KYBERNETIKA — VOLUME *19* (1983), NUMBER 3

DENOTATIONAL SEMANTICS OF PARALLEL PROGRAMMING LANGUAGES

PETER BREZÁNY

The concise explanation of the principles and then the development and application of denotational semantic for a class of parallel programming languages is given in the paper. The semantics of monitor type language constructs, which serve the synchronization of concurrent processes, is expressed by a designed system of semantic domains and semantic functions. The developed model is applied to a concrete parallel programming language.

1. INTRODUCTION

Parallelism has become an increasingly important topic of research in programming. The effective utilization of contemporaneous computer systems is conditioned by its successful management. The important trend in this research is the development of parallel programming languages which must meet many relatively pretentious criteria. In the process of design, implementation and using of these languages the precise definition of their semantics has an important role. In the last years a great deal of progress has been made towards the development of a theoretical framework appropriate to the formal analysis and the specification of the semantical aspects of computer languages.

In the existing approaches to the formalization of the semantics only little attention has been devoted to the parallel programming languages. This paper deals with problems of denotational semantics definition of parallel features of the subset of the language *CL*/1 (Concurrent Language 1). This language belongs to the class of parallel languages, which contain constructs for the description of concurrent processes synchronized by monitors.

The method of denotation semantics combines a powerful and lucid descriptive notation with an elegant and rigorous theory. This paper is aimed at the descriptive technique without going into background mathematics at all. A similar approach describing basic ideas of denotational semantic can be found in [4], [7].

2. DENOTATIONAL SEMANTICS

An essential part of our formalization will be the defining of various sets which have according to Scott's criteria features of complete lattices. We shall call these sets domains. At the application in practice we abstract from lattice features and intuitively domains will be thought of just like sets and therefore we shall use the normal set theoretic notation on them. For example $\{x \mid P(x)\}$ is the set of all x's satisfying the condition $P(x)$; $x \in \boldsymbol{S}$ means x belongs to $\boldsymbol{S}$; $f : \boldsymbol{S}_1 \to \boldsymbol{S}_2$ means f is a function from $\boldsymbol{S}_1$ to $\boldsymbol{S}_2$.

Denotational semantics gives a functional correspondence between syntactic constructs of a programming language and abstract mathematical entities, representing their meanings (semantical values). The language constructs, appearing in programs, are elements of syntactic domains. Each element of the syntactic domain is mapped by a suitable semantic function to the element of the semantic domain. The semantic functions are defined by the system of mutual recursive equations. The semantic equation expresses semantic interpretation of the considered element (phrase of the language) of the syntactic domain by means of the meanings of its components (subphrases), which are also elements of syntactic domain of the considered language.

3. DEFINING DOMAINS

In this section the symbols $\boldsymbol{D}$, $\boldsymbol{D}'$, $\boldsymbol{D}_1$, $\boldsymbol{D}_2$, ... etc. will stand for arbitrary domains. We suppose that each domain $\boldsymbol{D}$ contains an error element $\mathbf{err}_{\boldsymbol{D}}$; we shall drop the domain subscript and allow the proper domain to be determined from the context. **err** is called the improper element. All other elements are called proper.

3.1. Standard domains

The following domains are standard and will be used without further explanations:

numbers: **Num** $= \{0, 1, 2, \ldots\} \cup$ **err**
truth values: **B** $= \{true, false\} \cup$ **err**
identifiers: **Id** $= \{I \mid I$ is a string of letters or digits beginning with a letter$\} \cup$ **err**

$\cup$ is the operator of the set union.

3.1. Finite domains

Finite domains will be defined explicitly by listing their elements in the way we defined the domain **B**. In the following parts we shall be listing only proper domain elements and we shall assume that each domain $\boldsymbol{D}$ contains the error element $\mathbf{err}_{\boldsymbol{D}}$ too.

3.3. Derivations of domains by domain expressions

3.3.1. Function space $[D_1 \to D_2]$

$[D_1 \to D_2]$ is the domain of functions from D_1 to D_2.

$[D_1 \to D_2] = \{f \mid f : D_1 \to D_2\}$

D_1 is called the source of f and D_2 the target of f. Functions which source or target contains functions are called higher order functions.

We may write $D_1 \to D_2$ instead of $[D_1 \to D_2]$. By convention $\to$ associates to the right so, for example,

$$D_1 \to D_2 \to D_3 \to D_4 \quad \text{means} \quad [D_1 \to [D_2 \to [D_3 \to D_4]]] .$$

3.3.2. Product $[D_1 \times D_2 \times \ldots \times D_n]$

$[D_1 \times D_2 \times \ldots \times D_n]$ denotes the product of the domains $D_1, D_2, \ldots, D_n$, i.e. the domain of n-tuples $d_1, d_2, \ldots, d_n$ where $d_i \in D_i$. If $d \in [D_1 \times D_2 \times \ldots \times D_n]$ then $d \downarrow i$ is the ith coordinate of d.

$$[D_1 \times D_2 \times \ldots \times D_n] = \{(d_1, d_2, \ldots, d_n) \mid d_1 \in D_1, d_2 \in D_2, \ldots, d_n \in D_n\}$$

$(d_1, d_2, \ldots, d_n) \downarrow i = d_i$ (we automatically assume $0 < i < n + 1$)

D^n is the domain of all n-tuples of elements of D.

If $f : [D_1 \times D_2 \times \ldots \times D_n] \to D$ then there is an equivalent function of type $D_1 \to D_2 \to \ldots \to D_n \to D$.

3.3.3. Sum $[D_1 + D_2 + \ldots + D_n]$

$[D_1 + D_2 + \ldots + D_n]$ denotes the disjoint union of the domains $D_1, D_2, \ldots, D_n$. Each member of $[D_1 + D_2 + \ldots + D_n]$ corresponds to exactly one member of some D_i.

3.3.4. Sequences D^*

D^* is the domain of all finite sequences of elements of D. If $d \in D^*$ then either d is the empty sequence ( ), or $d = (d_1, d_2, \ldots, d_n)$ where $n > 0$ and each d_i is a member of D, i.e. $D^* = D^0 + D^1 + \ldots + D^n + \ldots$.

D^0 is equivalent to the empty sequence ( ). $d \dagger i$ is the sequence consisting of all but the first i elements of d.

$$(d_1, d_2, \ldots, d_{i-1}, d_i, d_{i+1}, \ldots, d_n) \dagger i = (d_{i+1}, \ldots, d_n) .$$

If $d = (d_1, d_2, \ldots, d_n)$, $d' = (d'_1, d'_2, \ldots, d'_n)$ then $d \circ d' = (d_1, d_2, \ldots, d_n, d'_1, d'_2, \ldots, d'_n)$.

4. DEFINING FUNCTIONS

For defining functions we shall use λ-notation.

Suppose $E(x)$ is some expression involving x such that whenever $d \in \boldsymbol{D}$ is substituted for x, the resulting expression $E(d)$ denotes a member of $\boldsymbol{D}'$. For example if both $\boldsymbol{D}$ and $\boldsymbol{D}'$ are **Num** then $E(x)$ could be $x + 1$. For such expressions the notations:

$$\lambda x \,.\, E(x)$$

denotes the function $f : \boldsymbol{D} \to \boldsymbol{D}'$ such that for all $d \in \boldsymbol{D}$ is valid: $fd = E(d)$. For example $\lambda x \,.\, x + 1$ denotes the function of type **Num** → **Num**.

An expression of the form $\lambda x \,.\, E(x)$ is called λ-expression, x is its bound variable and $E(x)$ its body. The body always extends as far to the right as possible, thus $\lambda x \,.\, x + 1$ is $\lambda x \,.\, (x + 1)$ not $(\lambda x \,.\, x) + 1$.

Just as we can form expressions like "$f1$" to denote the application of f to 1 so we can form expressions in which λ-expressions are applied to arguments, for example: $(\lambda x \,.\, x + 1)\, 2 = 2 + 1 = 3$.

When evaluating $(\lambda x \,.\, E(x))$ a to $E(a)$ one must only substitute a for those occurrences of x in $E(x)$ which are not bounded by inner λ's. For example $(\lambda x \,.\, (\lambda x \,.\, x + 1))\, a$ evaluates to $\lambda x \,.\, x + 1$ not $\lambda x \,.\, a$.

If $b \in \boldsymbol{B}$, $d_1, d_2 \in \boldsymbol{D}$ the conditional term $b \to d_1, d_2$ signifies the value d_1 when b is *true* and the value d_2 when b is *false*. For example $\lambda x \,.\, (x = 0) \to \mathit{true}, \mathit{false}$ denotes the test-for-zero function of the type **Num** → **B**.

$$b_1 \to d_1 \,,\quad b_2 \to d_2, \ldots, b_n \to d_n,\ d_{n+1} \text{ means}$$
$$b_1 \to d_1 \,,\quad (b_2 \to d_2 \,,\quad (\ldots (b_n \to d_n, d_{n+1}) \ldots)) \,.$$

If $f : \boldsymbol{D} \to \boldsymbol{D}'$, $d_1, \ldots, d_n \in \boldsymbol{D}$ and $d'_1, \ldots, d'_n \in \boldsymbol{D}'$ then $f[d'_1, \ldots, d'_n / d_1, \ldots, d_n]$ denotes the function identical to f except at $d_1, \ldots, d_n$ where it has values $d'_1, \ldots, d'_n$ respectively.

Sometimes it is convenient to structure expressions by writing

$$\text{let } x_1 = E_1 \,,\quad x_2 = E_2 \ldots, x_n = E_n \quad \text{in} \quad E(x_1, x_2 \ldots, x_n)$$

instead of $E(E_1, E_2, \ldots, E_n)$.

Fixed points of functional equations solve recursions. With $\mathit{fix}\,(F)$ we denote the least fixed point f_0 of the function F so that the equation $f_0 = F(f_0)$ is valid.

5. METHODOLOGY OF THE APPROACH

The idea of a denotational semantics for a language is perfectly well illustrated by the contrast between numerals on the one hand and numbers on the other in [8]. The numerals are expressions in a certain language; while the numbers are mathematical (abstract) objects.

In more detail we may consider the following explicit syntax for binary numerals

$$\nu ::= \mathbf{0} \mid \mathbf{1} \mid \nu\mathbf{0} \mid \nu\mathbf{1} .$$

Here we have used the Greek letter ν as a metavariable over the syntactical category (domain) of numerals which we call **Nml**. We introduce a semantical interpretation function $\vartheta : \mathbf{Nml} \to \mathbf{Num}$ which to each $\nu \in \mathbf{Nml}$ assigns the function value $\vartheta[\![\nu]\!]$ which is the number denoted by ν. Inasmuch as ν to be defined on a recursively defined set **Nml**, it is reasonable that ϑ itself should be given a recursively definition by the four following semantic equations

$$\begin{aligned} \vartheta[\![\mathbf{0}]\!] &= 0 \\ \vartheta[\![\mathbf{1}]\!] &= 1 \\ \vartheta[\![\nu\mathbf{0}]\!] &= 2 \cdot \vartheta[\![\nu]\!] \\ \vartheta[\![\nu\mathbf{1}]\!] &= 2 \cdot \vartheta[\![\nu]\!] + 1 . \end{aligned}$$

In the semantic equations we shall enclose the object language expressions in special brackets $[\![\ ]\!]$.

In specifying a syntax we shall provide only an abstract form of a syntax which specifies the compositional structure of programs while leaving open some aspects of their concrete representations as strings of symbols.

Let us suppose, we have a simple programming language, in which **Com** is defined as a syntactic domain of commands and **Exp** as a syntactic domain of expressions; their typical elements we denote Γ and E respectively. Let the grammar includes:

$$\begin{aligned} \Gamma &::= \ldots \mid \Gamma_1 ; \Gamma_2 \mid \ldots \\ E &::= \ldots \mid \mathbf{succ}\ E \mid \ldots \end{aligned}$$

so that a command may be also a sequence of two others commands and the expression **succ** E means adding one to the value of the expression E.

Let $\mathscr{C}$ denotes the semantic function for the evaluation of commands, $\boldsymbol{S}$ denotes the semantic domain of an abstract store states and $\sigma \in \boldsymbol{S}$. Considering a command to specify a transformation on a state σ, we can write $\mathscr{C}[\![\Gamma]\!] \in [\boldsymbol{S} \to \boldsymbol{S}]$ that is, the meaning of command execution is the transformation of a store state. When the store state is σ_1 before an execution of some command Γ, it is σ_2 after its execution. Thus we can write $\mathscr{C}[\![\Gamma]\!]\,\sigma_1 = \sigma_2$.

Then ignoring the possibilities of *goto*'s, errors and nontermination within commands, the natural meaning of $\Gamma_1 ; \Gamma_2$ would be the composition of effects of $\mathscr{C}[\![\Gamma_1]\!]$ and $\mathscr{C}[\![\Gamma_2]\!]$. The semantic equation for this is

$$\mathscr{C}[\![\Gamma_1 ; \Gamma_2]\!] = \lambda\sigma \,.\, \mathscr{C}[\![\Gamma_2]\!]\,(\mathscr{C}[\![\Gamma_1]\!]\,\sigma) .$$

This semantic equation can be interpreted as follows: if σ is the store state before the executing of Γ_1 first one new store state σ_1 is formed as the effect of executing Γ_1; the following executing of Γ_2 causes the new transformation to the state σ_2. Thus

the function $\mathscr{C}$ can be specified

$$\mathscr{C} : \mathbf{Com} \to [\boldsymbol{S} \to \boldsymbol{S}]$$

or without the brackets

$$\mathscr{C} : \mathbf{Com} \to \boldsymbol{S} \to \boldsymbol{S} .$$

The nature of the domain $\boldsymbol{S}$ varies from language to language but it must always contain at least enough information to give the contents of all the abstract store locations in use; it must therefore include a component with functionality $\boldsymbol{L} \to \boldsymbol{V}$. $\boldsymbol{L}$ is the domain of locations. Their contents determine the abstract store state. $\boldsymbol{V}$ is the domain of all storable values.

Dynamic allocation and deallocation of locations require a more complex model of the abstract store. The locations must be partitioned into "active" and "inactive" areas, as follows:

$$\boldsymbol{S} = \boldsymbol{L} \to [\boldsymbol{T} \times \boldsymbol{V}] .$$

Each location has associated (in addition to the usual stored value) with a truth value "tag" to record whether it is active or inactive in the current store state.

Programming languages allow names chosen by the programmer to stand for or denote certain objects. The relationship between the name and the thing it denotes is a function of type $\mathbf{Id} \to [\boldsymbol{D} \times \mathbf{Typ}]$ which is an element of the semantic domain of environments. This domain will be denoted by **Env** and ϱ will stand for an individual environment. $\boldsymbol{D}$ is the domain of denoted values and **Typ** is the domain of their types. In many semantical definitions **Env** is simply defined as $\mathbf{Id} \to \boldsymbol{D}$. However, this is insufficient for us when we want to express semantics of an abstract data types mechanism and the meaning of a data type compatibility.

We can specify the function $\mathscr{C}$ now as

$$\mathscr{C} : \mathbf{Com} \to \mathbf{Env} \to \boldsymbol{S} \to \boldsymbol{S} .$$

The effect of executing $\Gamma_1; \Gamma_2$ can be understood now as

$$\mathscr{C}[\![\Gamma_1; \Gamma_2]\!]\,\varrho = \lambda\sigma \,.\, \mathscr{C}[\![\Gamma_2]\!]\,\varrho(\mathscr{C}[\![\Gamma_1]\!]\,\varrho\sigma) .$$

It is appropriate to say here that in many languages (e.g. in Algol 60) the name of a variable denotes a location in the context of some environment. This location remains fixed throughout the scope of the name. The ordinary value associated with the name is the content of this location. The location is sometimes known as the *L*-value of the name and its content is called the *R*-value.

Executing an expression is intended primarily to produce a value. This value may be used in other expressions, commands or declarations; it may clearly depend on both the environment and the store, so that we might expect to find the valuations appropriate for expressions to be specified

$$\mathscr{E} : \mathbf{Exp} \to \mathbf{Env} \to \boldsymbol{S} \to \boldsymbol{E}$$

where $\boldsymbol{E}$ is the domain of expressed values. For languages allowing possibilities of side effects we must this specification modify to

$$\mathscr{E} : \mathbf{Exp} \to \mathbf{Env} \to \boldsymbol{S} \to [\boldsymbol{E} \times \boldsymbol{S}] .$$

For example $\mathscr{E}[\![\mathbf{succ}\ \boldsymbol{E}]\!]\,\varrho\sigma = \mathscr{E}[\![\boldsymbol{E}]\!]\,\varrho\sigma + 1$.

The main purpose of a declaration is to form, or alter an environment, in which part of a program is to be executed. A declaration may have a side effect, therefore the semantic function $\mathscr{D}$ evaluating declarations can be specified now as

$$\mathscr{D} : \mathbf{Dec} \to \mathbf{Env} \to \boldsymbol{S} \to [\mathbf{Env} \times \boldsymbol{S}] .$$

A continuation semantics is a special sort of denotational semantics. It enables us to model an abnormal flow of control through the program (e.g. jumps, subroutine calls, coroutines, concurrent programs). In the continuation semantics we make the denotations of constructs depend on the rest of the program – or continuation – following them. The intuitive idea is that each construct desides for itself where to pass its result. Usually it will pass it to the continuation corresponding to the "code" textually following it in the program – the normal continuation – but in some cases it will be ignored and the result passed to some other abnormal continuation.

The execution of a command Γ in the presence of the store state σ and the environment ϱ merely replaces σ by σ'. The part of the program following Γ is executed in a semantic context that differs from the one in which Γ was executed because the state of the store has changed. There should therefore be a function μ, which maps store states into "*answers*" and which is built from the part of the program following Γ in such a way that $\mu\sigma'$ is the answer obtained by first executing Γ and then continuing with the rest of the program. It is natural to write this answer $\mu\sigma'$ as $\mathscr{C}[\![\Gamma]\!]\,\varrho\mu\sigma$ and to assume that $\mathscr{C}$ is specified as

$$\mathscr{C} : \mathbf{Com} \to \mathbf{Env} \to [\boldsymbol{S} \to \boldsymbol{A}] \to \boldsymbol{S} \to \boldsymbol{A}$$

$\boldsymbol{A}$ is a domain of answers i.e. the anticipated results of the whole programs, in which commands are embedded. Answers can include store states, expressible values and other possible results such as environments, error messages and so on. The exact structure of $\boldsymbol{A}$ is language dependent.

What happens on executing the part of a program that follows a command can therefore be mimicked by a function belonging to the domain $\boldsymbol{S} \to \boldsymbol{A}$; such a function will be termed a command continuation. After letting $\boldsymbol{C} = \boldsymbol{S} \to \boldsymbol{A}$, $\mathscr{C}$ is specified as

$$\mathscr{C} : \mathbf{Com} \to \mathbf{Env} \to \boldsymbol{C} \to \boldsymbol{C} .$$

The semantic value of $\mathscr{C}[\![\Gamma]\!]\,\varrho\mu$ is the continuation which models what happens if Γ is executed in the environment ϱ before the rest of the program (as mimicked by μ) is executed. Thus $\mathscr{C}[\![\Gamma]\!]\,\varrho\mu\sigma$ should be the answer obtained by supplying the store state σ when Γ is followed by the continuation μ. If the execution of Γ terminates

without errors or jumps occurring the store state produced is passed on to μ; under all other circumstances μ is displaced by a different continuation.

The effect of executing $\Gamma_1; \Gamma_2$ can be understood now as

$$\mathscr{C}[\![\Gamma_1; \Gamma_2]\!]\,\varrho\mu = \mathscr{C}[\![\Gamma_1]\!]\,\varrho\{\mathscr{C}[\![\Gamma_2]\!]\,\varrho\mu\}\,.$$

Braces are used to mark complex continuation arguments as a notational aid.

Executing an expression can produce a result in $\boldsymbol{E}$ along with an altered store, so an expression continuation, which represents what happens on executing the part of a program that follows an expression, yields an answer in $\boldsymbol{A}$. We take the domain of expression continuations $\boldsymbol{K}$ to be $[\boldsymbol{E} \times \boldsymbol{S}] \to \boldsymbol{A}$ or rather $\boldsymbol{E} \to \boldsymbol{S} \to \boldsymbol{A}$; since we set $\boldsymbol{K} = \boldsymbol{E} \to \boldsymbol{C}$. Accordingly we can write

$$\mathscr{E} : \mathbf{Exp} \to \mathbf{Env} \to \boldsymbol{K} \to \boldsymbol{C}\,.$$

Since declarations pass an environment, together with a possibly changed state of the store, to the rest of the program following them we define $\boldsymbol{X}$ – the domain of declaration continuations by $\boldsymbol{X} = \mathbf{Env} \to \boldsymbol{S} \to \boldsymbol{A}$; since we set $\boldsymbol{X} = \mathbf{Env} \to \boldsymbol{C}$. Accordingly we can write:

$$\mathscr{D} : \mathbf{Dec} \to \mathbf{Env} \to \boldsymbol{X} \to \boldsymbol{C}\,.$$

6. MONITORS

An operation sequence described by a given part of a program is called a *process*. When a program is organized in a way that entails executing two portions of it simultaneously it is said to be *concurrent*. A language which enables to create concurrent programs is said to be *parallel*.

An interaction in two ways may originate between concurrent processes:

1. directly so that they are sharing or interchanging data, information, data structures and so on;
2. indirectly so that they are competing for the same resources.

The modular language construct – *monitor*, was developed [3] to master the conflicts between concurrent processes. From a language theory point of view a monitor is a facility for defining abstract data types. It is a collection of data and procedures, shared by several processes in a concurrent program. Syntactically, monitor declarations start with the reserved word monitor, followed by data and procedure declarations, and end with a part which initializes the declared data. Real shared components are represented by *monitor objects* which are defined and initialized by appertaining language constructs.

Access to a monitor object is restricted to calls to its procedures, written "I_{objm} . I_{proc}(... *parameters* ...)". I_{objm} is the name of the monitor object and I_{proc} identifies which of the monitor's procedures is to be called. Only the procedure bodies and the initialization part may refer to the monitor's data.

Usually a monitor object is known to several processes and is used to allocate

resources among or to communicate between them. To insure the orderly use, entry to a monitor object is subject to mutual exclusion. Calls to a monitor object are served one at a time; if while the object is processing a call a second call to the same object occurs, the second caller waits in a queue until the object finishes with the first call.

A monitor's procedures may schedule their actions upon and replies to calls by I_{cond} . **wait** and I_{cond} . **signal** operations. I_{cond} is a queue of defferred calls, declared in the monitor's data in the form "I_{cond} : **condition**". The I_{cond} . **wait** operation places the currently calling process into the queue I_{cond}, suspends its processing and releases the monitor's mutual exclusion so that other calls may be accepted. The I_{cond} . **signal** operation restarts processing of the first call waiting in I_{cond}.

In a concurrent program the execution of one portion of the program may be interleaved with the execution of another at the end of any of the "indivisible operations". Such operations may not be displayed overtly in the syntax of the language but they become apparent in the semantic equations.

The using of continuations enables us to model the execution of concurrent programs. But up to now there have appeared only a few works (e.g. [5], [6]) that form a theoretical basis on which it is possible to define the semantics of many constructs of parallel programming languages. In the literature no work has been dealing so far with the denotational semantics of monitors.

7. FORMAL DEFINITION OF THE SEMANTICS OF THE LANGUAGE *CL*/1

At the designing of the language *CL*/1 we have come out from the language Concurrent Pascal [3]. The complete definition of *CL*/1 is given in the work [1]. In this paper we present only those parts of the definition, which are connected with the parallel control structure of the language.

Syntactic domains and their elements:

$\Gamma \in \mathbf{Com}$	commands
$\Gamma_p \in \mathbf{Com}_p$	parallel commands
$\Gamma_d \in \mathbf{Com}_d$	delaying commands
$\Gamma_s \in \mathbf{Com}_s$	signalling commands
$E \in \mathbf{Exp}$	expressions
$\Delta V \in \mathbf{Dec}_v$	declarations of basic type variables
$\Delta\ Cond \in \mathbf{Dec}_{cond}$	declarations of condition variables
$\Delta\ Abs \in \mathbf{Def}_{abs}$	declarations of monitor type abstractions
$\Delta\ Obj_m \in \mathbf{Dec}_{obm}$	declarations of monitor objects
$\Delta\Gamma_m \in \mathbf{Def}_{pm}$	declarations of monitor procedures
$In_a \in \mathbf{Init}_a$	initialization parts of monitor type abstractions
$In \in \mathbf{Init}$	initializations of monitor objects

$\Pi \in$ **Par**	formal parameters
$P_f \in$ **Prist**$_f$	formal access rights
$P_a \in$ **Prist**$_a$	actual access rights
$T \in$ **Type**	types
$I \in$ **Id**	identifiers

Abstract syntax:

$$
\begin{aligned}
\Gamma &::= \ldots \mid \Gamma_1; \Gamma_2 \mid \textbf{begin } \Delta V^*; \Delta\, Abs^*; \Delta\, Obj_m^*; I_n; \Gamma \textbf{ end} \\
&\quad \mid \textbf{cobegin } \Gamma_{p1}, \Gamma_{p2}, \ldots, \Gamma_{pn} \textbf{ coend} \mid \ldots \\
\Gamma_p &::= \ldots \mid I := E \mid \textbf{if } E \textbf{ then } \Gamma_{p1} \textbf{ else } \Gamma_{p2} \mid \textbf{while } E \textbf{ do } \Gamma_p \\
&\quad \mid \Gamma_{p1}; \Gamma_{p2} \mid \textbf{begin } \Delta V^*; \Gamma_p \textbf{ end} \mid I_{objm} \cdot I_{proc}(E^*) \\
\Gamma_d &::= \Lambda \mid \textbf{if } E \textbf{ then } I_{cond} \cdot \textbf{wait} \\
\Gamma_s &::= \Lambda \mid I_{cond} \cdot \textbf{signal} \\
\Delta V &::= I_v : T \\
\Delta\, Cond &::= I_{cond} : \textbf{cond} \\
\Delta\ \ Abs &::= \textbf{monitor } I_m(P_f^*); \textbf{ begin } \Delta V^*; \Delta\, Cond^*; \Delta F_m^*; \textbf{ begin } In_a \textbf{ end end} \\
\Delta\, Obj_m &::= \textbf{var } I_{objm} : I_m \\
\Delta F_m &::= \textbf{procm } I_{pm}(\Pi^*); \textbf{ begin } \Delta V^*; \Gamma_d; \Gamma_p; \Gamma_s \textbf{ end} \\
In_a &::= \ldots \mid In \mid \ldots \\
In &::= \textbf{init } I_{obj}(P_a^*) \mid In_1; In_2 \mid \Lambda \\
\Pi &::= I : T \mid \textbf{var } I_{objm} : I_m \\
P_f &::= I : T \mid \textbf{var } I_{objm} : I_m \\
P_s &::= I_{obj} \mid E \\
T &::= \textbf{integer} \mid \textbf{real} \mid \textbf{boolean}
\end{aligned}
$$

The introduced syntax does not specify elements of the syntactic domains **Exp** and **Id**. We distinguish the different elements of the same domain by means of indices. The asterisk "*" denotes the operation of creating a list, which is semantically evaluated by the sequential application of an appropriate semantic function on the particular elements of this list. Λ denotes a null category. For the creation of n concurrent control paths there is designed the language construct
cobegin $\Gamma_{p1}, \Gamma_{p2}, \ldots, \Gamma_{pn}$ **coend**.

Semantic domains:

$\boldsymbol{T}$		booleans
$\boldsymbol{N}$		integers
$\boldsymbol{R}$		reals
$\alpha \in \boldsymbol{L}$		locations
$\delta \in \boldsymbol{D}$	$= \boldsymbol{L} + \boldsymbol{V} +$ **Ids**$_{cond}$ + **Abs** + **Proc**$_m$	denoted values
$\varepsilon \in \boldsymbol{E}$	$= \boldsymbol{D}$	expressed values
$\beta \in \boldsymbol{V}$	$= \boldsymbol{T} + \boldsymbol{N} + \boldsymbol{R}$	stored values
$\sigma \in \boldsymbol{S}$	$= \boldsymbol{L} \to [\boldsymbol{T} \times \boldsymbol{V}]$	store states

$\varrho \in \mathbf{Env}$	$= [\mathbf{Id} \to [\boldsymbol{D} \times \mathbf{Typ}]] \times [\{initobj\} \to \mathbf{Id}^*]$	environments
$\boldsymbol{A}$	$= \boldsymbol{S} + \mathbf{Env} + \boldsymbol{E} + \boldsymbol{H}$	answers
$\mu \in \boldsymbol{C}$	$= \boldsymbol{S} \to \boldsymbol{A}$	command continuations
$\varkappa \in \boldsymbol{K}$	$= \boldsymbol{E} \to \boldsymbol{C}$	expression continuations
$\chi \in \boldsymbol{X}$	$= \mathbf{Env} \to \boldsymbol{C}$	declaration continuations
TypB	$= \{integer, real, bool\}$	basic types
TypM	$= \{monitor\}$	abstraction type
$\mathbf{Ids}_A$	$= \{Is_{m1}, Is_{m2}, \ldots, Is_{mk}\}$	monitor object types
TypC	$= \{cond\}$	condition variable type
TypP	$= \{procm\}$	monitor procedure type
$\tau \in \mathbf{Typ}$	$= \mathbf{TypB} + \mathbf{TypM} + \mathbf{Ids}_A + \mathbf{TypC} + \mathbf{TypP}$	types
$\iota \in \boldsymbol{I}$	$= \{1, 2, \ldots, n\}$	processes
$\gamma \in \boldsymbol{P}$	$= [\mathbf{Com}_v + \{set, reset\}] \times \mathbf{Env} \times \boldsymbol{P}$	parallel continuations
$\tau' \in \boldsymbol{Q}$	$= \boldsymbol{P}^n$	comprehensive parallel continuations
$\eta \in \boldsymbol{H}$	$= \{1, 2, \ldots, n\}^\infty$	rosters
$\upsilon \in \boldsymbol{Y}$	$= [\mathbf{Id}_{obj} \to [\boldsymbol{I} \times \boldsymbol{I}^n \times \boldsymbol{Z}]]$	process states of monitor objects
$\zeta \in \boldsymbol{Z}$	$= \mathbf{Ids}_{cond} \to \boldsymbol{I}^n$	process delays
$\psi \in \boldsymbol{\Psi}$	$= \mathbf{Id}^*_{obj}$	monitor object stacks
$o \in \boldsymbol{O}$	$= \boldsymbol{\Psi}^*$	comprehensive monitor object stacks

$\mathbf{Ids}_{cond}$ is the domain of semantic values of identifiers of conditional variables. Each element of this domain is a reference to a list of processes waiting for a fulfilment of the condition joined with the appertaining conditional variable.

Abs is the domain, which elements are semantic meanings of abstract data types defined by monitors. Elements of the domain $\mathbf{Proc}_m$ are meanings of monitor procedures.

The second component of the environment enables to make accessible, by means of the so called implicit name *initobj*, a list of identifiers of those monitor objects, which are initialized in the given moment. The component $\mathbf{Ids}_A$ of the domain of types reflects the existence of the abstract data types mechanism in the considered language. Is_{m1}, Is_{m2}, ..., Is_{mk} are new data types introduced by declarations of monitors $m1$, $m2$, ..., mk. Elements of the processes domain $\boldsymbol{I}$ are natural numbers, by which we refer to individual, in the given moment existing, concurrent processes. To each process ι is connected its parallel continuation $\gamma_\iota \in \boldsymbol{P}$. It determines a command, which has to be executed as immediately following, in which environment and with which next parallel continuation (i.e. predicts a computational future

of the process ι). By the component *set*, *reset* we define the successful beginning respectively ending of a procedure execution of a monitor object. The interleaving of a process execution at the end of any of the "indivisible operations" is defined by certain element $\eta \in \boldsymbol{H}$. The indivisible operations may not be displayed overtly in the syntax of the language, but provided the language is given the denotational semantics they become apparent in the semantic equations. Before each execution of n concurrent processes one element η is created. It is the infinite list, items of which are randomly distributed elements of the domain $\boldsymbol{I}$. However, this distribution of elements must fulfil certain criteria of correctness. At the end of any of the indivisible operations the head of the list η is analysed. The result of this analysis influences the choice of the process which has to be running and the head is popped. If the control is given to another process it is necessary to preserve the parallel continuation of the process running till then. The domain $\boldsymbol{Q}$ serves for that. The process states of monitor objects $\upsilon \in \boldsymbol{Y}$ join three elements with any monitor object: a process $\iota \in \boldsymbol{I}$, which is just executing some procedure of this object, a list of processes $\iota^n \in \boldsymbol{I}^n$ waiting for the entry into some procedure of this object and a process delay $\zeta \in \boldsymbol{Z}$, which is a function joining a list of waiting processes to each conditional variable. The domain of comprehensive monitor objects stacks $\boldsymbol{O} = \psi^n$ explicitly binds with each process a list of identifiers of those monitor objects, the procedures of which an appertaining process is just executing. Each such a list ψ is an element of the semantic domain $\psi = \mathbf{Id}^*_{obj}$ and it is processed as stack.

Choosen semantic functions and equations:

1. $\mathscr{C} : \mathbf{Com} \to \mathbf{Env} \to \boldsymbol{C} \to \boldsymbol{C}$

$\mathscr{C}[\![\Gamma_1; \Gamma_2]\!]\,\varrho\mu = \mathscr{C}[\![\Gamma_1]\!]\,\varrho\{\mathscr{C}[\![\Gamma_2]\!]\,\varrho\mu\}$

$\mathscr{C}[\![\mathbf{begin}\ \Delta V^*;\ \Delta\ Abs^*;\ \Delta\ Obj^*;\ In;\ \Gamma\ \mathbf{end}]\!]\,\varrho\mu =$

$\quad \mathscr{D}^*_v[\![\Delta V^*]\!]\,\varrho\{\lambda\varrho_1' \,.\, \mathscr{F}^*_{Abs}[\![\Delta\ Abs^*]\!]\,\varrho_1\{\lambda\varrho_2 \,.\, \mathscr{D}^*_{obj\,m}[\![\Delta\ Obj_m]\!]\,\varrho_2$

$\quad \{\lambda\varrho_3 \,.\, \mathscr{J}[\![In]\!]\,\varrho_3\{\lambda\varrho_4 \,.\, \mathscr{C}[\![\Gamma]\!]\,\varrho_4\mu\}\}\}$

$\mathscr{C}[\![\mathbf{cobegin}\ \Gamma_{p1}, \Gamma_{p2}, \ldots, \Gamma_{pn}\ \mathbf{coend}]\!]\,\varrho\mu =$

$\quad \mathbf{let}\ Is_{cond} = \varrho_0[\![I_{cond}]\!] \downarrow 1\ \mathbf{in}$

$\quad\quad \mathbf{let}\ \zeta_0(Is_{cond}) = \Lambda^n$ for all I_{cond} declared in all

$\quad\quad\quad$ environments $\varrho_0 = \varrho[\![I_{obj}]\!] \downarrow 3\ \mathbf{in}$

$\quad\quad \mathbf{let}\ \upsilon_0[\![I_{obj}]\!] = (\Lambda, \Lambda^n, \zeta_0)$ for all I_{obj} declared in the environment ϱ

$\quad\quad\quad o_0 = (\Lambda^*, \Lambda^*, \ldots, \Lambda^*)$ (n-tuple)

$\quad\quad\quad \tau' = ((\Gamma_{p1}, \varrho, (\Lambda, \varrho, ?)), (\Gamma_{p2}, \varrho, (\Lambda, \varrho, ?)), \ldots, (\Gamma_{pn}, \varrho, (\Lambda, \varrho, ?)))$

$\quad\quad \mathbf{in}$

$\quad\quad New\ Roster\ \{\lambda\eta \,.\, (\eta \downarrow 1 = 1) \to \mathscr{S}[\![\Gamma_{p1}]\!]\,\varrho(\Lambda, \varrho, ?)\,1(\eta \dagger 1)\,o_0\upsilon_0\tau'\mu,$

$\quad\quad\quad\quad\quad\quad (\eta \downarrow 1 = 2) \to \mathscr{S}[\![\Gamma_{p2}]\!]\,\varrho(\Lambda, \varrho, ?)\,2(\eta \dagger 1)\,o_0\upsilon_0\tau'\mu,$

$\quad\quad\quad\quad\quad\quad\quad\quad \vdots$

$\quad\quad\quad\quad\quad\quad (\eta \downarrow 1 = n - 1) \to \mathscr{S}[\![\Gamma_{pn-1}]\!]\,\varrho(\Lambda, \varrho, ?)\,(n-1)\,(\eta \dagger 1)\,o_0\upsilon_0\tau'\mu,$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathscr{S}[\![\Gamma_{pn}]\!]\,\varrho(\Lambda, \varrho, ?)\,n(\eta \dagger 1)\,o_0\upsilon_0\tau'\mu$

2. $\mathcal{S} : \mathbf{Com}_v \to \mathbf{Env} \to \boldsymbol{P} \to \boldsymbol{I} \to \boldsymbol{H} \to \boldsymbol{O} \to \boldsymbol{Y} \to \boldsymbol{Q} \to \boldsymbol{C} \to \boldsymbol{C}$

$\mathbf{Com}_v = \mathbf{Com}_p + \mathbf{Com}_d + \mathbf{Com}_s$

$\mathcal{S}[\![I := E]\!]\,\varrho\gamma\iota\eta o\upsilon\tau'\mu =$
  let $\tau = \varrho[\![I]\!] \downarrow 2$ **in**
  $\mathcal{L}[\![I]\!]\,\varrho\tau\{\lambda\alpha\,.\,\mathcal{R}[\![E]\!]\,\varrho\tau\{\lambda\beta\,.\,Set\;\alpha\beta(Do\;\iota\gamma\eta o\upsilon\tau'\mu)\}\}$

$\mathcal{S}[\![\textbf{if } E \textbf{ then } \Gamma_{p1} \textbf{ else } \Gamma_{p2}]\!]\,\varrho\gamma\iota\eta o\upsilon\tau\,\mu =$
  $\mathcal{R}[\![E]\!]\,\varrho(bool)\,\{\lambda\beta\,.\,\beta \to Do\;\iota(\Gamma_{p1}, \varrho, \gamma)\,\eta o\upsilon\tau'\mu,$
  $\qquad Do\;\iota(\Gamma_{p2}, \varrho, \gamma)\,\eta o\upsilon\tau'\mu\}$

$\mathcal{S}[\![\textbf{while } E \textbf{ do } \Gamma_p]\!]\,\varrho\gamma\iota\eta o\upsilon\tau'\mu =$
  $\mathcal{R}[\![E]\!]\,\varrho(bool)\,\{\lambda\beta\,.\,\beta \to Do\;\iota(\Gamma_p, \varrho, (\textbf{while } E \textbf{ do } \Gamma_p; \gamma))\,\eta o\upsilon\tau'\mu,$
  $\qquad Do\;\iota\gamma\eta o\upsilon\tau'\mu\}$

$\mathcal{S}[\![\Gamma_{p1}; \Gamma_{p2}]\!]\,\varrho\gamma\iota\eta o\upsilon\tau'\mu = \mathcal{S}[\![\Gamma_{p1}]\!]\,\varrho(\Gamma_{p2}, \varrho, \gamma)\,\iota\eta o\upsilon\tau'\mu$

$\mathcal{S}[\![\textbf{begin } \Delta V^*; \Gamma_p \textbf{ end}]\!]\,\varrho\gamma\iota\eta o\upsilon\tau'\mu =$
  $\mathcal{D}_v^*[\![\Delta V^*]\!]\,\varrho\{\lambda\varrho'.\;Do\;\iota(\Gamma_p, \varrho', \gamma)\,\eta o\upsilon\tau'\mu\}$

$\mathcal{S}[\![I_{obj} \,.\, I_{nroc}(E^*)]\!]\,\varrho\gamma\iota\eta\iota o\upsilon\tau'\mu =$
  let $\varrho_{obj} = \varrho[\![I_{obj}]\!] \downarrow 1 \downarrow 3$ **in**
    let $\Gamma_{proc} = \varrho_{obj}[\![I_{proc}]\!] \downarrow 1 \downarrow 2,$
      $\varrho_1 = \varrho_{obj}[\![I_{proc}]\!] \downarrow 3,\; \Pi^* = \varrho_{obj}[\![I_{proc}]\!] \downarrow 1 \downarrow 1$ **in**
      let $\varrho_{proc} = fix\,(\lambda\varrho_1'\,.\,Var\,[\![\Pi^*]\!] \to \mathcal{L}^*[\![E^*]\!]\,\varrho'\tau^*\{\lambda\alpha^*.$
        $\varrho_1[\alpha^*/\Pi^*]\},$
        $\mathcal{R}^*[\![E^*]\!]\,\varrho_1'\tau^*\{\lambda\beta^*\,.\,\varrho_1[\beta^*/\Pi^*]\})$
  in $\varrho_{obj}[\![I_{proc}]\!] \downarrow 2 = proc\iota\iota) \to$
  $((\upsilon[\![I_{obj}]\!] \downarrow 1 = \Lambda \vee \upsilon\,[\![I_{obj}]\!] \downarrow 1 = \iota) \to$
    $((\textbf{let } o_1 = o[I_{obj} \circ (o \downarrow \iota)/(o \downarrow \iota)],\; \upsilon_1 = \upsilon[\iota/\upsilon[\![I_{obj}]\!] \downarrow 1]$
    in $\mathcal{S}[\![I_{proc}]\!]\,\varrho_{proc}(reset, \varrho, \gamma)\,\iota\eta o_1\upsilon_1\tau'\mu),$
      $(\textbf{let } \upsilon_2 = \upsilon[\upsilon[\![I_{obj}]\!] \downarrow \iota \circ \iota/\upsilon[\![I_{obj}]\!] \downarrow 2]$
    in $Do\;\iota\,(set, \varrho, (I_{obj}\,.\,I_{proc}(E^*), \varrho, \gamma))\,\eta o\upsilon_2\tau'\mu)),$?

$\mathcal{S}[\![\textbf{if } E \textbf{ then } I_{cond} \,.\, \textbf{wait}]\!]\,\varrho\gamma\iota\eta o\upsilon\tau'\mu =$
  let $I_{obj} = (o \downarrow \iota) \downarrow 1$ **in**
  let $\zeta = \upsilon[\![I_{obj}]\!] \downarrow 3$ **in**
  let $Is_{cond} = \varrho[\![I_{cond}]\!] \downarrow 1$ **in**
  $\mathcal{R}[\![E]\!]\,\varrho(bool)\,\{\lambda\beta\,.\,\beta \to$
  $((\upsilon[\![I_{obj}]\!] \downarrow 2 = \Lambda) \to$
  $((\textbf{let}\upsilon_1 = \upsilon[\zeta(Is_{cond})]\!] \circ \iota/\zeta(Is_{cond})]\,[\Lambda/\upsilon[\![I_{obj}]\!] \downarrow 1]$
  in $Do\;\iota\gamma\eta o\upsilon_1\tau'\mu),$
    $(\textbf{let } \iota_2 = (\upsilon[\![I_{obj}]\!] \downarrow 2) \downarrow 1$ **in**
    let $\upsilon_2 = \upsilon[\zeta(Is_{cond}) \circ \iota/\zeta(Is_{cond})]\,[\iota_2/\upsilon[\![I_{obj}]\!] \downarrow 1]$
      $[(\upsilon[\![I_{obj}]\!] \downarrow 2) \dagger 1/\upsilon[\![I_{obj}]\!] \downarrow 2]$
    in $Apply\;(\tau' \downarrow \iota_2 \downarrow 3)\;\iota_2\eta o\upsilon_2\tau'[\gamma/\tau' \downarrow \iota]\;\mu))),$
    $\mathcal{S}[\![\gamma \downarrow 1]\!]\,(\gamma \downarrow 2)\,(\gamma \downarrow 3)\,\iota\eta o\upsilon\tau'\mu$

$\mathcal{S}[\![I_{cond} \,.\, \textbf{signal}]\!]\,\varrho\gamma\iota\eta o\upsilon\tau'\mu =$
  let $Is_{cond} = \varrho[\![I_{cond}]\!] \downarrow 1$ **in**

$$
\begin{array}{l}
\textbf{let } I_{obj} = (o \downarrow \iota) \downarrow 1 \textbf{ in} \\
\textbf{let } \zeta = \upsilon[\![I_{obj}]\!] \downarrow 3 \textbf{ in} \\
(\zeta(Is_{cond}) = \Lambda \rightarrow \\
\quad ((\upsilon[\![I_{obj}]\!] \downarrow 2 = \Lambda) \rightarrow \\
\quad (\textbf{let } \upsilon_1 = \upsilon[\Lambda/\upsilon[\![I_{obj}]\!] \downarrow 1 \textbf{ in} \\
\quad Do\ \iota\gamma\eta o \upsilon_1 \tau' \mu), \\
\quad (\textbf{let } \iota_2 = (\upsilon[\![I_{obj}]\!] \downarrow 2) \downarrow 1 \textbf{ in} \\
\quad \textbf{let } \upsilon_2 = \upsilon[\iota_2/\upsilon[\![I_{obj}]\!] \downarrow 1] \\
\qquad\qquad [(\upsilon[\![I_{obj}]\!] \downarrow 2) \dagger 1/\upsilon[\![I_{obj}]\!] \downarrow 2] \textbf{ in} \\
\quad Apply\ (\tau' \downarrow \iota_2 \downarrow 3)\ \iota_2 \eta o \upsilon_2 \tau' [\gamma/\tau' \downarrow \iota]\ \mu))), \\
\quad (\textbf{let } \iota_3 = \zeta(Is_{cond}) \downarrow 1 \textbf{ in} \\
\quad \textbf{let } \upsilon_3 = \upsilon[\zeta(Is_{cond}) \dagger 1/\zeta(Is_{cond})] \\
\qquad\qquad [\iota_3/\upsilon[\![I_{obj}]\!] \downarrow 1] \\
\quad \textbf{in } Apply\ (\tau' \downarrow \iota_3)\ \iota_3 \eta o \upsilon_3 \tau' [\gamma/\tau' \downarrow \iota]\ \mu) \\
\upsilon[\![\Lambda]\!]\ \varrho\gamma\iota\eta o \upsilon\tau' \mu = Do\ \iota\gamma\eta o \upsilon\tau' \mu
\end{array}
$$

Also additional semantic functions have been defined in [1]. Those functions evaluate declarations and initializations. The used semantic functions $\mathscr{L}$ and $\mathscr{R}$ evaluate expressions which have values in the domains $\boldsymbol{L}$ and $\boldsymbol{V}$ respectively.

Auxiliary functions are important components of denotational definitions. The auxiliary function *Set* expresses the semantics of an assigning of a new value to a given location. The function *NewRoster* models a creation of a new roster. Other used auxiliary semantic functions are defined as follows:

1. $Var : \textbf{Par} \rightarrow \boldsymbol{T}$

$$
\begin{array}{l}
Var\ [\![\Pi]\!] = (\Pi = \textbf{var } I : T) \rightarrow true, \\
\qquad\qquad (\Pi = I : T) \rightarrow false, ?
\end{array}
$$

2. $Do : \boldsymbol{I} \rightarrow \boldsymbol{P} \rightarrow \boldsymbol{H} \rightarrow \boldsymbol{O} \rightarrow \boldsymbol{Y} \rightarrow \boldsymbol{Q} \rightarrow \boldsymbol{C} \rightarrow \boldsymbol{C}$

$$
\begin{array}{l}
Do\ \iota\gamma\eta o \upsilon\tau' \mu = \\
\quad \textbf{let } \iota_1 = \eta \downarrow 1 \textbf{ in} \\
\quad (\iota_1 = \iota) \rightarrow Apply\ \gamma \iota_1 (\eta \dagger 1)\ o \upsilon \tau' [\gamma/\tau' \downarrow \iota]\ \mu, \\
\qquad\qquad Apply\ (\tau' \downarrow \iota_1)\ \iota_1 (\eta \dagger 1)\ o \upsilon \tau' [\gamma/\tau' \downarrow \iota]\ \mu
\end{array}
$$

3. $Apply : \boldsymbol{P} \rightarrow \boldsymbol{I} \rightarrow \boldsymbol{H} \rightarrow \boldsymbol{O} \rightarrow \boldsymbol{Y} \rightarrow \boldsymbol{Q} \rightarrow \boldsymbol{C} \rightarrow \boldsymbol{C}$

$$
\begin{array}{l}
Apply\ \gamma \iota \eta o \upsilon \tau' \mu = \\
(\gamma \downarrow 1 = reset) \rightarrow Apply\ (\gamma \downarrow 3)\ \iota \eta o [(o \downarrow \iota) \dagger 1/(o \downarrow \iota)]\ \upsilon \tau' \mu, \\
(\gamma \downarrow 1 = set) \rightarrow Do\ \iota\gamma\eta o \upsilon \tau' \mu, \\
(\gamma \downarrow 1 = \Lambda \wedge \gamma \downarrow 3 = ?) \rightarrow (Terminatedothers\ \iota\tau' \rightarrow \mu,\ Do\ \iota\gamma\eta o \upsilon\tau' \mu\ , \\
o[\![\gamma \downarrow 1]\!]\ (\gamma \downarrow 2)\ (\gamma \downarrow 3)\ \iota\eta o \upsilon \tau' \mu
\end{array}
$$

4. $Terminatedothers: \boldsymbol{I} \rightarrow \boldsymbol{Q} \rightarrow \boldsymbol{T}$

$$
\begin{array}{l}
Terminatedothers\ \iota\tau' = \\
\bigvee_{i=1}^{n} [(\iota = i) \rightarrow (\bigwedge_{\substack{j=1 \\ j \neq i}}^{n} (\tau' \downarrow j \downarrow 1 = \Lambda \wedge \tau' \downarrow j \downarrow 3 = ?))] \rightarrow true, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad false
\end{array}
$$

$\wedge$ denotes the conjunction operator.

The technique used is a sort of formalised time slicing, but it does not imply that the implementation must necessarily be on a single processor. The process considered "current" in the semantics is merely the one, on which our mathematical attention happens to be fixed at the moment.

8. CONCLUSION

The establishing of the definition of the denotational semantics of parallel languages has an extensive importance. The formal definition of the semantics of a language together with the formal definition of its syntax provides an accurate and complete reference standard for designers, implementers and users of the language. The designers of the parallel languages have available tools for a precise formulation and validation of the semantic properties of the designed language constructs. The definition of the denotational semantics of parallel languages provides the basis for a derivation of methods for a systematic synthesis and verification of concurrent programs. However, the level of formalization used in the notation of the semantic equations is not high enough to ensure completely precise and unambiguous definition of the semantics of a language. If the semantics definition of the language *CL/*1 were processed by a compiler generator, we should write this definition in a metalanguage with an unambiguous grammar and semantics. The paper [2] deals with the problems of such metalanguages and with the possibilities of their utilization.

(Received June 23, 1981.)

REFERENCES

[1] P. Brezány: Towards Development and Application of the Functional Formal Semantics Theory to Some Elements of Modern Programming Languages (in Slovak). Ph. D. Dissertation, Faculty of Electrical Engineering, Slovak Technical University, Bratislava 1979.

[2] P. Brezány and L. Ščepánek: Denotational semantics and compiler generation of parallel programming languages. In: Zborník seminára "Moderní programování", Tatranská Lomnica 1980, 11—20.

[3] P. Brinch Hansen: The programming language Concurrent Pascal. In: Language Hierarchies and Interfaces (F. L. Bauer, K. Samelson, eds.), Springer-Verlag, Berlin—Heidelberg—New York 1976, 82—110.

[4] M. J. C. Gordon: The Denotational Description of Programming Languages. Springer-Verlag, Berlin—Heidelberg—New York 1979.

[5] G. T. Ligler: Proof Rules, Mathematical Semantics, and Programming Language Design. Ph. D. Dissertation, Oxford University, 1975.

[6] R. E. Milne: The Formal Semantics of Computer Languages and Their Implementations. Ph. D. Dissertation, Cambridge University, 1974.

[7] R. E. Milne and C. Strachey: A Theory of Programming Language Semantics. Chapman and Hall, London 1976.

[8] D. Scott and C. Strachey: Toward a mathematical semantics for computer languages. In: Proc. Symp. on Computers and Automata, Polytechnic Institute of Brooklyn, 1972.

Ing. Peter Brezány, CSc., Katedra kybernetiky SVŠT (Department of Cybernetics — Slovak Technical University), Gottwaldovo nám. 19, 812 43 Bratislava. Czechoslovakia.