General Concepts of the Simula 67 Programming Language

J. D. ICHBIAH and S. P. MORSE†

Compagnie Internationale pour L'Informatique, Les Clayes Sous Bois, France

Introduction

Simulation languages, like many problem oriented languages, serve a two-fold purpose. They provide the user with a number of frequently used operations. In addition, they offer him an ease of expression that is usually not found in general purpose programming languages.

A simulation program is indeed a program in the sense that it can be executed and will give results corresponding to the workings of the model it describes. It should not be overlooked that a simulation program is also a representation of a model and, as such, must be concise and readable. The design of a model is usually the result of successive redefinitions and modifications. This process will be easier if the representation of the model remains clear and understandable at each step.

Simulations involve complex entities, called processes, which consist of data structures and of algorithms that operate on these structures. Such associations of data and algorithms have general implications. However, languages referred to as "general purpose programming languages", such as FORTRAN, ALGOL, and PL/I, do not provide adequate facilities for the definition and handling of such entities.

These are some of the considerations which underlie the definition of the SIMULA 67 language. Dahl, Myhrhaug, and Nygaard [4] have, in effect, designed it as a general purpose language that is general enough to serve as the base of the definition of a simulation language. SIMULA 67 incorporates into ALGOL 60 the notions of classes and objects as well as facilities for the treatment of quasi-parallel systems. Classes permit the definition of complex entities which later can be used as elementary entities. In other words, these

† Present address: Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

complex entities can be used globally without having to refer to their components. Because of this feature SIMULA 67 can be used to conveniently define problem oriented languages. For example, facilities for handling symmetrical lists have been defined in the class SIMSET. Similarly, the class SIMULATION has been defined which offers facilities for handling quasi-parallelism analogous to the facilities found in the earlier language SIMULA [1, 2].

The first part of this paper describes the notions of class, subclass, and object. The second part treats reference variables and the means to assure that they can be safely used. The last part treats quasi-parallelism. The notion of virtual entity and an example on the use of SIMULA 67 are presented in the appendices.

1. Classes and Objects

1.1. Origin of the Notions of Class and Object

The notions of *class* and *object* in SIMULA 67 [3, 4] can be traced back to the notions of *block* and *block instance* in ALGOL 60 [6] and also to the notions of *record class* and *record* introduced by Hoare and Wirth [5, 7].

An ALGOL *block* is a description of a composite data structure and associated algorithms. When a block is executed, a dynamic instance of that block is generated. The *block instance* is the composite data structure described by the block and contains the local variables of the block as well as information needed for the dynamic linkage to other blocks. It is possible to have interaction between instances of different blocks and even between instances of the same block in the case of recursive procedure calls.

The notion of *class* is a generalization of the notion of block. As with instances of blocks, it is possible to generate multiple *objects* of the same class and each object thus generated is an instance of the class. However, control may pass from one object to another in more general ways than is possible for procedures. Thus an object may suspend its execution. It may also ask that the execution of a different object be initiated or resumed.

When there are no algorithms associated with the data structure of a class, the class reduces to a *record class* with its objects being *records*. In general, objects are records to which algorithms are associated. Different objects can coexist in memory; they may be of the same or of different classes. Objects of the same class can be at the same stage or at different stages of execution of their algorithm.

The definition of classes and subclasses, their concatenation, and the generation of simple and compound objects are described in the following sections.

1.2. Classes and Subclasses

SIMULA 67 is an extension of ALGOL 60. With some minor exceptions the rules of ALGOL 60 have been preserved in SIMULA. The definitions of certain syntactic units, however, have been extended to account for the new concepts of SIMULA.

For example, as in ALGOL, the head of a SIMULA block contains declarations. However, in SIMULA these can be the declarations of classes as indicated by the redefinition of the syntactic unit < declaration >.

< declaration > ::= < ALGOL declaration > | < class declaration >

1.2.1. Class Declaration

A *class declaration* can have the following form:

```
class A (PA); SA;
begin DA;
      IA
      ; inner;
      FA
end
```

The identifier A is the name of the class; PA is the parameter list of the class A; SA is the list of specifications of the parameters PA.

The *class body* begins with a list of declarations DA. The symbols IA and FA represent lists of instructions respectively called *initial operations* and *final operations* of the class A. The symbol **inner** represents a dummy instruction acting as a separator between the initial and final operations.

The quantities passed as parameters in PA or declared in DA are called the *attributes* of the class A and are hence also attributes of any object of that class. We will see later that attributes of an object may be accessed by other objects whereas an entity declared in a block which is a subblock of the class body may only be accessed in the block itself. Note that a class declaration can be one of the declarations in DA. This would then be a class attribute of the class A.

1.2.2. Object Generation

The expression

new A (. . .)

where A is a class, is an *object generator*. When encountered it creates an object which is an instance of class A and starts execution of the initial operations of A. The execution continues until

the end of the class body is encountered, at which time the execution is terminated. Execution will be suspended, however, if a call to the procedure "detach" is encountered. It is this suspension capability that makes it possible for many objects to exist simultaneously.

When the execution of an object is terminated, the data structure of the object remains in memory as long as it is possible to access it.

1.2.3. Subclasses

A class declaration may be preceded by a *prefix* which is the name of another class. The prefixed class is now called a *subclass* of the

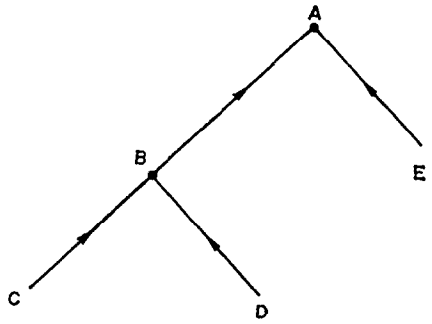


FIG. 1. Hierarchy of classes and subclasses.

prefix. The following is a declaration for the subclass B of the previously declared class A.

```
A class B (PB); SB;
begin DB;
    IB
    ; inner;
    FB
end
```

More generally a hierarchy of classes can be introduced by a succession of declarations of subclasses as follows (see Fig. 1):

```
class A . . . ;
A class B . . . ;
B class C . . . ;
B class D . . . ;
A class E . . . ;
```

A graph corresponding to this hierarchy is an oriented tree. The root of the tree is the class that has no prefix (in this case A). The

prefix sequence of a given class is the sequence of classes encountered on the path (unique) going from the given class to the root. In the above example the prefix sequence of class C consists of C, B, and A.

A subclass is said to be *inner* to its prefixes. Thus C is inner to B and A. Conversely B and A are said to be *outer* to C.

1.2.4. Class Concatenation and Compound Objects

A subclass is equivalent to the class obtained by the concatenation of those classes that are on its prefix sequence. Thus the subclass C is equivalent to the class obtained by the concatenation of A, B, and C. This section describes the data structure associated with the concatenated class; the next section defines its class body.

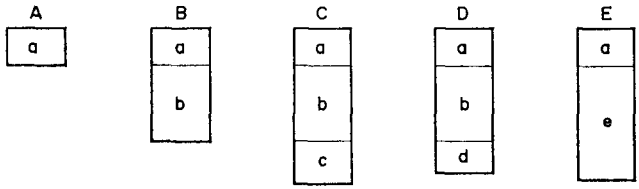


FIG. 2. Structure of compound objects in Fig. 1.

An object of a class resulting from a concatenation is a *compound object*. The concatenation process can be interpreted in terms of the space occupied by the data structure of a compound object as follows. This space is the union of the spaces occupied by the data structures of the various classes in the prefix sequence. Fig. 2 shows the structure of the compound objects in the previous example (Fig. 1). The spaces a, b, c, d, e represent the spaces occupied by the data structures defined by A, B, C, D, E.

1.2.5. Class Body of a Concatenated Class

Consider the class A and its subclass B defined previously. The subclass B is equivalent to the fictitious class K defined below.

```
class K (PA, PB); SA; SB;
begin DA; DB;
      IA;
      IB
      ; inner;
      FB;
      FA
end
```

The class K is obtained therefore by concatenating PA and PB, their specifications SA and SB, and the declarations DA and DB. The body

of class K is obtained by replacing the symbol **inner** of A by the initial operations of B followed by; **inner;** and the final operations of B. This illustrates the role played by the symbol **inner.** On the one hand it is a dummy instruction when executed, and on the other hand it specifies where code may be inserted for a compound object.

The execution of an object generator such as **new** B (. . .) is equivalent to that of an object generator of the fictitious class K, namely **new** K (. . .).

1.2.6. Example

A simulation model involves the interaction of independent entities called *processes.* It is convenient to be able to distinguish between the generation of a process and its activation, and also to test if the execution of a process is terminated or not. The following example shows how the class "process" can be defined within SIMULA 67 so that these conditions are satisfied. (Dot sequences . . . represent elements that are not essential to the understanding of the example.)

```
. . . class process;
begin . . . ; Boolean TERMINATED;
      detach;
      inner;
      TERMINATED:= true;
      . . .
end
```

Consider now a simulation of the spread of an epidemic disease [2]. The processes interacting in this simulation are sick persons and treatments. The definition of a sick person is given in the declaration of the following subclass of the class process.

```
process class sick person . . . ;
```

The effect of the object generator "**new** sick person" is as follows. The first action performed by the resulting compound object is a call to the procedure "detach". The execution of the object is thereby suspended. It is only after a subsequent activation that the actions indicated in the class body of "sick person" will be carried out. After completion of these actions the final operations of the class "process" are begun and the value "**true**" is assigned to the variable TERMINATED.

In the previous example two classes have been defined corresponding to the two successive levels of description.

—the class "process" accounts for the actions that are necessary because objects considered are processes.

—the class "sick person" states the actions resulting from the fact that the processes considered are also sick persons.

It is clear that the definition of the class "process" can be done without considering the subclasses of "process" which are defined afterwards.

1.2.7. Prefixed Blocks

The mechanism of concatenation can be extended to the case of prefixed blocks. The execution of a block prefixed by a given class consists of successively executing the initial operations of the class, the operations of the block, and the final operations of the class. In addition, the attributes of the class are accessible to the statements inside the prefixed block. Prefixed blocks are used mainly for the latter feature; accessibility to large sets of attributes is provided concisely by specifying only the prefix.

1.3. Relationship Between Class and Subclass

The relation between class and subclass may appear striking at first glance. In certain cases the relation appears similar to the relation between set and subset. If B and C are subsets of the set A, then B and C are included in A; also every element of B or C is an element of A.

For example consider the sets R and K of real and complex numbers. Note that R is included in K. Similarly, REAL may be made a subclass of the class COMPLEX in a SIMULA program.

```
class COMPLEX (real part, imaginary part);
real real part, imaginary part;
    . . .
end COMPLEX;
COMPLEX class REAL;
begin
    imaginary part:= 0;
    . . .
end REAL;
comment examples of generation of objects of these classes;

ref (REAL) X, Z; ref (COMPLEX) Y;
X :- new REAL (8, 0);
Y :- new COMPLEX (3, 2);
Z :- new REAL (3, 2);
```

The trouble with such a definition is that the treatment of reals is unnecessarily awkward. A real appears to be effectively a particular complex. Being treated as complexes, all reals possess therefore two fields—a real part and an imaginary part. The latter field is always null but nevertheless occupies space in memory. This inconvenience

can be avoided by realizing that a class definition specifies characteristics that are possessed by all objects of the class. The definition of the subclass in turn specifies additional characteristics possessed by objects that are members of the subclass. Note that whereas a subset implies a restriction, a subclass can be used to introduce additional attributes. In the definition which follows, the characteristic common to the objects of the class REAL is the possession of a real part. The objects which are members of the class COMPLEX possess a real part and, in addition, possess a complex part. COMPLEX should therefore be made a subclass of REAL.

```
class REAL (real part); real real part;
    . . .
end REAL;
REAL class COMPLEX (imaginary part); real imaginary part;
    . . .
end COMPLEX;

comment examples of generation of objects of these classes;
ref (REAL) X, Z; ref (COMPLEX) Y;
X :- new REAL (8);
Y :- new COMPLEX (3, 2);
Z :- new REAL (3);
```

Thus the notions of set and subset correspond to an analytic approach; sets are stratified into subsets. Classes and subclasses may be designed to correspond to such stratifications. However, it will often be preferable to use a synthetic approach for the design of classes. When designing a given class no *a priori* knowledge of its subclasses is necessary. More and more elaborate classes can thus easily be obtained by a step-by-step definition of subclasses which introduce additional characteristics.

1.4. Methodological Implications of the Concepts of Class and Concatenation

The indication of a class name in a prefix makes the entities defined in that class accessible within the prefixed class. Thus a simple and powerful means is available for the global communication of these entities.

Consider for example the classes SIMSET and SIMULATION. The class "linkage" and its two subclasses "link" and "head" are class attributes of SIMSET. The class SIMULATION is itself prefixed by SIMSET. The class attribute "link" is therefore accessible inside of SIMULATION and "link" can serve as a prefix to the classes "process" and "event notice". Similarly a class "TRAFFIC", prefixed by SIMULATION, can be defined. Class attributes of "TRAFFIC"

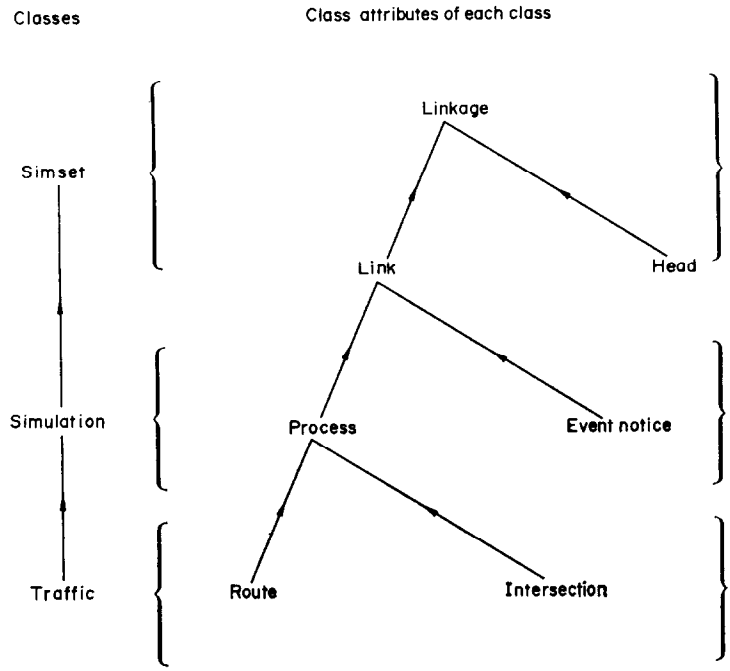


FIG. 3. Hierarchy of classes used for traffic simulation.

are the subclasses "route" and "intersection" of "process". A user doing a traffic simulation will thus be able to use these concepts inside a block prefixed by TRAFFIC (see Fig. 3).

```
class SIMSET;
begin class linkage; begin . . . end;
      linkage class head; begin . . . end;
      linkage class link; begin . . . end;
      . . .
end SIMSET;
SIMSET class SIMULATION;
begin link class process; begin . . . end;
      link class event notice; begin . . . end;
      . . .
end SIMULATION;
SIMULATION class TRAFFIC;
begin process class route; begin . . . end;
      process class intersection; begin . . . end;
end TRAFFIC;
```

The different classes correspond to the different levels from which a given problem can be viewed. SIMSET provides for the treatment of

lists. SIMULATION introduces the notion of simulation. The entities used in traffic simulation are defined in TRAFFIC. It is only inside a block prefixed by TRAFFIC that a traffic simulation of a particular network will be performed.

This has considerable implications for the design of large systems. In such design it is a common practice to introduce successive levels of definition and to use a coherent set of concepts within each level. In the traffic simulation problem, for example, the levels considered are the list processing level, the simulation level, and the traffic level. On the traffic level the concepts introduced are route and intersection. They form the coherent set of concepts normally used by traffic engineers. Similarly, link and head are coherent concepts in list processing. However, head and route, which appear on different levels, would normally not be used in the same context.

The prefixed classes of SIMULA bring this leveling design practice into a programming language. Because of this feature SIMULA 67 can be used as a powerful kernel language for the design of problem oriented languages.

2. Reference Variables and Remote Identifiers

2.1. Local Access and Remote Access

Consider the execution of an object. Some of the actions performed involve attributes of the object itself. This is called *local access* of attributes and the usual rules of ALGOL apply.

The term *remote access* is used for the accessing of attributes of other objects. In this case a given attribute is not uniquely specified by merely mentioning its attribute identifier. In fact, at any given time several instances of the same class, each having identical attribute identifiers, can coexist in memory. It is therefore necessary to specify the object to which the considered attribute belongs. A remote access is thus performed in two steps:

—The first step is the selection of a particular object
—The second step is the determination of the attribute of the selected object.

A new type of variable, the *reference variable*, is used for referring to objects. A reference to a newly created object is obtained during the execution of an object generator. This reference is analogous to a pointer. The language permits the calculation and assignment of references. For example, an object generator may be assigned to a reference variable. In such a case, the reference variable refers to the newly created object.

2.2. Inherent Difficulties of Remote Access

Some precautions must be taken if remote access is to be carried out efficiently and without loss of security.

Consider the access of attribute "b" of the object referred to by reference variable X. In an implementation of the language, the value stored in X may be the base address of the object's data or may serve to obtain that address. The displacement of a given attribute with respect to this base address is the same for all the objects of a class. The address of the attribute "b" can therefore be simply obtained by adding to the base address a value which is known at compile time.

If the requirement of efficient access is thus satisfied, then the same cannot be said about the security of use. The following risk is encountered. The language allows for the calculation of references and for their assignment to reference variables. It then can happen by accident that X refers to an object which belongs to a class other than the one intended. Even though this object has no "b" attribute, a value for the fixed displacement of that attribute is obtained at compile time. No error indication will be given until possibly much later in the execution of the program and the search for the cause of the error at that time may be difficult.

Hoare's solution, taken and expanded in SIMULA 67, is to have the syntax make this sort of error impossible. In the majority of cases these referencing errors can and will be detected at compile time. For the remaining cases a run-time check is necessary. However, these checks will not significantly degrade the run-time efficiency since they occur infrequently.

The elements of this syntactic solution to the problem of security of use for references are as follows.

(a) A qualification must be included in the declaration of each reference variable. This qualification indicates to which class an object referred to by the variable may belong. It also specifies the scope of the attributes which may be remotely accessed with the reference variable. Thus the legality of a remote access depends on static information only and can be determined at compile time.

(b) Reference assignments are defined and performed in such a way as to ensure that a reference variable will always designate an object whose run-time qualification corresponds to the declared qualification of the reference.

2.3. Reference Assignments

The qualification of a reference variable is used to test for the validity of reference assignments to that variable as illustrated in the

following example. The classes A, B and E are those defined in Fig. 1 (section 1.2.3).

```
ref (A) X; ref (B) Y;
Y :- new B;
X :- Y;
```

The validity of the above assignments may be established at compile time. In the first assignment the object **new** B and the reference variable Y are both qualified by B. Thus Y really refers to an object of a class that it was intended to refer to and there is no danger. In the second assignment, X is qualified by A and will serve for remote access to attributes of that class. Since each of these attributes will also be present in an object of any subclass of A, the object referred to by Y may be safely assigned to X. Now consider

```
Y :- X;
```

In this case it is necessary to check at run-time whether the object referred to by X is actually a member of B or of a subclass of B. If so the assignment may be carried out. Otherwise a run-time error results. Finally consider

```
ref (E) Z;
Z :- Y;
```

This assignment will be recognized as invalid at compile time. Reference variable Y is qualified by B, and Z by E. However, B and E are not in the same prefix sequence.

2.4. The Two Types of Remote Access

To carry out a remote access without ambiguity, an object and an attribute identifier must be known. In addition, to satisfy the security requirement the qualification of the object must be known. These requirements are satisfied by a *remote identifier* or, simpler yet, by the *connection.*

2.4.1. Remote Identifiers

A *remote identifier* is composed of a reference variable (or an expression delivering a reference value), a dot, and an attribute identifier. Consider the remote identifier Y.b for the previous example. When the reference variable Y was declared, its qualification was specified (in this case B). Hence it is possible to check at compile time whether it has meaning to look for a "b" attribute of such a class.

Thus if "a" and "b" represent respectively an attribute of class A and an attribute of subclass B prefixed by A the remote identifiers X.a,

Y.a, and Y.b are permitted. Indeed, X is qualified by A which has an "a" attribute. Similarly Y is qualified by B which has an "a" attribute and a "b" attribute after concatenation.

On the other hand, X.b is not permitted since "b" is not in the scope of A, the qualification of X. Note that X may nevertheless refer to an object of B (after the legal assignment X :- **new** B for example). However, it can be known only at run-time if the object referred to by X is an object of class A or of class B, or, even, of another subclass of A with or without a "b" attribute. Therefore this justifies rejection at compile time.

A local qualification may be placed on a reference variable for accesses which would not otherwise be permitted. The remote identifier X **qua** B.b where X is given the local qualification B is thus legal. However, a run-time check will be made to determine if X refers to an object either of class B or of one of the subclasses of B.

2.4.2. Connection

The *connection* is another mechanism for accomplishing remote access. The same principles that are valid for remote identifiers are also valid for the connection. The connection is a convenient type of access when several remote accesses to the attributes of a given object must be made. In such cases it is possible to factor, and therefore do only once, both the access to the object and the qualification check. Thereafter the attributes may be referred to by their identifiers alone. The format of a connection statement is the following:

inspect < object expression >
  when < class identifier > **do** < connection block >
  . . .
  when < class identifier > **do** < connection block >
  otherwise . . . ;

The evaluation of the object expression results in a reference to the object considered. Once the object is known, its qualification is also known. The qualification is then compared to the classes mentioned in the successive "when" clauses of the connection statement until a class is found that belongs to the prefix sequence of the qualification. The associated connection block is then executed. Inside this block the attributes of the object are referred to, exactly as with local access, by merely mentioning their identifiers.

2.5. Anotated Example—the Class SIMSET

The following is a simplified version of the class SIMSET used for list processing. Symmetric lists are considered. A list is made of a head

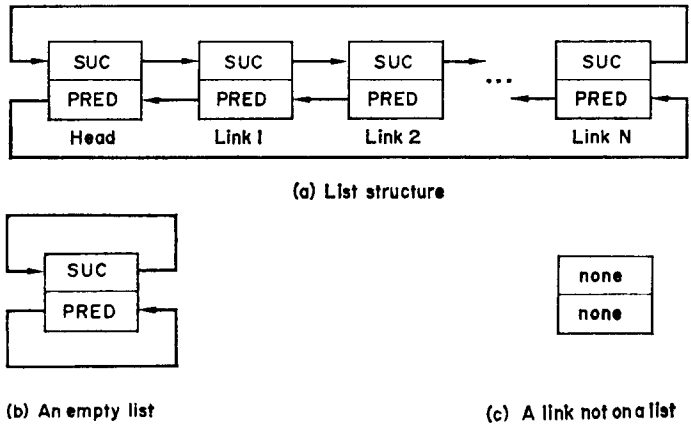


FIG. 4. SIMSET list structures.

```
(0)  class SIMSET;
(1)  begin
(2)       class linkage;
(3)       begin ref (linkage) SUC,PRED;
                 . . .
(4)       end linkage;
(5)       linkage class link;
(6)       begin
(7)            procedure out;
(8)            if SUC =/= none then
(9)            begin SUC.PRED :- PRED ; PRED. SUC :- SUC;
(10)                SUC :- PRED :- none
(11)           end;
(12)           procedure follow (X); ref (linkage) X; . . .
(13)           procedure precede (X); ref (linkage) X; . . .
(14)           procedure into (S); ref (head) S; . . .
(15)      end link;
(16)      linkage class head;
(17)      begin
(18)           procedure clear;
(19)           begin ref (linkage) X;
(20)                for X :- SUC while X =/= this head do
(21)                     X qua link.out
(22)           end clear;
                 . . .
(23)           SUC :- PRED :- this head
(24)      end head;
(25) end SIMSET;
```

and of links, possibly none if empty. A link does not necessarily belong to a list (see Fig. 4).

The class SIMSET possesses three attributes, all of which are classes. These are the class "linkage" (2–4) and its subclasses "link" (5–15) and "head" (16–24).

Only the attributes SUC and PRED of the objects of the class "linkage" (3) will be considered here. These attributes may refer to other objects of the class "linkage". It will be seen shortly how objects can be linked to form symmetric lists. Note that in SIMULA the reference variables are initialized to the null reference **none.**

The subclass "link" of "linkage" corresponds to the current list elements with the exception of the head of the list. All the attributes of this class are procedures. These attributes are accessible by means of remote identifiers. Thus in (21) X **qua** link.out constitutes a call of the procedure attribute "out" corresponding to the object of class "link" which is referred to by X. In instructions (8) and (10) SUC and PRED represent the fields of this object. Instruction (9) is more complicated. In SUC.PRED :- PRED, the variable SUC and the second occurrence of the variable PRED do indeed represent fields of the object referred to by X. On the other hand, SUC.PRED is a remote identifier. Thus that occurrence of PRED represents a field of the object referred to by SUC.

Each object of class "link" has procedure attributes namely "out", "follow", "precede", and "into". The procedure attribute "out" of an object of class "link" permits removing that object from the list to which it belongs. The procedures "follow (X)" and "precede (X)" provide for making the object a successor or a predecessor of the object referred to by X. Finally, if S denotes the head of a list, the object will be added to that list as a result of the execution of the procedure attribute "into (S)".

The subclass "head" corresponds to the list heads. The procedure "clear" is used for emptying a list. At each iteration of the **for** loop (20–21) the "link" which is the successor of the "head" is removed from the list. The local qualification "link" is given to the "linkage" X since "out" is an attribute of "link". The loop terminates when the list is empty, i.e., when the successor of the "head" is the "head" itself.

In (23) the fields of a list head are initialized. The value of the expression "**this** head" is a reference to the "head" object being executed.

In conclusion note that subclasses of the classes "head" and "link" can be defined in any subclass of SIMSET. The procedure attributes of "head" and "link" can therefore be used on behalf of the compound objects of those subclasses. Such compound objects know, in effect, the fields SUC and PRED on which these procedures operate. Note also that in SIMSET the concepts of list processing were defined without having to take care of the contents of the elements of the lists but

only taking into account the mechanism which permits them to be linked. It is in a subsequent stage, in the definition of subclasses of "head" and "link", that the contents of these elements are defined.

3. Quasi-parallel Systems

Simulating a system entails representing the different processes which act in parallel in the system. Usually only a single processor is used to carry out this simulation. This leads to a quasi-parallel representation of a parallel system. The parallel activity phases of different processes are, in fact, treated sequentially but the advancement of system time is done in such a way as to create the illusion of parallelism.

There are conceivably a large variety of mechanisms that permit the sequencing of parallel activity phases. The sequencing mechanism of the earlier language SIMULA used a list of event notices. An event notice contains a reference to a process and the "date" when that process must be activated. The event notices are ordered in the list by increasing dates. Hence the processes are treated in the order in which their event notices appear in the list.

In the case of continuous simulations, simpler mechanisms can be used. For example, the parallel phases of different processes can be treated successively and in an invariable order.

SIMULA 67 puts the sequencing mechanism of SIMULA into the class SIMULATION. It is also possible to define other sequencing mechanisms in SIMULA 67. For this purpose certain notions have been introduced into the language. These are the notions of quasi-parallel system, attached and detached component, as well as the procedures "detach" and "resume" which provide for detaching a component and resuming the execution of another component.

3.1. Definition of Quasi-parallel System

The execution of a SIMULA 67 program involves the generation and execution of block instances, of prefixed blocks, and of objects.

Consider the execution of a block. This is performed as in ALGOL. Instances of subblocks of a block are created and destroyed as these subblocks are entered and exited. This sequencing does not involve mechanisms other than the normal ALGOL stack mechanism. It is in this sense that these subblocks are said to be *attached* to the block and belong to the same *component* as the block.

Similarly, an object remains attached to the block calling for its generation until the delimiter **end** is reached or until a call to the procedure "detach" is issued as part of the execution of the object. As long as the object remains attached, it belongs to the same component as the block. When the delimiter **end** is finally reached, the object is no

longer attached but may still be kept in memory. Such an object can thus have an independent existence as a data structure. It is said to be in the "terminated" state.

Consider now the execution of an object which comprises calls to the procedure "detach". The first call to "detach" on behalf of a given object removes this object from the component to which it belongs and makes it an independent component. The object is said to be a *detached* object. This leads to a multi-component system. The main program and the detached objects are the components of this system. The multi-component system is called a *quasi-parallel system.*

3.2. Sequencing of the Execution of the Quasi-parallel System Components

At a given time only one component in a quasi-parallel system is being executed; the other components are temporarily suspended. The following definitions will be used to specify the sequencing effects of the procedures "resume" and "detach".

Definitions

—*The outer sequence control* (OSC) of a quasi-parallel system is the point of the system which is being executed at a given time.

—*The local sequence control* (LSC) of a component of a quasi-parallel system is:

(a) the actual execution point if the component is currently being executed (active component).

(b) the point at which execution will be resumed if the component is suspended.

At any given time it is clear that the OSC of the quasi-parallel system coincides with the LSC of the active component (see Fig. 5).

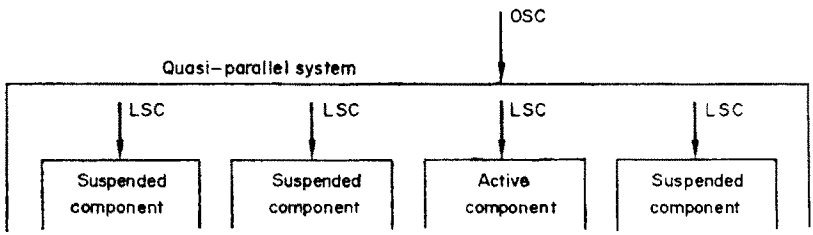
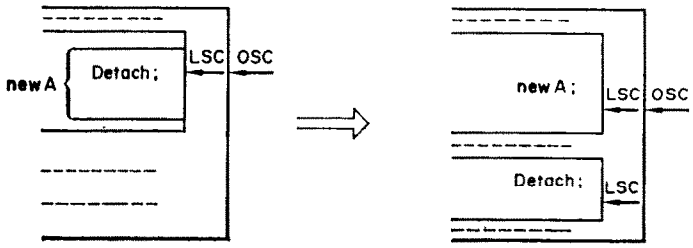


FIG. 5. Sequence controls of quasi-parallel system.

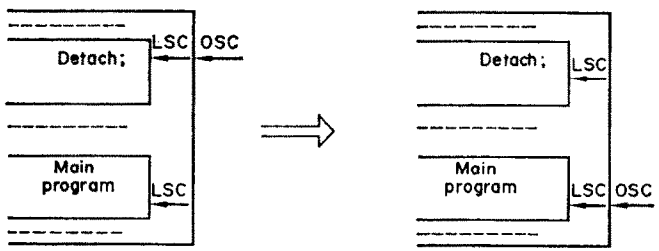
3.2.1. Effect of "Detach"

3.2.1.1. First call. Consider the first call to "detach" on behalf of a given object which is initially attached and is therefore a part of a component. After "detach" the object becomes an independent component. Its execution is suspended and its LSC is positioned after "detach". The OSC is positioned after the instruction that caused the generation of the object. This is illustrated in Fig. 6(a).



(a) First call

3.2.1.2. Later calls. Consider an object that issues more than one call to the procedure "detach". The object will be detached after the first call and remain detached when later calls are issued because it is not possible to reattach a detached object. For these later calls, just as for the first call, the LSC of the object is positioned after the call to "detach". But the OSC, on the other hand, takes the value of the LSC of the main program which therefore becomes active. This is illustrated in Fig. 6(b).



(b) Later calls

FIG. 6. Effect of a call to "detach".

3.2.2. Effect of "Resume"

The call of the procedure "resume (Y)" causes the execution of the object referred to by the reference variable Y to be resumed. The LSC

of the component containing the call to "resume" is placed after this call. The OSC of the quasi-parallel system takes the value of the LSC of the component referred to by Y. Note that "resume (Y)" can be called by the main program as well as by another component (see Fig. 7).

The parameter of the procedure "resume" must refer to a detached object. Specifically, Y cannot refer to a terminated object because such an object no longer has an LSC.

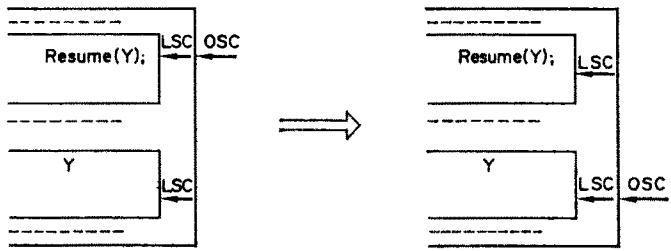


FIG. 7. Effect of a call to "resume".

3.3. SCHEDULING MECHANISM IN THE CLASS SIMULATION

Simulations describe the behavior of entities called processes which interact during a period of time. The basic problem faced in simulations is that of scheduling the simulated processes. Conceivably this may be done by a scheduler using a tag (event notice) associated with each process. Each tag contains the name of the corresponding process and the time at which that process is scheduled for activity. The scheduler will arrange these tags in chronological order and then always look at the first tag in order to know which process to activate next.

The class SIMULATION introduces into SIMULA 67 the notions of processes and event notices along with a scheduling mechanism (see Fig. 8). The class SIMULATION is prefixed by the class SIMSET. This allows the list processing facilities of SIMSET to be used for queuing event notices and for queuing processes. In addition, the scheduling mechanism of the class SIMULATION uses the primitive sequencing procedure "detach" and "resume" to achieve quasi-parallelism.

The class "event notice" defined in SIMULATION is a subclass of the class "link". Thus event notices have SUC and PRED attributes permitting them to be put on a list called the *sequencing set*. In

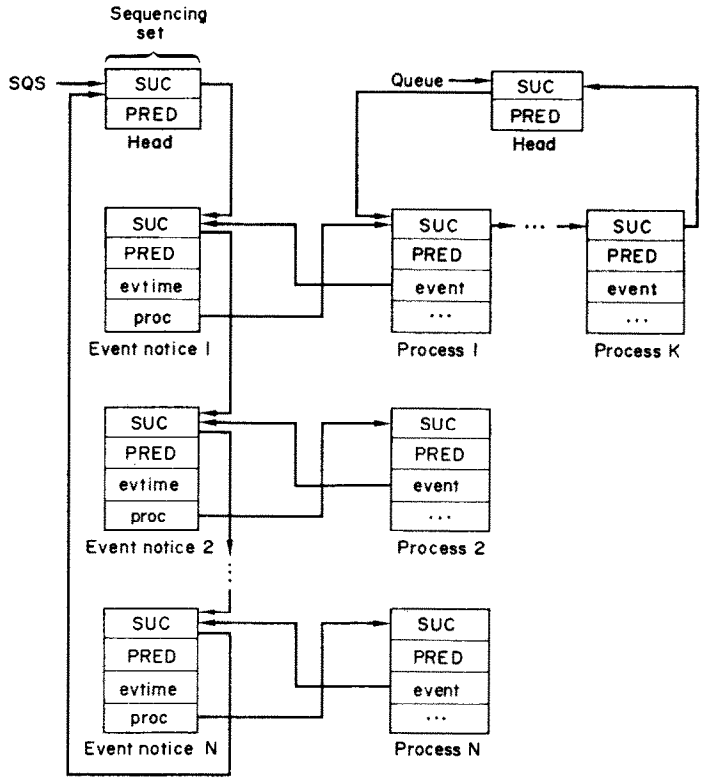


FIG. 8. Scheduling mechanism of the class simulation.

addition, event notices have the attributes "evtime" and "proc". The "proc" attribute is a reference to the associated process; the "evtime" attribute is the time at which that process is scheduled for execution. The event notices in the sequencing set are ordered by increasing values of "evtime". Hence the sequencing set represents the simulation time axis. At any stage of the simulation, the value of the simulation time is the value of the "evtime" attribute of the first event notice of the sequencing set. The head of the sequencing set is referred to by the reference variable SQS.

The class "process" is also a subclass of the class "link" and thus processes can be put on lists. In addition, a process has an "event" attribute which is a reference to its associated event notice. In most practical simulations the processes considered will actually be objects of subclasses of the class process rather than just objects of the class process itself.

```
SIMSET class SIMULATION;
begin
    link class process;
    begin ref (event notice) event;
        detach;
        inner;
        passivate;
        error
    end process;
    link class event notice (evtime, proc);
    real evtime; ref (process) proc;
    begin
        . . .
    end event notice;
    ref (head) SQS;
    procedure hold . . . ;
    procedure passivate . . . ;
    procedure activate . . . ;
    process class main program;
    begin
        L: detach;
          go to L
    end main program;
    ref (main program) main;
    SQS :- new head;
    main :- new main program;
    main.event :- new event notice (0, main);
    main.event.into (SQS)
end SIMULATION
```

3.3.1. Sequencing Procedures

A process is in one of four states with respect to scheduling. The process associated with the first event notice of the sequencing set is the process currently being executed and is said to be *active.* The processes which are associated with the other event notices of the sequencing set are scheduled for future execution and are said to be *suspended.* If a process is not associated with an event notice in the sequencing set it is said to be *passive* unless its execution has reached the end of the body of the class process in which case it is said to be *terminated.*

An active process can change its own state or the state of another process with a call to one of the following sequencing procedures.

(a) The procedure "hold" stops the execution of the active process and reschedules it for future execution. The next process scheduled for execution then becomes active. The procedure "hold" accomplishes these actions in two steps. First the "evtime" attribute of the first event notice is changed and the event notice is reinserted into the sequencing set so as to maintain the chronological order. Then the primitive sequencing procedure "resume" is called to update the OSC.

(b) The procedure "passivate" stops the execution of the active process but does not reschedule it for future execution. The next process scheduled for execution then becomes active. This is achieved by a call to the procedure "resume".

(c) The procedure "activate" schedules the execution of a process. It can either schedule the execution of a passive process, thereby making it either active or suspended, or reschedule the execution of an active or suspended process. In this paper the procedure call "activate (X)", with no other arguments, will be used for immediate activation of the process referred to by X.

3.3.2. Scheduling Active Phases of the Main Program

To perform a simulation a user will write a main program which is a block prefixed by the class SIMULATION. In the main program will be included the declarations of the subclasses of the class "process" which are needed for the simulated model. Also the main program will generate processes and interact with them. To provide for this interaction it is necessary to be able to schedule the main program.

The previous section described a mechanism for scheduling the active phases of processes. Although the main program is not a process, the same mechanism is also used to schedule its active phases. This is accomplished by associating a process "main program" with the main program. Whenever this process is activated, it issues a call to "detach" which has the effect of passing control to the main program.

The interaction between the main program and the processes is next illustrated by an example. Consider the simulation of a facility functioning on a first-come-first-served basis. Users arrive at the facility randomly (negative exponential distribution), enter a queue and wait their turn to be served. The first user in the queue is the one being served. When that user is finished with the facility he leaves the queue and passes control of the facility to the user who is then first in the queue.

```
SIMULATION begin
    Boolean facility busy; ref (head) queue;
    real service time;
    process class user;
    begin
        into (queue);
        if facility busy then passivate;
        facility busy:= true:
        hold (service time);
        facility busy:= false;
        out;
        if queue.SUC =/= queue then
            activate (queue.SUC qua process)
    end user;
    queue :- new head; service time:= 10;
    cycle: activate (new user);
        hold(negexp(. . .));
        go to cycle
end
```

The progression of the simulation is described below. During the initial operations of the class SIMULATION the sequencing set is created and it is initialized to contain the event notice for the process "main program". This is illustrated in Fig. 9(a). Thus when the main program is being executed, the process "main program" will indeed correspond to the first event notice in the sequencing set. Other processes will now be generated and scheduled by the main program. This is done in the "cycle" of the above simulation example; the generation is the result of the evaluation of "new user" and the scheduling is performed by the procedure "activate". Note that the first instruction in the class "process" is a call to detach so after the main program generates a process, control is immediately passed back to the main program. Fig. 9(b) shows the sequencing set at this time. The main program can relinquish control by calling either the procedures "hold" or "activate". In the above example "hold" is called. The primitive procedure "resume" called by "hold" (or "activate") passes control to the process next scheduled for execution and the simulation continues. The sequencing set after the call to "hold" is shown in Fig. 9(c).

3.4. Generalization: Multi-level Quasi-parallel Systems

Quasi-parallel systems with multiple levels may be introduced by means of prefixed blocks. An instance of a prefixed block is initially detached. The objects generated in the prefixed block and detached by the procedure "detach" form a quasi-parallel system which is at a

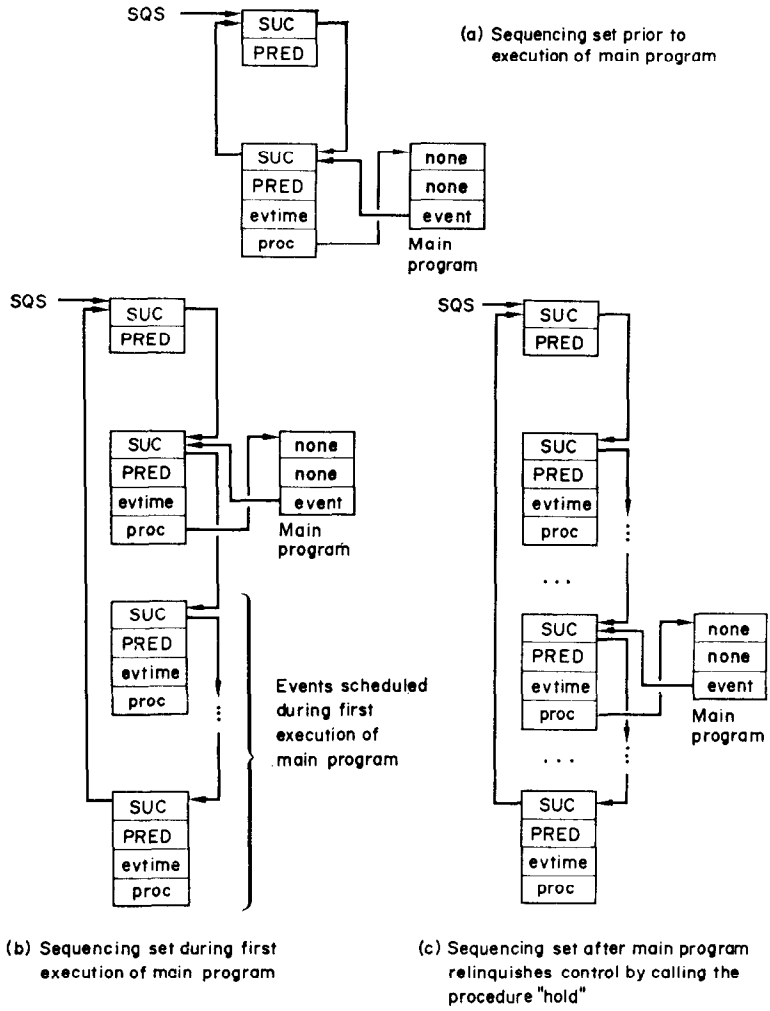


FIG. 9. Evolution of the sequencing set.

different level than the system to which the prefixed block belongs. The degrees of nesting of prefixed blocks correspond to the successive quasi-parallel system levels. For a quasi-parallel system at a given level, the prefixed block and the blocks and objects attached to it act as a main program. Using this definition, the procedures "detach" and "resume" can be generalized for a quasi-parallel system with an arbitrary number of levels.

Acknowledgements

The authors are grateful to Christian Scherer, Bernard Lang, and Jean-Paul Rissen for their perceptive remarks and suggestions on an earlier version of this paper.

This work was sponsored by the Délégation à l'Informatique under contract number 69-02-006-212-75-01.

Bibliography

1. DAHL, O. J., and NYGAARD, K., SIMULA—A Language for Programming and Description of Discrete Event Systems. Introduction and User's Manual. Norwegian Computing Center, Forskningsveien 1B, Oslo 3, Norway (1966).
2. DAHL, O. J., and NYGAARD, K., SIMULA—An ALGOL-Based Simulation Language. *Comm. ACM.* **9** (Sept. 66) pp. 671–678.
3. DAHL, O. J., and NYGAARD, K., Class and Subclass Declarations. In *Simulation Programming Languages* J. N. Buxton (Ed.), North Holland, Amsterdam (1968), pp. 158–174.
4. DAHL, O. J., MYHRHAUG, B., and NYGAARD, K., SIMULA 67—Common Base Language. Norwegian Computing Center, Forskningsveien 1B, Oslo 3, Norway (May 1968).
5. HOARE, C. A. R., Record Handling. In *Symbol Manipulation Languages and Techniques* J. G. Bobrow (Ed.), North Holland Amsterdam (1968), pp. 262–284.
6. NAUR, P. (Ed.) *et al.* Revised Report on the Algorithmic Language ALGOL 60. *Comm. ACM* **6**, 1 (Jan. 1963), pp. 1–17.
7. WIRTH, N., and HOARE, C. A. R., A Contribution to the Development of ALGOL. *Comm. ACM* **9**, 6 (June 1966), pp. 413–432.

Appendix A. Virtual Entities

A class defines a model for entities having common attributes. There are cases where an attribute of a given name is to take a different meaning for various subclasses of the class considered. This effect can be achieved in SIMULA by the inclusion of a *virtual* part in a class declaration. Such a declaration is of the following form:

```
K class M (P); S; V;
    begin D;
    I
    ; inner;
    F
end
```

V designates the *virtual part*. It is composed of **"virtual:"** followed by a list of virtual items. These can be labels, switches, or procedures. The rules of concatenation are extended to virtual parts in an obvious way.

The meaning of a virtual attribute is determined as follows. Given an object of class N. Suppose that an attribute X is specified virtual in the class M belonging to the prefix sequence of N. An occurrence of X is interpreted as virtual in M and in all classes inner to M. The matching definition for the virtual attribute X is the innermost definition of X.

This is illustrated by the following example. The procedure "show" is specified virtual in the class REAL. It is defined in both REAL and in COMPLEX. When "show" is called for X, the definition in the class REAL is used. When "show" is called for Y, two definitions are to be considered. The innermost definition, i.e. that given in the class COMPLEX, is the definition effectively used.

```
begin
    class REAL (real part); real real part;
        virtual : procedure show;
    begin
        procedure show;
        begin
            outfix (real part, 2, 5);
            outimage
        end
    end;
    REAL class COMPLEX (imaginary part); real imaginary part;
    begin
        procedure show;
        begin
            outfix (real part, 2, 5);
            if sign (imaginary part) > 0 then
                outtext (' + i')
            else
                outtext (' — i');
            outfix (abs(imaginary part), 2, 5);
            outimage
        end
    end;
    ref (REAL) X; ref (COMPLEX) Y;
    X :- new REAL (3); Y :- new COMPLEX (2, 4);
    X.show; Y.show;
    comment X.show prints 3.00
            Y.show prints 2.00 + i 4.00
end
```

Appendix B. Symbolic Differentiation

The concepts of class and prefixed block are illustrated in what follows by their application to symbolic differentiation of expressions. The SIMULA 67 version of this classical problem was inspired by the formulation given by Hoare [5].

Definition of Classes

The class SYMBOL serves as a framework for the definition of the concepts and entities used in symbolic differentiation. Expressions are objects of the class EXPR. This class, its subclasses CONSTANT, VARIABLE and PAIR, as well as the subclasses of the latter—namely SUM and DIFF—are the class attributes of SYMBOL. The reference variables "zero" and "one" which serve to designate the constants "0" and "1" are additional attributes of SYMBOL. The objects corresponding to these two constants are generated during the initial operations of SYMBOL.

The reference variable "delta" is the unique attribute of the class EXPR. Whenever the derivative of an EXPR object is calculated, a reference to this derivative will be assigned to the "delta" field of the object. The procedure "deriv" is specified virtual in the class EXPR.

Corresponding to the subclasses CONSTANT, VARIABLE and PAIR of the class EXPR are objects having attributes which are the value of a constant, the name of a variable, and the left and right parts of a PAIR respectively. In addition, the subclasses CONSTANT and VARIABLE of EXPR, as well as the subclasses SUM and DIFF of PAIR, have a procedure attribute called "deriv". Note that a different procedure is defined for each of these subclasses in order to illustrate the level by level elaboration of a SIMULA 67 program. The call to a procedure "deriv" on behalf of a given expression has the effect of calculating the derivative of the expression with respect to the variable passed as a parameter. Furthermore, a reference to this derivative is assigned to the variable "delta". Thus, after the execution of the instruction

L :- M.deriv(N)

the reference variable L refers to the derivative of M with respect to N. Furthermore, the relation

L == M.delta

has the logical value **true**.

```
class SYMBOL;
begin
    class EXPR; virtual: ref(EXPR) procedure deriv;
    begin
        ref (EXPR)delta;
    end EXPR;
    EXPR class CONSTANT(k); real k;
    begin
        ref(EXPR)procedure deriv(X); ref(VARIABLE)X;
        deriv :- delta :- zero;
    end CONSTANT;
    EXPR class VARIABLE(id); value id; text id;
    begin
        ref (EXPR) procedure deriv(X); ref(VARIABLE)X;
        if X == this VARIABLE then
          deriv :- delta:- one
        else
          deriv :- delta:- zero;
    end VARIABLE;
    EXPR class PAIR (left, right); ref(EXPR) left, right;;
    PAIR class SUM;
    begin
        ref(EXPR)procedure deriv(X); ref(VARIABLE)X;
        begin
            ref(EXPR) l prime, r prime;
            l prime :- left.deriv(X);
            r prime :- right.deriv(X);
            delta :- if l prime == zero then
                        r prime
                     else
                       if r prime == zero then
                         l prime
                       else
                         new SUM (l prime, r prime);
            deriv :- delta;
        end deriv;
    end SUM;
```

```
    PAIR class DIFF;
    begin
        ref (EXPR) procedure deriv (X); ref (VARIABLE)X;
        begin
            ref (EXPR) l prime, r prime;
            l prime :- left.deriv(X);
            r prime :- right.deriv(X);
            delta :- if rprime == zero then
                        l prime
                     else
                        new DIFF (l prime, r prime);
            deriv :- delta;
        end deriv;
    end DIFF;
    ref(CONSTANT) zero, one;
    comment the initial operations of the class SYMBOL
    are the following;
    zero :- new CONSTANT(0);
    one :- new CONSTANT(1);
    zero.delta :- one.delta :- zero;
end SYMBOL;
```

Using the Class SYMBOL

Now that the class SYMBOL has been defined, it is possible to utilize the entities defined in SYMBOL inside a block prefixed by SYMBOL. The short program which follows requires the generation of the expression e = (x + y) — (z — 4) using the auxiliary variables u = (x + y) and v = (z — 4). This generation is followed by the calculation of the partial derivative f of the expression e with respect to the variable y.

```
SYMBOL begin
    ref (VARIABLE) x, y, z;
    ref(EXPR)u,v,e,f;
    x :- new VARIABLE('x'); y :- new VARIABLE('y');
    u :- new SUM(x,y);
    z :- new VARIABLE('z');
    v :- new DIFF(z,new CONSTANT(4));
    e :- new DIFF(u,v);
    f :- e.deriv(x)
end
```