

ON PROVING PROGRAM CORRECTNESS
BY MEANS OF STEPWISE REFINEMENT METHOD

Lech Banachowski
Institute of Informatics
Warsaw University

The stepwise refinement method [4] and the notion of an implementation of one data structure type by another are formulated on the basis of algorithmic logic [2]. It is shown that the properties of partial and total correctness of an implementation are expressible by means of nondeterministic algorithmic formulas.

1. INTRODUCTION

In the programming practice the following situation occurs frequently. We wish to develop a program correct with respect to some conditions α and β ⁽¹⁾. This program is supposed to operate on objects drawn from a certain fixed many sorted algebra $\mathcal{A} = \langle \{U_s\}_{s \in S}, \theta \rangle$, where $\{U_s\}_{s \in S}$ is a family of basic data types (such as for example integers,

⁽¹⁾ We consider two notions of program correctness: partial and total correctness [2]. We assume the following definitions. Suppose we are given program P and two formulas α and β (α describing intended input data, β describing results), all of them written in the algorithmic language [2] of a certain relational system \mathcal{A} . P is said to be partially correct with respect to α and β iff for every input data satisfying α if the computation of P terminates, then the resulting output data satisfy β . P is said to be totally correct with respect to α and β iff for every input data satisfying α the computation of P terminates and the resulting output data satisfy β . Thus the partial correctness is expressed by the formula $(\alpha \wedge P \text{ true} \Rightarrow P\beta)$ and the total correctness by $(\alpha \Rightarrow P\beta)$.

reals, Boolean values, vectors of integers etc.) and Θ is a set of functions of the form

$$\varphi: U_{S_1} \times \dots \times U_{S_k} \rightarrow U_{S_{k+1}} \quad (s_1, \dots, s_k, s_{k+1} \in S).$$

(In this paper for notational convenience we treat relations as functions assuming one of the sets U_S to be the set of Boolean values $\{0, 1\}$.) However the problem in question may involve some other objects from a certain larger many-sorted algebra $\mathcal{Y} = \langle \{U_S\}_{S \in S_1}, \Theta_1 \rangle$ where we assume that $S \subseteq S_1$ and $\Theta \subseteq \Theta_1$. It is often much easier to write a program over \mathcal{Y} than over \mathcal{X} . Suppose that we have written a program P over \mathcal{Y} correct with respect to conditions α and β . To get a corresponding program P' over \mathcal{X} correct with respect to α and β , we must replace in some way objects not in \mathcal{X} by objects in \mathcal{X} and moreover we must program over \mathcal{X} all the operations $\varphi \in \Theta_1 - \Theta$ that are used in P i.e. we must provide an implementation for P . For example the standard system of programming (defined by a programming language) is frequently adjoined by the objects like graphs, sets, families of sets etc.

Example 1. Let us consider the following problem. Given a positive integer k and integers $e, f, c[1], \dots, c[l], d[1], \dots, d[l]$ drawn from the set $\{1, 2, \dots, k\}$ determine whether the pair (e, f) belongs to the least equivalence relation containing all the pairs $(c[i], d[i])$ for $1 \leq i \leq l$. We are to solve this problem over the algebra consisting of Boolean values, integers and vectors of integers. However it is more natural for this problem to think about it in terms of sets of integers and sets of equivalence classes of equivalence relations over the set $\{1, 2, \dots, k\}$. So let R be a variable assuming as its values partitions of the set $\{1, 2, \dots, k\}$ and let A and B be variables assuming as their values subsets of the set $\{1, 2, \dots, k\}$. Consider the following operations:

- (1) $\text{init}(k)$ whose value is the partition $\{\{1\}, \{2\}, \dots, \{k\}\}$;
- (2) $\text{find}(h, R)$ whose value is the set of the partition R that contains integer h ;
- (3) $\text{eq}(A, B)$ whose value is either Boolean value 1 if $A=B$ or Boolean value 0 if $A \neq B$;
- (4) $\text{union}(A, B, R)$ whose value is either the partition $(R - \{A, B\}) \cup \{A \cup B\}$ if sets A and B are in R , or R otherwise.

Now consider the program which uses these operations:

```

begin
  {  $\alpha: k \geq 1 \wedge l \geq 0 \wedge \bigwedge_{u=1}^l 1 \leq c[u], d[u] \leq k \wedge 1 \leq e, f \leq k \}$ 
1:  R := init(k);
   for i := 1 to l do

```

```

begin {invariant: R is the set of equivalence classes of the least
      equivalence relation containing the pairs  $\langle c[u], d[u] \rangle$ 
      for  $1 \leq u \leq i$  }
2:   A:=find( $c[i]$ , R);
3:   B:=find( $d[i]$ , R);
4:   R:=union(A, B, R);
   end;
5:   A:=find(e, R);
6:   B:=find(f, R);
7:   q:=eq(A, B)
{  $\beta : q=1 \Leftrightarrow \langle e, f \rangle$  belongs to the least equivalence relation over
   $\{1, 2, \dots, k\}$  containing all the pairs  $\langle c[u], d[u] \rangle$  for  $1 \leq u \leq i$  }
end

```

where q is a Boolean variable and i is an integer variable. Abstract instructions which should be replaced by concrete instructions are preceded by labels in the text. \square

After Hoare [4] we shall call the source program P an abstract program and the target program P' a concrete program. We shall admit the following notational conventions:

- (1) a vector of variables of types in S common to both P and P' will be denoted by x (these variables will be called common variables);
- (2) a vector of variables of types in $S_1 - S$ appearing only in P will be denoted by y (these variables will be called abstract data structure variables);
- (3) a vector of variables of types in S appearing only in P' will be denoted by z (these variables will be called concrete data structure variables).

Note that in order to represent one abstract data structure variable we must sometimes use several concrete data structure variables. For example: if we represent a set A , $\text{card}(A)=n$, by an ordered table $T[1:n]$ such that $n \leq m$ and $A = \{T[i] : 1 \leq i \leq n\}$ then two concrete data structure variables T and n correspond to one abstract data structure variable S . Hence it follows that the transformation from P to P' should be accomplished by replacing whole instructions by instructions. Returning to Example 1 the parts of the program that should be replaced are the whole assignment statements: $R := \text{init}(k)$, $A := \text{find}(c[i], R)$, $B := \text{find}(d[i], R)$, $R := \text{union}(A, B, R)$, $A := \text{find}(e, R)$, $B := \text{find}(f, R)$, $q := \text{eq}(A, B)$.

In the opinion of the author input and output specifications should be the same for both P and P' and should depend only on common variables x . We shall adopt here this point of view assuming "data structu-

ness" of P and P' as local means being initialized at the beginning of computations.

To simplify further considerations relations between values of corresponding data structure variables in the vectors y and z , respectively, will be extended to vectors of values of y and z . The notation $I(z)=y$ will denote the fact that the vector of values of concrete data structure variables z corresponds to the vector of values of abstract data structure variables y . The two main properties which should hold between corresponding computations of P and P' are: the invariance of the relation $I(z)=y$ and the invariance of the fact that the corresponding vectors of values of variables x are the same. Now it should not be surprising that these two properties allow to infer from partial or total correctness of P with respect to $\alpha(x)$ and $\beta(x)$, partial or total correctness of P' with respect to the same formulas.

In section 2 we introduce an appropriate notion of a data structure type. In section 3 we define a notion of an implementation of one data structure type by another. In section 4 we introduce notions of partial and total correctness of an implementation and we prove two sufficient conditions for these two kinds of correctness which have the form of validity of some deterministic algorithmic formulas. In section 5 we present the rules which from partial (total) correctness of a source, abstract program P and partial (total) correctness of an implementation allow to infer partial (total) correctness of a target, concrete program P' . In section 6 we prove that both the properties partial and total correctness of an implementation are expressible by means of formulas of certain extended nondeterministic algorithmic logic.

2. $\Sigma(K)$ -PROGRAMS AND THEIR COMPUTATIONS

Suppose we are given a many sorted universe of objects $\bigcup_{s \in S} U_s$ with operations forming many-sorted algebra $\mathcal{A} = \langle \{U_s\}_{s \in S}, \Theta \rangle$ where S is a set of sorts and Θ is a set of operations such that for each $\varphi \in \Theta$ there exists $k \geq 0$ and $s_1, \dots, s_{k+1} \in S$ such that

$$\varphi : U_{s_1} \times \dots \times U_{s_k} \rightarrow U_{s_{k+1}}.$$

We assume that one of the sets U_s is two-element Boolean algebra. Moreover we assume that for this algebra an algorithmic language $L(\mathcal{A})$ is defined which includes the following sets:

- (1) the set of deterministic instructions (programs) FS;
 - (2) the set of algorithmic terms FST;
 - (3) the set of algorithmic formulas FSF.
- (For the definitions consult [2].)

For notational convenience we shall denote operations and corresponding functors by the same symbols.

Since we wish to abstract from specific programs which contain certain objects and instructions to be implemented, we need to introduce the notion of a computation of a sequence of instructions (drawn from arbitrary programs which use these instructions).

So let $\mathcal{K} = \{K_0, K_1, \dots, K_n\}$ be a finite set of instructions in FS. We assume that three kinds of variables are distinguished:

- (1) common variables x (common to abstract and concrete programs);
- (2) data structure variables y (specific for only one program either abstract or concrete);
- (3) auxiliary variables (local to a single instruction).

Abstract and concrete programs usually carry out some initializations of values of data structure variables at the beginning. We assume that this is accomplished by the first instruction K_0 which should deliver initial values to all data structure variables in y . In addition we assume that K_0 has the stop property in \mathcal{A} .

By $\Sigma(\mathcal{K})$ we shall denote the set of all finite sequences

$$\delta = \delta_0, \delta_1, \dots, \delta_m$$

where $m \geq 0$, $\delta_0 = K_0$ and $\delta_i \in \mathcal{K} - \{K_0\}$ for $1 \leq i \leq m$.

The elements of the set $\Sigma(\mathcal{K})$ will be called \mathcal{K} -programs.

The intuitive meaning of \mathcal{K} -programs is as follows. Let us suppose that the instructions of the set $\mathcal{K} = \{K_0, K_1, \dots, K_n\}$ are used in a program P . P may be called the main program. Consider a finite computation of P for some initial valuation of variables. During this computation executions of some instructions from \mathcal{K} take place. Suppose that $\delta_0, \delta_1, \dots, \delta_m$ is the sequence of instructions used consecutively by the main program P . First the execution of $\delta_0 = K_0$ initializes the values of data structure variables. Afterwards some computations take place in P , which however do not change the values of data structure variables. The main program P prepares values of common variables for the instruction δ_1 and then δ_1 is called. The execution of δ_1 can change values of data structure variables and upon the completion δ_1 communicates the values of common variables to the main program P . This process is repeated for the next instructions $\delta_2, \dots, \delta_m$, consecutively. At any time values of data structure variables can be changed only by one of the instructions in \mathcal{K} . In this way the values of data structure variables form a certain kind of data type to be distinguished from other data. The main program P may use the values of data structure variables only by calling instructions in \mathcal{K} .

In this paper such a set \mathcal{K} of instructions will be called a

data structure type over α . With the presented above intuitions in mind, we adopt the following definition of a computation of a \mathcal{K} -program. Namely a computation of a \mathcal{K} -program $\delta = \delta_0, \delta_1, \dots, \delta_m$ consists of three sequences of vectors of values:

- (1) x_0, x_1, \dots, x_m
- (2) y_1, y_2, \dots, y_{m+1}
- (3) t_0, t_1, \dots, t_m

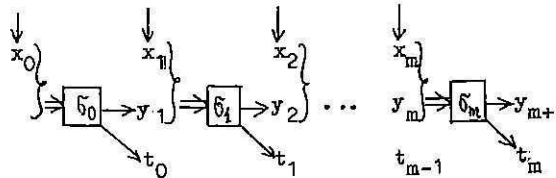
such that the following conditions hold:

- (i) for each i , $0 \leq i \leq m$, x_i and t_i are vectors of values of the common variables x ;
- (ii) for each i , $1 \leq i \leq m$, y_i is a vector of values of the data structure variables y ;
- (iii) y_1 is the vector of values of data structure variables upon the completion of the execution of $\delta_0 = K_0$;
- (iv) for each i , $1 \leq i \leq m$, the computation of δ_i for $x = x_i$ and $y = y_i$ terminates and upon the completion $y = y_{i+1}$ and $x = t_i$. It is convenient to present a computation of a \mathcal{K} -program $\delta = \delta_0, \delta_1, \dots, \delta_m$ by means of the following diagram (assuming the above denotations).

vectors of values of the
common variables x before
execution of an instruction

vectors of values of the
data structure variables y

vectors of values of the
common variables x after
execution of an instruction



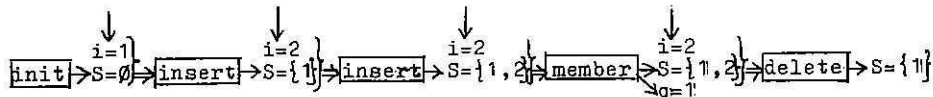
Example 2. Let N be a positive integer. Consider the system whose universe includes the set $\{1, 2, \dots, N\}$, the set of Boolean values $\{0, 1\}$ and the set $P(N)$ of all subsets of the set $\{1, 2, \dots, N\}$. Assume that the following set operations and relations are in the system under consideration:

- (1) constant \emptyset (empty set);
- (2) two-argument operations \cup and $-$ of union and subtraction of sets, respectively;
- (3) two-argument relation \in of being an element of a set;
- (4) the operation $\{ \}$ of composing one-element set.

Let i be a variable assuming its values from the set $\{1, 2, \dots, N\}$, let S be a variable assuming its values from the set $P(N)$ and let q be a Boolean variable. Let us consider the following data structure type $\text{DICTIONARY} = \{\text{init}, \text{insert}, \text{delete}, \text{member}\}$ where $\text{init} = [S := \emptyset]$,

$\text{insert} = [S := S \cup \{i\}]$,
 $\text{delete} = [S := S - \{i\}]$,
 $\text{member} = [q := i \in S]$.

We assume that i and q are common variables and S is a data structure variable. (N is treated as a constant.) These instructions do not contain any auxiliary variables. For example $\delta = \text{init}, \text{insert}, \text{insert}, \text{member}, \text{delete}$ is a DICTIONARY-program. If we provide the inputs 1, 2, 2, 2 for the common variable i of the consecutive instructions in δ we receive the following computation (only relevant values are indicated).



□

3. A NOTION OF AN IMPLEMENTATION

Let $\mathcal{X} = \langle \{U_s\}_{s \in S_1}, \Theta_1 \rangle$ be an algebra which is an extension of $\mathcal{A} = \langle \{U_s\}_{s \in S}, \Theta \rangle$ (i.e. $S \subseteq S_1$ and $\Theta \subseteq \Theta_1$). We shall assume that $L(\mathcal{X})$ is a sublanguage of $L(\mathcal{A})$. Suppose that we are interested in a data structure type $\mathcal{K} = \{K_0, K_1, \dots, K_n\}$ over \mathcal{X} and we wish to implement it by another data structure type $\mathcal{M} = \{M_0, M_1, \dots, M_n\}$ over \mathcal{A} . Assume that x is a vector of common variables of \mathcal{K} and \mathcal{M} , y is a vector of data structure variables of \mathcal{K} and z is a vector of data structure variables of \mathcal{M} . For example, suppose that we have an abstract program P such that it uses a data structure type \mathcal{K} and all its instructions which cannot be directly carried out by a computer are included in \mathcal{K} . We could look for a data structure type \mathcal{M} such that its instructions can be directly carried out by a computer and each instruction in \mathcal{M} simulates the corresponding instruction in \mathcal{K} . By replacing the instructions of \mathcal{K} by corresponding instructions of \mathcal{M} in P , we would get a program P' such that it solves the problem in question and it can run on a computer. Given common inputs for P and P' the program P calls consecutively some instructions $K_{i_0}, K_{i_1}, \dots, K_{i_m}$ of \mathcal{K} , while at the same time P' calls consecutively the corresponding instructions $M_{i_0}, M_{i_1}, \dots, M_{i_m}$ of \mathcal{M} .

We shall call \mathcal{K} an abstract data structure type and \mathcal{M} a concrete data structure type. We shall use the following notational convention. If

$$\delta = K_{i_0}, K_{i_1}, \dots, K_{i_m}$$

is a \mathcal{K} -program then by $\bar{\delta}$ we denote the corresponding \mathcal{M} -program

$$\bar{\delta} = M_{i_0}, M_{i_1}, \dots, M_{i_m}$$

(which is to simulate δ).

To wholly define an implementation of \mathcal{K} by \mathcal{M} we need to establish the connection between values of concrete and abstract data structure variables. We shall use the notion of an interpretation instruction, which is similar to Hoare's [4] concept of "abstract function". Namely to define an implementation we shall provide an instruction I in the language $L(\mathcal{Y})$ which is intended to determine the vector of values of abstract data structure variables y which corresponds to a given vector of values of concrete data structure variables z .

Any pair $J = (\mathcal{M}, I)$ satisfying all the conditions stated above will be called an implementation over \mathcal{A} of the data structure type \mathcal{K} .

Example 3. Let us consider again the data structure type **DICTIONARY** = {init, insert, delete, member} defined in Example 2. Its implementation can be defined as $J = (\mathcal{M}, I)$ where

$\mathcal{M} = \{ \text{INIT, INSERT, DELETE, MEMBER} \},$
 $\text{INIT} = (\text{for } j := 1 \text{ to } N \text{ do } T[j] := \text{false}),$
 $\text{INSERT} = [T[i] := \text{true}],$
 $\text{DELETE} = [T[i] := \text{false}],$
 $\text{MEMBER} = [q := T[i]],$

i and q are common variables, T is a concrete data structure variable of type Boolean vector of length N , j is an auxiliary variable of type integer and

$I = [S := \{ j : T[j] = \text{true} \}]. \quad \square$

4. PARTIAL AND TOTAL CORRECTNESS OF IMPLEMENTATIONS

The question arises under what conditions \mathcal{M} -programs can be regarded as correct simulations of \mathcal{K} -programs. We shall adopt the following definitions.

An implementation $J = (\mathcal{M}, I)$ over \mathcal{A} of the data structure type \mathcal{K} is said to be partially correct if for every \mathcal{K} -program $\delta = \delta_0, \delta_1, \dots, \delta_m$ and for every computation of the corresponding \mathcal{M} -program $\bar{\delta} = M_0, M_1, \dots, M_m$ given in Figure 1 there exists a computation of $\bar{\delta}$ of the form given in Figure 2 such that for each i , $1 \leq i \leq m+1$ the vector of values $I_{\mathcal{K}}(z=z_i)(y)$ is defined and equal to y_i (where by $I_{\mathcal{K}}(z=z_i)(y)$ we denote the vector of resulting values of variables y on the completion of I for $z=z_i$).

In virtue of the definition partial correctness of an implementation means that if we have a terminating computation of a sequence $\bar{\delta}$ simulating $\delta \in \Sigma(\mathcal{K})$, then we can obtain the results of the original sequen-

vectors of values of the
common variables x before
execution of an instruction
vectors of values of the
data structure variables z
vectors of values of the
common variables after
execution of an instruction

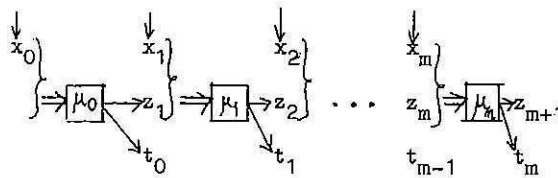


Figure 1. A computation of \mathcal{M} -program $\bar{\delta}$.

vectors of values of the
common variables x before
execution of an instruction
vectors of values of the
data structure variables y
vectors of values of the
common variables after
execution of an instruction

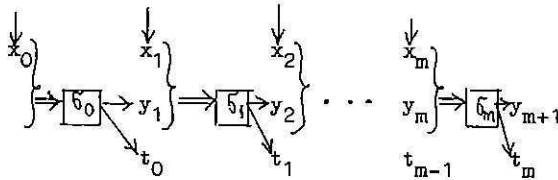


Figure 2. A computation of \mathcal{K} -program δ .

ce $\bar{\delta}$ directly from this computation. This corresponds (as we shall see in the next section) to the notion of partial correctness of programs. If we get results we know they are correct. Of course sometimes we wish to know something more. Namely if a sequence $\bar{\delta}$ simulates δ and the computation of δ terminates then the computation of $\bar{\delta}$ terminates as well.

An implementation $J=(\mathcal{M}, I)$ over \mathcal{A} of the data structure type \mathcal{K} is said to be totally correct if for every \mathcal{K} -program $\delta=\delta_0, \delta_1, \dots, \delta_m$ and for every its computation of the form given in Figure 2 there exists a computation of $\bar{\delta}$ of the form given in Figure 1 such that for each i , $1 \leq i \leq m+1$ the vector of values $I_{\mathcal{M}}(z=z_i)(y)$ is defined and equal to y_i .

Observe that if all the instructions of an abstract data structure type \mathcal{K} (in practice the instructions in \mathcal{K} are simply assignment statements or at most sequences of assignment statements) have the stop property in \mathcal{A} then the total correctness of J implies partial correctness of J . The converse may be not true.

It seems that the definitions of partial and total correctness of implementations cannot be written by means of ordinary deterministic algorithmic formulas. Nevertheless in practical situations we need not go away from algorithmic formulas. Our reasonings about sequences of instructions consists in considering each instruction separately.

Below there are two criteria of partial and total correctness of

an implementation, respectively. They use the notion of validity of algorithmic formulas. Let $\Delta(z)$ be an arbitrary algorithmic formula which plays the role of an invariant of the data structure type \mathcal{U} .

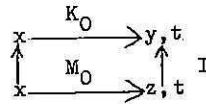
Theorem 1. If the following formulas are valid in \mathfrak{B}

- (1) $M_0 \Delta(z)$,
- (2) $\bigwedge_{1 \leq i \leq n} (\Delta(z) \wedge M_i \text{ true} \Rightarrow M_i \Delta(z))$,
- (3) $K_0 y = M_0 I y \wedge K_0 x = M_0 x$,
- (4) $\bigwedge_{1 \leq i \leq n} (\Delta(z) \wedge M_i \text{ true} \Rightarrow (M_i I y = I K_i y \wedge M_i x = I K_i x))$

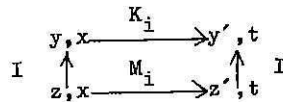
then $J = (\mathcal{U}, I)$ is a partially correct implementation over \mathfrak{A} of the data structure type \mathfrak{X} .

The validity of the consecutive formulas in Theorem 1 can be expressed as follows:

- (1) $\Delta(z)$ holds after performing M_0 ;
- (2) $\Delta(z)$ is preserved under each instruction M_i , $1 \leq i \leq n$;
- (3) the computations of all the instructions in the diagram below terminate and the diagram is commutative;



- (4) for each i , $1 \leq i \leq n$, if a vector of values of data structure variables z satisfies $\Delta(z)$ and the computation of M_i terminates then the computations of all the instructions in the diagram below terminate and the diagram is commutative.



Proof of Theorem 1. Let us consider an arbitrary computation of the \mathcal{U} -program $\overline{\mathfrak{B}}$ given in Figure 1. It follows that during the realization of $\overline{\mathfrak{B}}$ the computation of each instruction μ_i terminates, $0 \leq i \leq m$. Now carry out $\overline{\mathfrak{B}}$ providing the vector of values x_i for the i -th instruction, $0 \leq i \leq m$.

To prove the thesis of Theorem 1 it is sufficient to show that for each j , $1 \leq j \leq m+1$, there exists a computation of the \mathfrak{X} -program $\mathfrak{B}^j = \mathfrak{B}_0, \mathfrak{B}_1, \dots, \mathfrak{B}_{j-1}$ of the form given in Figure 3 such that for each i , $1 \leq i \leq j$, the vector of values $I_{\mathfrak{X}}(z=z_i)(y)$ is defined and equal to y_i . The proof proceeds by induction on j . The validity of the formula (3) implies the initial induction step for $j=1$. Since by (1) and (2) $\Delta(z)$

vectors of values of the
common variables x before
execution of an instruction
vectors of values of the
data structure variables y
vectors of values of the
common variables x after
execution of an instruction

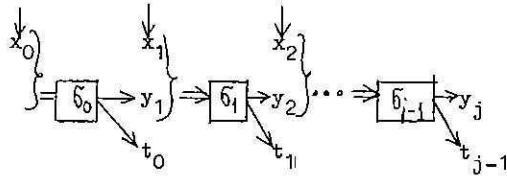


Figure 3. A computation of S_j .

is an invariant during the computation of \bar{S} the validity of the formula (4) allows to carry out the induction step from j to $j+1$. \square

Theorem 2. If the formulas (1)-(4) of Theorem 1 and the formula
(5) $\bigwedge_{1 \leq i \leq n} (\Delta(z) \wedge IK_i \text{ true} \Rightarrow M_i \text{ true})$

are valid in \mathcal{B} then $J = (\mathcal{M}, I)$ is a totally correct implementation over \mathcal{A} of the data structure type \mathcal{X} .

Proof. It is sufficient to observe that the validity of (5) together with the validity of the remaining formulas allows to infer from the termination of the computation of a \mathcal{X} -program S the termination of the corresponding \mathcal{M} -program \bar{S} . \square

In some situations when all the instructions of \mathcal{X} and \mathcal{M} have the stop property and no restrictions on values of concrete data structure variables z are needed, the formulas in Theorems 1 and 2 simplify their form very much.

Corollary. If the following formulas are valid in \mathcal{B}

- (1) $M_0 Iy = K_0 y \wedge M_0 x = K_0 x$,
(2) $\bigwedge_{1 \leq i \leq n} (M_i Iy = IK_i y \wedge M_i x = IK_i x)$

then $J = (\mathcal{M}, I)$ is a totally correct implementation over \mathcal{A} of the data structure type \mathcal{X} .

Proof. It is sufficient to apply Theorem 2 for $\Delta = \text{true}$ and observe that the validity of (2) ensures the validity of all the formulas $M_i \text{ true}$ and $IK_i \text{ true}$ for $1 \leq i \leq n$. \square

Example 4. Let us consider again the data structure type $\text{DICTIONARY} = \{\text{init}, \text{insert}, \text{delete}, \text{member}\}$ defined in Example 2 and its implementation $J = (\mathcal{M}, I)$ defined in Example 3. By Corollary in order to prove the total correctness of this implementation it is sufficient to show the validity of the following formulas (the equalities concerning variables whose values do not change in the corresponding instructions are omitted):

$[S := \emptyset] S = (\text{for } j := 1 \text{ to } N \text{ do } T[j] := \text{false}; S := \{j : T[j]\}) S,$

$$\begin{aligned}
 (T[i] := \text{true}; S := \{j: T[j]\})S &= (S := \{j: T[j]\}; S := S \cup \{i\})S, \\
 (T[i] := \text{false}; S := \{j: T[j]\})S &= (S := \{j: T[j]\}; S := S - \{i\})S, \\
 [q := T[i]] \ q &= (S := \{j: T[j]\}; q := i \in S)q.
 \end{aligned}$$

The above formulas can be reduced equivalently by means of simple rule for algorithmic formulas to the following ones:

$$\begin{aligned}
 \emptyset &= (\text{for } j:=1 \text{ to } N \text{ do } T[j] := \text{false}) \{j: T[j]\}, \\
 [T[i] := \text{true}] \{j: T[j]\} &= \{j: T[j]\} \cup \{i\}, \\
 [T[i] := \text{false}] \{j: T[j]\} &= \{j: T[j]\} - \{i\}, \\
 T[i] &= (i \in \{j: T[j]\})
 \end{aligned}$$

which are obviously valid. \square

5. PARTIAL AND TOTAL CORRECTNESS OF PROGRAMS WITH DATA STRUCTURES

Let \mathcal{A} and its extension \mathcal{B} will be defined as previously. Let $J = (\mathcal{U}, I)$ be an implementation of a data structure type \mathcal{X} , where $\mathcal{X} = \{K_0, K_1, \dots, K_n\}$ is an abstract data structure type over \mathcal{B} and $\mathcal{U} = \{M_0, M_1, \dots, M_n\}$ is a concrete data structure type over \mathcal{A} . As previously assume that x is a vector of common variables, y is a vector of abstract data structure variables (i.e. of \mathcal{X}) and z is a vector of concrete data structure variables (i.e. of \mathcal{U}). Now we want to apply the implementation J to transform "abstract" programs over \mathcal{B} into "concrete" programs over \mathcal{A} . Let P be a program over \mathcal{B} in which we distinguish instructions of the data structure type \mathcal{X} . Let P' be a program resulting from P by textual replacement of each occurrence of the instruction K_i by the instruction M_i for $0 \leq i \leq n$. Let $\alpha(x)$ and $\beta(x)$ be two formulas in the language $L(\mathcal{A})$.

Theorem 3. If the following conditions hold:

- (1) P' is a program in the language $L(\mathcal{A})$;
 - (2) P is partially correct (totally correct) with respect to $\alpha(x)$ and $\beta(x)$;
 - (3) J is a partially correct (totally correct) implementation over \mathcal{A} of the data structure type \mathcal{X}
- then P' is partially correct (totally correct) with respect to $\alpha(x)$ and $\beta(x)$.

Proof. Let us consider computations of P and P' for some input data $x = \bar{x}$ satisfying α . Assume that the computation of P' terminates. From this computation of P' we can extract a certain \mathcal{U} -program $\mu = M_0, \mu_1, \dots, \mu_m$ and its computation of the form presented in Figure 1. From the assumption (3) it follows that there exists a computation of the corresponding \mathcal{X} -program $\bar{\sigma}, \bar{\sigma} = \mu$, of the form presented in Figure 2 such that for each $i, 1 \leq i \leq m+1$, the vector of values $I_{\mathcal{X}}(z=z_i)(y)$ is defined and equal to y_i . Consider now the computation of P for input

data $x = \bar{x}$. Using the assumption (1) it can be easily proved by induction on the length of computations that excluding steps inside instructions of \mathcal{K} and \mathcal{M} the following three conditions are preserved in the corresponding steps of the computations of P and P' :

(4) the vector of values of ly is defined in P' and equal to the vector of values of y in P ;

(5) the vector of values of x in P is equal to the vector of values of x in P' ;

(6) the statements which are currently carried out in P and P' are either the same or one is K_i and the other one is M_i for some $0 \leq i \leq n$. It follows that the computation of P comes to the end point and the variables x have the same values as the variables x at the end point in P' . By the assumption (2) the values of the variables x satisfy β .

We have thus proved Theorem 3 for the case of partial correctness.

The proof for the case of total correctness is similar (the invariance of the same conditions (4), (5) and (6) should also be proved). Therefore we omit it here. \square

Example 5. Let us come back to the program in Example 1. We shall treat this program as an abstract one. It uses the following common variables $x = (k, l, e, f, c, d, q)$ and abstract data structure variables $y = (A, B, R)$. To be able to apply the methods presented in this paper we must add to this program two assignment statements which initialize the values of A and B . Thus we shall deal with the following program:

```
P: begin
    { $\alpha: k \geq 1 \wedge l \geq 0 \wedge 1 \leq c[u], d[u] \leq k$  for  $1 \leq u \leq l \wedge 1 \leq e, f \leq k$ }
    begin  $R := \text{init}(k)$ ;  $A := \emptyset$ ;  $B := \emptyset$  end;
    for  $i := 1$  to  $l$  do
        begin
            {invariant  $R$  is the set of equivalence classes of the least
              equivalence relation containing  $(c[u], d[u])$  for  $1 \leq u < i$ }
             $A := \text{find}(c[i], R)$ ;
             $B := \text{find}(d[i], R)$ ;
             $R := \text{union}(A, B, R)$ 
        end;
    end;
     $A := \text{find}(e, R)$ ;
     $B := \text{find}(f, R)$ ;
     $q := \text{eq}(A, B)$ 
    { $\beta: q \Leftrightarrow$  the pair  $(e, f)$  belongs to the least equivalence relation
      on the set  $\{1, 2, \dots, k\}$  containing the pairs  $(c[u], d[u])$ 
      for  $1 \leq w \leq l$ }
end
```

It is easy to prove the total correctness of P with respect to α and β by using the enclosed loop invariant. In P we distinguish seven abstract instructions:

K_0 : begin $R := \text{init}(k)$; $A := \emptyset$; $B := \emptyset$ end,
 K_1 : $A := \text{find}(c[i], R)$,
 K_2 : $B := \text{find}(d[i], R)$,
 K_3 : $R := \text{union}(A, B, R)$,
 K_4 : $A := \text{find}(e, R)$,
 K_5 : $B := \text{find}(f, R)$,
 K_6 : $q := \text{eq}(A, B)$

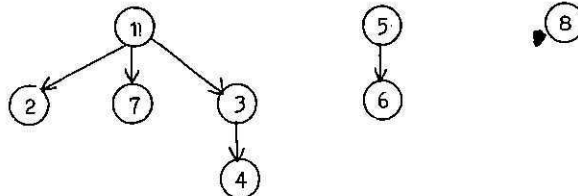
forming together a data structure type, call it find-union.

Now we shall present a certain implementation of this data structure type called a tree implementation [1]. An underlying algebra \mathcal{A} will consist of three basic types: Boolean values, integers and vectors of integers. The set of operations of \mathcal{A} will consist of all operations used in the concrete data structure type to be defined.

Each subset of the set $\{1, 2, \dots, k\}$ will be represented as a tree. The whole partition of the set $\{1, 2, \dots, k\}$ will be represented as a forest of trees, each tree representing one set of the partition. The vertices of this forest are integers $1, 2, \dots, k$. The edges of this forest are given by the contents of an integer array $T[1:k]$. Namely for each i , $1 \leq i \leq k$ the value $T[i]$ is an integer such that $0 \leq T[i] \leq k$ and if for some $1 \leq i, j \leq k$, $T[i] = j$ then the pair (i, j) forms an edge of the forest under consideration. If $T[i] = 0$ then this means that the vertex i is the root of a tree. For example the contents of the array T :

1	2	3	4	5	6	7	8
0	1	1	3	0	5	1	0

determines the following forest



(arrows go from sons to fathers of vertices). The forest above represents the partition $\{\{1, 2, 3, 4, 7\}, \{5, 6\}, \{8\}\}$. Each root of a tree represents all vertices composing this tree. For example the root 1 represents the set $\{1, 2, 3, 4, 7\}$.

Now we shall proceed to the definition of the concrete data struc-

ture type FU-tree = {INIT, FIND(c[i],a), FIND(d[i],b), UNION, FIND(e,a), FIND(f,b), EQ} with $z=(a,b,T)$ as a vector of concrete data structure variables. (Note that there are four distinct instructions whose names begin with FIND. However they can be considered as distinct calls of the same procedure having two integer parameters u and v called by name.) Let u and v be auxiliary integer variables. We define

INIT= begin

$u:=1;$

while $u \leq k$ do $(u, T[u]) := (u+1, 0);$

$(a,b) := (0,0)$

end

(INIT builds the forest composed of k one-element trees);

FIND(v,u)= begin

$(u,v) := (v, T[v]);$

while $v \neq 0$ do $(u,v) := (v, T[v]);$

end

(FIND begins at a given vertex v and follows the path to the root of the tree. Upon the completion u is the root of the tree);

UNION= if $1 \leq a, b \leq k \wedge a \neq b$ then

if $T[a]=0 \wedge T[b]=0$ then $T[a] := b$

(if a and b represent roots of distinct trees then UNION combines these trees into one tree);

EQ= [$q := (a=b)$] .

Now we shall proceed to the definition of the interpretation instruction I . Let for a nonnegative integer h and an integer p , $0 \leq p \leq k$, the notation $T^h[p]$ means p if $(h=0 \vee p=0)$ or $T^{h-1}[T[p]]$ otherwise. Recall that the contents of the array T and integer i , $1 \leq i \leq k$, determine a subset of the set $\{1, 2, \dots, k\}$ consisting of all vertices j , $1 \leq j \leq k$, from which the vertex i is reachable in the forest. Denote this set by $\text{set}(i, T)$ i.e. $\text{set}(i, T) = \{j : \exists h, 0 \leq h \leq k \wedge T^h[j] = i\}$. It should be evident that this set can be built by a program over the abstract system. The interpretation instruction I can be defined as follows $I = [A := \text{set}(a, T); B := \text{set}(b, T); R := \{ \text{set}(u, T) : 1 \leq u \leq k \wedge T[u] = 0 \}]$. Thus we have completed the definition of the tree implementation of the data structure type find-union, $J = (\text{FU-tree}, I)$.

Now using Theorems 1 and 2 we shall show that J is a partially and totally correct implementation. To do this we shall consider consecutive formulas, whose validity is sufficient for partial and total correctness. As an invariant $\Delta(z)$ we take an algorithmic formula stating that the contents of the vector T represents a partition of the set $\{1, 2, \dots, k\}$. It should be clear that such a formula can be constructed

if we are provided with classical quantifiers.

(1) Formula (1), namely $\text{INIT } \Delta(z)$ states that after initialization T represents a partition of the set $\{1, 2, \dots, k\}$.

(2) Formula (2) states that this property is preserved under any instruction. The instructions EQ and those prefixed with FIND do not change the current forest. The instruction UNION adds new edge only if this edge links two roots of distinct trees. Hence UNION preserves $\Delta(z)$.

(3) Formula (3) states that the initializations have the following properties:

- (i) they possess terminating computations;
- (ii) on the completion the values of the common variables k, l, e, f, c, d , i are the same in K_0 and INIT, respectively;
- (iii) on the completion the values of abstract data structure variables A, B, R and of concrete data structure variables a, b, T corresponds mutually according to the interpretation instruction I.

The conditions (i) and (ii) are obviously fulfilled. The condition (iii) holds since after initializations $R = \{\{1\}, \{2\}, \dots, \{k\}\} = \{\text{set}(u, T) : 1 \leq u \leq k \wedge T[u] = 0\}$, $A = \emptyset = \text{set}(0, T)$ and $B = \emptyset = \text{set}(0, T)$.

(4) Formula (4) states that for each pair of corresponding instructions for every values of common and concrete data structure variables, if T represents a partition of the set $\{1, 2, \dots, k\}$ and the concrete instruction possesses a terminating computation then the following conditions hold:

- (i) the interpretation instruction I possesses a terminating computation transforming given values of concrete data structure variables T , a, b into corresponding values of abstract data structure variables R , A, B , respectively;
- (ii) the abstract instruction possesses a terminating computation;
- (iii) on the completion of both the instructions the values of common variables are the same;
- (iv) on the completion of both the instructions the values of abstract data structure variables R, A, B and of concrete data structure variables T, a, b correspond mutually according to the interpretation instruction I.

The conditions (i) and (ii) are easy to check. Consider now (iii) and (iv) for each pair of instructions.

Each particular call of $\text{FIND}(v, u)$ computes the root u of the tree that contains v . Thus $\text{set}(u, T) = \text{find}(v, R)$ which equals either A or B . $\text{FIND}(v, u)$ computes nothing more, so the conditions (iii) and (iv) hold.

Observe that the values a and b are roots of distinct trees iff $A = \text{set}(a, T)$ and $B = \text{set}(b, T)$ are distinct nonempty sets of the partition R . Hence if a and b are roots of distinct trees then the new value of T represents the partition:

$\{set(u, T) : T[u] = 0 \wedge u \neq a, b\} \cup \{set(a, T) \cup set(b, T)\} = (R - \{A, B\}) \cup \{A \cup B\}$
 $= \text{union}(A, B, R)$ which is the new value of R . It follows that UNION satisfies (iii) and (iv).

Since $A = set(a, T)$ and $B = set(b, T)$ and the equality $set(a, T) = set(b, T)$ is equivalent to $a = b$, EQ satisfies (iii) and (iv).

(5) Formula (5) states that if T represents a partition of $\{1, 2, \dots, k\}$ and an abstract instruction has a terminating computation for given values of common variables and for values of abstract data structure variables resulting from given values of concrete data structure variables by applying of the interpretation instruction I , then the corresponding concrete instruction possesses also a terminating computation. Since the values $e, f, c[i]$ and $d[i]$ belong to the set $\{1, 2, \dots, k\}$, all FIND-instructions possess always terminating computations. EQ and UNION evidently possess terminating computations.

By Theorem 1 J is partially correct and by Theorem 2 J is totally correct. Hence by Theorem 3 the program resulting from P by textual replacement of abstract instructions by corresponding concrete instructions is totally correct ^{with respect} to the submitted formulas α and β . \square

6. EXPRESSIBILITY OF PARTIAL AND TOTAL CORRECTNESS OF AN IMPLEMENTATION BY MEANS OF FORMULAS OF NONDETERMINISTIC ALGORITHMIC LOGIC

Let an algebra \mathcal{A} and its extension \mathcal{B} be defined as previously. Let us extend the language of ordinary algorithmic logic by the following three nondeterministic constructs:

- (1) for a vector of variables x , $[?x]$ means the nondeterministic assignment to x of values drawn from the appropriate basic types of \mathcal{B} ;
- (2) for a sequence of statements P_1, P_2, \dots, P_s , choice $[P_1, P_2, \dots, P_s]$ means the nondeterministic choice of one of the instructions P_1, P_2, \dots, P_s and the execution of it;
- (3) for a formula α , $\Box\alpha$ has value 1 if α yields 1 for all possible nondeterministic assignments and choices in α , otherwise $\Box\alpha$ has value 0.

Let $J = (\mathcal{M}, I)$ be an implementation over \mathcal{A} of a data structure type \mathcal{X} where $\mathcal{X} = \{K_0, K_1, \dots, K_n\}$ is an abstract data structure type over \mathcal{B} and $\mathcal{M} = \{M_0, M_1, \dots, M_n\}$ is a concrete data structure type over \mathcal{A} . As previously, assume that x is a vector of common variables, y is a vector of abstract data structure variables and z is a vector of concrete data structure variables. Let q be a Boolean variable which appear neither in \mathcal{X} nor in \mathcal{M} .

Theorem 4. J is partially correct iff the following formula is valid in \mathcal{B} :

PC: $K_0x=M_0x \wedge K_0y=M_0y \wedge \Box(\neg P(\neg \alpha))$ where

$P = \text{begin}$
 $\quad [?x]; M_0; [?q];$
 $\quad \text{while } q \text{ do}$
 $\quad \quad \text{begin}$
 $\quad \quad \quad [?x]; [?q];$
 $\quad \quad \quad \text{choice}[M_1, M_2, \dots, M_n]$
 $\quad \quad \text{end}; [?x]$
 $\quad \text{end}$ and
 $\alpha = \bigwedge_{1 \leq i \leq n} (M_i \text{ true} \Rightarrow (IK_1x=M_1x \wedge IK_1y=M_1y))$.

J is totally correct iff the following formula is valid in \mathcal{J}

TC: $K_0x=M_0x \wedge K_0y=M_0y \wedge \Box(\neg P(\neg \beta))$ where P is defined above and

$\beta : \bigwedge_{1 \leq i \leq n} (IK_1 \text{ true} \Rightarrow (IK_1x=M_1x \wedge IK_1y=M_1y))$.

Proof. First observe that $(Q \text{ true} \Rightarrow Q\alpha)$ is equivalent to $\neg(\neg\alpha)$ for any deterministic program Q. The next observations concern equivalent formulations of the statements defining the notions of partial and total correctness of implementations. It follows that:

(1) J is partially correct iff the formula $K_0x=M_0x \wedge K_0y=M_0y$ is valid in \mathcal{J} and for any \mathcal{M} -program $\mu = \mu_0, \mu_1, \dots, \mu_m$, for any terminating computation of it which results in $z=\bar{z}$, for each i , $1 \leq i \leq n$, for any values $x=\bar{x}$ such that M_i terminates for $x=\bar{x}$ and $z=\bar{z}$, the formula $IK_1x=M_1x \wedge IK_1y=M_1y$ is satisfied by $x=\bar{x}$ and $z=\bar{z}$.

(2) J is totally correct iff the formula $K_0x=M_0x \wedge K_0y=M_0y$ is valid in \mathcal{J} and for any \mathcal{M} -program $\mu = \mu_0, \mu_1, \dots, \mu_m$, for any terminating computation of it which results in $z=\bar{z}$, for each i , $1 \leq i \leq n$, for any values $x=\bar{x}$ such that K_i terminates for $x=\bar{x}$ and $y=I_{\bar{z}}(z=\bar{z})(y)$, the formula $IK_1x=M_1x \wedge IK_1y=M_1y$ is satisfied by $x=\bar{x}$ and $z=\bar{z}$.

(1) and (2) imply that PC and TC express the desired properties. \square

REFERENCES

- [1] AHO A.V., HOPCROFT J.E., ULLMAN J.D., The design and analysis of computer algorithms, Addison-Wesley, Reading, Mass. 1974
- [2] SALWICKI A., An introduction to algorithmic logic, Banach Centre Publications, vol.2, PWN, Warsaw, 1977
- [3] BANACHOWSKI L., On implementations of abstract data types, 26. Mathematical Logic in Computer Science, Salgotarjan 1978, Coll. Math.Soc.Janos Bolyai, 1981, 143-166
- [4] HOARE C.A.R., Proof of correctness of data representation, Acta Inf. 1(1972), 271-281

SEMANTICS AND IMPLEMENTATION OF PREFIXING AT MANY LEVELS⁺

W.M. Bartol
A. Kreczmar
A.I. Litwiniuk
H. Oktaba

Institute of Informatics
University of Warsaw
00-901 Warsaw, P.K.i N. Poland

Abstract

A generalization of Simula's prefixing of classes is presented. The notion of one-level prefixing is first introduced by means of the example of Simula 67; the semantics of a programming language with prefixing at many levels is then discussed and analyzed. The principles for efficiently implementing programming languages with prefixing of classes at many levels are described. A generalized display mechanism is introduced and the correctness of a display update algorithm is proved. A new data structure for efficient identification of dynamic objects is also presented.

Keywords: block structured programming languages, classes, prefixing, methods of implementation, Simula 67.

⁺This research was supported in part by "Zjednoczenie MERA" of Poland.

1. Introduction

The prefixing of classes is one of the most attractive and powerful mechanisms incorporated into the programming language Simula 67 (cf [4]). This tool allows a programmer to design a program in a structural, abstract way. To present briefly the main ideas of prefixing we start with the notion of a class.

Let us consider the following scheme of class declaration:

```
class A;
  attributes  $a_1, \dots, a_n$ ;
   $I_1; \dots; I_p$ ; inner;  $I_{p+1}; \dots; I_r$ 
end A;
```

where a_1, \dots, a_n are attributes (variables or, perhaps, other syntactic units like classes, procedures, functions etc.) and I_1, \dots, I_p , I_{p+1}, \dots, I_r are instructions of the class A. With the help of an object generator ("new A") one can create an object of the class A, i.e. create a frame (activation record) in the memory for attributes a_1, \dots, a_n and execute the instruction list $I_1, \dots, I_p, I_{p+1}, \dots, I_r$. When control returns to the object where the expression "new A" has been executed, the frame is not deallocated and a reference to that frame is transmitted as the value of the expression "new A". Hence, a reference to the object may be retained in a reference variable (e.g. $X := \text{new A}$, where X is a reference variable qualified by the class A).

The attributes of objects are accessible from outside as well as from inside the object. Remote accessing (e.g. $X.a_1$) allows one to use the attributes a_1, \dots, a_n from outside. Internal access occurs while executing the instructions of the object of A and any unit nested within it or during calls of the class's procedure attributes.

Consider now declaration scheme of a class B:

```
A class B;
  attributes  $b_1, \dots, b_m$ ;
   $J_1; \dots; J_s$ ; inner;  $J_{s+1}; \dots; J_t$ 
end B;
```


Class B is prefixed by A, i.e. B has attributes $a_1, \dots, a_n, b_1, \dots, b_m$ and the instruction list $I_1, \dots, I_p, J_1, \dots, J_s, J_{s+1}, \dots, J_t, I_{p+1}, \dots, I_r$ and B is called a subclass of A. One can create an object of class B in a similar way as was done for A, i.e. by $Y: \text{new B}$. Here Y may be a reference variable qualified by class B as well as by class A (for the general rules of this kind of assignment statement see [4]).

The following class C is a subclass of the classes B and A:

```
B class C;
    attributes  $c_1, \dots, c_k$ ;
     $K_1; \dots; K_u$ ; inner;  $K_{u+1}; \dots; K_v$ 
end C;
```

and it has the attributes $a_1, \dots, a_n, b_1, \dots, b_m, c_1, \dots, c_k$ and the instruction list $I_1, \dots, I_p, J_1, \dots, J_s, K_1, \dots, K_u, K_{u+1}, \dots, K_v, J_{s+1}, \dots, J_t, I_{p+1}, \dots, I_r$. The sequence of classes A, B, C is called the prefix sequence of the class C. Class C may in turn be used as a prefix of some other class, and so forth, but no class can occur in its own prefix sequence. Hence prefixing has a tree structure.

Blocks may also be prefixed. For instance, a block:

```
A begin
    attributes  $c_1, \dots, c_k$ ;
     $K_1; \dots; K_u$ 
end
```

is prefixed by the class A, i.e. it has the attributes $a_1, \dots, a_n, c_1, \dots, c_k$ and the instruction list $I_1, \dots, I_p, K_1, \dots, K_u, I_{p+1}, \dots, I_r$.

In Simula 67, perhaps because of the method chosen for the original implementation, there is an important restriction on prefixing; namely, a class may be used as a prefix only at the block level at which it has been declared. Before we explain the reasons for this restriction and possible ways of abolishing it, let us look at some examples which illustrate the difficulties arising from this restriction.

Suppose we have a declaration of a class PQ which provides the data structure of a priority queue of integers with maximal capacity defined by an input parameter n:

```
class PQ(n); integer n;
begin
```

```

integer procedure deletemin;
...
end deletemin;
procedure insert(x); integer x;
...
end insert;
...
end PQ;
In the following program:
begin
  class PQ(n); integer n;
  ...
  end PQ;
  ...
  begin integer n;
  read(n);
  PQ(n) begin
    ...
    end
  end
end

```

the declaration of PQ is not at the same level as the prefixed block, hence this construction is incorrect in Simula 67.

If the class PQ were translated separately and treated as being declared in the block at level 0, it would never be possible to use this data structure as a prefix in other block except the outermost one.

In Simula 67 this problem has been partially solved, because system classes like SIMSET and SIMULATION may be used at any level. But the user is not able to extend the library of system classes, which still forces him to rewrite the declarations at relevant block levels.

This situation becomes even more cumbersome if we want to make use of two data structures simultaneously and both of them are subclasses of one class. Consider for instance, the data structures A and B using lists as an auxiliary data system. Hence they ought to be subclasses of a class LIST. We have the following declarations:

```

class LIST;
...

```

```

end LIST;
LIST class A;
...
end A;
LIST class B;
...
end B;

```

and now we would like to open two prefixed blocks:

```

A begin
...
  B begin
    ...
    end
  end
end

```

Because of the restriction one must redeclare classes B and LIST at the level where B is used as prefix. Thus, redundancy is unavoidable.

Observe that with the possibility of separate translation and allowing prefixing at many levels we can develop software in a structural way. Any system or user class may be easily extended by the user and attached to the catalog of system classes without the necessity of re-compiling already compiled units and without the redundancy of the program text. Moreover, as we showed before, the user is able to make use of arbitrary data structures simultaneously by means of a prefixing mechanism instead of remote accessing (what speeds-up run-time of a program and clarifies its source code).

To conclude, we emphasize that prefixing at many levels is not merely a sophisticated technical problem in programming languages, but an essential step forward in developing an effective software methodology.

The structure of the paper is the following. In section 2 we give an informal insight, illustrated by examples, into some important semantic questions concerning many-level prefixing. Section 3 contains definitions and facts concerning the block structured programming languages, which are well known but necessary. Section 4 contains the formal definition of access to attributes in one-level prefixing (Simula 67). In section 5 we prove that the proposed semantics of the rules for many-level prefixing is correct. Section 6 gives a description of addressing algorithms for many-level prefixing. In particu-

lar, a generalized display mechanism is introduced, a mechanism which realizes an efficient access to attributes. In section 7 we discuss the various strategies of storage management and their impact on the semantics of the proposed construct.

2. Many-level prefixing (informal presentation)

The prefixing in Simula-67 is subject to an important restriction: a class may be used as a prefix only at the syntactic level of its declaration. Hereafter we shall call this prefixing "at one level".

In this paper we consider a Simula-like language, in which there is no such restriction and "many-level" prefixing is possible i.e. a class may be used as a prefix whenever its declaration is visible. To speak about such a language we must be able first to determine its semantics. One might think that prefixing "at many levels" is a trivial generalization of prefixing "at one level", but this is not the case.

The semantics of such a language is not obvious: in particular the rules defining access to object attributes cannot be deduced from the analogous Simula rules.

Consider the following program scheme (we follow Simula syntax):

```

L1: begin
    class A; begin real x;
        .
        .
        .
    end A;
L2: A begin real y;
    class B; begin
        .
        .
        x:=y;
        .
        .
    end B;
    .
    .
    .
new B;

```

```

.
.
L3: A begin real y;
      B class C; begin
          .
          .
          .
          y:=x;
          .
          .
          .
          end C;
      .
      .
      .
      new C;
      .
      .
      .
      end
end;

```

This program has the following block structure: the class A is declared in the outermost block of the program. It prefixes two blocks (one contained in the other) labelled L2 and L3, respectively. Note that the use of the same prefix for two blocks - one nested in the other - is not allowed in Simula-67.

The first prefixed block contains the declaration of a class B, while the second contains the declaration of a class C prefixed by B.

Let us consider the structure of objects created during the execution of the program. Every object of a prefixed class or block contains all attributes belonging to classes from their prefix sequences. In the above program the first object is created upon entry to the block labelled L1. Denote this object by p1. The second, denoted by p2, is created upon entry to the block labelled L2. This object contains two local real variables: x and y. The execution of the statement new B yields a third object (denoted by p3) corresponding to the class B. As indicated in the program scheme, variables x and y occur in the statements of B. Both variables denote attributes of the

object p2.

Upon entry to the block L3 a new object p4 containing two variables x and y is created. The execution of the statement new C yields a new object p5 (see Fig.1) of the class C.

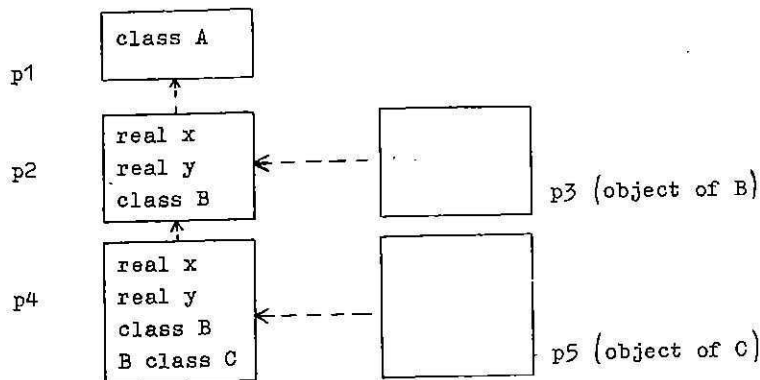


Fig.1.

According to the definition of prefixing the instruction list of C contains the instruction list of B. Therefore we must determine for each occurrence of the variables x and y in the instruction list of C the object from which the appropriate attribute is taken.

Consider first the statement $y:=x$ in the body of C of the object p5. Note that none of the occurrences of the variables x,y is local in C. The object p5 belongs to class C and the nearest block containing the attributes x,y and the declaration of C is the block L3. Hence, both variables denote attributes of the object p4, which represents the block L3.

There are, however, different ways of defining the semantics of the statement $x:=y$ from the class B of the object p5. The semantics of the statement can be based on a purely textual concatenation of the bodies of classes, as in Simula-67. We treat the declaration of class C as if it were concatenated with class B and declared in the block L3. Therefore both variables denote attributes of the object p4.

The semantics of the statement can be also defined in the following way: the syntactic unit to which the variable x is related is the class A, since A is the class in which x is declared; the syntactic unit to which the variable y is related is block L2. During the execution of statements in the object p5 the sequence of objects statically enclosing p5 is the following: p4,p3,p2,p1. In this sequence

p4 is the first object having attributes of the class A. Hence the variable x denotes an attribute of p4. The first object representing block L2 is the object p2, therefore the variable y denotes an attribute of p2.

From the above example it follows that there are some alternative ways of defining the semantics of assignment statement $x:=y$ executed in p5.

In this paper we chose the one described above as the second, and we present its precise and formal definition in Section 5.

Why is this way of defining the semantics preferable?

There are several reasons for this choice. The most important is that we are able to define it in a precise and formal way and we are able to implement it efficiently (cf Section 6).

In the semantics based on a purely textual concatenation we see no way of addressing attributes which would depend only on the place of variable declaration. In particular we are not able to assign a relative displacement (offset) to an identifier occurring in a class statement. Note that an identifier may relate to attributes with different relative displacements depending on the place where a class is used. Compare with the example: in the statement $x:=y$ of B the variable y relates to an attribute of p2 or p4 and these attributes may have different relative displacements. To illustrate the chosen semantics let us consider the program scheme structurally analogous to the example of Section 1.

```
begin
  class LIST;
    begin
      ref (...)head;
      procedure into(...);... head:=...; end;
    .
    .
    .
  end LIST;
LIST class QUEUE;
  begin
    procedure intoqueue; ... into(...) ...; ... end;
  .
  .
  .
end QUEUE;
```

```

LIST class DECK;
  begin
    procedure intodeck; ... into(...) ...; ... end;
    .
    .
    .
  end DECK;
L1: QUEUE begin
  L2: DECK begin
    .
    .
    .
    S1: intoqueue;
    S2: intodeck;
    end;
  end;
end

```

The above program contains declarations of classes: LIST, QUEUE, DECK. The class LIST describes the general structure of lists and contains the declaration of the variable "head" and the procedure "into", where that variable is used.

The classes QUEUE and DECK use the structure of LIST to describe the structures of queues and decks. In particular, they call the procedure "into" declared in LIST, and they use the variable "head" as its local attribute.

If we want to use both classes: QUEUE and DECK in a program, we may need two blocks prefixed by QUEUE and DECK, respectively. Moreover we wish the procedure "into" called in the body of "intoqueue" to be taken from the object representing the block prefixed by QUEUE; similarly, this procedure when called in the body of "intodeck" is to be taken from the object representing the block prefixed by DECK. Otherwise they should use the same attribute "head", which might destroy completely the proper execution of the program.

Denote the object created upon entry to the outermost block by p1. Objects created upon entries to blocks L1 and L2 will be denoted by p2 and p3, respectively.

The call of the procedure "intoqueue" (statement S1) yields a new object denoted by p4. The procedure "intoqueue" is an attribute of p2, so that the sequence of objects which statically enclose p4 is as follows: p4, p2, p1. The procedure "intoqueue" calls in turn the

procedure "into", which is declared in the class LIST: The first object in the sequence p_4, p_2, p_1 which contains attributes of LIST is p_2 , thus in our semantics "into" is an attribute of p_2 and "head" will be taken from p_2 . Analogous reasoning shows that the procedure "into" when called in the body of "intodeck" is an attribute of p_3 .

Thus the discussion shows that such informally presented semantics suits our purposes. In the subsequent sections the precise definition of this semantics and its implementation will be given.

3. Syntactic environment in programming languages without prefixing Static containers

Consider first the case of a programming language with block structure and without prefixing. By a syntactic unit in such a language we shall mean a block or a procedure. Arbitrary syntactic units will be denoted by U, V, W with indices or dashes, if necessary.

From the point of view of its block structure, any program may be treated as a tree T . The root of this tree $R(T)$ is the outermost block and for $U, V \in T$, U is the father of V iff V is declared in U (in definition blocks are treated as declarations in units where they appear). For the sake of simplicity of notation we shall write $V \text{ decl } U$ when V is declared in U (or alternatively, when U is the father of V in T).

Let decl^+ denote the transitive closure of the relation decl and let decl^* denote the transitive and reflexive closure of decl . So we have, in particular, $U \text{ decl}^* U$ and $U \text{ decl}^* R(T)$ for any U .

The level of a node in a tree T is introduced as usual, i.e. $\text{level}(R(T)) = 1$ and $\text{level}(U) = \text{level}(V) + 1$ if $V \text{ decl } U$.

Any variable and any syntactic unit except a block has a name, called an identifier, introduced at the moment of its declaration. The identifier is then used to represent the variable or the unit in a program. The question of distinction between identifiers and syntactic entities (variables and syntactic units) is essential, because the same identifier may be introduced by different declarations in the program text.

Let id denote an arbitrary identifier. We consider now an occurrence of an identifier id in a statement of a program. Since a declaration associates an identifier with a syntactic entity, for the occurrence of id one must determine a unit U such that a syntactic entity named id is declared in U . For the semantics of a program to be

unambiguous, the correspondence between occurrences of identifiers and syntactic entities should be unique, i.e. only one syntactic entity may be associated with the given occurrence of an identifier id . Let us assume that id occurs in a unit V , i.e. V is the innermost unit containing the considered occurrence of id . In the following definition we make precise what is meant by scope of declarations or visibility rules.

Definition 3.1.

By a static container of the occurrence of an identifier id in a unit V , denoted by $SC(id, V)$, we mean a syntactic unit U such that

- (a) id is declared in U ,
- (b) $V \text{ decl}^* U$,
- (c) there is no unit U' such that $V \text{ decl}^* U'$ and $U' \text{ decl}^+ U$ and id is declared in U' (i.e. U is the innermost unit enclosing V such that id is declared in U).

If $SC(id, V)$ does not exist, i.e. if there is no U such that (a) and (b) hold then of course the program is incorrect. Otherwise we say that the occurrence of id is local in V if $V = SC(id, V)$, and non-local in V if $V \neq SC(id, V)$.

Dynamic containers

During a program's execution we can deal at the same time with many objects of the same syntactic unit, hence a computation of any instruction in an object requires identification and access to all the syntactic entities that it uses. In Algol-60 instances of blocks and procedures may be treated as the examples of objects, (in Simula-67 this is augmented with the objects of classes). The collection of objects of a syntactic unit U will be denoted by $|U|$. The objects themselves will be denoted by small latin letters p, q, r with indices, if necessary.

Consider an object $p \in |U|$. If the occurrence of an identifier id is local in a unit U , then the syntactic entity identified by id is situated within the object p . Hence there is no problem either with identification or with access to this syntactic entity. In general, however, for any id such that $SC(id, U)$ exists, we must determine a unique object q such that $q \in |SC(id, U)|$. Then during the execution of the instruction list of U in the object p , the syntactic entity identified by id will be taken from q . Such an object q will be called a dynamic container of id with respect to p , and will be denoted by $DC(id, p)$. Dynamic containers are unequivocally determined by means

of static links.

Upon a unit U is entered an object of this unit is allocated and initialized. It contains some system pointers in addition to declared attributes, for example the dynamic link (DL) which points to the calling object and the static link (SL) pointing to the object which is its syntactic father. We shall write $p.SL=q$ when SL link of the object p points to the object q . (If $p.SL$ is not defined, then we shall write $p.SL=\text{none}$.)

An object q is called the syntactic father of an object p , since q must be the object of a unit V where U is declared, i.e. if $p.SL=q$, $p \in |U|$, $q \in |V|$, then $U \text{ decl } V$.

A sequence p_k, \dots, p_1 of objects is called the SL chain of the object p_k , if $p_1.SL=\text{none}$ and $p_i.SL=p_{i-1}$ for $i=k, \dots, 2$. The SL chain of an object p will be denoted by $SL(p)$.

The SL chains define completely and uniquely syntactic environment of objects. This follows from the well-known results quoted below:

Lemma 3.1.

- (a) If $SL(p_k) = p_k, \dots, p_1$ and $p_i \in |U_i|$ for $i=k, \dots, 1$, then the sequence U_k, \dots, U_1 is a path from U_k to $R(T)$ in the tree T ,
- (b) Let $SL(p_k) = p_k, \dots, p_1$ and $p_k \in |V|$. If $SC(id, V)$ exists, then there is a unique i , $1 \leq i \leq k$, such that $p_i \in |SC(id, V)|$. □

Lemma 3.1 (b) shows that the SL chain of an object defines completely and uniquely its syntactic environment. All syntactic entities which can be used in V are uniquely situated in $SL(p_k)$. Consequently the dynamic container $DC(id, p_k)$ of the occurrence of id with respect to the object p_k is defined as a unique object p_i belonging to $SL(p_k)$ such that $p_i \in |SC(id, V)|$.

The way SL links are defined during a program's execution induces the semantics of identifiers. The following algorithm determines exactly what should be done with SL links in order to obtain the most natural semantics (cf [7]).

Algorithm 3.1.

We can assume the only one object of the outermost block $R(T)$ may be entered and, of course, for that object $SL = \text{none}$. Consider now the call of a unit U in an object $r_k \in |V|$. If id identifies U , then according to the definition 3.1 U is declared in $SC(id, V)$. The syntactic father of $p \in |U|$ must be the object of the unit $SC(id, V)$, i.e. the

unit where U is declared. Let $SL(r_k) = r_k, \dots, r_1$. By Lemma 3.1 (b) there is a unique i , $1 \leq i \leq k$, such that $r_i \in |SC(id, V)|$. Then define $p.SL = r_i$, i.e. r_i becomes the syntactic father of p . (cf Fig.2). \square

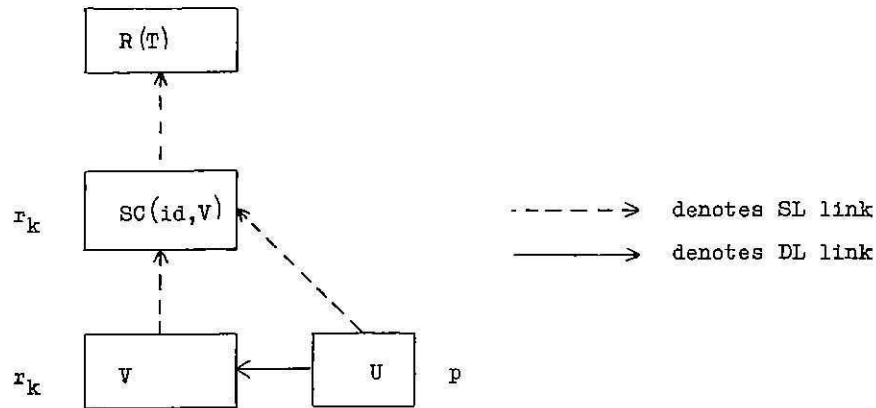


Fig.2.

4. Prefixing at one level

Prefix structure of a program

In this section we shall consider a programming language with block structure and one-level prefixing, i.e. exactly the case of Simula 67.

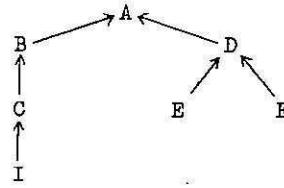
From the point of view of its prefix structure, any program may be treated as a forest of prefix trees $\{P_i\}$. Each prefix structure of a program is a tree P_i where for $U, V \in P_i$, U is the father of V iff U is the prefix of V and the root of P_i is a unique element of P_i without any prefix. Similarly to the relation $decl$ we introduce the relation $pref$, i.e. $U \text{ pref } V$ iff U is the prefix of V .

By a prefix sequence of a unit U (denoted by $prefseq(U)$) we mean a sequence V_1, \dots, V_k of units such that $V_k = U$, V_1 has no prefix and $V_i \text{ pref } V_{i+1}$ for $i = 1, \dots, k-1$. The example of the block and the prefix structures of a program are illustrated in Figure 3.

```

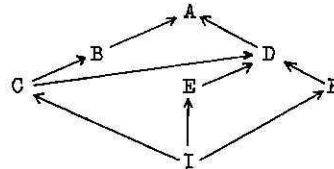
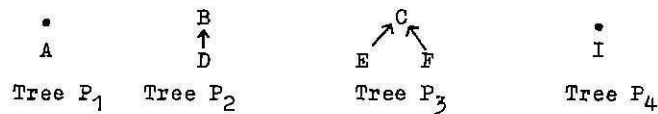
A: begin ref D Z;
  class B; begin ref(C)X1,X2;
    class C; begin
      class I; begin
        ...
      end I;
    end C;
  end B;
B class D; begin
  C class E; begin ref(I)Y1;
    ...
    Y1:- new I;
    ...
  end E;
  C class F; begin ref(I)Y2;
    ...
    Y2:- new I;
    ...
  end F;
  ...
  X1:- new E; X2:- new F;
  ...
end D;
Z:- new D;
end A;

```

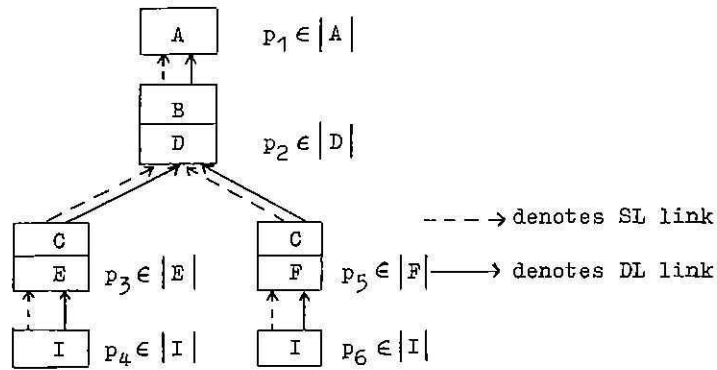


tree T

scheme of a block structure



Graph G



Graphs of SL' and DL's

Fig. 3.

Let pref^+ denote the transitive closure of pref and let pref^* denote the transitive and reflexive closure of pref . Then, in particular, $U \text{ pref}^* U$ for any U , $U \text{ pref}^* V$ for any $U \in \text{prefseq}(V)$ etc.

Note now that if $U \text{ pref}^* V$, then an attribute of U is an attribute of V as well. In particular, a syntactic unit W may be an attribute of U and, hence, it will be an attribute of V . Let us denote this extension of the relation decl by attr , i.e. $W \text{ attr } V$ iff there is a unit U such that $W \text{ decl } U$ and $U \text{ pref}^* V$. While the relation decl always defines a tree, the relation attr need not define a tree. Denote by G the graph determined by the relation attr . Since the relation attr is the extension of decl , the tree T is a subgraph of the graph G .

In Figure 3 the syntactic unit C is the attribute of the syntactic unit D , because $C \text{ decl } B$ and $B \text{ pref } D$. Thus C being the attribute of D may be used as a prefix of the syntactic units E and F . Finally, $I \text{ decl } C$ and $C \text{ pref } E$ implies $I \text{ attr } E$, similarly $I \text{ decl } C$ and $C \text{ pref } F$ implies $I \text{ attr } F$.

One-level prefixing is characterized by the following restriction:

(4.1) If $U \text{ pref } V$, then $\text{level}(U) = \text{level}(V)$.

(In words, U may prefix V only if both have the same level in the tree T .) This restriction has many interesting consequences which make the implementation problem almost trivial. First, as an immediate consequence of (4.1) we obtain the following lemma.

Lemma 4.1.

- (a) If $U \text{ attr } V$, then $\text{level}(U) = \text{level}(V) + 1$,
- (b) G is a directed acyclic graph with one sink $R(T)$,
- (c) Every path in G from U to $R(T)$ has length $\text{level}(U)$. □

The definition of a static container for the occurrence of an identifier in a unit is generalized in the following way:

Definition 4.1.

By a static container of the occurrence of an identifier id in a unit V denoted $SC(id, V)$ we mean a syntactic unit U such that id is declared in U and there is a syntactic unit W such that

- (a) $U \text{ pref}^* W$,
- (b) $V \text{ decl}^* W$,
- (c) there is no unit W' such that $V \text{ decl}^* W'$ and $W' \text{ decl}^+ W$ and id is the attribute of W' ,
- (d) there is no unit W' such that $U \text{ pref}^+ W'$ and $W' \text{ pref}^* W$ and id is declared in W' .

In block structured languages without prefixing we search for the innermost unit W such that id is declared in W and W contains a unit V with the occurrence of id . However, according to the definition of prefixing, the attributes of a prefixing unit are contained in the set of attributes of prefixed unit. This implies that the relation pref is stronger than the relation decl in the following sense: in the process of searching for a static container, we search for it first in the prefix sequence and then in the lower levels of the block structure of a program. Conditions (a)-(c) of definition 4.1 require that we search for the innermost unit W such that id is the attribute of W (U is a unit where the searched syntactic entity is declared). Condition (d) says that U is the nearest prefix of W satisfying the conditions (a)-(c).

We now present an algorithm determining the static container $SC(id, V)$.

Algorithm 4.1.

Start from V . If there is no declaration of id , look for it in $\text{pref-seq}(V)$ reading from right to left. If id is not an attribute of V , then take V' such that $V \text{ decl } V'$ and repeat the above process for V' . If id is not an attribute of V' , then take V'' such that $V' \text{ decl } V''$ and so on. When the algorithm terminates on the outermost block without finding the required declaration, the static container $SC(id, V)$ does not exist and a program is incorrect. □

Look at Figure 3. We have $SC(I, E) = C = SC(I, F)$, $SC(Y1, E) = E$, $SC(Y2, F) = F$, $SC(X1, D) = SC(X2, D) = B$ and $SC(D, A) = SC(Z, A) = A$.

According to the definition of prefixing, the attributes coming from a prefix sequence are the attributes of a prefixed unit, hence, all of them are local in that unit. Thus we say that the occurrence of an identifier id is local in U if $SC(id, U) \text{ pref}^* U$, otherwise the occurrence of id is non-local in U .

In the example on Figure 3 all occurrences of identifiers are local.

Dynamic containers

Let $\text{prefseq}(U_k) = U_1, \dots, U_k$ and let us consider an object $p \in |U_k|$. This object consists of layers corresponding to the syntactic units U_1, \dots, U_k . (In Figure 3 p_1 has a layer A , p_2 has layers B, D , p_3 has layers C, E , p_5 has layers C, F , and p_4, p_6 have a layer I .)

Now consider the execution of the instruction lists of units U_1, \dots, U_k . If an identifier id occurs in a unit $U_i, 1 \leq i \leq k$, then for any object $p \in |U_k|$ we must determine a unique object q such that $q \in |V|$ and $SC(id, U_i) \text{ pref}^* V$. It means that the object q has a layer which corresponds to the static container for the occurrence of id in a unit U_i . The object q will be called a dynamic container of the occurrence of id in a unit U_i with respect to the object p , and will be denoted by $DC(id, U_i, p)$. Dynamic containers will be uniquely determined by means of static links, as before. However, the definition of a syntactic father is more general. In fact, if $p.SL = q$, $p \in |U|$, $q \in |V|$, then U need not be declared in V .

Look at Figure 3. The object p_2 is created by the instruction $Z:-\text{new } D$, its syntactic father is, of course, the object p_1 . In this case D decl A . The object p_3 is created by the instruction $X1:-\text{new } I$ and its syntactic father is p_2 . In this case E decl D . The object p_4 is created by the instruction $Y1:-\text{new } I$ and its syntactic father is evidently p_3 . In this case I is not declared in E but in C . Hence the simple rule of Algol 60 does not work. The syntactic father of p_4 is the object p_3 such that I is the attribute of E (not necessarily declared in E). Similarly, the syntactic father of p_5 is p_2 , and F decl D , finally the syntactic father of p_6 is p_5 , and I attr F .

The example shows the necessity for a more general definition of syntactic father of an object: if $p \in |U|$ and $p.SL = q$, then q should be an object of a unit V such that $U \text{ attr } V$ (previously U decl V). The definition of SL chain remains the same as in Section 3. Before we present an algorithm of setting SL links, we prove a lemma analogous to Lemma 3.1 which is of basic importance for the whole con-

struction.

Lemma 4.2.

- (a) If $SL(p_k) = p_k, \dots, p_1$ and $p_i \in |U_i|$ for $i=k, \dots, 1$, then the sequence U_k, \dots, U_1 is a path from U_k to $R(T)$ in the graph G ,
 (b) Let $SL(p_k) = p_k, \dots, p_1$ and $p_i \in |U_i|$ for $i=k, \dots, 1$. If $SC(id, V)$ exists and $V \text{ pref}^* U_k$, then there is a unique i , $1 \leq i \leq k$, such that $SC(id, V) \text{ pref}^* U_i$.

Proof

By the definition of the syntactic father, if $p_{i+1} \in |U_{i+1}|$ and $p_i \in |U_i|$, then $U_{i+1} \text{ attr } U_i$, for $i = k-1, \dots, 1$. Hence U_k, \dots, U_1 is a path from U_k to $R(T)$ in the graph G . Thus (a) is proved.

Now by Lemma 4.1 $\text{level}(U_i) = i$ for $i=k, \dots, 1$. Assume that there are two such integers, i, j , $1 \leq i < j \leq k$, that $SC(id, V) \text{ pref}^* U_i$ and $SC(id, V) \text{ pref}^* U_j$. By the restriction 4.1 $\text{level}(SC(id, V)) = \text{level}(U_i)$ and $\text{level}(SC(id, V)) = \text{level}(U_j)$. Hence $\text{level}(SC(id, V)) = i = j$, which is impossible.

The proof that such an i exists is given in Section 5 (Lemma 5.3), where the more general case is considered; namely the case of prefixing at many levels. For this reason we do not repeat this proof in a much simpler case and leave it to the next section. \square

Now we are able to present an algorithm which is an immediate generalization of the algorithm 3.1.

Algorithm 4.2.

We assume the only one object of the outermost block $R(T)$ may be entered, and for that object $SL = \text{none}$.

Consider now an object $p \in |U|$ created in an object $r_m \in |V_k|$. Let $\text{prefseq}(V_k) = v_1, \dots, v_k$, and let the instruction which creates p occur in a unit v_i , $1 \leq i \leq k$. If id identifies U , then according to the definition 4.1 U is declared in $SC(id, v_i)$. The syntactic father of p should be an object containing $SC(id, v_i)$ as a layer.

Let $SL(r_m) = r_m, \dots, r_1$. By Lemma 4.2(b) there is a unique j , $1 \leq j \leq m$, such that $SC(id, v_i)$ is the layer of r_j . Then define $p.SL = r_j$. \square

Figure 4 shows this general situation. When the statement of a unit v_i with the occurrence of id is being executed, in the SL chain of r_m there is a unique object r_j which may be the syntactic father of p .

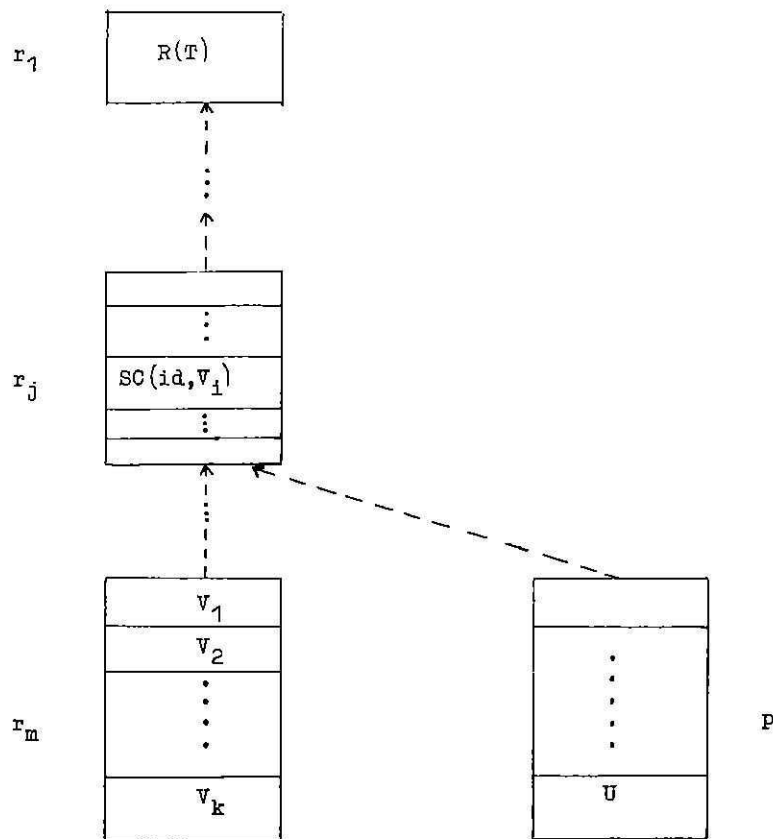


Figure 4

5. Syntactic environment in a programming language with prefixing at many levels.

Existence of a syntactic environment

In this section we shall analyze the situation when Simula's restriction (4.1) is left out. A programming language with block structure and prefixing at many levels, i.e. when (4.1) is not binding, possesses some amazing properties. First we are not able to prove a lemma analogous to Lemma 4.2, where the existence and the uniqueness

ness of the syntactic environment for prefixing at one-level is proved. In particular, the analogon of Lemma 4.2(b) does not hold. However, we can show that for a static container $SC(id, V)$ and $V \text{ pref}^* U_k$ there is at least one i , $1 \leq i \leq k$ such that $SC(id, V) \text{ pref}^* U_i$, where $SL(p_k) = p_k, \dots, p_1$, U_1 . Lemma 4.2(b) shows the uniqueness of such an i , and thus there is no problem with definition of Simula's semantics. Here the situation is not so clear.

The proof of the existence of such an i , $1 \leq i \leq k$, is given in the following three lemmas. Lemma 5.1 is auxiliary and justifies the implication which is used later in the proof of Lemma 5.2. Lemma 5.2 is crucial for the whole proof. It shows that graph G satisfies the desired property. The proof of this lemma is carried out by double induction, with respect to the length of a path U_k, \dots, U_1 in G , and with respect to the length of the prefix sequence of U_k . At last Lemma 5.3 is a simple corollary of the Lemma 5.2.

Lemma 5.1.

Let the sequence U_k, \dots, U_1 be a path in the graph G from U_k to $R(T)$.

Assumption If $V \text{ pref}^* U_k$ and $V \text{ decl } W$, then there exists j , $1 \leq j < k$, such that $W \text{ pref}^* U_j$.

Conclusion If $V \text{ pref}^* U_k$ and $V \text{ decl}^* W$, then there exists t , $1 \leq t \leq k$, such that $W \text{ pref}^* U_t$.

Proof.

First note that the above implication has the following meaning. Assumption says that for any V from $\text{prefseq}(U_k)$ and declared in W there is U_j on the path U_{k-1}, \dots, U_1 such that $W \text{ pref}^* U_j$ (cf Fig.5). Conclusion generalizes this property. Namely, for any V from $\text{prefseq}(U_k)$ and for any W such that $V \text{ decl}^* W$ there is U_t on the path U_k, \dots, U_1 such that $W \text{ pref}^* U_t$, (cf Fig.5).

We shall prove the conclusion by induction on the length of path from V to W in the tree T . If $V=W$ and $V \text{ pref}^* U_k$, then $W \text{ pref}^* U_k$. Hence $t=k$ in this case.

Now consider units V and W such that $V \text{ decl}^+ W$. Hence there exists a unit W' such that $V \text{ decl } W'$ and $W' \text{ decl}^* W$. If $V \text{ pref}^* U_k$ and $V \text{ decl } W'$, then it follows from the assumption that there is $1 \leq j < k$ such that $W' \text{ pref}^* U_j$. Now $W' \text{ pref}^* U_j$ and $W' \text{ decl}^* W$, where the length of the path from W' to W is less than the length of the path from V to W . Hence by inductive assumption there exists $1 \leq t \leq j$ such that $W \text{ pref}^* U_t$. □

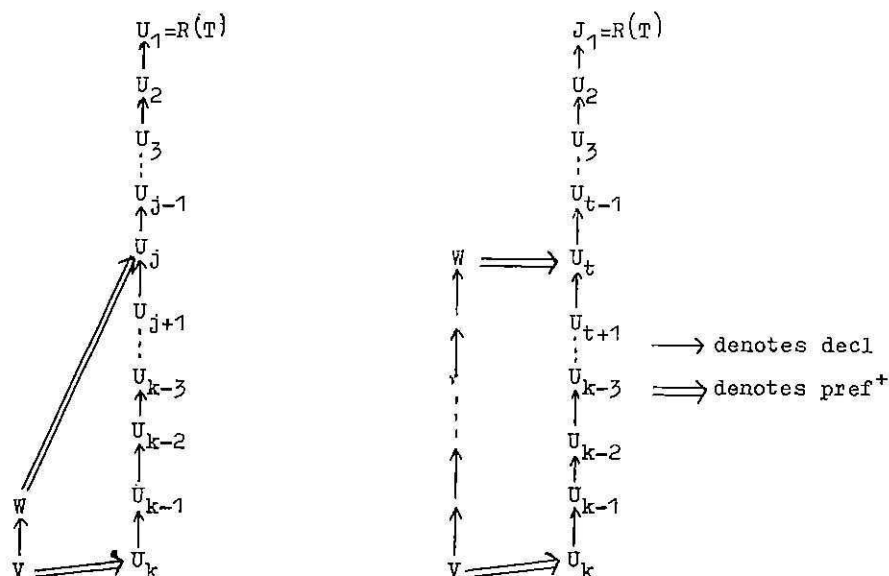


Figure 5

Lemma 5.2.

Let the sequence U_k, \dots, U_1 be a path in the graph G from U_k to $R(T)$. If $V \text{ pref}^* U_k$ and $V \text{ decl}^* W$, then there exists i , $1 \leq i \leq k$ such that $W \text{ pref}^* U_i$.

Proof.

First note that the lemma is simply the conclusion of the previous one. However, it should be proved without the assumption. Since Lemma 5.1 has just been proved, it is sufficient to prove its assumption, i.e.

(5.1) if $V \text{ decl} W$ and $V \text{ pref}^* U_k$, then there is j , $1 \leq j < k$ such that $W \text{ pref}^* U_j$.

The proof is carried out by induction on the length of the sequence U_k, \dots, U_1 . For $k=1$, $U_k = R(T)$. Thus $V \text{ pref}^* R(T)$ iff $V = R(T)$ and $V \text{ decl} W$ for no W .

Assume now that (5.1) holds for all sequences of length less than k , $k \geq 2$. For a sequence U_k, \dots, U_1 let $V \text{ pref}^* U_k$ and $V \text{ decl} W$. We shall now use induction on the length of the prefix sequence of the unit U_k to prove that $W \text{ pref}^* U_j$ for some $j < k$.

The beginning is simple since for $V=U_k$ we have $U_k \text{ decl } W$ and consequently $W \text{ pref}^* U_{k-1}$ (by the definition of the relation attr). Assume that (5.1) holds for all prefix sequences of the length less than h . Let $V \text{ pref}^* U_k$ and suppose that the length of the prefix sequence from V to U_k is $h \geq 2$. For some units V', W' we have $V \text{ pref}^* V' \text{ pref}^* U_k$ and $V' \text{ decl } W'$. The length of the prefix sequence from V' to U_k is $h-1$. We infer from the inductive assumption that $W' \text{ pref}^* U_j$ for some $j < k$. Now, since $V \text{ pref}^* V'$ and $V' \text{ decl } W'$, the syntactic container $SC(id, W')$ exists, where id identifies V (V occurs in W'). By Definition 4.1 and because $V \text{ decl } W$, there is a unit W'' such that $W \text{ pref}^* W''$ and $W' \text{ decl}^* W''$. The length of the sequence U_j, \dots, U_1 is less than k and $W' \text{ pref}^* U_j$ so, from the inductive assumption on k , we infer that for a unit \bar{W} such that $W' \text{ decl } \bar{W}$ there is $m < j$ and $\bar{W} \text{ pref}^* U_m$.

By Lemma 5.1 if $W' \text{ decl}^* \bar{W}$ and $W' \text{ pref}^* U_j$ there is $1 \leq m \leq j$ such that $\bar{W} \text{ pref}^* U_m$. Since $W' \text{ decl}^* W''$ and $W' \text{ pref}^* U_j$, taking \bar{W} as W'' , we obtain $W'' \text{ pref}^* U_m$. Finally, $W \text{ pref}^* W''$ and $W'' \text{ pref}^* U_m$, hence $W \text{ pref}^* U_m$ where $1 \leq m < k$. Thus we have proved (5.1) and the lemma. \square

Lemma 5.3.

Let $SL(p_k) = p_k, \dots, p_1$ and $p_i \in |U_i|$ for $i=1, \dots, k$. If $SC(id, V)$ exists and $V \text{ pref}^* U_k$, then there is i , $1 \leq i \leq k$ such that $SC(id, V) \text{ pref}^* U_i$.

Proof.

From the definition of the SL chain, U_k, \dots, U_1 is a path in the graph G . Since $SC(id, V)$ exists, there is a unit W such that $V \text{ decl}^* W$ and $SC(id, V) \text{ pref}^* W$. We have $V \text{ pref}^* U_k$ and $V \text{ decl}^* W$, and by Lemma 5.2 there is i , $1 \leq i \leq k$ such that $W \text{ pref}^* U_i$. But $SC(id, V) \text{ pref}^* W$ and $W \text{ pref}^* U_i$ implies $SC(id, V) \text{ pref}^* U_i$. \square

Dynamic containers

During the execution of the instruction list of an object $p \in |U|$, we must be able to indicate the dynamic container $DC(id, V, p)$ for any identifier id occurring in any unit V belonging to $\text{prefseq}(U)$. To achieve this goal we wish to use the SL chain of the object p , as in Simula 67. Unfortunately, in the case of many-level prefixing the SL chain does not uniquely define the syntactic environment of p , since the same unit may occur more than once as a layer in $SL(p)$. (Lemma 5.3 guarantees a dynamic container belongs to SL chain but not exactly once).

This new complication is well illustrated on Figure 1. The SL chain of the object $p_5 \in |C|$ contains the layer A twice, in the object p_2 and p_4 .

Hence it is necessary to introduce a uniform rule for determining dynamic containers. It seems that there are only two concurrent choices. We may take the nearest or the farthest from the given object on its SL chain. However, the second choice is impossible because it contradicts the standard understanding of locality. Consider an occurrence of id local in V and an object $p \in |U|$ containing a layer corresponding to a syntactic unit V . Assume the chain $SL(p)$ contains another object q with a layer corresponding to V . Then, of course, a dynamic container $DC(id, V, p)$ should be the object p , not the object q (for a concrete example see Section 2, where the program with two data structures `QUEUE` and `DECK` is considered).

From the above discussion we can infer a new definition of a dynamic container as well as an algorithm which computes SL links.

Definition 5.1.

Let $SL(r) = r_m, \dots, r_1$ be the SL chain of an object $r \in |V_k|$ and let $\text{prefseq}(V_k) = V_1, \dots, V_k$. Consider an occurrence of an identifier id in a unit V_i . We shall say that r_j is the dynamic container for the occurrence of id in a unit V_i with respect to the object r if r_j is the nearest object to r in $SL(r)$ such that $SC(id, V_i)$ is a layer of r_j .

Algorithm 5.1.

The start is the same as usual. Consider an object $p \in |U|$ created in an object $r \in |V_k|$. Let $\text{prefseq}(V_k) = V_1, \dots, V_k$ and let the instruction which creates p occur in a unit V_i , $1 \leq i \leq k$. If id identifies U , then according to the definition 4.1., U is declared in $SC(id, V_i)$. Let $SL(r) = r_m, \dots, r_1$ be the SL chain of r . By lemma 5.3. there is j , $1 \leq j \leq m$, such that $SC(id, V_i)$ is the layer of r_j . Let j' be the largest j satisfying this condition i.e. $r_{j'}$ is the dynamic container of the occurrence of id in the unit V_i with respect to r . Then define $p.SL = r_{j'}, \dots$



6. The addressing algorithm and its correctness.

In this section we shall describe an addressing algorithm for a language with many-level prefixing. The correctness of this algorithm will be proved.

Addressing in Algol and Simula

Let us start with some remarks on an addressing algorithm for the Algol-like language invented by E. Dijkstra ([6], [7]). Let id be a name of a variable v occurring in U and let $SC(id, U) = V$. Then the

variable v is identified by a pair:

$(\text{level}(V), \text{offset}(v))$

where $\text{offset}(v)$ is a relative displacement of v in a memory frame. Note that both quantities $\text{level}(V)$ and $\text{offset}(v)$ may be computed at compile time. The run-time address of v is evaluated by a simple formula:

$\text{DISPLAY}[\text{level}(V)] + \text{offset}(v)$

where DISPLAY is a running system array updated during run-time.

When an object $p \in |U|$ is being executed, $\text{DISPLAY}[i]$ for $i = \text{level}(U), \dots, 1$ must point to the members of the SL chain of p .

When an object $p \in |U|$ is being generated, it is sufficient to set

$\text{DISPLAY}[\text{level}(U)] := p;$

since for $m < \text{level}(U)$, $\text{DISPLAY}[m]$ must be well defined. But when p is reentered the next time (i.e. through DL or goto statement), the following DISPLAY update algorithm is used:

```
X := p;
for k := -level(U) step -1 until 1 do
  begin
    DISPLAY[k] := X; X := X.SL;
  end
```

For a language with many-level prefixing we postulate that the addressing algorithm is efficient as in the case described above.

However, from the discussion given below, it follows that the same method of attributes identification as in Algol-60 (and Simula-67) is not possible.

Let U be an arbitrary unit with prefix sequence U_1, \dots, U_n . It is easy to observe that the prefix sequence has the following property: for every i , $1 \leq i \leq n$, $\text{level}(U_i) \leq \text{level}(U_{i+1})$, where $\text{level}(U_i)$ is determined from the tree T . Due to this property it is not possible to assign one level to all attributes of a given object p since they may be declared in units of different syntactic levels. Hence the local attributes of the object p should be addressed relative to many elements of DISPLAY . (Note that in Simula 67 the equality $\text{level}(U_i) = \text{level}(U_{i+1})$ holds for all U_i belonging to the prefix sequence of U . Thus, the addressing algorithm is exactly the same as in Algol 60). Consider the following example:

```

B1:begin
  class A;... end A;
  B2:begin
    A class B;... end B;
    ...
    new B;
    ...
  end;
...
end

```

When the object r of class B (generated by `new B`) is executed, the SL chain of r is described at Fig.6.

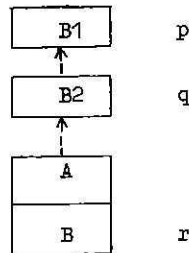


Fig.6.

The Algol-like rule, that $\text{DISPLAY}[3]=r$ and $\text{DISPLAY}[2]=q$, is not valid because the attributes of the object r declared in the unit A ought to be addressed with respect to $\text{level}(A)=2$.

In order to avoid these difficulties the assignment of numbers to syntactic units is modified so that levels determined by the program tree T must not be used.

Generalized DISPLAY

To every unit U of a given program we assign a unique number, called a unit number $\text{nr}(U)$, determined by any enumeration of tree T . To every id occurring in a unit U we assign a pair of numbers $\text{nr}(\text{SC}(\text{id}, U))$ and an offset, where the offset is evaluated taking into account all attributes of $\text{prefseq}(\text{SC}(\text{id}, U))$.

A prefix number sequence $\text{pns}(U)$ of a unit U is a sequence

$nr(U_1), \dots, nr(U_n)$, where $U_1, \dots, U_n = \text{prefseq}(U)$.

The vector DISPLAY is replaced by the vector GDISPLAY, the length of which is equal to the number of vertices of T.

Now we present an algorithm which computes relevant items of GDISPLAY every time an object $p \in |U|$ is entered. Let $SL(p) = p_m, \dots, p_1$; then the GDISPLAY update algorithm has the form:

Algorithm 6.1.

for $k := 1$ step 1 until m do
update $GD(p_k)$;

The instruction update $GD(p_k)$ consists of the assignment:

$GDISPLAY[n_1] := GDISPLAY[n_2] := \dots := GDISPLAY[n_{d_k}] := p_k$,

where $p_k \in |U_k|$ and the prefix number sequence of U_k is $pns(U_k) = n_1, \dots, n_{d_k}$. \square

Observe that for every object $p \in |U|$ the cost of update $GD(p)$ is constant, depending only on the unit U prefix sequence length. The correctness of the GDISPLAY update algorithm can be proved with the help of the following lemma.

Lemma 6.1.

Let $SL(p) = p_m, \dots, p_1$, where $p_i \in |U_i|$ for $i = m, \dots, 1$. If id is non-local in V , $V \text{ pref}^* U_m$ and p_j is a dynamic container for id ($p_j = DC(id, V, p)$), then id is non-local in any U_k for $k = j+1, \dots, m$.

Proof follows immediately from the definition 5.1 of a dynamic container. \square

Theorem 6.1 (correctness of the GDISPLAY update algorithm)

Let $SL(p) = p_m, \dots, p_1$, where $p_i \in |U_i|$ for $i = m, \dots, 1$, and assume that the GDISPLAY update algorithm has been executed for an object p . If the occurrence of id is represented by a pair (n, offset) and id occurs in V such that $V \text{ pref}^* U_m$, then $GDISPLAY[n] = p_j$ ($m \geq j \geq 1$), where $p_j = DC(id, V, p)$.

Proof:

When id is local in U_m , then the dynamic container of id is equal to p and n belongs to $pns(U_m)$. It follows from the algorithm that $GDISPLAY[n] = p_m = p$.

When id is non-local in V , $V \text{ pref}^* U_m$ and $p_j = DC(id, V, p)$, then by Lemma 6.1 for every $k = j+1, \dots, m$ id is non-local in U_k , hence $nr(SC(id, V)) = n$ does not belong to $pns(U_k)$. Since p_j is a dynamic container of id , it follows that $SC(id, V) \text{ pref}^* U_j$; thus n belongs

to $\text{pns}(U_j)$. Therefore, after executing the update algorithm loop "for $k:=j$ " we have $\text{GDISPLAY}[n] = p_j$ and by the Lemma 6.1 this value will not be changed for $k=j+1, \dots, m$. \square

This theorem implies the correctness of the run-time addressing algorithm given by a formula:

$$\text{GDISPLAY}[n] + \text{offset}$$

where the pair (n, offset) represents an attribute in a program.

The following example illustrates the use of the GDISPLAY mechanism. Let us consider the extended scheme of the program given in the previous example.

```

B1[1]: begin
    class A[2]; begin real x[2,m]; ... end A;
    B2[3]: begin real y[3,n];
        A class B[4]; begin real z[4,k]; ... end B;
        ...
        new B;
        ...
    end;
    ...
end

```

In this program every unit has a unit number given in brackets and every variable is identified by a pair of numbers: the first is a unit number of the static container of this variable and the second is a displacement in a memory frame. Consider the execution of the statement new B. A new object r of class B is created, the SL chain of r (see Fig.6) consists of the objects r, q (the block B2) and p (the block B1). Before control passes to the object r we must execute the GDISPLAY update algorithm. Fig.7 shows the contents of the vector GDISPLAY after its execution.

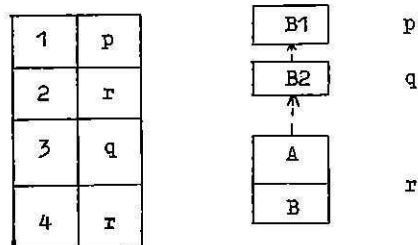


Fig.7.

Note that the attributes x and z of the object r are identified by two different unit numbers. However, due to the GDISPLAY update algorithm, all the elements of the vector GDISPLAY corresponding to the prefix number sequence of the unit B refer to the object r . Thus, the addressing formulas:

$$\begin{aligned} & \text{GDISPLAY}[2] + m \\ \text{and } & \text{GDISPLAY}[4] + k \end{aligned}$$

compute the addresses of x and z respectively in the frame of the object r .

7. Storage management

In this section we discuss briefly possible strategies of storage management and their influence on the semantics of the language with many-level prefixing. We propose a new approach to the problem and some principles of implementation.

Terminated objects accessibility

Consider first the problem of the accessibility of terminated objects. By a terminated object we mean an object in which control has passed through the final end.

Two different cases occur in Simula 67. A block (or a procedure) object is not accessible after its termination while the termination of a class object does not affect its accessibility. The property that a block object becomes inaccessible after its termination results only from the static properties of the correct program and may be statically checked.

Note another important property of Simula 67. The SL chain of the object being executed contains no terminated objects. It follows from the above properties that the activation record for a block or a procedure may be deleted from a memory as soon as this object is terminated.

The situation is quite different when many-level prefixing is allowed. Consider the following example:

```
L1: begin ref(A)X;
      class A;
      ...
      end A;
```

```

L2: begin integer j;
    A class B;
    begin
        procedure P;
        ...
        j:=j+1;
        ...
    end P;
end B;
X:-new B;
end L2;
XquaB.P; comment XquaB.P denotes instantaneous qualification which
changes the qualification of X;
end L1;

```

After the execution of the assignment X:-new B there exist three objects: p of block L1, q of block L2 and r of class B, the latter pointed by X. Recall that this assignment is valid because X is qualified by class A and A prefixes B.

Observe now the instruction XquaB.P after the termination of objects r and q. This instruction denotes a call of the procedure P. The created object of the procedure would have in its SL chain two terminated objects: q and r. Note that P may use the attribute j from the terminated block object q. As we see, Simula's access rules are violated. Therefore the semantics of such a call must be determined. (Is the call of procedure P legal or would it cause a runtime error?)

Is the access to j of object q legal or would it cause a run-time error? Two solutions are admissible, each implying a possible storage management strategy (cf [2]).

Retention semantics

The first semantics is called "retention". The object remains accessible as long as at least one user's or system pointer (e.g. SL or DL link) refers to that object. The retention strategy of storage allocation corresponds to the above semantics. This strategy may be accomplished either by the use of reference counters or by garbage collection.

Observe however, that within the retention semantics the concepts of block and procedure become trivial. A procedure would be a kind of a crippled class without a remote access mechanism. A block would only be an abbreviation of an anonymous class declaration and a generation at the same time. In this semantics the call of procedure P

from the example is legal because the objects q and r are accessible.

Deletion semantics

Following the Simula principles we choose the other semantics, which may be called "deletion". It consists in the principle that a non-class object becomes inaccessible after its termination while a class object remains accessible as long as at least one user's or system pointer refers to that object. We regard this semantics proper for two reasons. First, it keeps the distinction between classes and blocks or procedure. Second, it admits the deletion of terminated non-class objects from a memory (but whether terminated non-class objects are actually deallocated immediately after their termination still depends on the implementation).

Since we are aiming at the possibility of deallocating non-class objects, we must provide the following property:

(7.1) The object being executed has no terminated non-class objects in its SL chain.

The implementation we propose makes use of SL links defining the SL chains for objects. These links are additional attributes of objects. We intend to treat system reference variables and user's reference variables uniformly. Hence, an SL link should become inaccessible after non-class object termination. (Observe also that when an object contains in its SL chain a terminated non-class object, it can not become an active object, because the display update algorithm (Algorithm 6.1) would fail in searching through the SL chain. In such a case a syntactic environment of the object would not be recovered even if the object requiring the display updating does not refer to inaccessible attributes).

Recall the statement XquaB.P from the example. The new created instance of P has a terminated non-class object q in its SL chain. The property (7.1) fails in this case.

Referencing mechanism

The new method of referencing must carry the information about the termination of non-class objects. Thus that method should realize the dictionary operations: insert, delete and member on the collection of all accessible objects.

In this paper we are not concerned with the strategy of allocating new frames for objects. Therefore we may omit some details and assume the existence of the function newframe (appetite) yielding an address of a new allocated frame of length appetite. Similarly we assume

the existence of a procedure free(X) which releases the frame indicated by an address X.

The operation insert corresponds to the creation of a new object and should be understood as making the new object accessible. Insert does not deal with memory allocation itself.

Operation delete corresponds to the termination of a non-class object, and member yields information whether a reference points to an accessible object.

We will use an auxiliary data structure, an array H, containing references to objects. Roughly speaking, objects will be addressed indirectly through array H. It is obvious that the operation member should be as efficient as possible, for it is the most frequently used. (In our implementation the cost of member is really low: only two machine instructions).

Array H occupies low addresses of core, from 0 to the position pointed by a variable LASTITEM. (Objects may be allocated in high addresses of core). Each item in H is represented by two words, the physical address of an object and an integer called an object number. The algorithms presented below also use a procedure "intolist", a function "deletefrom" and a boolean function "empty", operating on the auxiliary list of released items of H. Because of their obvious meanings, details are omitted. Let the variable LIST be the head of this list.

Now objects are referenced by the so-called virtual addresses defined as pairs (address in H, object number). The object number will be used for checking whether the object is accessible, while address in H will be the indirect address of the object (if accessible).

For a reference X denote the first and the second component of the virtual address of X by Xadd and Xob. The method of referencing will satisfy the following properties:

(7.2) If X refers to an accessible object, the $H[Xadd]$ contains the physical address of the object,

(7.3) X refers to an accessible object iff $Xob = H[Xadd+1]$ (i.e. iff object numbers are the same in the virtual address of X and the corresponding item of H).

Hence, the algorithm for the member operation is as follows:

```
boolean procedure member (Xadd,Xob,physical address);
name physical address; integer Xadd,Xob,physical address;
begin
  if Xob= $H[Xadd+1]$  then
    begin physical address:= $H[Xadd]$ ; member:=true
```

```

    end else member:=false
end member;

```

Consider now delete operation. Following property (7.3) it is sufficient to change the object number in an item of H to guarantee that the subsequent executions of a member concerning this item return value false. All items in H which previously pointed to some objects, subsequently being made inaccessible, are linked together into a list (started by the variable LIST) and may be reused for addressing some new objects.

The algorithm of delete operation is as follows:

```

procedure delete (Xadd,Xob);
integer Xadd,Xob;
begin integer addr;
  if member (Xadd,Xob,addr) then
    begin free (addr); comment a frame in memory may be released;
      H[Xadd+1] := H[Xadd+1] + 1;
      intolist (Xadd,LIST)
    end
  end delete;

```

When a new activation record is allocated, a new element must be inserted into H. If the list of released items of H is not empty, one of the previously used elements of H may be reused. Otherwise array H is extended (LASTITEM:=LASTITEM+2).

```

procedure insert (appetite,Xadd,Xob);
name Xadd,Xob; integer appetite,Xadd,Xob;
begin
  if empty (LIST) then
    begin Xadd:=LASTITEM+1; H[Xadd+1]:=0; LASTITEM:=LASTITEM+2
    end else Xadd:=deletefrom (LIST); comment one element has been
    taken from the list of released elements;
      Xob:=H[Xadd+1];
      H[Xadd]:=newframe (appetite)
    end insert;

```

Moreover we intend to treat uniformly references to terminated objects of non-classes and the reference to the empty object none. This is easily accomplished by the following initialization:

```

none := (0,0); H[0] := H[1] := 1;

```

Hence none does not refer to any accessible object because its object number equals 0 and H[1] equals 1.

Finally, we recall now that the SL chain may be cut off. Therefore the display update algorithm must be modified.

Algorithm 7.1.

Let $SL(p) = p_m, \dots, p_1$, then the GDISPLAY update algorithm has the form:

```
X := p;
while X.SL  $\neq$  none do X := X.SL;
if  $X \notin R(T)$  then error else
for k := 1 step 1 until m do updateGD( $p_k$ );
```

□

Let us now discuss the cost of the proposed referencing method. Each accessible object needs two extra words for an item in the array H. Each reference variable needs two words for a virtual address. Thus, with respect to a standard method we lose two words for each accessible object and one word for each reference variable. (However, the pair of integers forming a virtual address may sometimes be packed into one machine word; the same may be done for an item of two words in the array H.)

On the other hand, we profit in an essential increase of the total number of different objects which may be used through the program lifetime without garbage collection. This number exceeds by far the capacity of H, though the number of objects accessible at the same time is limited by H. The new strategy has the advantage of the standard one when a program uses many procedures (what is natural and very common). Then the terminated objects of these procedures are deallocated on line and the corresponding space may be immediately reused by the other objects (as in the case of stack-implementable language). Observe that the lack of on-line deallocation of terminated non-class objects was the main snag to efficient implementations of Simula-67. Moreover, by virtue of this indirect addressing (in case of memory segmentation), the memory compactification may be done without traversing a graph of objects and without updating the reference variables. It may be accomplished by removing inaccessible objects and changing the corresponding addresses stored in the array H.

Finally the time-cost of these three operations (delete, insert, member) is as follows. The cost of the operation member is constant and very low. It may be compared with the cost of testing on none in standard implementation. The cost of insert and delete depends on the cost of other operations like newframe, free, intolist, deletefrom, which maintain the frames of inaccessible objects. Apart from the cost resulting from these operations, the cost of delete and insert is constant. These operations may be implemented in many different

methods. However, with the use of good algorithms and data structures (i.e. linear lists, heaps etc.) one can obtain the same time complexity as in the case of standard solutions. Moreover, observe that due to the property (7.1) display may contain physical addresses instead of virtual ones, so an access to the visible attributes is not charged by the cost of member operation.

Programmed deallocation

To end this section, as a consequence of the reference mechanism introduced above, we can propose the new operation to be introduced to the programming language. This new operation is called usually programmed deallocation and may be denoted by kill(X), where X is a reference. The semantics of kill(X) is as follows. If X is a reference to an accessible object, then kill(X) makes this object inaccessible (and in consequence this object may be deallocated). Otherwise kill(X) is equivalent to the empty statement.

We got kill operation as a benefit from the referencing method introduced because of the other reasons. Roughly speaking, kill is realized by the delete operation described previously. Thus, after the execution of kill(X) the object pointed (if any) by X becomes inaccessible. Moreover, any remote access to such a being made inaccessible object will cause a run-time error. The realization of this is possible as a result of the operation member already existing in the set of storage management operations. Here the simple test on none is extended to the test on being accessible (member operation). We showed that the cost of member operation is constant and very low, and may be compared with the cost of the test on none. Thus, with some loss of space and a minimal loss of time we can solve the problem of "dangling reference".

We are confident that a programmer when allowing the use of programmed secure deallocation will be able to perform an efficient storage management by conscious deletions of useless objects. Therefore in most cases the time consuming garbage collection may be omitted.

Acknowledgement

The authors are very grateful to Tomasz Muldner for many useful comments and for careful reading of the manuscript.

References

- [1] Bartol W.M. "The definition of the semantics of some instructions of a block structured language with type prefixing.", manuscript, 1980.
- [2] Berry D.M., Chirica L., Johnston J.B., Martin D.F. and Sorkin A. "Time required for reference count management in retention block-structured languages." Part 1, Int. J. Comp. and Inf. Sciences, Vol.7, No.1 (March 1978), pp.11-64.
- [3] Bobrow D.G., Wegbreit B. "A model and stack implementation of multiple environments", Comm.A.C.M., Vol.16, No.10 (Oct.1973), pp.591-603.
- [4] Dahl O-J., Myrhaug B., Nygaard K., "Simula 67 Common Base Language", Norwegian Computing Center 1970.
- [5] Dahl O-J., Wang A., "Coroutine sequencing in a block structured environment", B.I.T. Vol.11 (1971), pp.425-449.
- [6] Dijkstra E.W., "Recursive programming", Numerische Mathematik 2, Vol.2 (1960), pp.312-318.
- [7] Gries D., "Compiler construction for digital computers." New York, Wiley 1971.

ON PRIORITIES OF PARALLELISM:

PETRI NETS UNDER THE MAXIMUM FIRING STRATEGY

Hans-Dieter Burkhard

Sektion Mathematik

Humboldt-Universität

DDR-1086 Berlin, PSF 1297

Abstract: The computational power of Petri nets is extended up to the power of counter machines by realizing certain priorities of parallelism. Hence certain concurrent computations can not exactly be reflected by the sets of all sequentialized computations in related systems. Moreover, the reachability, boundedness and liveness problems are undecidable under the modified firing rule.

0. Introduction

The states and the processed sequences in concurrent systems may be heavily affected by the assumptions about the occurrences of parallelism. To show this we consider concurrent computations using the Petri net model where we claim that maximal sets of simultaneously firable transitions have to fire in parallel ("Maximum Firing Strategy" 2.1). Petri nets under this firing rule are of more computational power than the nets under the common firing rule (3.2).

While the common firing rule (1.2) for Petri nets corresponds to all possible sequentialized computations (executable by one processor) (1.3), the Maximum Firing Strategy allows only those concurrent computations which make use of the maximally possible parallelism (with a related number of processors). This concept is related to the strategy MAX for concurrent computations, which was introduced by Salwicki and Müldner /SM/. The extended computational power under the Maximum Firing Strategy implies that there are concurrent computations which can not be faithfully represented by the set of sequentialized runs.

Furthermore, the Petri nets working under the Maximum Firing Strategy are able to simulate counter machines (3.2). As a consequence the boundedness, reachability and liveness problems are undecidable (4.1). This result may be unpleasant with respect to practical use.

But, as it can be seen by the used constructions, these results already hold for parallelism of two processors: If at least two

transitions have to fire simultaneously whenever this is possible, then the computational power is again extended up to the power of counter machines (4.4). In this sense it can be stated that the use of parallelism must be paid by undecidability results.

1. Preliminaries. The common firing rule.

1.1 \mathbb{N} is the set of all non-negative integers. For a finite alphabet A , A^* is the free monoid with the empty word e . Operations and relations on vectors are understood componentwise.

A (generalized initial) Petri net is given by $\mathcal{N} = (P, T, F, m_0)$, where P and T are the finite sets of places and transitions, respectively. $F: (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ is the flow function, $m_0 \in \mathbb{N}^P$ is the initial marking. For a transition $t \in T$ we define the vectors $t^-, t^+ \in \mathbb{N}^P$ by $t^-(p) := F(p, t)$, $t^+(p) := F(t, p)$ ($p \in P$).

1.2 The transition $t \in T$ is firable under the common firing rule at a marking $m \in \mathbb{N}^P$ iff $t^- \leq m$. After its firing the new marking is $m + \Delta t$, where $\Delta t := t^+ - t^-$.

A sequence $u = t_1 \dots t_n \in T^*$ is a firing sequence under the common firing rule iff each transition t_i ($i=1, \dots, n$) is firable at the marking $m_0 + \Delta t_1 + \dots + \Delta t_{i-1}$ under the common firing rule, it leads to the new marking $m_0 + \Delta u$, where $\Delta u := m_0 + \Delta t_1 + \dots + \Delta t_n$. If only one processor is working, then the firing sequences may be considered as the computational sequences which can be processed by this processor.

1.3 The set of all firing sequences under the common firing rule of a Petri net \mathcal{N} is denoted by $L_{\mathcal{N}}$. The following pumping lemma /B2/ holds:

There are numbers k, l for each language $L_{\mathcal{N}}$ such that the following holds:

If the length of a sequence $u \in L_{\mathcal{N}}$ is greater than k ,

then there is a decomposition $u = u_1 u_2 u_3$ such that

$1 \leq \text{length of } u_2 \leq l$ and $u_1 u_2^{n+1} u_3 \in L_{\mathcal{N}}$ for all $n \in \mathbb{N}$.

By the modified firing rule, which we shall define later on (2.1), we get sets of firing sequences which are subsets of $L_{\mathcal{N}}$. In general, such a pumping lemma is not valid for these sets.

1.4 The set of all reachable markings under the common firing rule is defined by $R_{\mathcal{N}} := \{ m_0 + \Delta u \mid u \in L_{\mathcal{N}} \}$. For a given subset $X \subseteq P$

of places the (non-terminal) Petri net predicate $M_{\mathcal{N}, X}$ is defined as the projection of $R_{\mathcal{N}}$ on the places of X :

$$M_{\mathcal{N}, X} := \{ x \in \mathbb{N}^X \mid \exists m \in R_{\mathcal{N}} : m(p) = x(p) \text{ for all } p \in X \}.$$

There is again a pumping lemma /B2/:

There are vectors $y', y'' \in \mathbb{N}^X$ for each set $M_{\mathcal{N}, X}$ such that the following holds:

If $x \in M_{\mathcal{N}, X}$ covers y' (i.e. $x \geq y'$), then there exists a vector $z \in (\mathbb{N} \setminus \{0\})^X$ such that $z \leq y''$ and $x + n \cdot z \in M_{\mathcal{N}, X}$ for all $n \in \mathbb{N}$.

In general, the Petri net predicates computable by the modified firing rule do not satisfy such a pumping lemma.

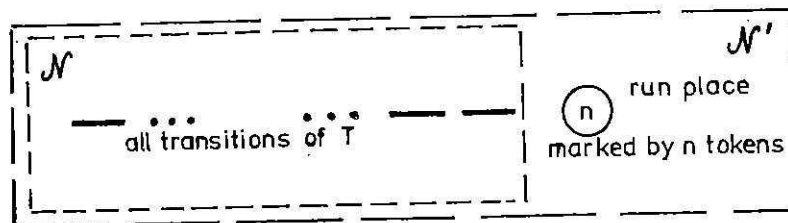
2. Firing under the Maximum Strategy.

2.1 The strategy "MAX" for concurrent computations was introduced by Salwicki and Muldner /SM/: As many processes as possible (limitations may arise by conflicts) have to work concurrently. Thus we want to make use of maximal parallelism. This can be represented in Petri nets by the following firing rule called the Maximum Firing Strategy:

In a marking m we choose a maximal set T' of simultaneously firable transitions, i.e. $\sum_{t \in T'} t^- \leq m$ and $\sum_{t \in T''} t^- \not\leq m$ for all $T'' \supsetneq T'$.

Then the transitions of T' are fired simultaneously. After this firing the new marking is $m + \sum_{t \in T'} \Delta t$. For that marking a new set T' is chosen ...

2.2 The number of simultaneously firable transitions is bounded by the number of transitions in the net. Additionally it can be bounded by the structure of the net. By adding a "run place" it is possible to change the net (thereby preserving the internal structure) such that not more than a given number n of transitions may fire simultaneously:

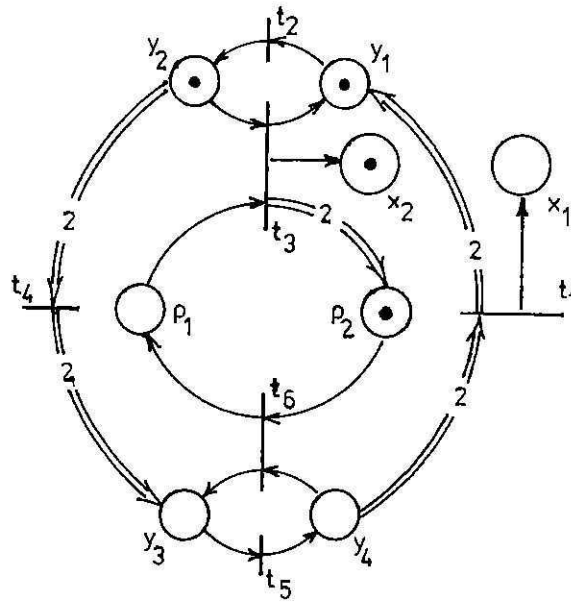


2.3 The set $R_{\mathcal{N}}^{\text{MAX}}$ of all reachable markings under the Maximum Firing Strategy contains all those markings which can be reached from m_0 by firing the maximal sets T' of simultaneously firable transitions, i.e., only those markings are valid which are reached when all transitions of a set T' have fired. However, the results presented in this paper remains true if we consider the sets additionally containing the intermediate markings (where some transitions of T' have fired — this would be related to the languages as in 3.4).

The Petri net predicate $M_{\mathcal{N}, X}^{\text{MAX}}$ under the Maximum Firing Strategy is the projection of $R_{\mathcal{N}}^{\text{MAX}}$ on the places of the set $X \subseteq P$. For each net \mathcal{N} we have:

$$R_{\mathcal{N}}^{\text{MAX}} \subseteq R_{\mathcal{N}} \quad \text{and} \quad M_{\mathcal{N}, X}^{\text{MAX}} \subseteq M_{\mathcal{N}, X}.$$

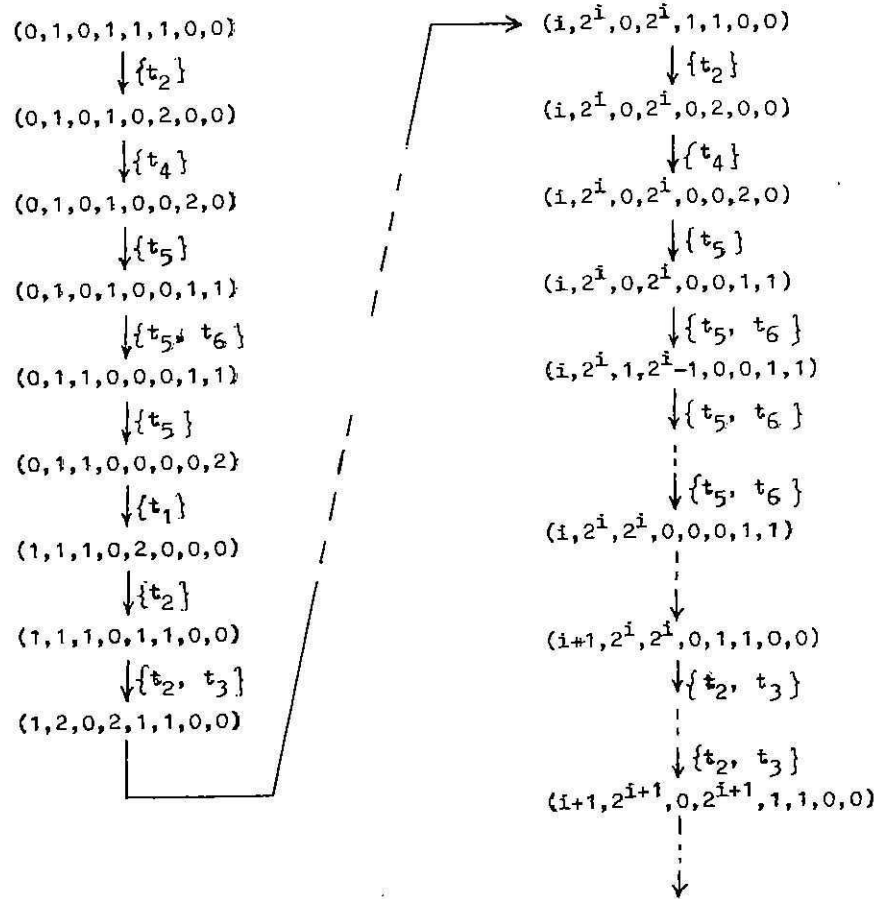
2.4 As an example we consider the following net (a modified version of Hack's example for the weak computation of 2^i):



We have $M_{\mathcal{N}, X} = \{ (i, j) / i \in \mathbb{N} \wedge 1 \leq j \leq 2^i \}$ under the common firing rule for $X = \{x_1, x_2\}$.

The computations under the Maximum Firing Strategy lead to the following reachability graph, whereby

$m = (m(x_1), m(x_2), m(p_1), m(p_2), m(y_1), \dots, m(y_4)) :$



Here we have $M_{W,X}^{\text{MAX}} = \{(i,j) / i \in \mathbb{N} \wedge 2^{i-1} \leq j \leq 2^i\}$. Since this set contains no infinite linear subset, it does not satisfy the conditions of the pumping lemma in 1.4. Hence it can not be computed in any Petri net under the common firing rule.

2.5 . In the example a while-loop is realized: If the places y_1 and y are each marked by one token, then the transitions t_2 and t_3 have to fire simultaneously as long as there are tokens in place p_1 . Thus we have under the Maximum Firing Strategy:

```

while  $m(p_1) > 0$  do begin
     $m(p_1) := m(p_1) - 1$ ;
     $m(p_2) := m(p_2) + 2$ ;
     $m(x_2) := m(x_2) + 1$  end

```

Another while-loop is realized by the transitions t_5 and t_6 . Under the common firing rule it can not be possible to realize while-loops in Petri nets. Otherwise the set $M_{N, X}^{\text{MAX}}$ of our example would also be computable under the common firing rule.

2.6 The reachability graphs under the Maximum Firing Strategy (as well as under the common firing rule) may be infinite as in our example. In general they may also have branchings (if there are two or more maximal sets of simultaneously fireable transitions). The reachability graph is finite iff the net is bounded (iff all reachable markings are bounded).

2.7 Certain properties of Petri nets - especially boundedness - under the common firing rule can be examined with help of the well-known construction of the coverability tree /KM/, /K1/, /H1/. An important fact used for this construction is the following one (which is also related to the pumping lemma 1.4):

If a marking m is reachable from m' by firing of a sequence u , then $m + a$ is reachable from $m' + a$ by firing of u under the common firing rule for each $a \in \mathbb{N}^P$.

This is not true for the Maximum Firing Strategy as it can be seen by the example. Hence a coverability tree with respect to the Maximum Firing Strategy can not be constructed. Furthermore, boundedness is not decidable in Petri nets working under this firing rule (4.1).

3. The computational power of Petri nets under the Maximum Firing Strategy.

3.1 The Maximum Firing Strategy is more "selective" than the common firing rule, thus we have $M_{N, X}^{\text{MAX}} \subseteq M_{N, X}$. By this selecting, the Maximum Firing Strategy is more powerful with respect to computations:

Let \mathcal{M} and \mathcal{M}^{MAX} be the classes of all Petri net predicates $M_{N, X}$ and $M_{N, X}^{\text{MAX}}$, respectively. Then we have

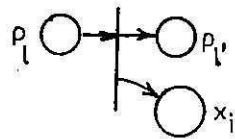
$$\mathcal{M} \subsetneq \mathcal{M}^{\text{MAX}}.$$

For the proof we refer to 2.2 and 2.4: If $n = 1$, then we are able to fire in the net N' constructed in 2.2 exactly all sequences of $L_{N'}$ even under the Maximum Firing Strategy, and hence $M_{N', X}^{\text{MAX}} = M_{N', X}$. For the example 2.4 we have $M_{N, X}^{\text{MAX}} \in \mathcal{M}^{\text{MAX}} \setminus \mathcal{M}$.

3.2 Petri nets under the Maximum Firing Strategy are able to simulate deterministic counter machines. Other possibilities to simulate counter machines by modified Petri nets were given by several authors (cf. 3.5). The crucial point is the simulation of zero-testing, which is not possible in Petri nets working under the common firing rule /K/. The consequence of the ability to simulate counter machines are the undecidability results given in 4.1. The instructions of a deterministic counter machine can be simulated by Petri nets working under the Maximum Firing Strategy in the following way (for more details the reader is referred to the literature):

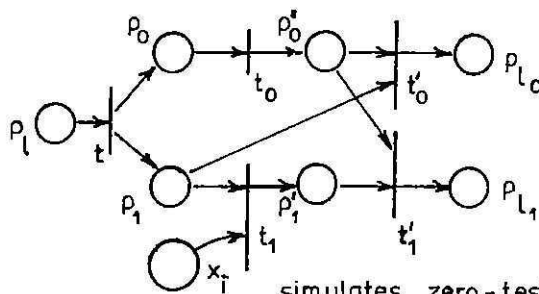


simulates "start in state l"



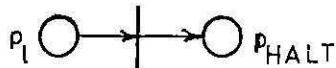
simulates " $l: x_i := x_i + 1; \text{ goto } l';$ "

(counters are simulated by the places x_i)



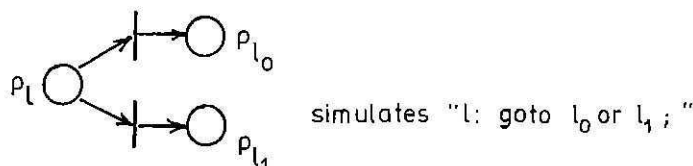
simulates zero-testing:

" $l: \text{ if } x_i = 0 \text{ then goto } l_0 \text{ else } x_i := x_i - 1; \text{ goto } l_1;$ "



simulates " $l: \text{ halt}$ "

Non-deterministic counter machines may also be simulated if we make use of the additional choice-construction:



3.3 As it was shown in /B1/, the set $\{(i, 2^i) / i \in \mathbb{N}\}$ is not in the class \mathcal{M}^{MAX} , and hence it is not possible to compute all recursively enumerable predicates in the sense of \mathcal{M}^{MAX} . To do this, termination is needed: Only those computations (markings) are valid for which a given submarking $y \in \mathbb{N}^{P \setminus X}$ is reached on the places of the set $P \setminus X$ (where X denotes the places on which the predicate is computed as before). By such predicates

$$\mathcal{M}_{\mathcal{N}, X, y}^{\text{MAX}} := \{x \in \mathbb{N}^X / \exists m \in \mathcal{R}_{\mathcal{N}}^{\text{MAX}} \forall p \in X \forall p' \in P \setminus X : m(p) = x(p) \wedge m(p') = y(p')\}$$

all recursively enumerable predicates can be represented /B1/.

Remark: It is an open problem which predicates can be represented using termination in Petri nets under the common firing rule. But it is conjectured that not all recursively enumerable predicates can be generated in this way.

3.4 The order of transitions in a firing sequence of $L_{\mathcal{N}}$ may be artificial in the case of concurrently firable transitions. For reasons of comparing results we can also introduce such an artificial order for the firings under the Maximum Firing Strategy: For each maximal set T' of simultaneously firable transitions the transitions of T' may fire in an arbitrary order (each transition exactly once before the next set T' is chosen). Then we obtain that the Maximum Firing Strategy is more powerful also with respect to the representation of languages by Petri nets. Using termination and a transition labelling function (homomorphism) $h: T \rightarrow \Sigma \cup \{e\}$ we can generate all recursively enumerable languages over the alphabet Σ /B1/.

3.5 The power of counter machines is also met by the modified Petri net versions given by several authors. In /H2/ inhibitor arcs and priorities for transitions, respectively, are used. In the nets defined in /JLL/ and /MATTK/ the transitions have to fire during fixed (individual) time intervals after their enabling. Concepts of firing

in the order of enabling (realized by certain queue regimes) have those effects, too /B1/. In /V/ the numbers of transported tokens are modified by the markings on certain places. The concept in /MPS/ is the closest one to our Maximum Firing Strategy: There the firings of transitions are synchronized by external events such that all enabled transitions connected to the actual event have to fire. The construction for the simulation of inhibitor arcs given in /MPS/ would also work under the Maximum Firing Strategy. But, on the other hand, the construction given there is quite opposite to parallelism since all concurrent firings of the essential transitions are suppressed by a run loop (similar to the construction in 2.2).

4. "The price of parallelism"

4.1 It is well established in the literature that the ability of Petri nets to simulate deterministic counter machines (whereby the Petri nets are modified in some sense) results in the undecidability of the boundedness (are all reachable markings bounded with respect to certain places), the reachability (is a given marking/submarking reachable) and the liveness (can certain transitions always become firable sometime later) problems. Since the halting problem is not decidable for deterministic counter machines, it is not decidable if a token can arrive at the place P_{HALT} (cf. 3.2) and hence the reachability problem for submarkings is undecidable. By connecting certain simple subnets to the place P_{HALT} the undecidability of the reachability, boundedness and liveness problems can be proved (cf. for instance /H1/, /JLL/, /B1/).

4.2 In the constructions for the simulation of the counter machines all places excluding the counter-place x_i may only be marked by 0 or 1. It is known from the theory that the halting problem is undecidable even for counter machines with two counters. Hence the undecidability results hold for Petri nets under the Maximum Firing Strategy where the nets have only two unbounded places. By a construction given in /B1/ the number of bounded places can also be limited by two. Hence the total number of places need not be greater than four. On the other hand, the reachability sets R_{μ} of Petri nets with 4 places under the common firing rule are always semilinear /HP/. This illustrates the difference between the firing rules once more.

4.3 Since we always have $M_{\mathcal{N}, X}^{\text{MAX}} \subseteq M_{\mathcal{N}, X}$, a place which is bounded under the common firing rule must also be bounded under the Maximum Firing Strategy. Hence it is possible that a place which was formerly unbounded becomes bounded under the Maximum Firing Strategy. But it is not decidable in general if this happens. Still more important could be the fact that a transition which was live under the common firing rule may become not live with respect to the Maximum Firing Strategy and vice versa [B1]. Here the undecidability results are very strongly affecting the practical use.

4.4 In the Maximum Firing Strategy we make use of maximal parallelism. But for simulating the counter machines the parallelism which is used may also be restricted: Only the parallelism of two transitions is needed for the zero-testing device (3.2): If the transition t_1 is firable (if there are tokens on the place x_1), then the transition t'_0 must not become firable (the places p_1 and p'_0 must not be marked at the same time). That can be ensured if the transition t_1 must start working (with taking the token from p_1) before the transition t'_0 has ended its actions (has given the token to p'_0). This condition can be satisfied if we claim that in a net simulating a deterministic counter machine (3.2) at least two transitions have to fire simultaneously whenever this is possible. Moreover, both transitions t_0 and t_1 become firable at the same time. Hence it should be reasonably accepted that under the assumptions of a parallel system both transitions are simultaneously acting in reality. Thus deterministic counter machines can be simulated. In this sense we can state that the use of parallelism (the priority of parallelism) must be paid by the undecidability of the reachability, boundedness and liveness problems.

4.5 Of course, the constraints of firings by parallelism of at least two transitions (as far as it is possible) lead also to more computational power (in comparison to the common firing rule as in 3.1). The consequence of those extensions is the impossibility of faithful simulations by all one-processor-computations: There are concurrent computations by nets working under constraints by parallelism such that no net working under the common firing rule can exactly simulate them.

4.6 As it was pointed out in 3.5, all known related extensions of Petri nets (together with termination, cf. 3.3, 3.4) give the nets the power of Turing machines. What we can say now is that already the use of parallelism can give the nets this computational power. The restric-

tions of this power under the common firing rule result from this point of view from irresolution with respect to parallelism. On the other hand, firing by fair scheduling (sequentializing instead of parallelism results in the power of Turing machines, too /B1/ (consider the zero-testing device in 3.2 under the assumption that a firable transition has to fire which was enabled the longest time).

Thus the restricted computational power (and the decidability of the boundedness problem, for instance) of Petri nets under the common firing rule can be understood as the consequence of allowing "too much": If there is made a decision concerning parallelism (or fair scheduling or one of the modifications mentioned in 3.5), then these restrictions may be overcome.

5. Conclusions.

The restrictions of firability by the use of parallelism extends the computational power of Petri nets. It is not possible to simulate all these computations by nets working under the common firing rule: The nets under the common firing rule are in general computing "too much". Hence there are concurrent computations executed by several processors using the possibilities of parallel working which cannot be exactly reflected by all computations which one processor could execute in the same system or even in any other system of the same kind.

Under the aspects of practical use the power of Turing machines (or at least of deterministic counter machines) may not be welcome. The decidability of liveness, for instance, is desirable. On the other hand, the use of parallelism as far as it is possible is desirable with respect to efficiency, too. Now the question arises for which classes of nets the mentioned problems (or at least some of them) are decidable with respect to the modified firing rules. A positive answer can trivially be given for the class of bounded nets.

Acknowledgement

The inspiration for the study of the Maximum Firing Strategy was given by Prof. A. Salwicki. I would like to thank him for his suggestions.

References

- /B1/ Burkhard, H.D., Ordered Firing in Petri Nets,
Elektron.Informationsverarb.Kybernetik 17(1981)2/3, 71-86.
- /B2/ Burkhard, H.D., Two Pumping Lemmata for Petri Nets.
To appear in Elektron.Informationsverarb.Kybernetik.
- /H1/ Hack, M., Decision Problems for Petri Nets and Vector
Addition Systems, MAC-TM 59, Proj.MAC, M.I.T. 1975.
- /H2/ Hack, M., Petri net languages,
CSG Memo 124, .Proj.MAC, M.I.T. 1975.
- /HP/ Hopcroft, J., Pansiot, J.J., On the reachability problem for
5-dimensional vector addition systems,
Theor.Comp.Science 8(1979), 135-159.
- /JLL/ Jones, N., Landweber, L., Lien, E., Complexity of some
problems in Petri nets, Theor.Comp.Science 4(1977), 277-299.
- /KM/ Karp, R.M., Miller, R.E., Parallel Program Schemata,
Journ. Comp. and System Sciences 3(1969), 147-195.
- /K1/ Keller, R.M., Vector replacement systems: a formalism for
modelling asynchronous systems,
Tech.Rep.117, Comp.Science Lab., Princeton Univ., 1972/74.
- /K2/ Keller, R.M., Generalized Petri nets as models for system
verification,
Tech.Rep.202, Dept.Electrical Eng., Princeton Univ., 1975.
- /MATTK/ Mori, M., Araki, T., Taniguchi, K., Tokura, N., Kasami, T.,
Some decision problems for time Petri nets and applications
to the verification of communication protocols,
Trans.IECE'77/10 Vol.J60-D, No.10, 822-829.
- /MPS/ Moalla, M., Pulou, J., Sifakis, J., Synchronized Petri nets:
A model for the description of non-autonomous systems,
Lecture Notes in Computer Science 64(1978), 374-384.
- /SM/ Salwicki, A., Müldner, T., On algorithmic properties of
concurrent programs, Manuscript, Warsaw 1979.
- /V/ Valk, R., Self-modifying nets, a natural extension of Petri
nets, Lecture Notes in Computer Science 62(1978), 464-476.