MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 87

Translation of Simula 67 Into the Common Base Language

by

Philippe Coueignoux

Philippe Janson

This work was submitted for credit in Subject 6.534,
"Semantic Theory for Computer Systems," Spring 1973.

August 1973

I. Scope of the Paper

In an earlier paper of this term (3), two of us have given a general and informal description of Simula 67 and have outlined how the main primitives of the language could be translated into the Common Base Language defined by Dennis [1] and Amerasinghe [2]. That brief study showed us that the CBL could support most of the concepts of Simula 67. However, some features of Simula 67 introduce new problems (e.g. coroutines, cycles, garbage collection, ...). Since the problems seemed challenging and interesting, we decided to carry on our study, to give a more complete definition of Simula 67, to give guidelines for the translation of each primitive and to solve the problems introduced by our translation rules.

The paper is unfortunately divided into two parts, although we both worked out all topics during common work sessions. This first part will describe Simula 67 and the rules finally adopted for his translation into CBL. The second part will explain what the problems are, how they can be solved in general, and it will show the interpretation of a simple one queue-one server problem. This part will in fact justify the translation rules explained in the first part. Like in the first paper, we will only consider features proper to Simula, and we assume that all Algol features can be translated with no problem.

II. Description of Simula 67

The description given here is based on the definitions given in [4, 5, 6, 7, 8]. The CDC manual gives the complete and formal definition of the primitives but also describes built-in and base functions which were implemented only to support the compiler and are actually not part of the language and inaccessible to the user. We will not describe these primitives. We are not interested, for instance, in statistical distribution generators, I/O handlers, etc.

1. Fundamental concepts

a. Systems

Simula is an Algol-like block structured language designed for the simulation of systems. A program describes a system; its execution simulates the evolution of the system. Two concepts are embedded in any system described by Simula: set manipulation and pseudo-parallel sequencing. Set manipulation gives to the program the ability to declare and use sets (i.e. queues). Pseudo-parallel sequencing gives to the program the ability to be run on a real world sequential machine and yet to simulate the evolution of a system containing concurrent components.

b. Objects:

In addition to the Algol-like notions of block and procedure, Simula has the notion of class; Simula also uses a text manipulation technique called prefixing. A class is a compound statement declared just like a procedure. The difference between both appears in the activations. Classes have to be activated by a reserved primitive (not a call) and when all instructions of a class are executed, the termination of the class activation is not equivalent to a return. The notion of class, unlike the notion of procedure, allows the user to define pseudo-coroutines. Prefixing means writing the name of a class in front of the declaration of another class or of a block. Prefixing inserts the whole text of the prefix class between the declaration part and the statement part of the prefixed class or prefixed block. Prefixing is nothing else than a powerful tool for concatenation of texts. Any activation of a block, procedure, class, prefixed block or prefixed class is called an object. Objects are the components of the simulated system.

2. Implementation of the concepts

a. Detaching: System structure:

In Algol, block and procedure activations are either attached or terminated. While executing, the activation of a block or procedure is said to be attached to the block or procedure which created it. After the end of the block or the return of the procedure the block or procedure activation is terminated and inaccessible. The same is true in Simula.

b. Resuming: system discrete simulation time scale:

We have just mentioned that a detached object can be active or passive. We will now explain what it means in terms of the simulation of the system. Any Simula program has a built-in privileged set called the sequencing set (SQS) which is the time scale of the simulation. Each element of SQS is called an event notice. In each event notice there exist two components: a pointer to a detached object and the real number standing for its scheduled activation time. All event notices are ordered in SQS by increasing time, thereby creating a discrete time scale. This discrete time scale implies a discrete simulation. When the object, pointed to by the first event notice of SQS, is active, the system time is fixed and equal to the time associated with the event notice. Execution proceeds to the next event notice when the active object hits the instruction "resume," thereby becoming passive (erasing the current event notice). As execution proceeds to the next event notice, time flows at once to the time of this new current event notice.

N.B.: "Resume" is also inaccessible to the user. It is implicitly contained in instructions like "end," "passivate," etc.

3. Aspect of a Program and Generation of a System

```
begin   {any set of Algol declarations and class declarations
        {any set of Algol statements
simulation begin  {any set of Simula declarations
                  {any set of Simula statements
            end;
        {any set of Algol statements
end;
```

Figure 1

Now we come to the case of a prefixed block activation. Prefixed block activations are always detached or terminated. Detaching a prefixed block activation means that one defines a system (or subsystem) at a certain level. Let us ignore for a while the existence of any compound statement except prefixed blocks. In doing so we define a proper hierarchy of prefixed blocks in the Simula program. The outermost prefixed block of the program must be prefixed by "simulation" (a built-in class; see further). Only this block can be prefixed by "simulation." It is detached at level one and defines the system simulated by the program. Any prefixed block nested in the first one will be detached at an upper level defined by the prefixed block hierarchy and will define an independent subsystem. The termination of a prefixed block implies the termination of all objects detached at the same level, i.e. detached class objects.

Finally we come to the case of classes (prefixed or not). Activations of classes can be attached, detached, or terminated. When initially created, a class activation is attached to the object which created it. A class may contain the instruction "detach" causing any activation to be detached at the level of the smallest enclosing prefixed block activation. Detaching a class activation for the first time returns the control back to the activation statement but the class activation object still remains in existence and is accessible. If, later on, control enters again the object, execution will proceed after the "detach" statement. If control hits another "detach" statement, it again quits the object but returns to the smallest enclosing prefixed block activation instead of returning to the object which created the current class object. If control hits the "end" statement of a class, the class activation is terminated, i.e. no more executable, but still accessible (its environment is not discarded). Control returns to the activation statement or the smallest enclosing prefixed block depending whether the class activation object was still attached or not.

Class detached objects together with the smallest enclosing prefixed block detached object define the components of a pseudo-parallel system (subsystem) at the level of the prefixed block object. Other block, procedure or class objects can be attached to a detached object. When an object is detached and has control it is said to be active; when it does not have control, it is passive. When control is in an object attached (directly or transitively) to a detached object, the detached object is active.

N.B.: "Detach" should not be used by the programmer. It is implicitly in the text of all classes prefixed by "process" (see further).

which also can be a set member and describes properties of event notices. In addition, simulation contains a few procedure declarations and initialization statements. When control enters the block prefixed by simulation, these statements will be the first ones to be executed before the actual user statements. Their purpose is to initiate a simulation, i.e. to generate the first object (the current prefixed block activation), to set up SQS and an event notice for the current prefixed block known under the name MAIN.

4. Description of Simula primitives

(Primitives of linkage, process, and event notice are inaccessible to the programmer and therefore uninteresting to us here.)

a. Declarations

In addition to Algol declarations, Simula uses classes and references. Classes have been described above. References are pointers to detached objects: e.g.:

ref(head) Q declares a set;
ref(truck) MAC declares an object MAC of the class truck.

b. Macros recognized by the translator

- new: is the privileged instruction generating an activation of a class or a prefixed class.
  e.g.: MAC := new(truck) generates an object of class truck called MAC.
- inspect: is an instruction which allows the object executing it to access the environment (CBL local structure) of another object.
  e.g.: inspect X do B allows the executing object to interpret the variables in block B as if it were working in the environment of object X. This is called remote access.
- dot notation: is a means much like inspect to access the environment of an external object (remote access).
  e.g.: X.proc (arg1) allows the executing object to execute procedure proc with arguments arg1 in the environment of object X. The arguments must be defined in the caller's environment but the procedure must exist in the external object environment.

- this: is a pointer to the current active object unlesss this appears in an inspect block.
- time: returns the current system time.

c. Primitives declared in class head:

- Q. first is a procedure which returns a pointer to the first element of set Q.
- Q. last returns the last element of set Q.
- Q. cardinal returns the number of members of set Q.
- Q. empty returns the boolean state of set Q.
- Q. clear empties set Q.

d. Primitives declared in class link:

- P. into (Q) inserts object P as the last member of set Q.
- P. precede (X) inserts P in front of X in whatever set X is.
- P. follow (X) inserts P behind X in whatever set X is.
- P. out removes P from whatever set P is in. This procedure is implicitly called by the other three before anything else.

e. Primitives declared in class simulation:

Sequencing

- passivate: passivates the current object.
- wait(Q): passivates the current object and inserts it at the end of set Q.
- hold(T): passivates the current object and schedules its next active phase in SQS, T time units later.
- cancel(P): erases the event notice corresponding to object P in SQS, if any. This procedure is implicitly called by the other three before anything else.

Scheduling

In all these statements, the event notice corresponding to the object P in SQS if any, is first removed. (Cancel (P).)

- activate ⟨P⟩⟨T⟩⟨S⟩: schedules an active phase for object P at time T with respect to S.
- reactivate ⟨P⟩⟨T⟩⟨S⟩: passivates the current object and schedules an active phase for object P at time T with respect to S.

⟨P⟩ is any kind of pointer referencing a detached object.

⟨S⟩ is blank or prior, meaning after or before all objects already scheduled for time ⟨T⟩.

⟨T⟩ is at T: at time T

delay T: at current time +T

before X: before X in SQS

after X: after X in SQS

blank: schedules P immediately, reschedules the current object after P and passivates it now.

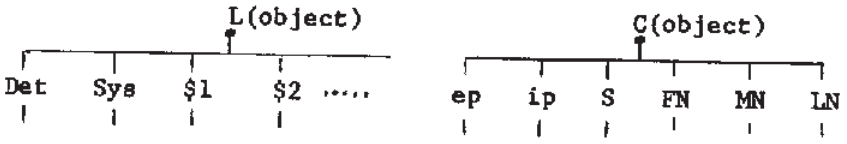
III. Translation of Simula Primitives

1. Text Representation

The translator should first of all have knowledge about the internal classes simset and simulation. It then should know how to manipulate prefixing in order to insert text of prefixes where appropriate. It also should distinguish prefixed blocks and classes from other compound statements in order to correctly translate the activation and termination statements like attach, detach, resume, and end. (See further.)

2. Object Representation

Any Simula object is characterized by two structures:



L(object) is a component of the 'local' component of the interpreter state. Detached is a boolean of which the meaning is obvious. System is a component of which the value is the name of the smallest enclosing block activation to be used when the object is detached. $1 and $2 are used for calls and remote access (see IV.5). C(object) is a component of the 'control' component of the interpreter state. It is a processor in the CBL sense. Ep, ip and s are as in CBL. FN, MN, and LN are needed to represent coroutines and avoid physical cycles (see IV.6). Their scope of use is broader than just Simula. In addition to these components which are created for any object, several components are created with objects of classes being, or containing as a prefix, head, link, process or event notice. An object containing head, when created, adds to its local structure a 'succ' component which will point to the first object of the set and a 'pred' component which will point to the last set member.

An object containing the properties of link also has two such components pointing to its successor and its predecessor in a set. It is, therefore, clear that a set is a cycle with two-way pointers (see IV.6 for their elimination). In addition an event notice object has a time component and a process component. An object with the properties of "process" has an 'event' component pointing to an associate event notice object. All objects of a class containing "process" are detached because the code of process contains "detach" as its first instruction.

3. Declaration Primitives.

- class: declaring a class in Simula has the same effect as declaring a procedure in Algol. The translator creates a closure with a text and a list of externals.
- ref (classname) objectname: creates a pointer initially null, to be used to point to a Simula object local structure.
- Notice that objects of the class event notice are not declared by the programmer. They are implicitly declared in primitives using, creating or deleting them.

4. Compile Time Macros

- time: is easy to translate. It is just a reference to SQS.succ.time.

The translation of other primitives is much more complex and poses problems -- not proper to Simula -- which result of the use of (pseudo-) coroutines and the ability to access the local structure of an external object (remote access). The translation of these features requires new instructions to be created for CBL. They will be explained in the second part of this paper.

5. Head Primitives

- first: returns the external 'succ' component of the local structure of the head object where it is called.
- last: returns the external 'pred' component of the local structure of the head object where it is called.
- cardinal: starting with the object pointed to by the 'succ' component of the local structure of the head object where it is called, it counts the number of set members until the 'succ' component of an object (the last one) points to the head object. Notice that this implies that the procedure access the local structures of all set members. This is done by a new CBL primitive (see p 15).
- empty: returns a boolean which is true if the 'succ' component of the local structure of the head object where it is called is null.
- clear: resets the 'succ' and 'pred' components of the head object to "null" and collects the garbage of set members by destroying all 'succ' and 'pred' components. This also implies remote access to all set members local structures.

6. Link Primitives

Previously to anything else all following procedures call the procedure "out."

- out: must be executed in an object local structure with link properties. It reads the 'succ' and 'pred' components of the local structure. It copies 'succ' into the 'succ' component of the object pointed to by 'pred' and copies 'pred' into the 'pred' component of the object pointed to by 'succ.' It needs remote access to these objects. It then sets 'succ' and 'pred' to null in the local structure where it is called.

- follow(X): gets remote access to X's local structure, copies the 'succ' component of X into the 'succ' component of the current link object where it is called and sets the 'succ' component of X to point to the current object. It then gets remote access to the 'succ' object of the current object, copies its 'pred' component into the current local structure 'pred' component and sets the 'pred' component just read to point to the current object.
- precede(X): is just like follow (substitute 'pred' for 'succ' above).
- into(Q): gets remote access to the set Q's local structure, copies Q.pred into the current local structure 'pred' component, and sets Q.pred to point to the current object. It then gets access to the 'pred' object of the current object, copies its 'succ' component into the current 'succ' component and sets its 'succ' component to point to the current object.

7. Simulation Primitives

The following lines express the redundancy of a lot of primitives: the translator should use them like macros rather than have a procedure for each. This brings down the overhead at run time.

wait(Q) = this. into(Q); passivate;
hold(T) = activate this at (time+T); passivate;
reactivate ⟨P⟩⟨T⟩⟨S⟩ = activate ⟨P⟩⟨T⟩⟨S⟩; passivate;
activate P delay T = activate P at (time+T);
activate P = activate P after this; activate this after P; passivate;

Given these macros, we will translate only the following primitives. They are called in the local structure of the current active object which executes them.

- cancel(X): gets remote access to X and from there to the corresponding event notice E if any. It then performs the procedure E.out, terminates E and erases the pointer to E in X.
- activate P at T (prior): first calls cancel (P). It then gets remote access to SQS and searches until it finds an event notice E' with a time component greater than or equal to T. If the greater relation holds, an event notice E is generated with a

time component equal to T and the processor component pointing to P, the procedure E.precede(E') is executed and a pointer to E is copied into the 'event' component of P. If the equal relation holds and there is a prior argument, the above procedure is also executed. If prior is not present then the search goes on again until the greater relation holds, and the above procedure is then executed.

- activate P before/after P': first calls cancel (P). It then remotely accesses P', reads 'event', accesses the event notice E', creates a new event notice E with the processor component pointing to P and the time component equal to the time component of E' and finally executes E.precede/follow (E').
- passivate: reads the 'event' component of the current detached object, i.e. the currently active processor or the detached processor to which it is attached. It then accesses the corresponding event notice E, executes E.out and terminates E. It finally executes a "resume" instruction on the new current object, i.e. the new first object of SQS (see p 24).

IV. Solutions and Problems

1. If the reader has not already done so, he is welcomed to skip this section and look at the example, to see how the language works. Knowing the previous sections and going through the example, he may criticize the implementation, discover some problems, find his own solutions, then compare what he has found to the present section. For he must not consider the following as the ultimate truth: the CBL is undefined enough to allow each user to modify it, hoping for improvements, at least to raise questions.

2. Coroutines

The most powerful concept of Simula 67 is the use of pseudo-coroutines. A coroutine is a CBL-process which can be interrupted and restarted without being killed and created again. Therefore, it is necessary to keep all its components at hand, which may even be modified in the meantime by some other CBL-process through data-sharing. A process, in the usual sense of time-sharing, is such a coroutine, generally implemented on a virtual system; parallelism can be modelled

with coroutines. Therefore we felt free to modify the CBL to account for such a broad construction, without being guilty of warping the CBL to fit our problem. In fact Simula does not use all the possibilities of coroutines: there is no parallelism in its execution sequence, only a pseudo-one; hence only one coroutine is working at any given time and every interruption is self-performed.

3. The CBL-73

a. At this point we present our version of the CBL and of the state space. The latter is composed of a set of processes called the control structure. A process has six components (see p 8).

π: (FN) the name: a character string; see the formation of names for Simula-67 below.

(σ) the state: a boolean 0 if the process may not be executed
1 if it may

(MN) the somname: the name of another process or an empty string

(ip) the ip: a pointer towards the next instruction to be executed

(ep) the ep: a pointer towards the local structure of the process

(LN) the fathername: the name of another process

The primitives are the following:

select, link, const, assign are as stated in CBL

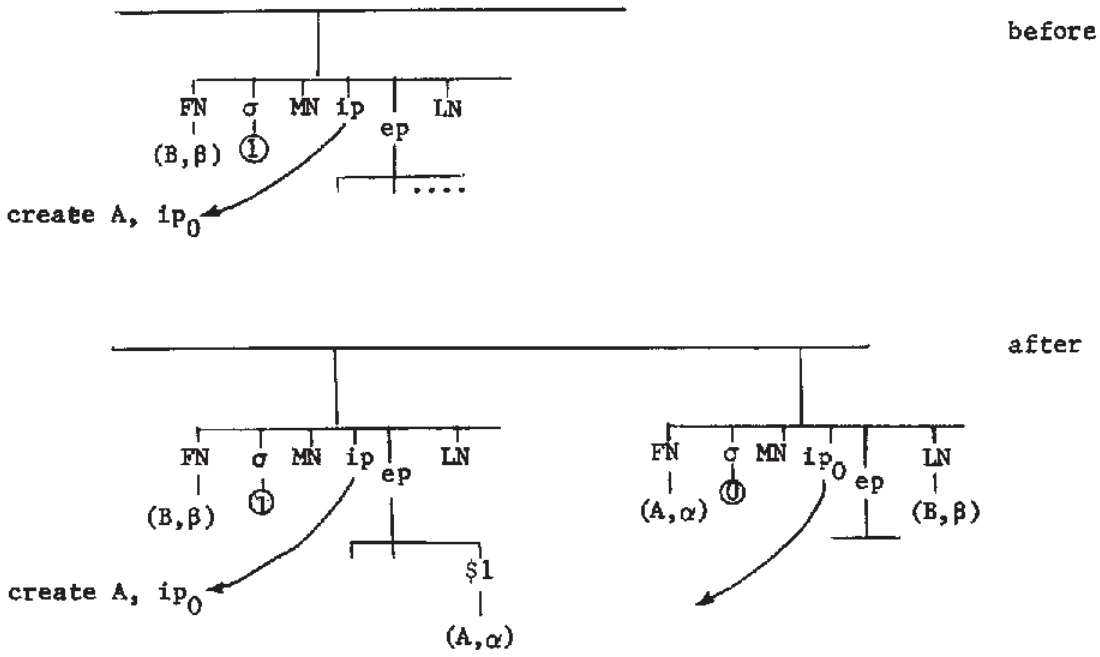
However, it is allowed that an instruction refers to FN, MN or LN of the process which executes it, just where any component of the local structure would have been legal.

mark replaces create

unmark replaces delete

The primitives create, sense, resume, wash are introduced.

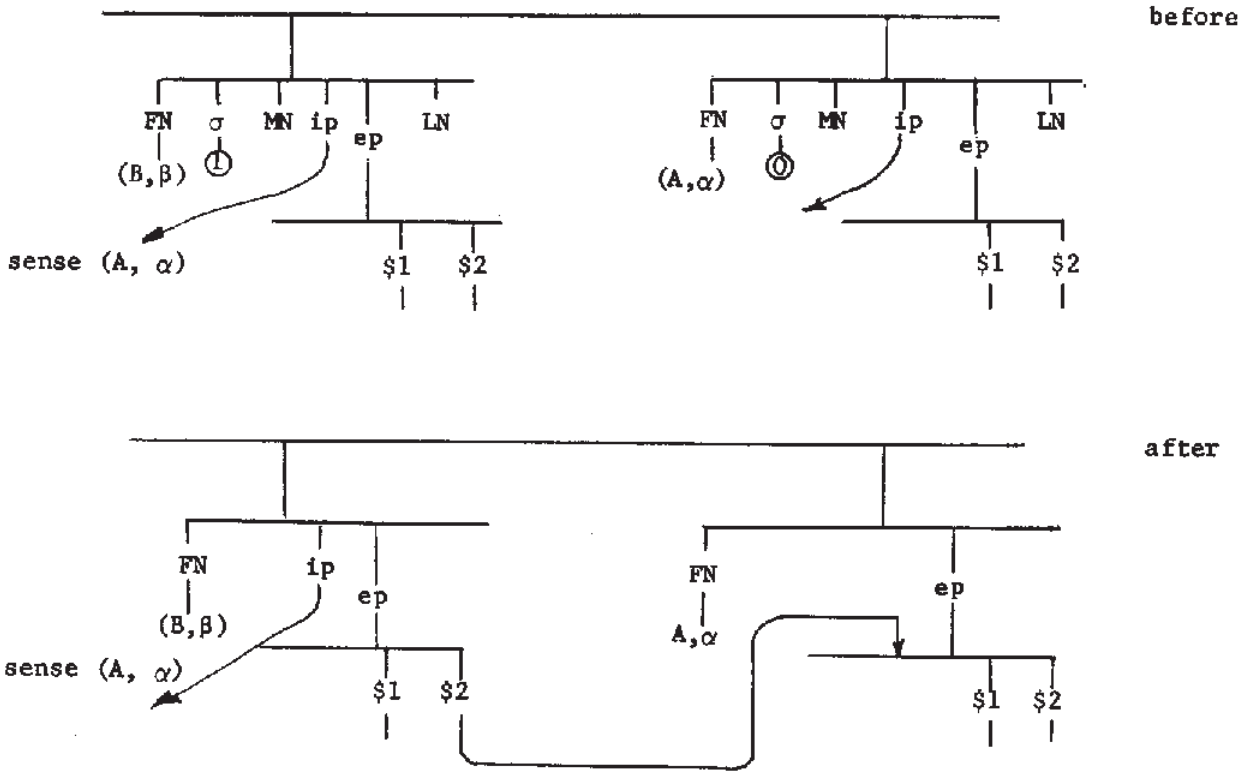
b. create (generic-name), (text pointer)



The create instruction generates a new process with a unique name formed by concatenating the generic name A with a unique number α; its state is 0; it has no son-name; ip_0 is as provided; its local structure is empty; its father-name is the name of the process which has executed the create instruction. Note that the new name is recorded in a special component, $1, of the old local structure, thus providing a path between old and new processes, as well as between new and old ones.

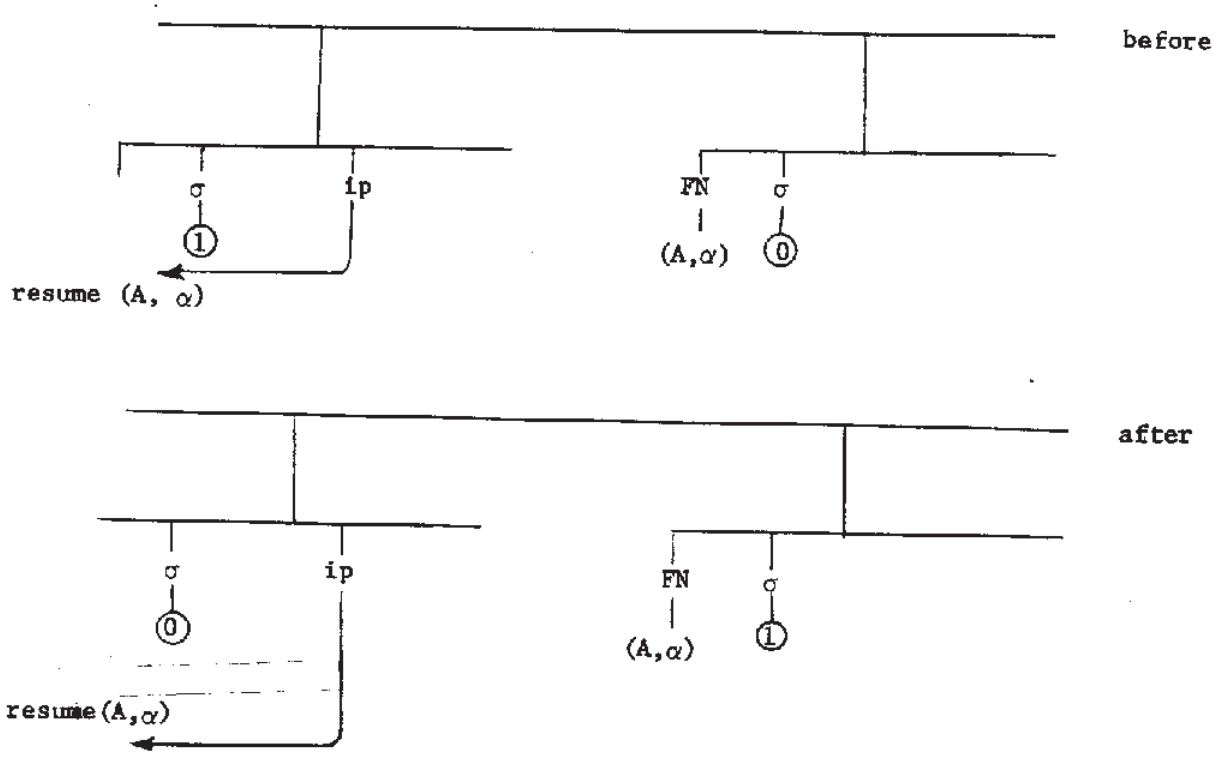
Note: Whenever a (unique-name) is written in an instruction, it must be understood as the value of a referenee variable in the local structure (ex.: Sense $1).

c. sense (unique-name)



The sense instruction makes the local structure of the process (A, α) known to the process (B, β), which executes the instruction, by copying the ep of (A, α) into a special location, $2, of the local structure of (B, β).

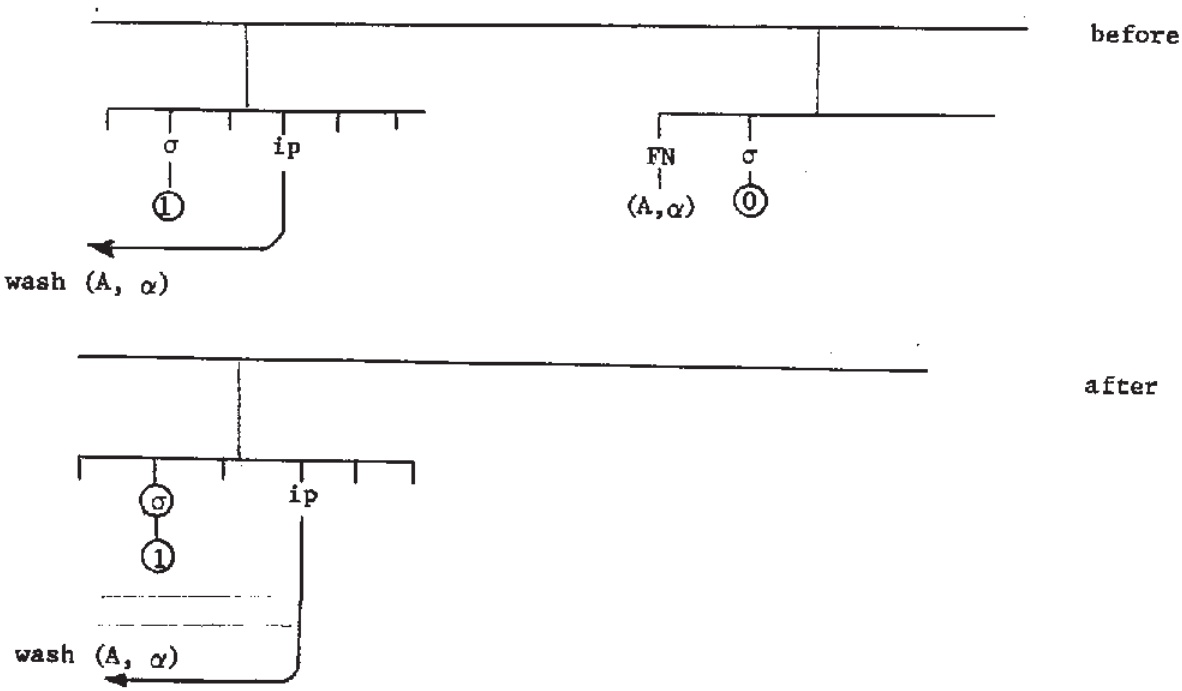
d. resume (unique-name)



The resume instruction transfers the site of activity from one process to another. See below an addition to the semantics: the resume * instruction (* means indirect).

If the name is not a process name, an error message is produced and the instruction ignored.

e. wash (unique-name)



The wash instruction gets rid of the operand process.

Note: Those instructions are very powerful and their use should be strictly under control. A way is to disallow direct programming in CBL, letting the task to enforce security and safety to the compiler (or to some macro-assembler). The last step would be to include good primitives for parallelism, yet to be discovered: for in our model, as in CBL, only one process is in state 1 at any given time.

The sense primitive can lead to cycles if used without precautions.

f. The use of the son-name and the "resume*" instruction

By now the use of every component of a process is clear: the father-name has replaced the $RET component, the other components are unchanged from CBL. The unique-name is a way to avoid pointers, which could produce cycles which would fool a simple garbage collection. The son-name is yet to be explained.

A process is created without a son-name.

However, when in state 1, it can gain one by executing the instruction:

assign (unique name) son-name

Once again, the unique name is the value of some reference variable within the local structure, and "son-name" is recognized as the MN component of the process under execution.

Rule: The "resume*" instruction has the following effect:

move the ip of the process under execution past the instruction "resume*";
switch the state of the executing process from 1 to 0;
until the son-name of the designated process is empty:
  take this son-name as the name of the designated process;
switch the state of the designated process from 0 to 1;

4. An Example of CBL-73

Let us translate this piece of code written in CBL:

```
                                f: begin: return;
apply f, $ARG      ;               end;
```

We obtain:

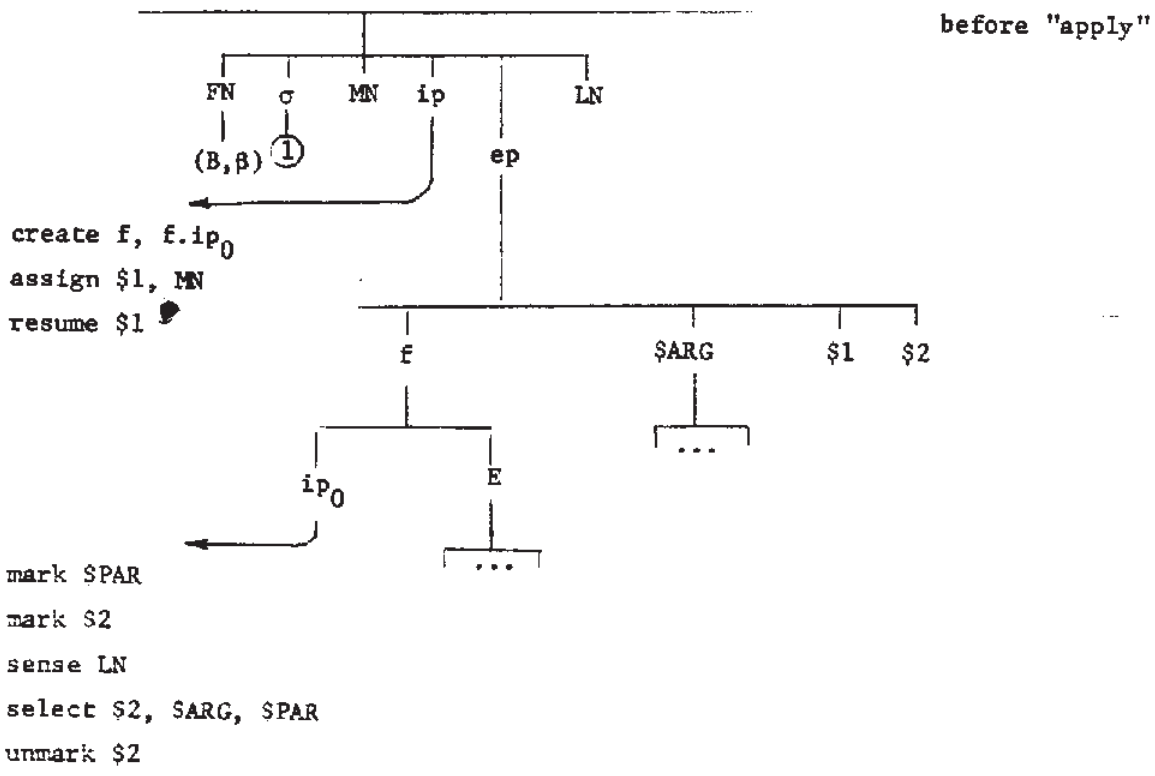
```
create f, f.ip                  f: mark $PAR
assign $1, MN                      mark $2
resume $1                          sense LN
-------------------- ;             select $2, $ARG, $PAR
unmark MN                          unmark $2
wash $1                         ---------------------
                                   resume LN
```

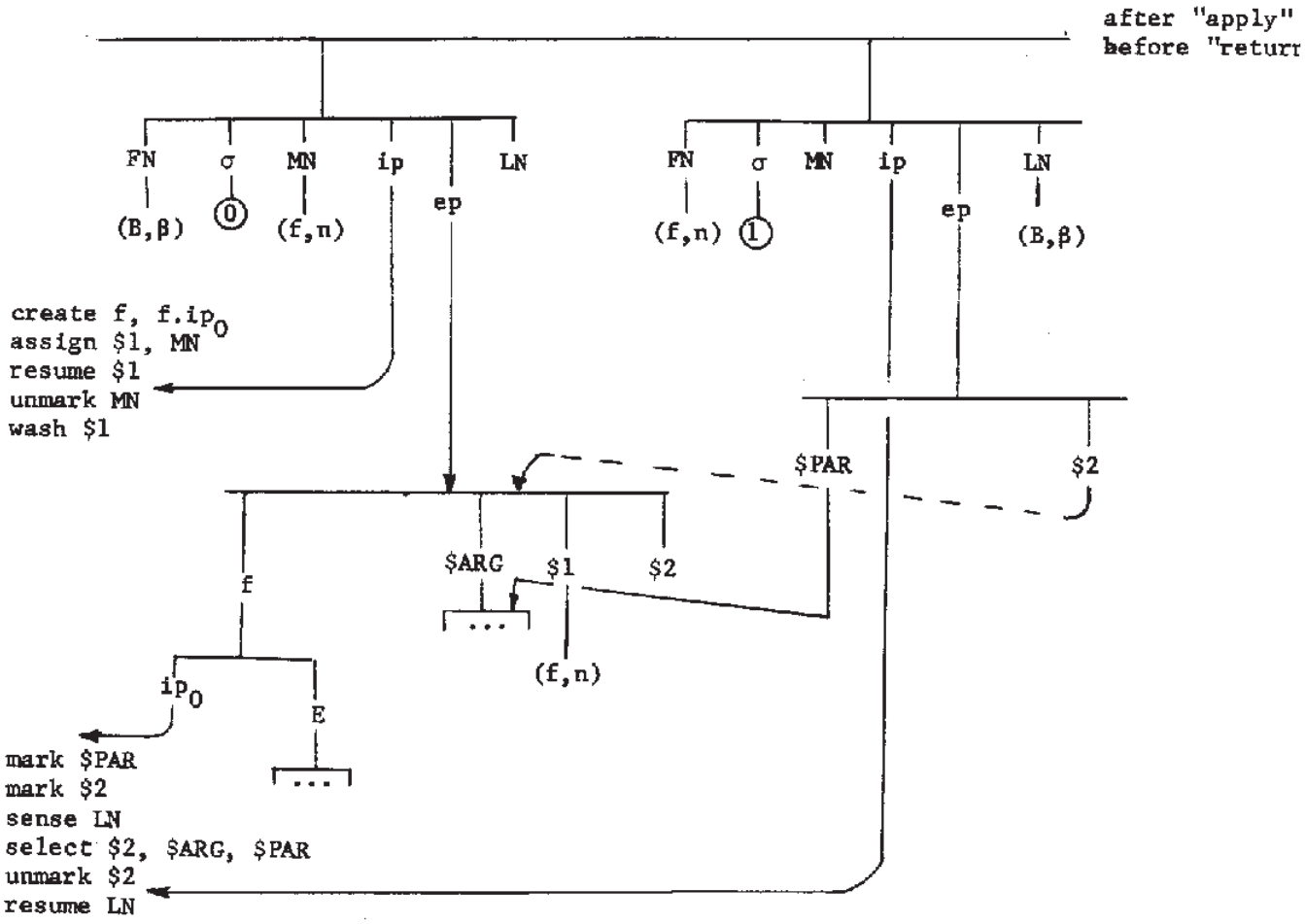
Note: f.ip is a shorthand for: select f, ip, ip_0
create f, ip_0

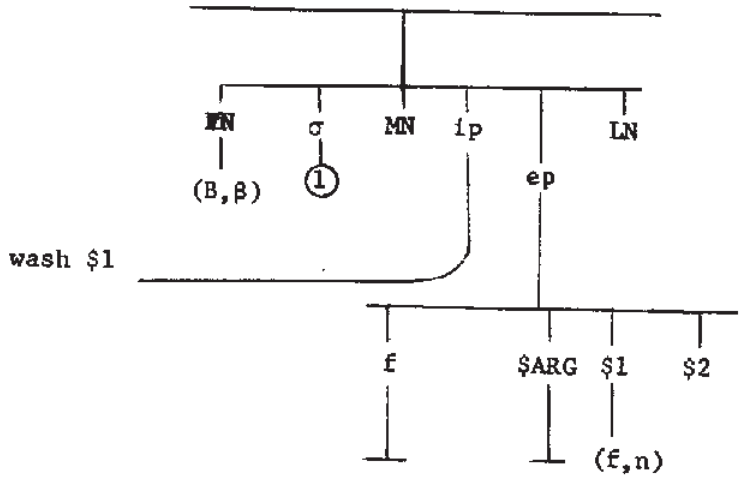
This contraction will be used over again as a symbolic facility.

The apply primitive of CBL is taken as a macro and behaves in this way:
see pp 19-20.

The return primitive of CBL undergoes a similar treatment: see p 20.







after "return"
(the parts which are inaccessible are not shown)

Note: Suppose the operating system, which is some process in concurrent activity, wants to interrupt the process B, which it happens to know by name. It will need some primitive stop* which would be the contrary of resume indirect. In such a way, even if B is not in state 1, but if it has relinquished the control to some descendant f, stop* will interrupt the compound process and resume* will restart it exactly as if it were a simple process.

To keep the simplicity of the CBL representation requires those possibly long primitives: a physical implementation might put an upper bound on the number of descendants, but it does appear feasible.†

5. The Translation of Simula-67 Into CBL-73

The following must be specified:

how to build generic-names and represent objects and references to objects
how to set up the local structures
how to translate the primitives: this_new_detach_resume_end_inspect of Simula-67

The previous study (of Part III) has shown how to compile any program using these facilities.

We invite the reader to draw diagrams as he goes through the translation of those primitives.

a. The task to build generic names is left to the language at hand. For Simula-67, we propose the following:

Each of the blocks, procedures, classes declared in the program is given a unique generic name. If the block or the class B is prefixed by a class A, the generic name of A is concatenated to the generic name of B, in effect forming A.B; this convention is recursive, when A is already prefixed.

† I strongly recommend to store the layer of processes in an associative memory: a proper scheme for names could use this to speed up indirect instructions.

A reference variable is a new type of the language which has two components:

name: which is empty or the unique-name of a process

generic-name: which is a generic-name

Reference variables are used for referencing class-objects only.

b. The local structure of any process contains four components special to Simula in addition to $1 and $2: (see p 8).

Succ and Prede which are reference variables initialized to

name: empty

generic-name: linkage

System: which is a reference variable initialized to the name and genericename of the prefixed block defining the quasi-parallel system to which the object was originally attached.

Detached: which is a boolean variable equal to 1 if the object is detached, 0 otherwise

Return, a unique-name, appears in the local structure of prefixed blocks to ensure proper block exits despite the fact that prefixed blocks are born detached.

CBL processes from a class prefixed by "process" and from class "event-notice" have other special components as mentioned earlier.

c. this (see p 7) is translated into FN within a class or a prefixed block; or it may be an external passed as an argument to procedures and simple blocks (it may also be a general reference variable in special cases not discussed here as in "inspect" blocks).

new (see p 6): taken as an example of creation and call (almost similar translations would be performed at the entry of blocks and call of procedures).

Let (B, β) be in execution;

let A be the generic-name of a closure;

let X be a reference variable of generic-name compatible with A

let x := new(A) be the program instruction.

In the text of B, instead read:

```
create A, A.ip
assign $1, X-name
assign $1, MN
resume $1
unmark MN
```

Wash $1 only appears for any block entry or a procedure call, after the "unmark MN" instruction. $ARG is supposed to be already in L(B,β).

In the text of A, read:

```
mark      Succ
assign    Succ-generic name, linkage
mark      Prede
assign    Prede-generic name, linkage
mark      System
mark      Detached
mark      $PAR
mark      $2
sense     LN
select    $2, $ARG, $PAR
unmark    $2
select    $PAR, system, system
assign    "0", Detached   /*Boolean "0"*/
```

Notice System is always passed as an argument except for prefixed blocks. In that case (which is not a translation of new, but of the entry of a prefixed block!) the sequence is slightly different:

in the text of B:

```
suppress        assign $1, MN      for a prefixed
and             unmark MN      ;   block is born detached
```

in the text of A:

```
suppress        select $PAR, System, System
                assign "0", Detached
and instead put select $PAR, generic-name, temp
                assign temp, System-generic name
                assign LN, return
                assign FN, LN
                assign FN, System-name
                assign "1", Detached   /*Boolean "1"*/
```

The prefixed block gets his generic-name from the parent-object as a parameter. At this point read again II, 2), a) on p. 2.

d. Detach: Suppose the first instruction of A, in the preceding example, is "detach" as it would be the case, were A a class prefixed by "process." Then, in the text of A, there follows:

```
if Detached then resume * LN
          else do:  mark     temp
                    assign   LN, temp
                    assign   System-name, LN
                    assign   "1", Detached
                    resume   temp
               end;
```

(We use Algol-like notation as a symbolic short-cut!)

As mentioned in the earlier discussion: the effect of a detach statement within an object already detached is equivalent to a resume procedure for the associated prefixed block. A detach statement within a prefix block is, therefore, a dummy statement.

Resume: At this point read II, 2), b) on page 4 and II, 4), e) on page 7. Resume operates on the current event notice (after the erasing of the old current one) through the value returned by a built-in procedure "current." This value is a reference variable and the general format of the instruction resume is as follows:

Let (B, β) be in execution
let X be a non-empty reference variable (A.α, G)
let resume (X) be the program instruction

in the text of B, read:

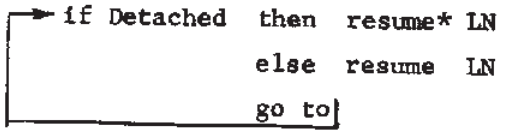
```
sense X.name
select $2, Detached, temp
unmark $2
if ¬ temp then ERROR
          else resume *  X.name
```

Note: Appreciate the use of indirect resume instructions.

End: At this point read over II, 2), a) on page 3.

end in a block which is not prefixed or a procedure is as translated in paragraph 4): resume LN and there is a corresponding wash statement in LN as the next executable instruction.

end in a class has two possible outcomes which can be molten in the following:



end in a prefixed-block is translated by: resume return

Note: An end statement is reached in a detached class only after an error has been notified. The normal exit is a "passivate" statement inserted just before the end statement.

Inspect: Let (B, β) be operating;
let X be a non-empty reference variable (A.α, G);
let inspect (X) be a program instruction.

The compiler has determined the set of attributes needed from A; it outputs:

```
sense  X-name
select $2, attr 1, temp 1
  :
select $2, attr n, temp n
unmark $2
```

6. Problems, Explanations and Difficulties

a. A first problem was to choose how to translate block, procedure and class activation. We felt that a very consequence of the definition is the creation of a new local structure: hence, in our model, there must be a corresponding process, and a formal call to the process with argument transmission. This is simpler than to distinguish between a simple block and others.

b. The first serious problem was a cycle problem. Amerasinghe has tackled the problem of cycles within a local structure and we do not look again at this. But a more deeper type of cycle arises when one deals with coroutines: while a simple block-structured language builds a tree of processes, Simula builds general

graphs of them. Hence keeping pointers leads to cycles. Our philosophy is that no physical cycle must appear, to allow a simple garbage collection at the system level: instead we transform a physical cycle into a conceptual cycle by use of names instead of pointers to objects, and we transfer the task of general garbage collecting to specialized procedures of the language under consideration.

Rule 1: no physical cycles are allowed.

Rule 2: the system level simple garbage collection applies to everything except processes (obviously the layer of processes must be known as such for interpretation).

Rule 3: every program must wash its own processes in its own way (dealing with its own kind of conceptual cycles).

Hence the last instruction of the process controlling "a" program, is always: "wash FN" (this supposes the operating system is running concurrently).

A block structured language is readily implementable this way, for procedure returns and block exits invoke a corresponding wash operation on their processes.

However, class objects never get washed out, to allow for the performing of sense instructions on them.

Therefore, each time the end of a prefixed block (A, α) has been performed and the instruction wash (A, α) is read as the next executable instruction (see above), a call to a special subroutine is executed instead, which has the ability to sense every process generated by the program (such a table must be kept by the system anyhow; or it is the whole control structure of the machine if the latter is not shared). It then executes the loop:

```
        select $2, System, Temp
if Temp = (A, α)  then  wash $2
                        unmark $2
```

Another specialized subroutine may be called within the simulation block if further storage is lacking. It is allowed to know every process so far generated within the simulation block. The rule would be:

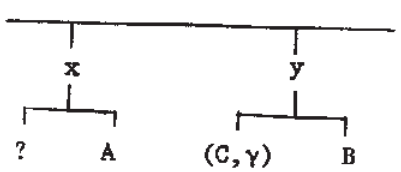
list every proeess within the SQS in list L
take the first one in L
→ for it, list its son-name, father-name, ep.SUCC, ep.PREDE and any other reference name in its local structure; let l be the list:
merge l after L.
if the end of L is not reached then take the next in L and go to ┐
else:
take the first process in the control structure to be known
→ if it is not in L, wash it off
if there is a next one, take it and go to ┐

This gets rid of implicit deaths.

Note: The fact that this subroutine ought to gain direct access to the control structure (to read names) is very bad, for we tried to make the control layer accessible by a process only through instructions create, wash, resume; and direct access of itself. Maybe we need to copy the names into the local structure of each process.

c. The problem of cycles leads to give names to processes. A feature of Simula 67 determined the form of the names and the form of references. A reference variable is qualified in the language by a class name, so that an assignment to it of a process which is not an instance of the same class, or one of its subclasses, produces an error message at run time, when it has not been discovered at compile time. The way the compiler would translate a check is made simple by the concatenated form we have adopted.

Let x = y be the instruction at run time.



The check is to see if A is a prefix of C by string manipulations; then to assign y.name to x.name. Note that B must be a prefix of C.

d. Simula has a notion of objects slightly different from ours: procedure calls, for example, are done by executing the procedure "within" (or "attached to") the caller; it may passivate itself (= "this" = the caller) and resume another process; after return to the first object, the rest of the procedure is executed. But since we decided to represent a procedure by a separate process, we had to provide for resuming a called procedure when awaking its caller, without knowing the call from outside. This leads us to the notion of son-name and indirect resuming, which are felt to be inherent to coroutines.

e. Create, resume, and wash were thus used to implement coroutines. Sense is needed to allow communications between local structures, without gaining access to the control layer. The "apply" primitive of CBL makes use of it implicitly. It is needed explicitly by Simula through remote access which makes known to a process attributes of another process which may be independent of the first one: a bad way would be to attempt to make the second process a subroutine of the first, to return attributes to it.

f. Difficulties are not absent in CBL-73, however, and a more careful study of the primitives to be added is required.

Notice that in the apply-return example of (4), the sequences:

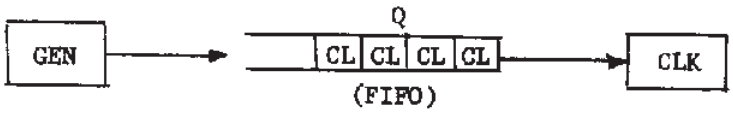
```
assign $1, MN        ;   and:   resume LN
resume $1                       unmark MN
```

are not interruptable

(Interruptions would make resume* instructions from outside illegal.)

V. An Example

1. Sample Program




```
(Algol)     begin
Simulation begin int I, NRCL;                    (count, nr of clients to generate)
                 real SVTM, MQT;                 (service time, mean queuing time)
                 bool IDLE;                      (flag for CLK)
                 real array QT [1:100];          (queuing time array)
                 ref (head)Q;                    (a set)
                 ref (clerk)CLK;                 (a clerk)
                 ref (clgen)GEN;                 (a client generator)
       process class clgen; begin real T;        (random generation time)
                                ref (client)NEXT;   (a client generated)
                              L:I := I + 1;         (iterate)
                                if I > NRCL then goto FIN;  (all generated → die)
                                T := random;        (next client arrival time)
                                hold(T);            (wait until then)
                                NEXT := new client; (generate it)
                                activate NEXT after this;   (schedule it)
                                goto L;             (loop)
                            FIN: end;               (suicide!)
       process class client;begin real TIN, TOUT;  (arrival and exit times)
                                int ID;             (arrival number)
                                ID := I - 1;        (get it)
                                TIN := time;        (remember input)
                                this . into(Q);     (get into the queue)
                                if IDLE then reactivate CLK;   (ask for help)
                                        else passivate;        (wait, he is busy)
                                TOUT := time;       (remember output)
                                this . out;         (get out of Q)
                                QT(ID) := TOUT - TIN; (how long did I wait?)
                                end;                (served → die)
```

(*)

```
process class clerk; begin ref (client)NEXT;      (a client)
                           int ID;                (a client number)
                       L:  IDLE := FALSE;         (somebody called?)
                           ID := ID + 1;          (iterate)
                           NEXT := Q.first;       (who is first?)
                           activate NEXT;         (get it out of the Q)
                           hold(SVTM);            (serve it)
                           if ¬ (Q.empty) then goto L;     (who is next?)
                           if ID = NRCL then reactivate MAIN; (Nobody? → close
                           IDLE := TRUE;          (expect somebody yet)
                           passivate;             (wait for him)
                           goto L;                (loop)
                           end;                   (... )
    Q := new head;
    GEN := new clgen;                                        }
    CLK := new clerk;                                        } (initialize)
    SYTM := 5.; NRCL := 2; IDLE := TRUE;                     }
    activate GEN after this;                       (start)
    passivate;                                     (wait until the end)
    for I := 1 step 1 until NRCL do MQT := MQT + QT(I)/NRCL; (compute statist
    end;                                            (terminated)
    end (Algol);
```

Note: For the sake of understandability, some syntactic features of Simula have been omitted (denotation, etc.). This program is semantically correct but could not be run as such on a CDC 6000.

2. Comments on the Example

- This is a program simulating a one-server, one-queue system. Clients are generated by the object GEN of class CLGEN at random intervals determined by calls to a built-in random number generator procedure. Each time a new client is created, it is activated and puts itself in Q. It then tests to see if the clerk CLK of class CLERK is IDLE or not. If not it passivates itself; if yes, it activates the CLK -- CLK then activates the first CLIENT in Q. This client gets itself out of the Q -- CLK then holds SVTM (service time) units with the flag IDLE off and then tries to activate the new first CLIENT in Q if any, otherwise it passivates itself and sets IDLE on. Simsim initializes the system by creating Q, GEN, and CLK -- it also terminates the simulation by calculating the mean queuing time of all CLIENTS.

- By the time control goes to Q := new(head), there already exist 5 objects in the system: the outer Algol block, the prefixed Simula block Simsim, SQS and a current event notice in SQS for Main, a companion of Simsim.

- The reader may easily simulate the whole system according to our rules. We have only represented here a partial snapshot of the interpreter state before the execution of the first statement: Q := new(head), followed by a partial snapshot of the interpreter state at step (*) for the second client generated. It is assumed that a first client was generated at time 1., and a second at time 2. .

- We have left out possible assignments of $1 and $2. For instance in L(GEN) $1 would in fact be (CLIENT, 1), the last object created, $2 would be (HEAD, 3) while performing hold(T) since hold(T) implies "sensing" the local structure of SQS.

- All "Sys" components when non-null are (Simsim, 2) since this is the only level existing (1 prefixed block). Terminated objects are kept around though no more executable.

- Terminated Event Notices are not represented here.

- ip pointers point to texts in closure of the universe (not represented here).

- At step (*) , only the second client is active.

- All MN are null, though it is not always the case. During a call to "random", a procedure attached to it, GEN MN would be the unique name MN of "random" control structure

- All LN are (Simsim, 2) because all processors are detached from or terminated and formerly attached to (Simsim, 2).

3. Snapshots

Generic names in use have the following abbreviated versions:

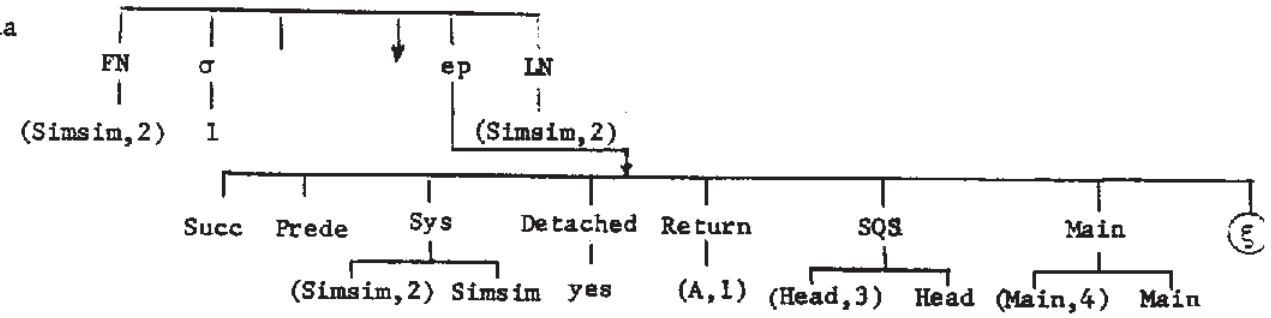
Algol	A	(the outer algol block)
Linkage	L	(the Simula-67 block)
Simset - Simulation	SimSim	
Linkage - Head	Head	
Link - Event notice	Ev	
Link - Process	Pr	
Link - Process - Main program	Main	
Link - Process - Clgen	Clgen	
Link - Process - Client	Client	
Link - Process - Clerk	Clerk	

This is a departure from the real notation which requires concatenation of names.

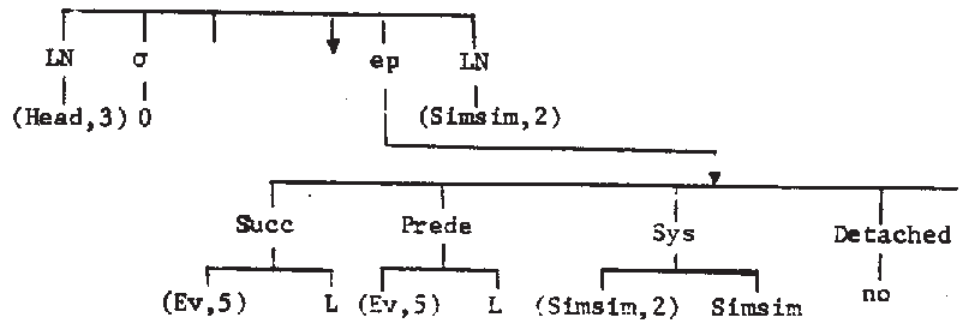
Initial Snapshot { The use of MAIN is to give the Simulation block the equivalence of the scheduling capacities of a process. Whenever a scheduling statement refers to this block in the program, it is translated by making the statement to refer to MAIN, instead. Notice that the Simulation block does not have the prefix process.

Algol

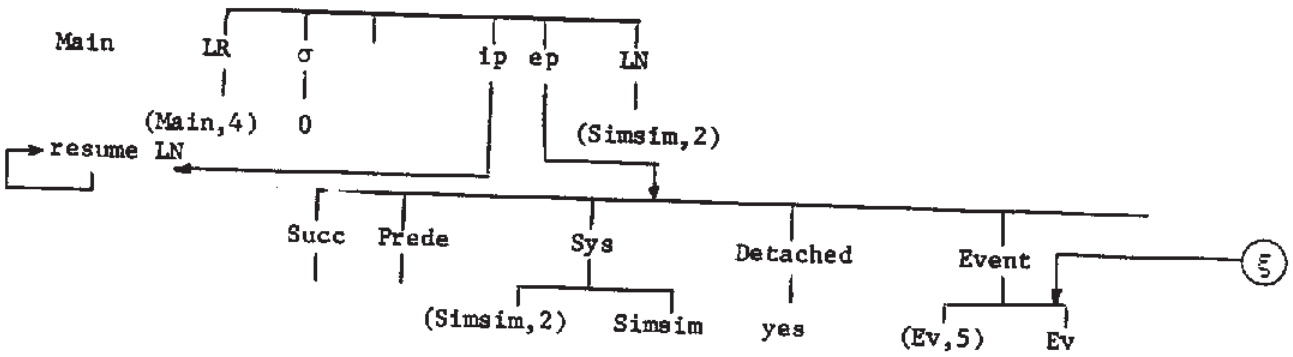
Simula

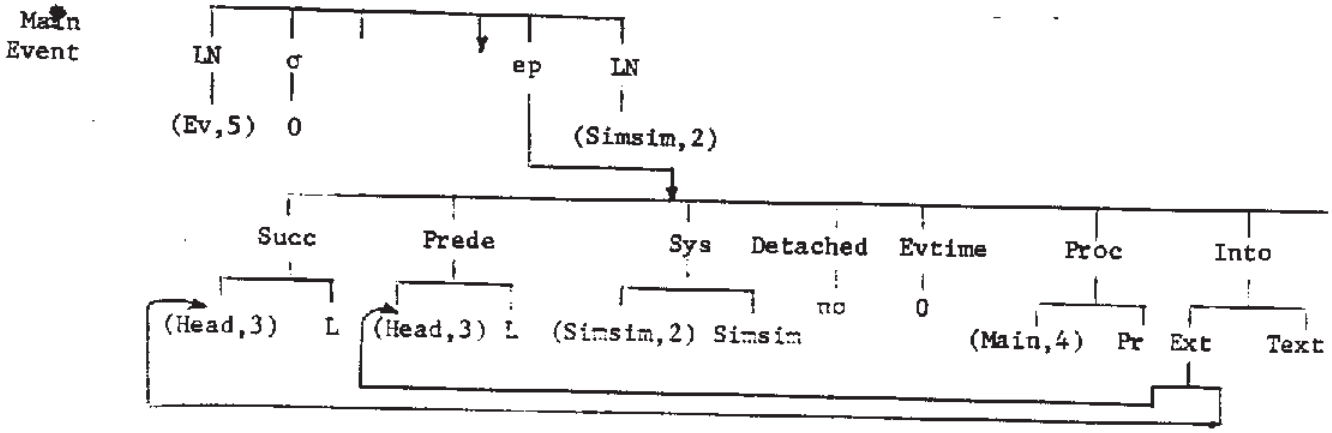


SQS



NB: (ξ) : See next page.

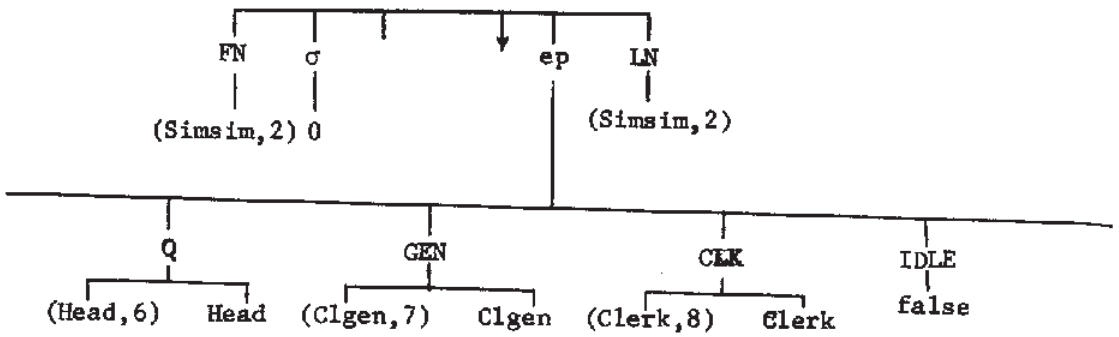

Main
LR
σ
ip
ep
LN
(Main,4)
0
(Simsim,2)
resume LN
Succ
Prede
Sys
Detached
Event
ξ
(Simsim,2)
Simsim
yes
(Ev,5)
Ev


Main
Event
LN
σ
ep
LN
(Ev,5)
0
(Simsim,2)
Succ
Prede
Sys
Detached
Evtime
Proc
Into
(Head,3)
L
(Head,3)
L
(Simsim,2)
Simsim
no
0
(Main,4)
Pr
Ext
Text

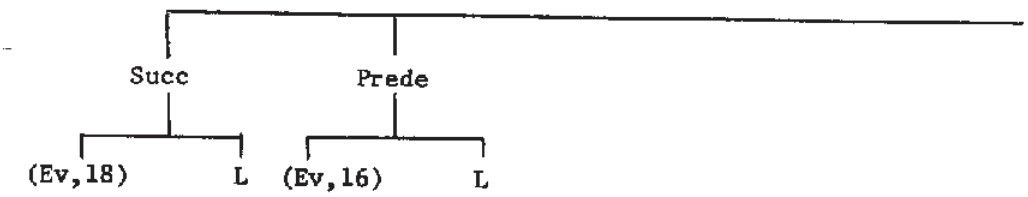
Second Snapshot

Algol: as previously

Simula:

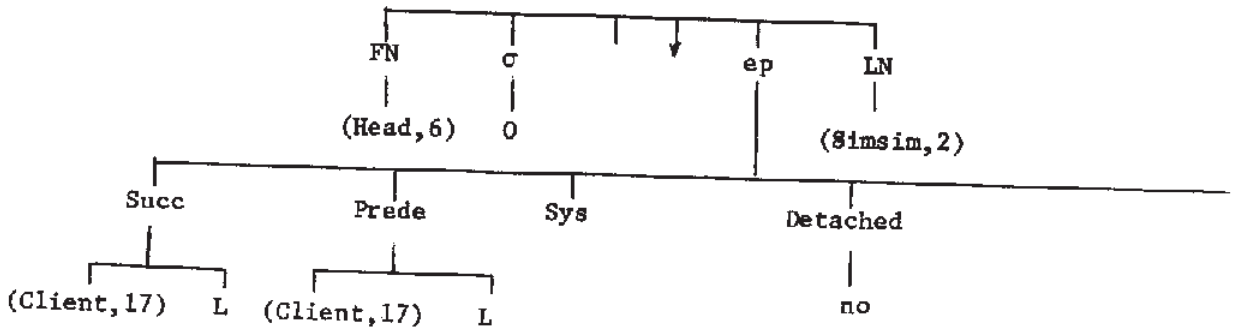


SQS: the local structure has become

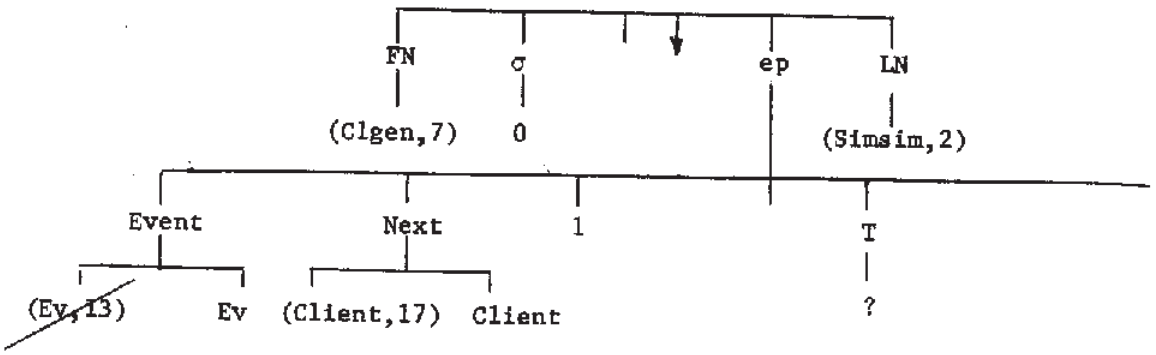


Main and Main-Event: Not shown; notice that (Ev,5) is logically dead since the execution of: "activate GEN after this; passivate", by the Simula block. If the program is correct, (Ev,5) is in fact implicitly inaccessible. However, it is still in the local structures of Simula and Main and will be overwritten only when CLK executes: "reactivate MAIN".

Q

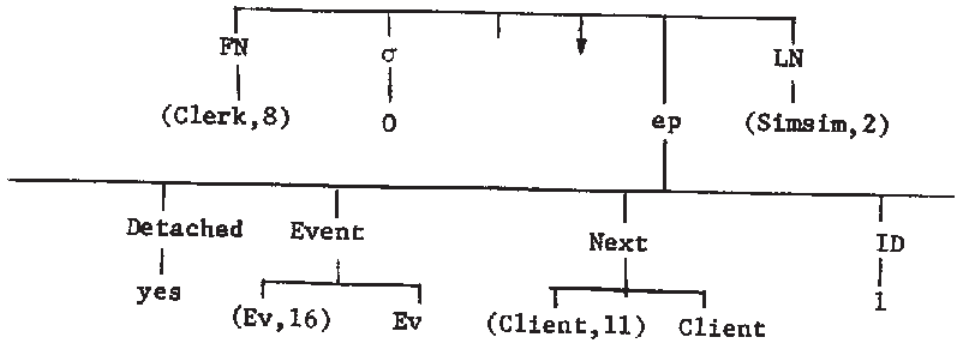


GEN



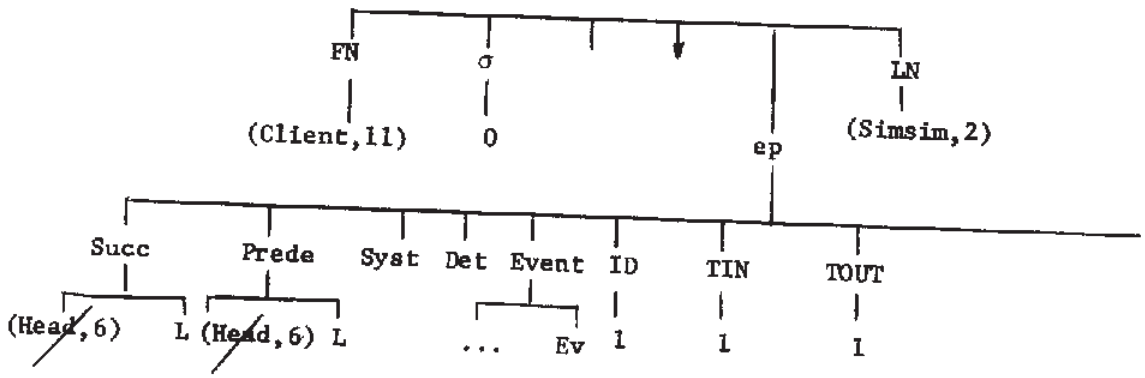
(Ev,9), at time 0, (Ev,10), at time 1, (Ev,13) at time 3 are the three activations of CLGEN.

CLK



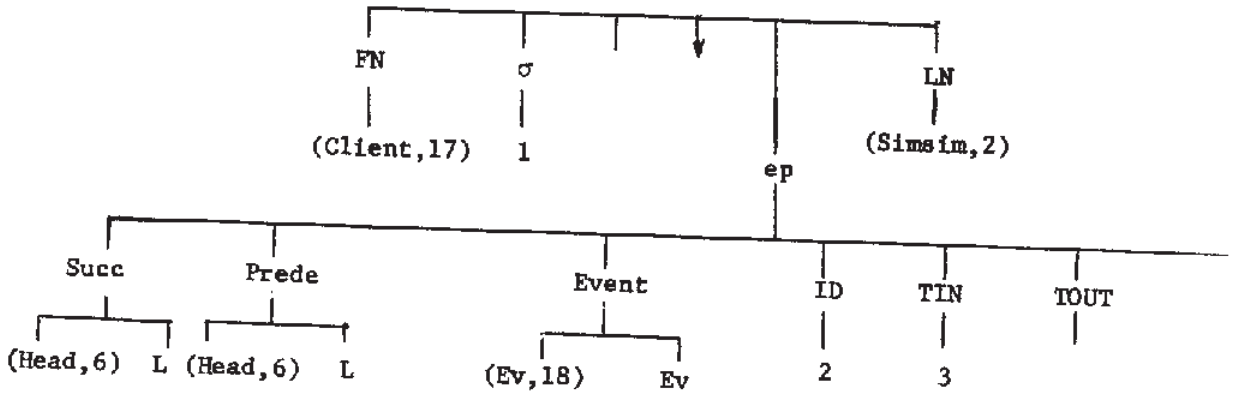
(Ev,14) corresponds to the service of the first client at time 1;
(Ev,16) is scheduled in SQS at time 6.

First Client



(Ev,12) and (Ev,15) correspond to the two activations of "first client", at time 1.

Second Client



Of the Event notices, only two are currently alive and are in SQS:

(Ev,18): { process: second client (currently in activity)
time : 3

(Ev,16): { process: clerk
time : 6

The client will wait in the queue until the clerk wakes up at time 6 and services him.

Time 0	Time 1					Time 3		Time 6
(Ev,5); (Ev,9);	(Ev,10);	(Ev,12);	(Ev,14);	(Ev,15);	(Ev,14);	(Ev,13);	(Ev,18);	(Ev,16)
(Main),	(generator),	(1st client),	(Clerk),	(1st Client),	(Clerk),	(Gener-ator),	(Client),	(Clerk)

References

1. J. B. Dennis, On the Design and Specification of a Common Base Language. Project MAC Technical Report TR-101. M.I.T., Cambridge, Mass., June 1972.

2. N. Amerasinghe, Translation of Blkstruc Programs to the Base Language. Computation Structures Group Notes, Project MAC, M.I.T., Cambridge, Mass., February 1973.

3. P. Coueignoux and P. Janson, On the Translation of Simula 67 into a Common Base Language. Course 6.534 Review Paper, Department of Electrical Engineering, M.I.T., Cambridge, Mass., Spring 1973.

4. Elgsaas, A Short Introduction to Simulation and Simula. Norsk Regnen Zentraal, 1970.

5. K. Nygaard, System Description by Simula. Norsk Regnen Zentraal, 1970.

6. Palme, Simula 67. Norsk Regnen Zentraal, 1970.

7. Simula Reference Manual 6000, Revision E. CDC Documentation Dept., 1971.

8. Chavy, Introduction au Langage Simula 67. Unpublished notes, European Software Development, Paris 1970.