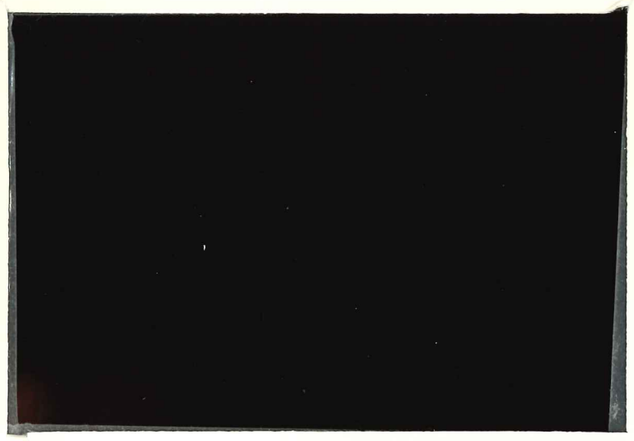
NORWEGIAN COMPUTING CENTER
FORSKNINGSVEIEN 1 B
OSLO 3 - NORWAY
JUNE 1967



SIMULA 67

COMMON BASE DEFINITION

BY

OLE-JOHAN DAHL AND KRISTEN NYGAARD

Introduction

The SIMULA 67 language is based upon experiences with the SIMULA I language [1] and is intended as a general programming language with a well developed simulation capability.

As SIMULA I, SIMULA 67 is a true extension of ALGOL 60 [2], except that the own concept and integer labels are excluded. Specifications are required for formal parameters. The language will consist of two levels:

- the SIMULA 67 Common Base, which should be a part of any implementation of the language

- the SIMULA 67 Complete Version, which contains the Common Base as a true subset. The main difference is that the Complete Version also will contain a unification of the concepts "class" and "type", with the natural consequences of such a feature.

The development and maintenance of SIMULA 67 will be supervised by a SIMULA Standards Group.

The main new features of SIMULA 67 are:

- the "class" declaration, with the possibility of introducing subclasses by means of a prefix notation. This allows the construction of hierarchies of data structures and associated actions.

- the "virtual quantities". These quantities are specified in a class declaration and their precise meaning may be defined (or redefined) in subclasses.

- new means for referencing of objects, built upon the ideas of C.A.R. Hoare [3], but extended to exploit the class-subclass concept.

A class may also be used as prefix to ordinary blocks, the effect being that all concepts defined in this class are directly available within the block. In this way a user may provide himself and others with augmented versions of the language, containing aggregated concepts oriented towards a specialized area of applications.

SIMULA 67 will contain one such problem oriented class as part of the language: the class "SIMULATION". When it is used as a prefix to a block features corresponding to those of SIMULA I are available. Transcription of SIMULA I programs into SIMULA 67 will be very simple when the prefix "SIMULATION" is used.

Standard classes oriented towards data processing, engineering problems, etc. are possible and natural additions to the language.

The range of application will be the same for both the Common Base and the Complete Version, the difference being greater user convenience and improved machine efficiency in certain areas by using the Complete Version.

2. Class Declarations

2.1 Syntax

<declaration> ::= <ALGOL declaration>|<class declaration>|<empty>
<class identifier> ::= <identifier>
<prefix> ::= <empty>|<class identifier>
<virtual part> ::= <empty>|virtual : <specification part>
<class body> ::= <statement>|<split body>
<split body> ::= <block head><statement list>
; inner; <compound tail>
<statement list> ::= <empty>|<statement list>;<statement>
<class declaration> ::= <prefix><main part>
<main part> ::= class <class identifier>
<formal parameter part>;
<specification part><virtual part>
<class body>

2.2 Semantics

A class declaration serves to define the class associated with a class identifier. The class consists of objects, each of which is a dynamic instance of the class declaration (cf. 4.1 and 4.2.2) on object designators.

The class body always acts like a block. If it takes the form of a statement which is not an unlabelled block, the class body is identified with an implicit block enclosing the statement.

For a given object the formal parameters, the quantities specified in the virtual part, and the quantities declared local to the class body, are called the "attributes" of the object. The statements of the class body constitute the "operation rule" of the object. The operation rule of a split body is divided into the "initial operations" and the final operations", separated by the symbols ";inner;". For an object whose class body is a split body the symbol "inner" represents a dummy statement.

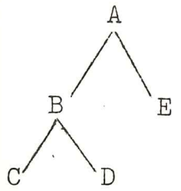
2.2.1 Subclasses

A class declaration with the prefix "C" and the class identifier "D" defines a subclass D of the class C. An object belonging to the subclass consists of a "prefix part", which is itself an object of the class C, and a "main part" described by the main part of the class declaration. The two parts are "concatenated" to form one compound object. The class C may itself have a prefix.

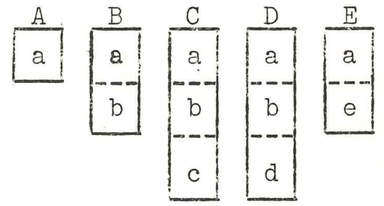
Let $C_1, C_2, \ldots, C_n$ be classes such that C_1 has no prefix and C_k has the prefix C_{k-1} $(k=2,3,\ldots,n)$. Then $C_1, C_2, \ldots, C_{k-1}$ is called the prefix sequence of C_k $(k=2,3,\ldots,n)$. The subscript k of C_k $(k=1,2,\ldots,n)$ is called the "prefix level" of the class. C_i is said to include C_j if $i \leq j$, and C_i is called a subclass of C_j if $i > j$ $(i,j=1,2,\ldots,n)$. The prefix level of a class D is said to be "higher" than that of a class C if D is a subclass of C.

The diagram below depicts the following class hierarchy and the structure of objects belonging to each class.

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



Class relationships



Structures of objects

A capital letter denotes a class. The corresponding lower case letter represents the attributes of the main part of an object belonging to that class. In an implementation of the language the object structures shown above may indicate the allocation in memory of the values of those attributes which are simple variables.

2.2.2 Concatenation

A compound object can be described formally by a "concatenated" class declaration, which is defined by the following rules.

1) If the prefix refers to a concatenated class declaration, in which identifier substitutions have been carried out, then the same substitutions are effected within the main part.

2) If identifiers declared or specified within the main part coincide with identifiers occurring within the prefix declaration, then name conflicts are avoided through suitable systematic changes of these identifiers within the main part. Identifiers corresponding to virtual quantities of the prefix class are not changed.

3) The formal parameter list of the concatenated declaration consists of the formal parameters of the prefix followed by those of the main part.

4) The specification part is that of the prefix followed by that of the main part.

5) The virtual part contains the virtual specifications of the prefix followed by those of the main part.

6) The block head of the class body contains the declarations in the block head of the prefix body followed by those of the main part.

7) In the concatenation of operation rules, that of the main part is considered "inner" to that of the prefix. From the prefix the operation rule of the main part may be referenced through the symbol "inner". At the place of "inner" the operation rule of the main part will be inserted. If the prefix does not have a split body, it will be interpreted as if the symbols ";inner;" is inserted between the last statement of the operation rule and the final end.

8) Name conflicts within the concatenated class declaration are resolved according to rules stated in section 2.2.4.

It follows that for a prefixed class declaration the attributes of the concatenated prefix sequence are accessible as local quantities from within the class body. If an attribute identifier of the main part coincides with a non-virtual one (cf. 2.2.4) of the prefix sequence, then the declaration or specification occurring in the prefix sequence is superceded by the one occurring within the main for references to the identifier occurring within the main part or within any subclass.

An expression, which is a subscript bound of an array declaration belonging to a class body, may reference any formal parameter of the concatenated class declaration, virtual quantities excepted.

.2.3 Formal Parameters of Class Declarations

Specification is required for each formal parameter of a class declaration. The following specifiers are accepted:

<type>, array and <type> array.

<type> parameters are transmitted "by value", i.e. to a <type> parameter of an object, an initial assignment is made of the value denoted by the corresponding actual parameter of the generating reference (cf. 4.2.2).

Array parameters are transmitted "by location", i.e. an array parameter of an object designates the array denoted by the corresponding actual parameter. The type of the actual array parameter must be subordinate (cf. 3.2.4)to the type of the formal parameter specification.

2.4 Virtual Quantities

A declaration or specification D1 is said to match another one D2 of the same identifier, if

1) D1 and D2 define quantities of the same kind, and

2) the type of D1 coincides with, or is subordinate to (cf. 3.2.4) the type of D2.

A virtual quantity may be declared or specified as parameter within the class declaration containing the virtual specification. Name conflicts caused by concatenation (cf. 2.2.2) are dealt with in the following way. Re-declaration or -specification of an attribute is permitted, if and only if

1) it matches any conflicting one given at a lower prefix level, and

2) it matches a virtual specification of the same identifier.

Conflicting virtual specifications are illegal. For an object belonging to a given class, a virtual quantity is identified with the attribute defined at the highest prefix level, whose definition matches the virtual specification. The identification is valid at all prefix levels which, after concatenation, contain the virtual specification. Reference to a virtual quantity, for which no matching definition has been given, constitutes a run time error.

It follows that a virtual quantity introduced at one prefix level makes it possible to refer, at this level, to a quantity defined at some higher prefix level. Alternative definitions may be

provided within different subclasses of the given class. Furthermore, a definition given at one prefix level (which may be the level of the virtual specification itself) will be superceded by a matching definition provided within a subclass.

The following specifiers are accepted for virtual quanitities:

label, switch, procedure and <type> procedure.

3. Types

3.1 Syntax

<type> ::= <ALGOL type>|ref<qualification>|string
<qualification> ::= <empty>|(<class identifier>)
<expression> ::= <ALGOL expression>|<string expression>|
<reference expression>

3.2 Semantics

3.2.1 Reference Types

Associated with an object is a unique value of type ref, which is said to "reference" or "point to" the object. A reference value may, by qualifying a declaration or specification by a class identifier, be restricted to point to objects belonging either to the stated class or to any of its subclasses.

The reference value "none" is a legal value of any quantity of type ref, regardless of its qualification.

3.2.2 Strings

ALGOL 60 is extended by including the symbol "string" as a type declarator. String operations and expressions to be included in the Common Base will be defined by the SIMULA Standards Group.

.2.3 Initialization

Any dynamic instance of a declared variable is given an initial value at the time of entry into the block to which it is local. One initial value is associated with each <type>:

real	0
integer	0
Boolean	false
string	implementation dependent
ref	none, regardless of qualification

.2.4 Subordinate Types

The concept "subordinate" is defined for reference types. One reference type is said to be subordinate to a second one if the qualification of the former is a subclass of the class which qualifies the latter. No qualification will be regarded as qualification by a universal class, which includes all declared classes.

A proper procedure is said to be of "type universal". Any type is subordinate to the universal type.

+. Reference Expressions

+.1 Syntax

```
<simple reference expression> ::= none|<variable>|
<function designator>|<generating reference>|
<local reference>|<qualified reference>|(reference expression)
<reference expression> ::= <simple reference expression>|
if <Boolean expression> then
<simple reference expression> else <reference expression>
<generating reference> ::= new <class identifier>
                           <actual parameter part>
<local reference> ::= this <class identifier>
<qualified reference> ::= <simple reference expression>.
                          this <class identifier>
```

2 Semantics

A reference expression is a rule for computing a reference value. Thereby reference is made to an object, except if the value is "none", which is a reference to "no object".

2.1 Qualification

Any reference expression has a qualification associated to its type. A variable or function designator is qualified according to the declaration or specification of the variable or procedure. A generating reference is qualified by the class identifier following the symbol "new". A local reference or a qualified reference is qualified by the class identifier following the symbol "this". The expression "none" is not qualified (or is qualified by the universal class).

The qualification of a conditional reference expression is the class at the highest possible prefix level, which includes the qualifications of both alternatives.

2.2 Object Generation

As the result of evaluating generating reference, an object belonging to the stated class is generated. Each actual parameter is evaluated, and its value is assigned to the corresponding formal parameter of the object. (The "value" of an array identifier is a pointer to the referenced array, cf. 2.2.3.) Then the (concatenated) class body is called into execution. Control returns to the generating reference when exit is made out of the class body through its final end, or whenever the basic procedure "detach" is executed (cf. 8.1). The value of the generating reference is a reference to the generated object.

2.3. Local Reference

A local reference "this C" is a meaningful expression within a block of any of the following kinds:

1) a class body

2) a prefixed block (cf. sect. 7),

3) a <connection block 1>, and

4) a <connection block 2> (cf. 6.2),

provided that the qualification associated with the block is C or a subclass of C, and that the local reference is within the scope of the class declaration C.

The qualification associated with each of the above block types is respectively

1) the class identifier of the class declaration,

2) the class identifier of the block prefix,

3) the class identifier of the preceding connection clause,

4) the qualification of the preceding reference expression.

The value of a local reference is a reference to the object associated with the smallest enclosing block in which the local reference is meaningful. If there is no such block, the local reference is illegal. The object associated with the block is either the current instance of the block (case 1,2) or the connected object (case 3,4).

2.4 Instantaneous Qualification

Let X represent any simple reference expression, and let C and D be class identifiers such that D is the qualification of X. The qualified reference "X.this C" is then a legal reference expression, provided that C includes D or is a subclass of D. Otherwise, i.e. if C and D belong to disjoint prefix sequences, the qualified reference is illegal.

If the value of X is none or refers to an object belonging to a class not included in C, the evaluation of X.this C constitutes a run time error. Otherwise, the value of X.this C is that of X.

For the purpose of this **rule** the "scope" is defined after the execution of all indicated concatenations.

The last rule represents a compile time check which has several important consequences.

1) It simplifies the problem of combining security and run-time efficiency. Increased economy can be obtained with respect to storage space as well as execution time.

2) It protects the user against making any reference assignment, whose consequences for the storage economy are difficult to overlook. The rule guarantees that any retrievable reference value is useful in the sense that the attributes of the referenced object are accessible through remote referencing (cf. sect. 6).

3) The dynamic scope of an object becomes limited by that of its class declaration, which makes "wholesale" de-allocation of data space possible.

Rules similar to those above apply to assignments of reference values implicit in for clauses and in the initializations of formal parameters called by value.

5.2 Relations

5.2.1 Syntax

<relation> ::= <ALGOL relation>|
<reference expression> = <reference expression>|
<reference expression> ≠ <reference expression>|
<reference expression> is <class identifier>|
<reference expression> in <class identifier>

5.2.2 Semantics

The operators "=" and "≠" may be used to compare reference values for equality. Two reference values which are equal refer to the same object, or they are both none.

The operators "is" and "in" may be used to test the class membership of a referenced object. "X is C" has the value true if the value of X refers to an object belonging to the class C, otherwise the value is false. "X in C" has the value true if the value of X refers to an object belonging to a class included in C, otherwise the value is false. Both relations are false if X has the value none.

5.3 For statements

5.3.1 Syntax

<for list element> ::= <ALGOL for list element>|
<reference expression>|
<reference expression> while <Boolean expression>

5.3.2 Semantics

The extended for statement will facilitate the processing of list structures. The implied assignment operations are subject to the rules of section 5.1.2.

6. Attribute Referencing

An attribute of an object is identified completely by the following items of information:

1) the value of a <reference expression> identifying an object,
2) a <class identifier> specifying a class, which includes that of the object, and
3) the <identifier> of an attribute declared for objects of the stated class.

The class identification, item 2, is implicit at run time in a reference value, however, in order to obtain run time efficiency, it is necessary that this information is available to the compiler.

For a local reference to an attribute, i.e. a reference from within the class body, items 1 and 2 are defined implicitly. Item 1 is a reference to the current instance (i.e. object), and item 2 is the class identifier of the class declaration.

Non-local (remote) referencing is either through remote identifiers or through connection. The former is an adaptation of the technique proposed in [3], the latter corresponds to the connection mechanism of SIMULA I [1].

6.1 Remote Identifiers

6.1.1 Syntax

```
<remote identifier> ::= <reference expression>.<identifier>
<identifier 1> ::= <identifier>|<remote identifier>
<variable identifier 1> ::= <identifier 1>
<simple variable 1> ::= <variable identifier 1>
<array identifier 1> ::= <identifier 1>
<variable> ::= <simple variable 1>|<array identifier 1>
                                    [<subscript list>]
```

<procedure identifier 1> ::= <identifier 1>
<label 1> ::= <identifier 1>
<switch identifier 1> ::= <identifier 1>
<actual parameter> ::= <expression>|<array identifier 1>|
<switch identifier 1>|<procedure identifier
<function designator> ::= <procedure identifier 1>
<actual parameter part>
<switch designator> ::= <switch identifier 1>[<subscript expression
<simple designational expression> ::= <label 1>|<switch designator>|
(<designational expression>)
<procedure statement> ::= <procedure identifier 1><actual parameter part>

6.1.2 Semantics

A remote identifier identifies an attribute of an individual object. Item 2 above is defined by the qualification of the reference expression. If the latter has the value none, the evaluation of the remote identifier constitutes a run time error.

6.2 Connection

6.2.1 Syntax

<connection block 1> ::= <statement>
<connection block 2> ::= <statement>
<connection clause> ::= when<class identifier>do<connection block 1>
<otherwise clause> ::= <empty>|otherwise<statement>
<connection part> ::= <connection clause>|<connection part>
<connection clause>
<connection statement> ::= inspect <reference expression> do
<connection block 2><otherwise clause>|
inspect <reference expression>
<connection part><otherwise clause>

6.2.2 Semantics

The connection mechanism serves a double purpose:

1) To define item 1 above implicitly for attribute references within connection blocks. The reference expression of a connection statement is evaluated once and its value is stored. Within a connection block this value is said to reference the connected object. It can itself be accessed through a <local reference> (cf. 4.2.3).

2) To discriminate on class membership at run time, thereby defining item 2 implicitly for attribute references within alternative connection blocks. Within a <connection block> item 2 is defined by the class identifier of the connection clause. Within a <connection block 2> it is defined by the qualification of the reference expression of the connection statement.

Attributes of a connected object are thus immediately accessible through their respective identifiers, as declared in the class declaration corresponding to item 2. These identifiers act as if they were declared local to the connection block.

The otherwise clause following a <connection block 2> is entered if and only if the computed reference value is none. An empty otherwise clause represents the symbol "otherwise" followed by a dummy statement.

Prefixed Blocks

.1.1 Syntax

<block prefix> ::= <class identifier><actual parameter part>
<main block> ::= <unlabelled block>
<unlabelled prefixed block> ::= <block prefix><main block>
<prefixed block> ::= <unlabelled prefixed block>|<label>:<prefixed block>
<block> ::= <ALGOL block>|<prefixed block>
<program> ::= <block>

7.1.2 Semantics

An instance of a prefixed block is a compound object, which is the result of concatenating an object of the stated class and an instance of the main block. The formal parameters of the former are initialized as indicated by the actual parameters of the block prefix. Any virtual quantity is identified by the quantity defined by a matching declaration in the block head of the main block, or by the matching definition at the highest prefix level of the prefix sequence. The operation rule of the concatenated object is defined by principles similar to those given in 2.2.2. The object belongs to an anonymous subclass of the class associated with the block prefix. The scope of this subclass is the prefixed block.

An instance of a prefixed block is an initially detached object (cf. sect. 8). When exit is made through the final end of the concatenated operation rule, control proceeds to the statement following the end of the prefixed block.

An instance of a program is an initially detached object, whether it is a prefixed block or not.

8. Quasi-parallel Sequencing

The basic constituent parts of a SIMULA program execution are dynamic instances of blocks. There are the following types of blocks:

1) Prefixed blocks
2) Sub-blocks
3) Procedure blocks
4) Connection blocks
5) Class blocks

In ALGOL 60 the block instances form a nested structure. A block instance is said to be "attached to" the one containing it. A block instance A is said to "enclose dynamically" an instance B, if B is attached to A, or if there is a block instance C attached to A, such that C dynamically encloses B.

A block instance is said to be the "local to" the one which contains its describing program text (or, if concatenated, the text describing its main part). An instance of a sub-block is attached to and local to the same block instance.

The "program sequence control", PSC, identifies that program point within a block instance which is currently being executed. For brevity, we shall say that the PSC is "positioned" at the program point and is "contained" in the block instance. If A is the block instance containing the PSC, then A and any block instance dynamically enclosing A are said to be "operating".

A block instance attached to another one is said to be in the "attached state", or simply "attached". When the PSC leaves an instance of a block, the instance is said to become "terminated".

Since ALGOL 60 is a subset of SIMULA 67, the above definitions are valid also for SIMULA program executions. In SIMULA 67, however, there are block instances which are, or may be, neither attached nor terminated. They are said to be in the "detached state". The table below shows the possible states and the initial state of block instances of the different types.

Block types	Possible states	Initial state
1	D, (T)	D
2,3,4	A, (T)	A
5	A, D, T	A

A: attached D: detached T: terminated

Any terminated block instance, except possibly instances of class blocks is inaccessible and therefore can be regarded as "not present". Thus, within its "dynamic scope" an instance of type 1 is permanently detached and an instance of type 2,3 or 4 is permanently attached. The program as a whole is a block of type 1 (cf. section 7).

An object is defined as a block instance of type 1 or 5, together with the block instances dynamically enclosed by it. These block instances are said to be "part of" the object. An object of type 5 can be attached and thereby part of another object.

A detached object A is said to "enclose" a detached object B, if

1) B is local to a block instance which is part of A, or
2) there is a detached object C, such that C is local to a block instance which is part of A, and C encloses B.

Let A be an instance of a prefixed block. All detached type 5 objects, whose smallest enclosing instance of a prefixed block is A, comprise together with A a "quasi-parallel system". These objects, including A, are said to be the "components" of the quasi-parallel system, and are said to be detached at the same "system level". A is called the "main program" of the quasi-parallel system. A detached object enclosing A is said to be detached at a "lower" system level. The program as a whole is detached at system level zero.

A component of a quasi-parallel system can not be referenced from outside the quasi-parallel system. It follows that the dynamic scope of any detached object is limited by that of the main program of its quasi-parallel system.

Each detached object has an associated "local sequence control", LSC. Associated with a quasi-parallel system is an "outer sequence control", OSC, which is the LSC of the smallest object enclosing its main program. The OSC of the quasi-parallel system at level zero is the PSC.

For any given quasi-parallel system, one and only one of its detached objects is said to be "active with respect to the OSC", or simply "active". The LSC of that object coincides with the OSC of the quasi-parallel system.

An instance of a prefixed block is initially active, i.e. it contains the OSC of its own quasi-parallel system. The OSC may move from one detached object to another one of the same system as described in section 8.2. The LSC of a detached object not containing the OSC remains positioned at the program point at which the OSC left the object the last time.

There exists at any given time a sequence of active detached objects $X_0, X_1, \ldots X_n$ such that:

1) X_k is active at system level k ($k = 0,1 \ldots,n$).
2) X_{k+1} is a member of a quasi-parallel system whose main program is enclosed by X_k ($k = 0,1,\ldots,n - 1$).

The objects of this sequence, all containing the PSC, are said to be operating. The LSC of a detached object remains fixed as long as it is not a member of the sequence of operating objects.

.1 <u>The "detach" statement</u>

Let the smallest operating and attached object be X. Then the execution of the statement "detach" has the following consequences.

1) The object becomes detached at the level of the smallest enclosing prefixed block instance, its LSC positioned at the end of the statement.

2) The PSC returns to the generating reference of the object, within the block instance to which it was attached. The reference value associated with the object is transmitted as the function value of the generating reference.

If there is no operating and attached object, the statement constitutes a run time error.

2 The "resume" statement

The resume statement is a basic statement for quasi-parallel sequencing. "resume" is formally a procedure with one unqualified reference parameter.

Let the actual parameter of a resume statement be a reference to a detached object Y, which is a component of a quasi-parallel system S. Since Y cannot be referenced from outside S, there must be a component X of S which is operating. The resume statement has the following effects.

1) The OSC of S leaves X. As a consequence the PSC leaves X and all operating objects detached at system levels higher than X. The LSC of each object remains at the end of the resume statement.

2) The OSC of S enters Y at the current position of its LSC. As a consequence Y, and possibly a sequence of objects detached at system levels higher than that of Y, become operating.

If the actual parameter of a resume statement does not reference a detached object, the statement constitutes a run time error.

3 Object "end"

The final end of the operation rule of an object has an effect which depends on the type and state of the object.

Attached object: The object becomes terminated. The PSC returns to the generating reference of the object, within the block instance to which it was attached. The reference value associated with the object is transmitted as the function value of the generating reference.

Detached object: Exit through the final end of a detached type 5 object constitutes a run time error. When control passes through the end of a prefixed block, the object becomes terminated. The LSC of the enclosing object (and the PSC) continues to the next statement.

3.4 go to Statements

A go to statement leading to a program point outside the smallest operating detached object, has an undefined effect.

If control leaves an attached object by a go to statement, the object becomes terminated.

3.5 SIMULA I Sequencing Statements

The special syntax of the "scheduling statements" of SIMULA I (1, chapter 4) is part of the SIMULA 67 Common Base, except that a scheduling statement may not be followed by connection clauses. A compiler (or a preprocessor) will transform any statement of the form.

activate X ....., or

reactivate X .....

into

X. schedule (.....),

where there is a one to one correspondence between the form of the scheduling statement and the parameter values of the "schedule" procedure.

It follows that the special syntax may be used if the referenced object belongs to a class for which a procedure "schedule" is appropriately declared.

For the system defined class "SIMULATION" (cf. section 10), "schedule" is declared local to the class "process" (and thus also its subclasses) in a manner which corresponds to the SIMULA I scheduling statements.

All sequencing statements of SIMULA I, except "terminate", will have counterparts declared within the class "SIMULATION".

9. Random Drawing and Data Analysis

SIMULA 67 random drawing facilities will be those of SIMULA I (1, chapter 7).

The procedure "histo" of SIMULA I (1, chapter 8) will be part of SIMULA 67. The procedure "accum" in the same chapter will be a part of the class "SIMULATION". The procedure "hprint" is machine-dependent and will not be a part of SIMULA 67.

It should be noted that it will be possible in SIMULA 67 to define more powerful data collection facilities than those of SIMULA I. The inclusion of such facilities in the Common Base will have to be decided upon by the SIMULA Standard Group.

10. System Classes

The SIMULA I extensions of ALGOL 60 consist of

1) The "process" concept, classes of processes being described by "activity" declarations.

2) Means of referencing processes through items called "elements", and the introduction of circular two-way lists called "sets".

Along with these extensions are introduced a number of procedures and special syntax statements and declarations to allow efficient exploitation of the concepts - for quasi-parallel sequencing according to a "system time" criterion and for set operations.

In SIMULA 67 the SIMULA I concepts may be made available through two system defined classes:

1) "SIMSET", which introduces "sets", and

2) "SIMULATION", having "SIMSET" in its prefix sequence, which in addition introduces the "processes" and special sequencing facilities.

Transcription from SIMULA I to SIMULA 67 is straightforward within blocks having the prefix "SIMULATION".

10.1 The Class "SIMSET"

The class "SIMSET" contains facilities for the manipulation of circular two-way lists, called "sets". SIMULA 67 sets correspond to SIMULA I sets, but are implemented without the implicit use of "elements". "SIMSET" will be a prefix to the class "SIMULATION".

The reference variables and procedures necessary for set handling are introduced in standard classes declared within the class "SIMSET" which are considered parts of the language. Using these classes as prefixes, the relevant data and other properties are made parts of the objects themselves.

Both sets, and objects which may acquire set membership, should have references to a successor and a predecessor. Consequently they are made subclasses of the "linkage" class.

The sets are objects belonging to a subclass "set" of "linkage", objects which may be set members belong to subclasses of "link" which is itself another subclass of "linkage".

As stated in section 13, it will be permitted within the Common Base to exclude unqualified reference variables. The inconvenience caused by such a restriction may be circumvented by introducing an explicit "universal" class. The class is empty of attributes:

```
class univ;;
```

If "univ" is used as class prefix whenever possible, qualification by "univ" will for most purposes function as no qualification.

The definition of the other "SIMSET" classes are:

```
univ class linkage;
     begin ref (linkage) succ, prede ;
           ref (link) procedure suc ;
           suc := if succ is link then succ else none ;
           ref (link) procedure pred ;
           pred := if prede is link then prede else none ;
end linkage ;
```

The references "succ" and "prede" will only be available through the procedures "suc" and "pred" and not for direct assignment by the user. The handling of "succ" and "prede" in connection with set operations will be through the system procedures defined below.

```
linkage class link ;
       begin procedure out ; if succ ≠ none then begin
             succ.prede := prede ; prede.succ := succ ;
             succ.:= prede := none end out ;
             procedure follow (X) ; value X ; ref (linkage)X ;
             begin out ; if X ≠ none then begin
             if X.succ ≠ none then begin
             prede := X ; succ := X.succ ;
             succ.prede := X.succ := this linkage end end
             end follow ;

             procedure precede(X) ; value X ; ref (linkage)X ;
             begin out ; if X ≠ none then begin
             if X.succ ≠ none then begin succ := X ;
             prede := X.prede ; prede.succ := X.prede := this
             linkage end
             end end precede ;

             procedure into (S) ; value S ; ref (set) S ;
             begin out ; if S ≠ none then begin
             succ := S ; prede := S.prede ;
             prede.succ := S.prede := this linkage end
             end into ;
end link ;
```

```
linkage class set ;
        begin ref (link) procedure first ; first := suc ;
        ref (link) procedure last ; last := pred ;
        Boolean procedure empty ;
        empty := succ = this linkage ;
        integer procedure cardinal ;
        begin integer i ; ref (linkage) X ; i = 0 ; X :=
        this linkage ;
        for X := X.suc while X ≠ none do i := i + 1 ;
        cardinal := i end ;
        procedure clear ;
        begin ref (link) X ; for X := first while X ≠ none do
        X.out end ;
        succ := prede := this linkage
        end set ;
```

The last statement (the body) of the class declaration initializes the set to refer to itself through "succ" and "prede".

```
link class element (object) ; ref (univ) object ;;
```

Objects of the class element are similar to the "elements" of SIMULA I and may be used to simulate multiple set memberships.

Many of the procedures associated with sets in SIMULA I will either be omitted or substituted by corresponding "SIMSET" procedures:

proc(X) : substituted by the generative expression new element (X).
head(S) : no meaning within SIMSET
suc(X) : substituted by "suc" local to "linkage"
pred(X) : substituted by "pred" local to "linkage"
same(X,Y) : of little interest since elements are not a basic part of SIMULA 67
similar(X,Y),X = Y and X ≠ Y will be substituted by suitable Boolean expressions.
prcd(X,Y) : substituted by "precede" local to "link"
remove(X) : substituted by "out" local to "link"
first(S) : substituted by "first" local to "set"
last(S) : substituted by "last" local to "set"

successor(n,X) : omitted, since it has not been proved useful in SIMULA I.
number(n,S) : as for successor (n,X)
member(X,S) : omitted
exist(X) : substituted by X ≠ none, since "sethead" does not appear in SIMSET
empty(S) : substituted by "empty" local to "set"
ordinal(X) : as for successor (n,X). Also inefficient in execution
cardinal(S) : substituted by "cardinal" local to "set"
precede(X,Y) : substituted by "precede" local to "link"
follow(X,Y) : substituted by "follow" local to "link"
transfer(X,S) : substituted by "into" local to "link"
include(X,S) : substituted by "into" local to "link"
clear(S) : substituted by "clear" local to "set"

"SIMSET" may be used as prefix to user defined classes when the above set-handling facilities are wanted and the other "SIMULATION" concepts are unnecessary.

10.2 The Class "SIMULATION"

The class "SIMULATION" has "SIMSET" as a prefix. It contains two additional classes - "process" and its subclass "main program".

The definition of the class "SIMULATION" in terms of SIMULA 67, similar to the above definition of "SIMSET", will be given in a report which will be an appendix to this report and will be part of the SIMULA 67 Common Base.

The definitions will follow the pattern given in [4], but

1) the class "process" will have "link" in its prefix sequence

2) the procedure "schedule" is declared local to "process".

3) the "final operations" (cf.2.2.) of "process" is:
"; L : current := nextev ; resume (nextev) ; go to L end".

The use of "SIMULATION" as block prefix makes all features corresponding to SIMULA I available. The procedure "terminate" is however excluded, and the declarator "activity" is substituted by "process class".

11. Input - output

The quasi-parallel sequencing features of SIMULA 67 offers possibilities of convenient and powerful input-output handling within the language. These possibilities have not yet beeen sufficiently explored, and it has been decided to transfer the decisions on input-output to the SIMULA Standards Group. The SIMULA 67 Common Base Conference in Oslo, June 1967, felt that inclusion of input-output was "extremely important".

12. Separate Compilation

If an implementation permits user defined procedure and class declarations to be separately compiled, then a program should have means of making reference to such declarations as external to the program.

The following additional declarations are recommended as an optional part of the Common Base.

12.1
```
<external item> ::= <external identifier>|
                    <identifier> = <external identifier>
<external list> ::= <external item>|
                    <external list>,<external item>
<external declaration> ::=
        external procedure <external list>|
        external <type> procedure <external list>|
        external <prefix>class <external list>
```

12.2 Semantics

An external identifier is an identification with respect to an "operating system" of a separately compiled declaration.

An external declaration introduces local identifier for such declarations. A local identifier is either identical to the corresponding external one, or different as defined by an equality sign.

13. Optional Restrictions

The following language restrictions are optional within the Common Base.

1) Rule "S": All classes of the prefix sequence of a class declaration must be declared within the block head containing the given one. The block head may be that of a concatenated class declaration or prefixed block.

2) The unqualified "ref" is excluded as <type> declarator and specifier.

3) A local reference (4.2.3) referring to an instance of a prefixed block is not accepted as the right hand side of any reference assignment (including implicit ones).

These restrictions together imply the satisfaction of rule "R" of section 5.1.2 and may be easier to implement. They have also several other consequences which may simplify an implementation.

Although being quite restrictive the above rules preserve the essential capabilities of the SIMULA 67 language. It should be noted that rule "S" is too restrictive for an implementation permitting separate compilation of class declarations.

However, rule "S" does not prevent the use of system declared classes or other classes as block prefixes at different block levels. System classes to be used as prefixes to class declarations may be brought in "protected" by an enclosing class, which is used as block prefix.

REFERENCES

1. O-J Dahl and K. Nygaard: "SIMULA - a Language for Programming and Description of Discrete Event Systems. Introduction and Users's Manual". Norwegian Computing Center, Oslo 1965.

2. P. Naur, ed.: "Report on the Algorithmic Language ALGOL 60".

3. C.A.R. Hoare: "Record Handling". Lectures delivered at the NATO Summer School, Vilard-de-Lans, September 1966. (Academic Press).

4. O-J Dahl and K. Nygaard: "Class and Subclass Declarations". Paper presented at the IFIP Working Conference on Simulation Languages. Oslo, May 1967.

Table of Contents