

$$< X \cup \{<\alpha,\mathbf{sel}\ \ \mathbf{les}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow err.$$

● When all guards are false, the result is an error configuration.

(d)

$< X \cup \{<\alpha_1, \textbf{sel}\ g_1\ \textbf{or}\ \cdots\ \textbf{or}\ g_i\ \textbf{or}\ \cdots\ \textbf{or}\ g_n\ \textbf{les}>,$

$\qquad\qquad <\alpha_2, \alpha_1 ! m(\gamma_1, \cdots, \gamma_k)>\}, \sigma, \tau, U > \rightarrow$

$< X \cup \{<\alpha_1, e \Rightarrow \alpha_2\ ; s_i>, <\alpha_2, \textbf{wait}>\}, \sigma', \tau, U >$

if for all $t \in \{1, \cdots, n\}$ :

$\quad g_t = \textbf{answer}(m_{t1}, \cdots, m_{tk_t})\ \textbf{then}\ s_t$

and if for all $t \in \{1, \cdots, i-1\}$

$\quad k_t > 0$, that is: $g_t \neq \textbf{answer}(\ )\ \textbf{then}\ s_t$

and

$\quad m \notin \{m_{t1}, \cdots, m_{tk_t}\}$

and if

$\quad m \in \{m_{i1}, \cdots, m_{ik_i}\}$

where

$\quad \mu(U, \tau(\alpha_1), m) = < (u_1, \cdots, u_k), (v_1, \cdots, v_t), e >,$

$\quad \sigma' = < \sigma_1, \sigma_2\{push\ [f, \sigma_2(\alpha_1)]\ / \alpha_1\} >,$

$\quad f(u_j) = \gamma_j$ for $j = 1, \cdots, k,$

$\quad f(v_j) = nil$ for $j = 1, \cdots, t.$

● This axiom can be applied only when all the guards are evaluated. (The 'false' guards have disappeared.) That is the meaning of the first condition. Guarded command $g_i$ can be chosen to be executed if the following three conditions hold. First, no guards with an 'empty' message expression are allowed to the left side of $g_i$. Second, the method name $m$ should not occur in any of the guarded commands to the left side of $g_i$. Finally, the answer expression of $g_i$ must contain the method name $m$. The state $\sigma$ changes in the same manner as in axiom 5.(a). (e)

$< X \cup \{<\alpha_1, \textbf{sel}\ g_1\ \textbf{or}\ \cdots\ \textbf{or}\ \textbf{answer}(\ )\ \textbf{then}\ s_i$

$\qquad\qquad \textbf{or}\ \cdots\ \textbf{or}\ g_n\ \textbf{les}>, \sigma, \tau, U > \rightarrow$

$< X \cup \{<\alpha_1, s_i>\}, \sigma, \tau, U >$

if for all $t \in \{1, \cdots, n\}$ :

$\quad g_t = \textbf{answer}(m_{t1}, \cdots, m_{tk_t})\ \textbf{then}\ s_t$

and if for all $t \in \{1, \cdots, i-1\}$

$\quad k_t > 0$, that is: $g_t \neq \textbf{answer}(\ )\ \textbf{then}\ s_t$.

● A guarded command with an 'empty' message expression can be chosen if all guards are evaluated and if there are no guarded commands with an 'empty' message expression to the left of this guarded command.

### 4.4.2. Rules

$$\frac{1.\ < X \cup \{<\alpha,e>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow < X' \cup \{<\alpha,e'>\}\ ,\ \sigma'\ ,\ \tau'\ ,\ U >}{}$$

$(i)$ $< X \cup \{<\alpha,x \leftarrow e>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow < X' \cup \{<\alpha,x \leftarrow e'>\}\ ,\ \sigma'\ ,\ \tau'\ ,\ U >$

$(ii)$ $< X \cup \{<\alpha,u \leftarrow e>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow < X' \cup \{<\alpha,u \leftarrow e'>\}\ ,\ \sigma'\ ,\ \tau'\ ,\ U >$

$(iii)$ $< X \cup \{<\alpha,e!m(e_1,\ \cdots\ ,e_n)>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow$

$< X' \cup \{<\alpha,e'!m(e_1,\ \cdots\ ,e_n)>\}\ ,\ \sigma'\ ,\ \tau'\ ,\ U >$

$(iv)$ $< X \cup \{<\alpha,\beta!m(\beta_1,\ \cdots\ ,\beta_{i-1},e,\ \cdots\ ,e_n)>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow$

$< X' \cup \{<\alpha,\overline{\beta}!m(\overline{\beta_1},\ \cdots\ ,\overline{\beta_{i-1}},e',\ \cdots\ ,e_n)>\}\ ,\ \sigma'\ ,\ \tau'\ ,\ U >$

$(v)$ $< X \cup \{<\alpha,m(\beta_1,\ \cdots\ ,\beta_{i-1},e,\ \cdots\ ,e_n)>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow$

$< X' \cup \{<\alpha,m(\overline{\beta_1},\ \cdots\ ,\overline{\beta_{i-1}},e',\ \cdots\ ,e_n)>\}\ ,\ \sigma'\ ,\ \tau'\ ,\ U >$

$(vi)$ $< X \cup \{<\alpha,\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow$

$< X' \cup \{<\alpha,\textbf{if}\ e'\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}>\}\ ,\ \sigma'\ ,\ \tau'\ ,\ U >$

$(vii)$ $< X \cup \{<\alpha,\textbf{sel}\ g_1\ \textbf{or}\ \cdots\ \textbf{or}\ g_{i-1}\ \textbf{or}\ (e\ \textbf{answer}(m_{i1},\ \cdots\ ,m_{ik_i})\ \textbf{then}\ s_i)\ \textbf{or}$

$g_{i+1}\ \textbf{or}\ \cdots\ \textbf{or}\ g_n\ \textbf{les}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow$

$< X' \cup \{<\alpha,\textbf{sel}\ g_1\ \textbf{or}\ \cdots\ \textbf{or}\ g_{i-1}\ \textbf{or}\ (e'\ \textbf{answer}(m_{i1},\ \cdots\ ,m_{ik_i})\ \textbf{then}\ s_i)\ \textbf{or}$

$g_{i+1}\ \textbf{or}\ \cdots\ \textbf{or}\ g_n\ \textbf{les}>\}\ ,\ \sigma'\ ,\ \tau'\ ,\ U >,$

if for all $j \in \{1,\ \cdots\ ,i-1\}$

$g_j = \textbf{answer}(m_{j1},\ \cdots\ ,m_{jk_j})\ \textbf{then}\ s_j.$

$(viii)$ $< X \cup \{<\alpha_1,e \Rightarrow \underline{\alpha_2}>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow < X' \cup \{<\alpha_1,e' \Rightarrow \underline{\alpha_2}>\}\ ,\ \sigma'\ ,\ \tau'\ ,\ U >.$

• (iii),(iv)and (v): Evaluation of the expressions in a send expression takes place from left to right.
(vii): The same holds for the guards of a select statement.

$$2.\ \frac{< X \cup \{<\alpha_1,s_1>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow < X' \cup \{<\alpha_1,s_2>\}\ ,\ \sigma'\ ,\ \tau'\ ,\ U >}{< X \cup \{<\alpha_1,s_1;s>\}\ ,\ \sigma\ ,\ \tau\ ,\ U > \rightarrow < X' \cup \{<\alpha_1,s_2;s>\}\ ,\ \sigma'\ ,\ \tau'\ ,\ U >}$$

### 4.5. Fairness conditions

Now, having described the transition relation $\rightarrow$, we shall have to specify what we want to consider as the meaning of a program. In the present case, the meaning of a unit U is the set of all (finite and infinite) sequences of configurations $<c_1,c_2,\ \cdots\ >$ that satisfy the following conditions:
(i) The initial configuration $c_1$ for our transition system will be:

$<\{<\hat{1},s>\}\ ,\ \sigma\ ,\ \tau\ ,\ U>,$

where $U$ is our fixed unit,

$U ::= < C_1 \Leftarrow d_1\ ,\ \cdots,\ C_n \Leftarrow d_n >,$

$d_n = < (x_1,\ \cdots\ ,x_p)\ ,\ (m_1 \Leftarrow \mu_1,\ \cdots\ ,m_k \Leftarrow \mu_k)\ ,\ s >,$

$\sigma_1(\hat{1}) = \lambda x.nil,$

$$\sigma_2(\hat{1}) = \epsilon,$$
$$\tau(\hat{1}) = C_n.$$

(ii) For all $c_i$ and $c_{i+1}$ we have:

$$c_i \rightarrow c_{i+1}.$$

(iii) The whole sequence satisfies the fairness condition. This condition can be intuitively described by saying that when an object is infinitely often allowed to participate in a transition, it will eventually do so. More formally the condition states that a sequence $<c_1,c_2, \cdots >$ is not allowed if there is an object $\hat{n}$ and an infinite set $\{i_0,i_1, \cdots \}$ of indices such that:
- the sequence $<c_1,c_2, \cdots >$ is infinite
- there is a (fixed) statement s such that for all $i > i_0$ the labeled statement $<\hat{n},s>$ is an element of the first component of $c_i$.
- for each $i$ in $\{i_0,i_1, \cdots \}$ there is a configuration $c$ with $c_i \rightarrow c$ such that $<\hat{n},s'>$ is an element of the first component of $c$, with $s'$ different from $s$.

**Remark**
This fairness condition is formulated in the informal language definition [America 1985] by stating that the speed of execution of each individual object is positive, but arbitrary, and by stating the meaning of a select statement in a slightly different way, using queues. The first condition states that if an object can make progress on its own, without cooperation of another object, it will do so after a finite number of steps. The equivalence of the two formulations of the meaning of the select statement relies on the fact that the situation in which an object has sent a message, which has not yet been accepted, is completely indistinguishable from the situation in which the object has fallen to sleep before sending the message, except for the fact that in the latter case the sleeping may not continue forever.

## 5. From minimal to maximal parallelism

In this section we will show how, by slight modifications, we can apply the above technique to express real parallelism: more than one activity going on at the same time. Note that the above transition system treats parallelism as interleaving of atomic actions: "concurrent" actions occur one after another, but in arbitrary order. Now we will present a system that allows parallel actions to occur in one transition.
In order to do that we will make some modifications. First we divide the global state information into the local states for each object. A (global) configuration $X$ will then consist of a set of local configurations. That will make it possible to express the transitions locally. Instead of axioms of the form

$$< X \cup \{ < \alpha , s > \} , \sigma , \tau , U > \rightarrow < X \cup \{ < \alpha , s' > \} , \sigma' , \tau' , U >$$

we can formulate them like this:

$$\{ < \alpha , s , \sigma , \tau , U > \} \rightarrow \{ < \alpha , s' , \sigma' , \tau' , U > \}.$$

Now we can give the rule for parallel composition:

$$\frac{X \rightarrow X' \quad Y \rightarrow Y'}{X \cup Y \rightarrow X' \cup Y'}$$

where the sets of objects in $X$ and $Y$ must be disjoint. Note that in the conclusion of this rule transitions in $X$ and $Y$ occur concurrently. We also need the following rule for the case that the objects in one set do not move:

$$\frac{X \rightarrow X'}{X \cup Y \rightarrow X' \cup Y}$$

where the sets of objects in $X$ and $Y$ must be disjoint.
The second modification is necessary to make it possible for an object to create a new object with a unique name, without having global information about the whole system. Therefore we change the definition of the set of active objects into:

$$AObj = \mathbb{N}^*$$

(the set of finite sequences of natural numbers).
Also each object maintains a count of the objects it has already created. When it creates a new object its counter is incremented. The name of the new object will consist of the name of its creator with a new counter appended to the end.
To be more specific, we show some of the modified definitions. For the local states the new definition of $\Sigma$ will be:

$$\Sigma = (\, IVar \rightarrow Obj \,) \times (\, LVar \rightarrow Obj \,)^* \times \mathbb{N}$$

where the last component represents the creation counter.
For typing information we will simply use $\tau \in CName$. A configuration $X \in Conf$ will now consist of a finite set $\{ l_1 , \cdots , l_n \}$ of local configurations $l \in LConf$, with $l$ of the form $< \alpha , s , \sigma , \tau , U >$.
We also give one example of a modified axiom:

$$\{ < \alpha , \textbf{new}\,(C) , \sigma , \tau , U > \} \rightarrow \{ < \alpha , \beta , \bar{\sigma} , \tau , U > , < \beta , s , \sigma' , \tau' , U > \}$$

where $\beta = \alpha \cdot < \sigma_3 >$ ( $\cdot$ stands for concatenation),
$\bar{\sigma} = < \sigma_1 , \sigma_2 , \sigma_3 + 1 >$,
$s$ is the body of class $C$:
$U = < \cdots , C \Leftarrow d , \cdots >$,
$d = < ( x_1 , \cdots , x_n >, ( m_1 \Leftarrow \mu_1 , \cdots , m_k \Leftarrow \mu_k ) , s >$,
$\sigma' = < \lambda x \,.\, nil , \epsilon , 0 >$,
$\tau' = C$.
The modifications to the other axioms and rules are straightforward.
The initial configuration for a unit $U$
with $U = < C_1 \Leftarrow d_1 , \cdots , C_n \Leftarrow d_n >$,
$d_n = < ( x_1 , \cdots , x_k ) , ( m_1 \Leftarrow \mu_1 , \cdots , m_p \Leftarrow \mu_p ) , s >$
is now

$$\{ < \epsilon , s , \sigma , \tau , U > \}$$

where $\epsilon \in N^*$ is the empty sequence,
$\sigma = < \lambda x \,.\, nil , \epsilon , 0 >$,
$\tau = C_n$.
The fairness condition can also easily be reformulated.
Note that with this formalism we can even express the notion of *maximal parallelism* (introduced in [Salwicki & Müldner 1981 ]; in [Koymans et al 1985] this notion is treated in a denotational semantics for a real-time language): we allow no set of objects that *can* make a move to remain silent in a transition. More formally, we might require for the sequences $< X_1 , X_2 , \cdots >$ that constitute the semantics of a unit, that for *no* index $i$ there are $X$, $X'$, $Y$ and $Y'$ such that $X_i = X \cup Y$, $X_{i+1} = X' \cup Y$ and $Y \rightarrow Y'$. In our opinion, this is a very natural way of expressing maximal parallelism. However, we do not apply this restriction to the *standard* semantics of POOL.

## 6. Conclusions

After having presented the transition system semantics for POOL, this is the place to compare the technique used with the more common ones [De Bakker et al 1984]. First there is clearly a difference in the amount of state information that must be carried around in the configurations. Our configurations are certainly more complex than the ones cited above, but notice that we give a full semantics of a realistic language, where the contents of variables, and typing and declaration information are relevant. We think that this justifies the additional complexity.

A more important difference lies in the way in which we treat parallelism. In the referenced systems, this was mainly expressed by the parallel composition operator ||, which was treated just like other constructs like sequential composition (;) and conditional **if then else fi**. In this way, however, all the interactions of the operator || with other operators must be explicitly mentioned in the axioms and rules of the system. (It turns out that there *is* a great deal of interaction between || and the other operators, while the other operators hardly interact among themselves.) This results in a very complex system, in which it is a non-trivial task to ensure that it yields the intended results.

In our system, we represented several concurrently executing processes as a *set*. In this way commutativity and associativity of parallel composition is automatically guaranteed. Also the interaction between parallel composition and the other constructs can easily be described, without the need for a number of axioms and rules that grows quadratically with the number of constructs. Finally, note that in our system it is very easy to describe dynamic process creation, where the new process, created possibly very deeply nested in an existing process, begins to execute in parallel with all other processes in the system, including its own creator. It is easily imagined how difficult it is to describe such a mechanism in a system like the cited ones. Furthermore, we saw that in our approach it is very easy to express the maximal parallelism condition. We think that these advantages of the current system outweigh the cosmetic disadvantage of treating the parallel composition in an asymmetrical way with respect to the other constructs.

Finally let us point out some directions in which further work could be done. First, it would certainly be worthwhile to see whether for this kind of language also a denotational semantics can be developed, and possibly proved equivalent to the current operational semantics (like the work done in [De Bakker et al 1984]). Maybe the representation used here for parallel processes could be adapted to denotational semantics in such a way that a clear description is possible. Finally, this kind of operational semantics could be a good basis to explore the possibility of automatic implementation of parallel languages by means of a interpreter.

## 7. REFERENCES

[America 1985] Pierre America: Definition of the programming language POOL-T, ESPRIT project 415, Doc Nr. 0091, Philips Research Laboratories, Eindhoven, the Netherlands, June 1985.

[Apt 1981] K.R. Apt: Recursive Assertions and Parallel Programs, Acta Informatica 15, pp. 219-232, 1981.

[Apt 1983] K.R. Apt: Formal Justification of a Proof System for Communicating Sequential Processes, Journal Assoc. Comput. Machn., 30(1), pp. 197-216, 1983.

[ANSI 1983] Reference Manual for the Ada Programming Language. ANSI / MIL-STD 1815 A., United States Department of Defense, Washington D.C., January 1983.

[De Bakker et al 1984] J.W. De Bakker, J.-J.Ch. Meyer, E.-R. Olderog & J.I. Zucker: Transition Systems, Infinitary Languages and the Semantics of Uniform Concurrency, Report nr. IR-95, Department of Mathematics and Computer Science, Free University of Amsterdam, November 1984.

[Goldberg & Robson 1983] Adele Goldberg & David Robson: Smalltalk-80, The Language and its

Implementation, Addison-Wesley 1983.

[Hennessy & Plotkin 1979] M. Hennessy, G.D. Plotkin: Full Abstraction for a simple Parallel Programming Language, Proceedings 8th MFCS, LNCS 74, pp. 108-120, Springer, Berlin/New York, 1979.

[Hewitt 1977] C. Hewitt: Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, 8, 1977.

[Koymans et al 1985] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, S. Arun-Kumar: Compositional semantics for real-time distributed computing to appear in: Proceedings of "Logic of Programs" 1985, Brooklyn, Springer Lecture Notes in Computer Science.

[Plotkin 1981] G.D. Plotkin: A Structural Approach to Operational Semantics, DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[Plotkin 1983] G.D. Plotkin: An Operational Semantics for CSP, in: D. Bjørner (ed.), Formal Description of Programming Concepts II, North-Holland, Amsterdam, pp.199-223.

[Salwicki & Müldner 1981] A.Salwicki & T.Müldner: On the algorithmic properties of concurrent programs, LNCS 125, Springer Verlag 1981.